Современные вычислительные системы и используемые в них методы управления нагрузкой и обработки информации сложны и разнообразны, состоят из большого количества разнородных элементов и учитывают множество разноплановых факторов. Это, с одной стороны, актуализирует задачи анализа их производительности и оптимизации, с другой стороны, делает построение достоверной аналитической модели практически недостижимым. В этой связи на первый план выступает такой мощный и гибкий инструмент как имитационное моделирование, при корректном использовании позволяющий учесть всю сложность системы в полном объеме. Цель настоящей книги состоит в том, чтобы научить заинтересованного читателя делать это. В ней рассмотрены реальные задачи из различных разделов теории вычислительных систем, для каждой разобрано построение и объектная программная реализация на языке C++ имитационной модели. Большое внимание уделено анализу и верификации данных моделирования, их соответствию здравому смыслу и в частных случаях известным аналитическим моделям. Книга будет полезна студентам старших курсов, аспирантам и всем специалистам, связанным с имитационным моделированием и задачами теории вычислительных систем.

Труб Илья Иосифович, Московский исследовательский центр SAMSUNG, ведущий инженер-программист. Труб Наталья Васильевна, Московский государственный гуманитарно-экономический университет, старший преподаватель кафедры математики.

978-3-659-78366-1

Моделирование вычислительных систем (C++)

Труб, Труб

Илья Труб
Наталья Труб

Имитационное моделирование вычислительных систем

Учебный практикум

LAP LAMBERT Academic Publishing

Илья Труб
Наталья Труб

Имитационное моделирование вычислительных систем

Илья Труб
Наталья Труб

Имитационное моделирование вычислительных систем

Учебный практикум

LAP LAMBERT Academic Publishing

Impressum / Выходные данные
Bibliografische Information der Deutschen Nationalbibliothek: Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
Alle in diesem Buch genannten Marken und Produktnamen unterliegen warenzeichen-, marken- oder patentrechtlichem Schutz bzw. sind Warenzeichen oder eingetragene Warenzeichen der jeweiligen Inhaber. Die Wiedergabe von Marken, Produktnamen, Gebrauchsnamen, Handelsnamen, Warenbezeichnungen u.s.w. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Библиографическая информация, изданная Немецкой Национальной Библиотекой. Немецкая Национальная Библиотека включает данную публикацию в Немецкий Книжный Каталог; с подробными библиографическими данными можно ознакомиться в Интернете по адресу http://dnb.d-nb.de.
Любые названия марок и брендов, упомянутые в этой книге, принадлежат торговой марке, бренду или запатентованы и являются брендами соответствующих правообладателей. Использование названий брендов, названий товаров, торговых марок, описаний товаров, общих имён, и т.д. даже без точного упоминания в этой работе не является основанием того, что данные названия можно считать незарегистрированными под каким-либо брендом и не защищены законом о брендах и их можно использовать всем без ограничений.

Coverbild / Изображение на обложке предоставлено: www.ingimage.com

Verlag / Издатель:
LAP LAMBERT Academic Publishing
ist ein Imprint der / является торговой маркой
OmniScriptum GmbH & Co. KG
Heinrich-Böcking-Str. 6-8, 66121 Saarbrücken, Deutschland / Германия
Email / электронная почта: info@lap-publishing.com

Herstellung: siehe letzte Seite /
Напечатано: см. последнюю страницу
ISBN: 978-3-659-78366-1

Copyright / АВТОРСКОЕ ПРАВО © 2015 OmniScriptum GmbH & Co. KG
Alle Rechte vorbehalten. / Все права защищены. Saarbrücken 2015

Содержание

Памяти наших друзей-программистов

Натальи Сашенко (1956-1998),

Юрия Ветчинова (1972-2001),

Любови Соболевой (1978-2004)

Введение

В 2006 году была опубликована книга одного из авторов [35], в которой были систематически изложены основы дискретно-событийного моделирования систем с помощью объектно-ориентированного подхода к проектированию моделирующего ПО, предложена и в деталях проиллюстрирована практическими примерами схема реализации этого подхода на языке C++. Книга эта оказалась достаточно востребованной, получила хорошие отзывы, представлена на многих тематических информационных ресурсах и в каталогах библиотек вузов и научных учреждений.

Однако, наличие в [35] только учебных задач несколько ограничило полноту рассмотрения вопроса и изложения тех идей, которые изначально планировал донести до читателей автор. Для их убедительности работе явно недоставало имитационных моделей, построенных для ярко выраженных научных, «нестуденческих» задач, моделей, за основу которых брались бы не искусственные (пусть даже достаточно содержательные) постановки, а реальные задачи, поставленные и рассмотренные известными специалистами в научной периодике. Это оставляло ощущение некоей недосказанности, незавершенности работы.

Настоящая книга предназначена для восполнения указанного недостатка и является естественным продолжением [35]. При этом цель, поставленная авторами, подразумевает вполне естественный вопрос: какую предметную область авторы собираются использовать для ее достижения, ведь применимость имитационного моделирования практически неограниченна. В качестве такой предметной области авторами выбрана теория вычислительных

систем не только в силу того, что она является областью их профессиональных интересов и научных исследований, а и потому что практически все ее задачи являются идеальным «полигоном» для построения и практического использования имитационных моделей.

Это обстоятельство несколько смещает тематическую направленность данной книги в сравнении с предшествующей. Если в [35] акцент делался в большей мере на основы имитационного моделирования и овладение языком C++ как его инструментом, то данная книга, во многом сохраняющая и поддерживающая эту направленность, все же в значительной степени адресуется и специалистам по теории вычислительных систем, т.к. имеет, помимо учебной, и существенную научную составляющую, и совершенно иной список литературы.

Теоретические основы имитационного моделирования, включая статистический анализ, исчерпывающе изложены в таких превосходных книгах, как [61] и [14]. В настоящей работе показано как смоделировать объектными средствами конкретную систему. Авторы постарались предоставить по каждой задаче всю информацию, чтобы сделать описанные имитационные эксперименты и их результаты полностью воспроизводимыми заинтересованным читателем, имеющим в своем распоряжении компилятор C++. Для каждой задачи подробно описан процесс объектного проектирования, приведены прокомментированные протоколы классов, составлен план проведения имитационных экспериментов и проанализированы полученные численные результаты.

В главе 1 описаны основы реализации генераторов случайных чисел для различных распределений вероятностей и их реализация на C, без

чего в принципе невозможно никакое имитационное моделирование. В главе 2 рассмотрена принципиальная архитектурная схема построения моделирующих программ на C++, используемая во всех последующих главах. Эти две главы в основном повторяют соответствующий материал [35] и представлены в настоящей книге для того, чтобы она имела полностью самостоятельный характер.

В главах 3–9 рассмотрены некоторые реальные задачи, относящиеся к различным разделам теории вычислительных систем. Взяты они из оригинальных публикаций, в том числе из собственных исследований авторов. В числе этих разделов — организация веб-кластеров и оптимизация выполнения запросов, маршрутизация, потоки данных, программные каналы и др. Основное внимание, как уже было отмечено, уделяется не столько проектированию моделирующих программ (хотя их текст прокомментирован самым подробным образом), а истории возникновения задачи, ее месте в современной теории вычислительных систем, различных вариантах постановки, обзоре литературы, анализе известных результатов и результатов, полученных с помощью написанной программы. В этих главах приведены упражнения для самостоятельной работы.

Представленные в книге тексты программ не ориентированы на какую-либо конкретную реализацию C++ и могут быть откомпилированы с помощью любого компилятора, реализующего стандарт ANSI/ISO C++, например пакета GNU g++ в среде ОС Linux.

Для понимания материала, изложенного в книге, читатель должен иметь определенную подготовку, впрочем, больших знаний не требуется. Эта подготовка включает знание основ объектно-

ориентированного программирования; базовые знания в области теории массового обслуживания, математического анализа, численных методов и теории вероятностей в объеме, предлагаемом в любом техническом вузе. Желателен также практический опыт разработки и отладки программ на C++, а также интерес к имитационному моделированию и задачам теории вычислительных систем.

Авторы надеются, что книга будет полезна студентам старших курсов и аспирантам, использующим объектный подход для разработки прикладного программного обеспечения, а также специалистам в области моделирования и теории вычислительных систем.

Помимо основного списка литературы, на элементы которого в тексте книги имеются ссылки, авторы на основе своего богатого опыта сочли полезным привести также еще один список рекомендованной литературы – учебной и монографической, состоящий из хорошо известных трудов отечественных и зарубежных специалистов и покрывающий фундаментальные вопросы в области имитационного моделирования и теории массового обслуживания.

Авторы выражают искреннюю благодарность своим работодателям, предоставляющим им возможности для творчества и профессионального развития – Московский исследовательский центр Samsung и Московский государственный гуманитарно-экономический университет.

Авторы благодарят ответственного редактора издательства Lambert Academic Publishing Юлию Артуровну Бурденюк за помощь и внимание при подготовке рукописи к печати.

Глава 1. Генерация вероятностных распределений

Вопрос генерации случайной последовательности, распределенной по заданному закону, хоть и не связан явно с технологией объектного моделирования, имеет исключительно важное значение. При отсутствии генератора случайных чисел (ГСЧ) программа просто не сможет имитировать ни одного случайного события в системе, как-то прибытие новой заявки или завершение ее обслуживания. ГСЧ играет, если можно так выразиться, роль кочегара, регулярно подбрасывающего в топку «уголь» — одно случайное число за другим. Без этого даже самый конструктивно совершенный паровоз двигаться не сможет. В этой главе мы кратко рассмотрим математические основы ГСЧ, примеры их реализации на языке С для различных законов распределения, а также ряд проблемных ситуаций и «ловушек», возникающих при этом.

Рассматриваемые в этой части книги примеры ГСЧ ни в коем случае не претендуют на эталонность программной реализации, присущую коммерческим библиотекам, хотя результаты работы этих программ и прошли успешно все необходимые статистические проверки. Автор всего лишь ставил целью отразить свое отношение к теме и свой личный опыт, приобретенный при работе с ГСЧ. У квалифицированного в данном вопросе читателя вполне могут быть иная точка зрения и иной стиль реализации.

1.1. Метод обратной функции

Предположим, что нам нужно генерировать независимые реализации

случайной величины X с заданной функцией распределения $F(x)$. Такая задача всегда возникает при моделировании систем с очередями, если входные потоки и длины заявок описываются случайными величинами. Чтобы отследить прибытие новой заявки или завершение обслуживания имеющейся, такую случайную величину предварительно нужно разыграть, а для этого необходим определенный математический метод, реализованный программно. Как мы далее убедимся, достаточно иметь в своем распоряжении генератор чисел, равномерно распределенных на интервале [0; 1]. Предположим, что такой генератор у нас уже имеется, и рассмотрим метод, известный как метод обратной функции [10], [28]. Прежде всего заметим, что $F(x)$ по смыслу является неубывающей функцией на конечном или бесконечном интервале. Для значения $0 \le u \le 1$ определим обратную функцию $F^{-1}(u)$ следующим образом: в точке разрыва $F(x)x = x_0$ зададим некоторое достаточно малое $\varepsilon > 0$ и, если $F(x_0 - \varepsilon) < u \le F(x_0 + \varepsilon)$, определим $F^{-1}(u) = x_0$; иначе, назначим для $F^{-1}(u)$ такое значение x, для которого $F(x) = u$. Далее зададим случайную величину X с помощью преобразования

$$X = F_x^{-1}(U), \qquad (1.1)$$

где U — случайная величина, равномерно распределенная на интервале [0; 1]. Идея метода заключается в следующем. Генерируем значение U, затем решаем уравнение (1.1) относительно X. Тогда справедливо

$$P\{X \le x\} = F(x), \qquad (1.2)$$

то есть случайная величина X имеет функцию распределения $F(x)$. Доказать (1.2) несложно.

Заметим, что $P\{X \le x\} = P\{F^{-1}(U) \le x\} = P\{U \le F(x)\}$. Теперь — ключевое рассуждение: если величина U равномерно распределена на интервале [0; 1], то $P\{U \le u\} = F(u) = u$. Значит, $P\{U \le F(x)\} = F(x)$, и (1.2) доказано. Рассмотрим простейший пример. Предположим, что X имеет экспоненциальное распределение со средним значением τ. Тогда $F(x) = 1 - e^{-x/\tau}$ $(0 \le x \le \infty)$, и уравнение (1.1) принимает вид $U = 1 - e^{-x/\tau}$. Решив его, получим $x = -\tau \ln(1 - U)$. Нетрудно показать, что $1 - U$ также равномерно распределена на [0; 1], как и U. Тогда вычисление можно упростить: $x = -\tau \ln U$. Таким образом, для реализации случайной величины X получаем реализацию случайной величины U и вычисляем X по формуле $x = -\tau \ln U$.

Отдельно рассмотрим случайную величину X, заданную дискретным распределением $P\{X = x_i\} = p_i$, $i = 1, 2, ..., n$. Для получения выборки из такого распределения вновь сгенерируем значение u и примем $X = x_i$, если

$$\sum_{j=1}^{i-1} p_j < u < \sum_{j=1}^{i} p_j \quad (j \;\; 1, 2, \ldots, n),$$

где сумма, не имеющая слагаемых, полагается равной нулю. Предположим, например, что нужно разыграть случайную величину, принимающую два значения, которые соответствуют таким событиям: следующее поступление заявки из входного пуассоновского потока произойдет раньше или позже завершения обслуживания находящейся в данный момент на сервере заявки. Длительность обслуживания распределена экспоненциально со средним $1/\mu$. Вероятность первого из этих событий есть на самом деле вероятность того, что экспоненциальная случайная величина с параметром λ

меньше экспоненциальной случайной величины с параметром μ (учли свойство отсутствия последействия). Эта вероятность вычисляется как

$$\int_0^{+\infty} f_\lambda(t)\left(1 - F_\mu(t)\right)dt = \int_0^{+\infty} \lambda e^{-\lambda t} e^{-\mu t} dt = \frac{\lambda}{\lambda + \mu}.$$

Тогда можно утверждать, что следующая заявка поступит раньше, чем будет завершено обслуживание предыдущей, если реализация u случайной величины U, равномерно распределенной на интервале [0; 1], удовлетворяет неравенству $0 < u < \lambda/(\mu + \lambda)$; если же выполняется $\lambda/(\mu + \lambda) < u < 1$, то раньше завершится обслуживание заявки.

Несмотря на то что описанный метод обратной функции обладает достаточной общностью, его применение существенно зависит от вида функции распределения $F(x)$ и зачастую требует определенного опыта в программировании численных методов. Иначе можно получить совсем не такую случайную последовательность как хотелось бы, что, разумеется, исказит результаты моделирования. Проведем классификацию функций $F(x)$ и рассмотрим особенности реализации метода в каждом из случаев.

1.2. Равномерное распределение

Рассмотрим ГСЧ, лежащий в основе метода обратной функции при любой $F(x)$. Нашей задачей является получение последовательности заданной длины, состоящей из случайных чисел, равномерно распределенных на отрезке [0; 1]. Сделать это можно так.

Листинг 1.1. Реализация равномерного распределения

```
#include<cstdlib>
```

```
#include<ctime>
using namespace std;
const int N=1000;    //задаем длину случайной
последовательности
srand((unsigned)time(0));
for (int i=0; i<N; i++)
 r=get_uniform();
 . . . . . .
float get_uniform()
{
 int number=rand();
 float result=(float)number/(RAND_MAX+1);
 return(result);
}
```

Функция `srand()` используется для инициализации ГСЧ, для чего нужно задать в качестве ее аргумента некоторое целое число. Удобно использовать результат функции `time()`, которая возвращает количество секунд, прошедших с начала суток 1 января 1970 г. Этот прием гарантирует, что случайная последовательность при каждом вызове будет разная. Функция `rand()` возвращает целое число, равномерно распределенное от 0 до `RAND_MAX`. Константа `RAND_MAX` определена в заголовочном файле cstdlib, ее типичное значение равно 32767. Приведение к типу `float` необходимо для того, чтобы результат деления не оказался равен нулю. Для получения числа в интервале [0; 1] делить лучше на `RAND_MAX+1`, а не на `RAND_MAX`, так как, если `rand()` возвратит максимальное значение, что не исключено, `get_uniform()` возвратит единицу, а уравнение

$F(u) = 1$ решения не имеет.

Интересно отметить, что большинство генераторов равномерного распределения используют алгоритмы, выдающие строку из цифр, которая на каждом такте цикла подвергается преобразованиям, так что следующая строка кажется случайной. Это относится, например, к методу срединных квадратов или к методу остатков [18], который в [7] назван методом конгруэнций. Алгоритм перехода от одной строке цифр к другой — детерминированный (возведение в квадрат, деление и др.), поэтому и случайные числа на самом деле являются детерминированной последовательностью, хотя и прошедшей, разумеется, тесты на случайность и некоррелированность. Из-за этого часто говорят также о псевдослучайных числах.

1.3. Уравнение $F^{-1}(u) = x$ имеет решение, являющееся элементарной функцией

Рассмотрим случай, когда u явно выражается через x. Такой ГСЧ реализовать наиболее просто — достаточно написать одну функцию с прототипом

```
float get_<имя распределения> (<параметры распределения>)
```

Вызов функции — генератора случайных чисел обычно осуществляется в цикле, потому что одно случайное число само по себе не имеет смысла. Говорить о реализации случайной величины, распределенной по заданному закону, можно только при наличии последовательности, длина которой достаточна для проведения проверочных статистических тестов на соответствие этому закону.

Количество и типы параметров распределения, а также ограничения, накладываемые на них, зависят от вида распределения (см. примеры в табл. 1.1).

Приведем пример ГСЧ для распределения Парето. Параметры распределения вводятся с клавиатуры, результаты записываются в выходной файл.

Листинг 1.2. Реализация распределения Парето

```
#include<stdio.h>
#include<cmath>
#include<cstdlib>
using namespace std;
const int VOLUME=1000; /*количество членов
случайной последовательности*/
/*Прототип ГСЧ. A и B - параметры распределения
Парето, A>0, B>0*/
float get_pareto(float A, float B);
int main()
{
 FILE *out;  /*Указатель выходного файла*/
 float A, B, result;
 out=fopen("out_pareto", "wt");
 /*ввод параметров*/
 printf("\nA="); scanf("%f", &A);
 printf("\nB="); scanf("%f", &B);
 srand((unsigned)time(0)); /*инициализация датчика
равномерного распределения*/
 for (int i=0; i<VOLUME; i++)
```

```
 {
  /*вызов этой функции возвращает случайное число,
полученное с помощью распределения Парето*/
  result=get_pareto(A, B);
  fprintf(out, "%f\n", result);
 }
 fclose(out);
 return 0;
}

float get_pareto(float A, float B)
{
  int r_num; float root, right;
  r_num=rand(); /*получение случайного целого
числа*/
  right=(float)r_num/RAND_MAX+1;   /*проекция на
интервал (0;1)*/
  root=A/(pow(1-right, 1.0/B));  /*вычисление
значения обратной функции*/
  return(root);
}
```

Заметим, что в методе `get_pareto()` мы не проверяем условие $B \neq 0$, так как, согласно табл. 1.1, $B > 0$. Предполагается, что корректность значений параметров распределения проверяется до вызова ГСЧ.

В табл. 1.1 перечислены несколько вероятностных распределений, для которых функция $F^{-1}(u)$ является элементарной.

Таблица 1.1. Распределения вероятностей с элементарной обратной функцией

Назв-ие	***F*(*x*)**	***F*$^{-1}$(*u*)**
Bradford	$\frac{\ln\left(1+\frac{C(x-A)}{B-A}\right)}{\ln(C+1)}, A<x<B, C>0$	$A+\frac{(B-A)\left((C+1)^{u}-1\right)}{C}$
Burr	$\left(1+\left(\frac{x-A}{B}\right)^{-C}\right)^{-D+1}, y>A, B>0,$ $C>0, D\le 100$	$A+B\left(u^{-\frac{1}{D+1}}-1\right)^{-\frac{1}{C}}$
Cauchy	$\frac{1}{2}+\frac{1}{\pi}\frac{1}{\mathrm{tg}\left(\frac{x-A}{B}\right)}, B>0$	$A+B\cdot\mathrm{arctg}\left(\frac{2}{\pi(2u-1)}\right)$
Exponential	$1-e^{\frac{A-x}{B}}, x\ge A, B>0$	$A-B\cdot\ln(1-u)$
ExtremeLB	$\exp\left(-\left(\frac{x-A}{B}\right)^{-C}\right), x>A, B>0,$ $0<C\le 100$	$A+B\cdot(-\ln u)^{-\frac{1}{C}}$
Fisk	$\frac{1}{1+\left(\frac{x-A}{B}\right)^{-C}}, x>A, B>0,$ $0<C\le 100$	$A+B\left(\frac{1}{u}-1\right)^{-\frac{1}{C}}$
Gumbel	$\exp\left(-\exp\left(\frac{A-x}{B}\right)\right), B>0$	$A-B\cdot\ln(-\ln u)$
Laplace	$\begin{cases}\frac{1}{2}\exp\left(\frac{x-A}{B}\right), x\le A;\\ 1-\frac{1}{2}\exp\left(\frac{A-x}{B}\right), x>A,\end{cases} B>0$	$\begin{cases}A+B\cdot\ln(2u), u\le 0{,}5;\\ A-B\cdot\ln\left(2(1-u)\right), u>0{,}5\end{cases}$
Logistic	$\frac{1}{1+\exp\left(\frac{A-x}{B}\right)}, B>0$	$A-B\cdot\ln\left(\frac{1}{u}-1\right)$

Pareto	$1-\left(\frac{A}{x}\right)^{B}$, $0 < A \le x$, $B > 0$	$A(1-u)^{-\frac{1}{B}}$
Reciprocal	$\frac{\ln\left(\frac{A}{x}\right)}{\ln\left(\frac{A}{B}\right)}$, $0 < A \le x \le B$	$B^{u}A^{1-u}$
Weibull	$1-\exp\left(-\left(\frac{x-A}{B}\right)^{C}\right)$, $x > A$, $B > 0$, $C > 0$	$A + B(-\ln(1-u))^{\frac{1}{C}}$

1.4. Функция $F^{-1}(u)$ не является элементарной

Если функция $F^{-1}(u)$ не является элементарной, уравнение $F(x) = u$ для каждого u нужно решать численно. Эта процедура существенно облегчается тем фактом, что $F(x)$ по своему смыслу — монотонно возрастающая, а следовательно, при любом x уравнение имеет единственное решение на [0; 1]. Можно применить любой численный метод решения уравнений, например, метод половинного деления, сходящийся, как известно, со скоростью геометрической прогрессии [8]. Для ГСЧ потребуется написать две дополнительные функции — реализации метода половинного деления и для вычисления $F(x)$. Здесь от пользователя наряду с заданием параметров распределения требуется ввести еще один — точность вычислений. Этот параметр регулирует количество итераций при решении уравнения. В качестве примера приведем ГСЧ для распределения Эрланга

$$F_k(x) = 1 - \sum_{j=0}^{k-1} \frac{(\mu x)^j}{j!} e^{-\mu x},$$

часто используемого при моделировании систем с очередями.

Листинг 1.2. Реализация распределения Эрланга

```
#include<cstdio>
#include<cmath>
#include<cstdlib>
using namespace std;
const int VOLUME=1000;
/*Прототип ГСЧ.
   k - порядок распределения Эрланга,
   mu - параметр,
  eps - требуемая точность
*/
float get_erlang(float mu, int k, float eps);
/*Численное решение уравнения методом половинного
деления - вычисление обратной функции.
 right - правая часть уравнения */
float equ(float mu, float right, int k, float
eps);
/*Вычисление функции распределения - левой части
уравнения - в заданной точке x*/
float function(float mu, int k, float x);
int main()
{
 FILE *out;
 float result, mu, eps; int i, k;
 out=fopen("out_erlang", "wt");
 printf("\nMu="); scanf("%f", &mu);
 printf("\nk="); scanf("%d", &k);
```

```
 printf("\nPrecision="); scanf("%f", &eps);
 srand((unsigned)time(0));
 for (i=0; i<VOLUME; i++)
 {
  result=get_erlang(mu, k, eps);
  fprintf(out, "%f\n", result);
 }
 fclose(out);
 return 0;
}

float get_erlang(float mu, int k, float eps)
{
  int r_num; float root, right;
  r_num=rand();
  right=(float)r_num/RAND_MAX+1;
  root=equ(mu, right, k, eps);
  return(root);
}
/*вычисление функции F(t) - левой части уравнения
- в заданной точке t*/
float function(float mu, int k, float t)
{
 float prod, s; int i;
 prod=1;
 s=1; ___//сумма инициализируется единицей,
//так как нулевая степень и 0! равны 1
```

```
 for(i=1;i<k; i++)
 {
  prod=prod*mu*t/i;
  s+=prod;
 }
 s=s*exp(-mu*t);
 return(1-s);
}

float equ(float mu, float right, int k, float eps)
{
 float edge1, edge2, middle, value;
/*инициализация интервала, на котором ищется
корень.
  Правая граница - среднее плюс десятикратное
среднеквадратичное отклонение. Будем считать, что
вероятностью попадания случайной величины в
область правее этого значения можно пренебречь*/
 edge1=0.0;
edge2=(float)k/mu+10*sqrt((float)k)/mu;
 /*если длина начального интервала все-таки мала -
удваиваем его, пока корень уравнения не окажется
внутри интервала. В правой границе edge2 интервала
приближения корня значение левой части уравнения
должно быть больше значения правой – это означает,
что корень уравнения лежит внутри отрезка
[edge1;edge2]*/
 while(function(mu, k, edge2)<right) edge2*=2;
```

```
 /*итерируем, пока не достигнем заданной
точности*/
 while((edge2-edge1)>eps)
 {
  middle=(edge1+edge2)/2;
  value=function(mu, k, middle);  /*вычисляем
значение левой части в середине текущего интервала
локализации корня*/
  /*корень лежит в правой половине текущего
интервала*/
  if ( (value-right)<0 )  edge1=middle;
  /*корень лежит в левой половине текущего
интервала*/
  else  edge2=middle;
 }
 return((edge1+edge2)/2);
}
```

Другим примером является гиперэкспоненциальное распределение

$$F_k(x) = \sum_{j=1}^{k} p_j \left(1 - \exp\left(-\mu_j x\right)\right); \ \sum_{j=1}^{k} p_j = 1.$$

Листинг 1.3. Реализация гиперэкспоненциального распределения

```
#include<stdio.h>
#include<math.h>
#include<cstdlib>
#define VOLUME 32000
```

```
/*ГСЧ. mu и p - массивы параметров, k – порядок
(размер массивов), eps - заданная точность
вычислений*/
float get_hyper(float *mu, float *p,  int k, float
eps);
float equ(float *mu, float *p, float right, int k,
float eps);
float function(float *mu, float *p, int k, float
x);
main()
{
 FILE *out;
 float s, result, *mu, *p, eps;
 int i, k;
 out=fopen("out_hyper", "wt");
 printf("\nk="); scanf("%d", &k);
 printf("\nPrecision="); scanf("%f", &eps);
 s=0;
 mu=new float[k];
 p=new float[k];
/*ввод параметров распределения pi и μi. Проверка
на равенство суммы вероятностей pi единице */
 for(i=0; i<k; i++)
 {
  printf("\np[%d]=", i+1); scanf("%f", p+i);
  s+=p[i];
  printf("\nmu[%d]=", i+1); scanf("%f", mu+i);
 }
```

```
 /*Проверка. Сумма вероятностей P(i) должна быть
равна единице*/
 if (fabs(s-1)>0.0001) {printf("Сумма P(i) не
равна 1!!!\n"); exit(1); }
 srand((unsigned)time(0));
 for (i=0; i<VOLUME; i++)
 {
  result=get_hyper(mu, p, k, eps);
  fprintf(out, "%f\n", result);
 }
 fclose(out);
 delete []mu;;
 delete []p;;
}
float get_hyper(float *mu, float *p, int k, float
eps)
{
  int r_num;
  float root, right;
  r_num=rand();
  right=(float)r_num/RAND_MAX+1;
  root=equ(mu, p, right, k, eps); /*см. листинг
1.2*/
  return(root);
}

float function(float *mu, float *p, int k, float
t)
```

```
{
 float s; int i;
  s=0;
 for(i=0;i<k; i++)
  s+=p[i]*(1-exp(-mu[i]*t));
 return(s);
}
```

Существуют и многие другие распределения, подпадающие под данную категорию, например, косинусное $F(x) = \frac{1}{2\pi}\left(\pi + \frac{x-A}{B} + \sin\left(\frac{x-A}{B}\right)\right)$, $A - \pi B \le x \le A + \pi B$, $B > 0$.

1.5. Функция $F(x)$ не является элементарной

Если функция $F(x)$ не является элементарной, для реализации ГСЧ необходимо запрограммировать еще один численный метод — вычисления самой функции $F(x)$. Это может быть метод численного интегрирования, решения дифференциального уравнения или какой-либо другой, в зависимости от функции $F(x)$. При этом нужно принять решение о выборе метода и шага сетки.

Классический пример распределения такого типа — нормальное,

$$F(x) = \frac{1}{\sqrt{2\pi}B}\int_{-\infty}^{x} \exp\left(-\frac{1}{2}\left(\frac{t-A}{B}\right)^2\right)dt\,.$$

Далее приведен пример ГСЧ для этого распределения.

Листинг 1.4. Реализация нормального распределения

```
#include<stdio.h>
```

```
#include<math.h>
#include<cstdlib>
#define VOLUME 32000
/*ГСЧ нормального распределения. mean - математическое ожидание, disp - дисперсия, eps - заданная точность*/
float get_normal(float mean, float disp, float eps);
/*вычисление интеграла от A до B методом Симпсона*/
float simpson(float A, float B, float mean, float disp);
/*вычисление обратной функции.
bottom_bound и top_bound - аппроксимация бесконечных пределов интегрирования;
almost_all - значение интеграла, взятого в этих конечных пределах;
mean и disp - параметры нормального распределения;
eps – заданная точность;
right – правая часть уравнения*/
float equ(float bottom_bound, float top_bound, float mean, float disp, float almost_all, float eps, float right);
/*вычисление подынтегральной функции в заданной точке x*/
float function(float mean, float disp, float x);
int main()
{
```

```
 FILE *out;
 float result, hint, mean, disp, eps; int i;
 out=fopen("out_normal", "wt");
 printf("\nMean="); scanf("%f", &mean);
 printf("\nDisp="); scanf("%f", &disp);
 /*По смыслу нормального распределения точность не
должна превышать значения hint*/
 hint=1.0/(disp*sqrt(2*M_PI));
 printf("\nPrecision(<%f)=", hint); scanf("%f",
&eps);
 if (eps>hint) { printf("Illegal presicion\n");
exit(1); }
 srand((unsigned)time(0));
 for (i=0; i<VOLUME; i++)
 {
  result=get_normal(mean, disp, eps);
  fprintf(out, "%f\n", result);
 }
 fclose(out);
 return 0;
}

float get_normal(float mean, float disp, float
eps)
{
  int r_num;
  float root, bottom_bound, top_bound, almost_all,
right;
```

```
  /*вычисление конечных аппроксимаций пределов
интегрирования в соответствии с заданной
точностью*/
  bottom_bound=mean-disp*sqrt(-
log(2*M_PI*eps*eps*disp*disp));
  top_bound=mean+disp*sqrt(-
log(2*M_PI*eps*eps*disp*disp));
  /*вычисление интеграла в этих пределах*/
  almost_all=simpson(bottom_bound, top_bound,
mean, disp);
  r_num=rand();
  right=(float)r_num/RAND_MAX+1;
  root=equ(bottom_bound, top_bound, mean, disp,
almost_all, eps, right);
  return(root);
}

float simpson(float A, float B, float mean, float
disp)
{
 float k1, k2, k3, s, x, h1, h;
 /*шаг интегрирования принимается равным 0,01. В
«товарных» реализациях метода применяется
процедура автоматического выбора шага с помощью
апостериорных оценок*/
 h=0.01;
 s=0;  h1=h/1.5;
 k1=function(mean, disp, A);
```

```
 for(x=A; (x<B)&&((x+h-B)<=h1); x=x+h)
 {
  k2=function(mean, disp, x+h/2);
  k3=function(mean, disp, x+h);
  s=s+k1+4*k2+k3;
  k1=k3;
 }
 s=s*h/6;
 return(s);
}

float function(float mean, float disp, float x)
{
 float result;
 result=(1.0/(disp*sqrt(2*M_PI)))*exp(-0.5*((x-
mean)/disp)*((x-mean)/disp));
 return(result);
}

float equ(float bottom_bound, float top_bound,
float mean, float disp, float almost_all, float
eps, float right)
{
 float edge1, edge2, middle, cover, value;
 edge1=bottom_bound; edge2=top_bound;
 if (right>almost_all) return(top_bound); else;
```

```
 if (right<(1-almost_all)) return(bottom_bound);
else;
 cover=0;   /*введена для повышения
производительности. В новой точке вычисление
интеграла производится не от bottom_bound, а от
edge1, в то время как значение интеграла от
bottom_bound до edge1 уже накоплено в cover*/
 while((edge2-edge1)>eps)
 {
  middle=(edge1+edge2)/2; value=simpson(edge1,
middle, mean, disp);
  if ( (cover+value-right)<0 ) { edge1=middle;
cover=cover+value; }
  else edge2=middle;
 }
}
```

Интеграл вычисляется методом Симпсона, шаг интегрирования взят 0,01. Переменная h_1 введена для корректного завершения цикла, так как при увеличении переменной x с шагом h равенство $x = B - h$ практически никогда не выполнится. Поэтому необходимо поставить дополнительное условие, чтобы значение $x + h$ не «заскочило» за верхний предел интегрирования. Коэффициент 1,5 был подобран экспериментальным путем.

Возникает еще одна вычислительная проблема — какими конечными пределами аппроксимировать несобственный интеграл. В приведенной программе вопрос решен следующим образом. Выбирается малый параметр $\varepsilon > 0$, задающий точность, и уравнение $G(u) = \varepsilon$, где $G(u)$ — подынтегральная функция, решается

относительно u. Конкретно для нормального распределения получим $u_{1,2} = A \pm B\sqrt{-\ln\left(2\pi\varepsilon^2 B^2\right)}$. Этими значениями аппроксимируются пределы интегрирования. Далее вычисляем определенный интеграл $L = \int_{u_1}^{u_2} G(u)du$.

При решении уравнения $F(u) = x$, если $x > L$, возвращаем u_2; если $x < 1 - L$, возвращаем u_1. Иначе численно решаем уравнение $F(u) = x$.

Помимо теоретических проблем возникают также и проблемы реализации. Например, на каждом шаге цикла нужно перевычислять x, а u_1, u_2 и L остаются постоянными. Поэтому их вычисление в теле функции `get_normal` никак нельзя считать оправданным. Здесь возможны два решения:

1) Вычислить u_1, u_2 и L вне тела цикла в вызывающей функции и передавать их в `get_normal()` в качестве параметров. Однако тогда прототип функции `get_normal()` потеряет наглядность, и ею сможет пользоваться только ее автор. Ведь от пользователя мы можем требовать лишь задание точности, среднего и дисперсии, а все остальное — подробности реализации, отягощать которыми других программист не имеет права;
2) Сделать переменные u_1, u_2 и L внешними. В принципе, это тоже не решает проблему, а лишь маскирует ее — значения этих переменных все равно нужно вычислять вне функции `get_normal()`, кроме того, взятая сама по себе она не будет компилироваться без подключения специального `header`-файла.

Приведенный в листинге вариант скрывает все подробности реализации ГСЧ, но при серийном использовании неэффективен.

Решением проблемы является использование локальных статических переменных внутри функции `get_normal()`. По умолчанию эти переменные инициализируются нулями. Инициализация статической переменной производится только однажды — при загрузке программы, и ее значение после выхода из функции не «забывается». В зависимости от параметров нормального распределения значения u_1 и u_2 могут оказаться равны нулю не только после инициализации — это могут быть их реальные значения. Того же нельзя сказать об L, так как это значение определенного интеграла от неотрицательной функции. Поэтому проверять, вычислены или нет значения статических переменных, надежнее именно по переменной `almost_all`, которая соответствует L. Модифицируем фрагмент функции `get_normal()`:

```
/*u1,u2 и L соответствуют статическим переменным*/
static float bottom_bound, top_bound, almost_all;
if (almost_all==0.0)
{
 bottom_bound=mean-disp*sqrt(-
log(2*M_PI*eps*eps*disp*disp));
 top_bound=mean+disp*sqrt(-
log(2*M_PI*eps*eps*disp*disp));
 almost_all=simpson(bottom_bound, top_bound,
mean, disp);
}
```

В [28] приведен другой метод ГСЧ для нормального распределения, основанный на центральной предельной теореме. Существует и другие методы генерации нормального распределения.

Кроме нормального распределения, существует множество других, имеющих неэлементарную функцию $F(x)$ — логнормальное, гамма-, бета-распределения и различные их модификации.

1.6. Распределения фазового типа. Композиция ГСЧ

Существуют такие распределения, что случайную величину, которую они описывают, можно разложить на составляющие, каждая из которых задается более простым распределением. К таким относятся распределения Эрланга и гиперэкспоненциальное.

Рис. 1.1. Схема распределения Эрланга

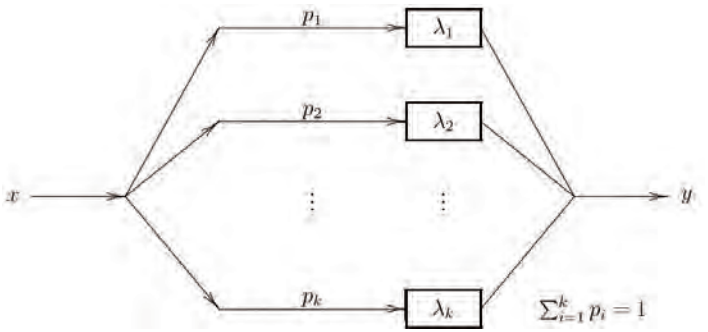

Рис. 1.2. Схема гиперэкспоненциального распределения

Случайную величину, распределенную по закону Эрланга k-го порядка, можно представить в виде суммы k экспоненциально распределенных случайных величин. Если этой случайной величиной является длительность обслуживания, то такое обслуживание эквивалентно k последовательным экспоненциальным этапам

(рис. 1.1). Для гиперэкспоненциального распределения H_k одна из k экспоненциальных случайных величин выбирается в соответствии с некоторым дискретным распределением p_i (параллельные этапы, рис. 1.2). Такие схемы полностью определяют алгоритм ГСЧ, даже если функция распределения $F(x)$ не задана явно. Приведем альтернативный ГСЧ для H_k. Он построен на основе двух ГСЧ — для экспоненциального распределения и дискретного, заданного набором вероятностей. Численное решение уравнения здесь не требуется.

Листинг 1.5. Реализация гиперэкспоненциального распределения как распределения фазового типа

```
/*функция main() остается без изменений, см.
листинг 1.3*/
float get_hyper(float *mu, float *p,  int k)
{
  int r_num, j; float root, right;
  r_num=rand();
  right=(float)r_num/RAND_MAX+1;
  /*разыгрываем параметр экспоненциального
распределения (номер фазы)*/
  j=get_discrete(k, p);
  /*вычисляем обратную функцию для
экспоненциального распределения с выбранным
параметром*/
  root=-log(1-right)/mu[j];
  return(root);
}
```

```
/*ГСЧ для дискретной случайной величины 0,1, ...,
k-1,
заданной распределением P(i)*/
int get_discrete(int k, float *p)
{
 int i, test; float a, s;
 test=rand();
 a=(float)test/RAND_MAX+1;
 s=0;
 for(i=0; i<k; i++)
 {
  if ( (a>=s) && (a<s+p[i]) ) return(i);
  else s=s+p[i];
 }
}
```

Существуют и более сложные фазовые распределения (рис. 1.3).

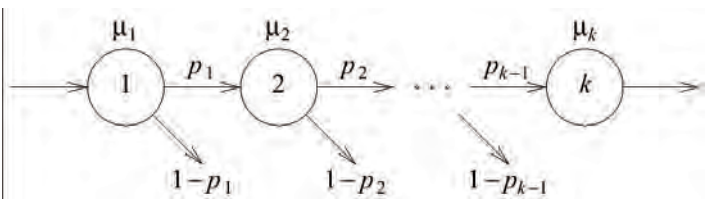

Рис. 1.3. Распределение фазового типа

μ_i — это параметры экспоненциального распределения длительности пребывания в состоянии i, p_i — маршрутные вероятности переходов между состояниями. Схемы такого типа часто используются для аппроксимации распределений «с тяжелым хвостом» [62]. Глядя на эту схему, не так-то просто записать $F(x)$, тем не менее, ГСЧ пишется

без проблем.

Листинг 1.6. Реализация распределения фазового типа

```
float get_phase(float *mu, float *p,  int k)
{
 int r_num, j, i;
 float right, s;
 s=0;
 for(i=0; i<k; i++)
 {
  r_num=rand();
  right=((float)r_num)/RAND_MAX+1;
  /*наращивание значения случайной величины в
соответствии с очередной пройденной фазой*/
  s=s-log(1-right)/mu[i];
  if (i==k-1) return(s); /*все фазы пройдены*/
  /*разыгрываем, перейти к следующей фазе или
завершить процесс*/
  j=get_side(p[i]);
  if (j==1) return(s); else;
 }
}

/*розыгрыш двухвариантного (0 или 1) события,
заданного вероятностью f.*/
int get_side(float f)
{
 int i, test; float a;
```

```
 test=rand();
 a=(float)test/RAND_MAX+1;
 if (a<=f) return(0); else return(1);
}
```

Заметим, что схему, изображенную на рис. 1.3, можно перерисовать в другом виде (рис. 1.4), где вероятности p_i и r_i связаны соотношением $r_i = (1-p_i)\prod_{j=1}^{i-1} p_j$. Вероятность r_i — это вероятность того, что состоятся $i-1$ этапов обслуживания, после чего (с вероятностью $1-p_i$) процесс обслуживания завершится. Иными словами, это распределение можно представить совокупностью распределений Эрланга, каждое своего порядка, одно из которых выбирается в соответствии с вероятностью r_i. Тогда $F(x)$ удается записать как

$$F(x) = \sum_{i=1}^{k} r_i \left(1 - \sum_{j=0}^{i-1} \frac{(\mu x)^j}{j!} \exp(-\mu x)\right) = 1 - \exp(-\mu x) \sum_{i=1}^{k} r_i \sum_{j=0}^{i-1} \frac{(\mu x)^j}{j!}$$

и отнести, таким образом, распределение к группе элементарных $F(x)$. Если же на каждом из этих последовательно-параллельных этапов параметр μ задать разным, фазовый метод будет, пожалуй, наилучшим вариантом, ибо функция распределения станет очень сложной.

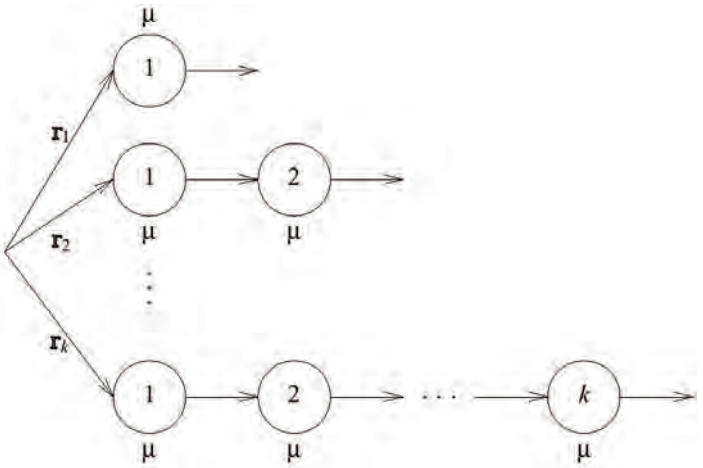

Рис. 1.4. Альтернативная схема распределения фазового типа

Таким образом, генератор равномерного распределения в совокупности с методом обратной функции позволяет создать ГСЧ для произвольного распределения вероятностей. Функции-датчики равномерного распределения имеются во всех языках программирования и специализированных моделирующих средах, в том числе, разумеется, в C и C++. Более того, в некоторых реализациях C++ имеются готовые классы для генерации наиболее популярных распределений. Эти функции и классы написаны квалифицированными специалистами и всесторонне протестированы.

Глава 2. Общая схема построения объектных моделирующих программ

2.1. Объектный анализ

Успех программного продукта в целом и быстрота его отладки в частности во многом зависят от того, насколько удачно программа спроектирована, насколько много мыслительных усилий потрачено при работе «за столом» — без компьютера. Особенно значимо это для объектно-ориентированных приложений. Если объектная модель построена неудачно, программа либо вообще не будет работать, либо ее результаты можно будет смело отправлять в мусорную корзину сразу же по получении. Поэтому при написании программ имитационного моделирования очень важно построить такую объектную модель, которая соответствовала бы основным реалиям моделируемой системы и процессам, в ней происходящим. Примеры подобного рода анализа в достаточном количестве представлены во второй части книги. Здесь же сформулируем некоторые рекомендации общего характера.

- Выделить в системе объекты, наличие которых, а также роль, которую они играют, являются принципиальными для системы и не могут быть проигнорированы.
- Попытаться сгруппировать объекты по каким-то общим признакам и сделать вывод — какие совокупности объектов могут быть описаны в рамках одного класса или классов, связанных отношением наследования.
- Решить, всякий ли объект системы несет в себе настолько содержательное информационное наполнение и

функциональность, что «достоин» собственного класса, либо он может быть представлен как поле некоторого более «важного» класса. Как правило, постоянно присутствующие в системе (еще говорят — *персистентные*) объекты (например, серверы) следует описывать отдельным классом, а временные (например, клиентские заявки) — нет.

- Иногда в виде класса целесообразно описывать объекты, не имеющие явного материального эквивалента, например, «Движение», «Производство», «Группировка». Обычно объекты «материальных» классов выступают в них в качестве полей данных.
- Решить, нужны ли отношения дружественности между какими-то парами классов.
- Для объектов каждого класса сформировать набор данных, который полностью идентифицирует состояние объекта в текущий момент и достаточен для принятия решения о дальнейшем поведении объекта в любой ситуации. Каждый элемент этого набора должен быть описан как поле данных (переменная состояния) класса с соответствующим типом. В качестве полей данных могут выступать как скалярные величины, так и массивы.
- Решить, какие поля данных следует объявить закрытыми (private), какие — защищенными (protected), а какие — открытыми (public).
- Если полем данных является объект другого класса — решить, как он соотносится с данным классом. Если имеет место отношение *композиции*, то есть один объект вложен в другой,

является его составной частью и вне «хозяина» существовать не может (например, хвост у собаки), то такой объект следует хранить по значению. Если же включаемый в число полей данных объект не принадлежит целиком объемлющему объекту, может существовать независимо от него и использоваться другими объектами, его лучше описать как указатель. На языке Cache Object Script [36] в таких случаях можно использовать конструкцию `Relationship` — задать связь между объектами разных классов (но нельзя задать связь «многие ко многим»), в C++ — действовать по обстоятельствам.

- Оценить, не является ли набор полей данных избыточным, то есть нет ли полей, значения которых можно вычислить по остальным. В реляционной алгебре такая ситуация называется нарушением третьей нормальной формы. Ее наличие может нарушить целостность данных, так как поля данных объекта нельзя будет изменять независимо друг от друга. Вычисление избыточных полей данных следует оформить в виде методов класса. Отметим, что в редких случаях это правило можно нарушать, но тогда вся ответственность за контроль над непротиворечивостью данных ложится на программиста. Внутри метода, который меняет какую-то из связанных зависимостью переменных, значения зависимых переменных нужно перевычислить. Типичный пример — комплексные числа. Чтобы повысить эффективность работы с ними, в полях данных комплексного числа хранятся четыре значения — действительная часть, мнимая часть, модуль и аргумент. Одна из этих пар значений однозначно определяет другую.
- Решить, с какими значениями полей данных объект начнет

участвовать в процессе моделирования. Эти значения использовать при написании конструктора по умолчанию.

- Рассмотреть каждое поле данных класса и ответить на вопрос, в результате наступления каких событий в системе значение этого поля может измениться. Каждому такому событию следует сопоставить некоторый метод класса, который будет его обрабатывать. Это может быть метод как данного класса, так и другого класса, если изменение поля данных есть косвенный результат события, очевидным образом соотносимого с объектом другого класса. Отношение «событие — метод-обработчик» можно считать взаимно-однозначным, а отношение «изменение переменной — метод-обработчик» довольно сложное и имеет в общем случае вид «многие ко многим»: значение переменной может измениться в результате наступления нескольких событий, а наступление события может повлечь за собой изменение значений нескольких полей данных, и даже у разных объектов. Кроме того, изменение переменной — это не всегда событие, то есть метода-обработчика может и не быть.
- При написании кода методов-обработчиков следует тщательно сверяться с описанием моделируемой системы, алгоритмами ее функционирования и принятия решений, чтобы корректно отразить реакцию объекта (объектов) на событие.

2.2. Модельное время

При проведении имитационного эксперимента программа должна постоянно генерировать всевозможные события, происходящие в системе. Простейшие из них — это прибытие новой заявки и

завершение обслуживания заявки на сервере. Возникает вопрос — в какой момент генерировать то или иное событие? Для этого каждый поток событий и каждая случайная величина, описывающая работу системы (например, длина заявки), должны быть заданы своей функцией распределения вероятностей — эта функция задается обычно при описании системы на этапе постановки задачи. Рассмотрим, что нужно сделать на уровне реализации.

Предположим, что некоторый объект *A* принимает и обрабатывает случайный поток событий. В протокол класса для объекта *A* вводим целочисленную переменную состояния *t*, имеющую следующий смысл: время, которое осталось до наступления ближайшего события из потока. Если объект принимает несколько потоков, то для каждого потока создаем свою переменную. При этом очень важно, чтобы все величины системы, так или иначе связанные со временем, были приведены к одной единице измерения — той, которая выбрана в качестве одного такта модельного времени (выбор такта модельного времени подробно рассмотрен в разделе 7.2 главы 7 и последующих главах). На каждом такте от значения переменной *t* отнимается единица. Если *t* стало равным нулю, вызывается обработчик события, после чего значение *t* разыгрывается заново, в соответствии с законом распределения. Генератор случайных чисел может вызываться как из самого метода-обработчика, так и из метода `A.run()` — на усмотрение программиста в зависимости от конкретной ситуации.

В качестве альтернативного проектного решения, чтобы не усложнять описание реального объекта искусственными переменными, можно создать класс `Timer`, экземпляры которого возьмут работу со временем на себя, каким-то образом взаимодействуя с объектами

системы. Очевидно, что эту, как и любую другую задачу, можно решать по-разному, и осознанный выбор всегда остается за разработчиком.

2.3. Основной цикл

Как уже отмечалось в предыдущей главе, кроме методов-обработчиков, которые, как правило, следует делать закрытыми, у класса должен существовать открытый метод-диспетчер, который мы по аналогии с классом `Thread` языка Java назовем `run()`. Этот метод производит декремент переменных времени объекта, опрашивает значения других переменных состояния и принимает решение о том, следует ли в данный момент вызывать какой-то закрытый метод-обработчик или же не вызывать никакого. Метод `run()` играет роль связующего звена между классом и его пользователем, в то время как методы-обработчики относятся к внутренним особенностям реализации логики (сейчас модно говорить о *бизнес-логике*) класса. В зависимости от этой логики `run()` может иметь параметры, а может и не иметь. При использовании такого подхода структуру функции `main()` можно унифицировать для довольно широкой совокупности моделирующих программ.

Листинг 2.1. Общая схема функции main() моделирующих программ

```
//подключение header-файлов, содержащих протоколы классов
#include<ctime>
#include<cstdlib>
#include "Class1.h"
```

```
#include "Class2.h"
. . . . . .
//задание общего времени моделирования
#define TOTAL_TIME 10000
int main()
{
/*объявление объектов. Это может происходить по-
разному: вызов конструктора без параметров,
конструктора с параметрами, объявление массива
объектов.*/
Class1 Object1;
Class2 Object2[5];
Class3 Object3(параметры конструктора);
. . . . . ..
//инициализация ГСЧ
srand((unsigned)time(0));
//основной цикл – потактовый опрос всех объектов
//системы
for(int i=0;i<TOTAL_TIME;i++)
{
 Object1.run();
for(int j=0; j<5; j++)
  Object2[j].run();
 Object3.run();
  . . . . . .
 } //завершение тела основного цикла
```

```
 return 0;
} //завершение программы
```

При желании можно, видимо, добиться того, чтобы метод `run()` в любом случае не принимал параметров и для всех классов имел бы одинаковый прототип. Решение в конкретной ситуации всегда остается на усмотрение разработчика.

В теории имитационного моделирования такая схема называется time-driven (управляемая временем), в отличие от альтернативной схемы event-driven (управляемая событиями), рассматриваемой в следующем разделе.

2.4. Параллельное моделирование и порядок обхода объектов

Параллельное дискретно-событийное моделирование относится к тем областям компьютерных наук, которые в данный момент еще нельзя назвать вполне сформировавшимися и в которых вопросов больше, чем ответов. Работы в области параллельного моделирования ведутся в основном в стенах академических лабораторий, и ни одна из имеющихся систем пока не получила повсеместного признания. На протяжении 90-х гг. прогнозы ведущих ученых и даже названия их работ были далеки от оптимизма. В обзорной работе [96] сказано: «Для разработки и анализа эффективных параллельных моделирующих программ требуются значительные усилия. Не существует “серебряной пули”, позволяющей программисту гарантированно написать хорошую программу. Вообще говоря, параллельное моделирование — это очень трудная задача». А названия работ «Выживет ли параллельное дискретно-событийное

моделирование?» [67] и «Параллельное дискретно-событийное моделирование: факт или фикция?» [48] говорят сами за себя. Чем же вызван такой скептицизм авторитетных ученых? Его причина — те проблемы, которые в большом количестве возникают перед исследователем, когда речь заходит о параллельной реализации проверенных и надежных методов. Хорошо известно, что при некорректном распараллеливании, например, методов вычислительной математики может оказаться так, что решается совсем другая задача, отличная от той, которую решала бы последовательная программа. Для имитационного моделирования подобная опасность имеет еще более ярко выраженную форму, что, разумеется, не добавляет надежности результатам работы параллельных программ. Коротко опишем возникающую проблему, при этом имея в виду, что речь идет о распараллеливании календарно-событийного (event-driven) метода моделирования.

В этом методе время изменяется не непрерывно, а скачками. Вместо того чтобы для каждого объекта вести счетчик времени до наступления ближайшего события, используется глобальный календарь событий. Программа разбивается на некоторое множество параллельных логических процессов. Каждый процесс планирует изменения состояния некоторого объекта системы и заносит в централизованно ведущийся календарь описание события, а также планируемое время его наступления. Каждый раз моделируется событие с наименьшим временем наступления, которое затем удаляется из календаря.

Здесь и возникает основная проблема, получившая название causality error (нарушение причинно-следственной зависимости) и описанная

во всех фундаментальных работах (например, [81], [66], [86]). Если моделировать ближайшие события параллельно для всех объектов, возникает вопрос — можно ли одновременно моделировать события, происходящие в разные моменты модельного времени? Пусть для объекта A ближайшее событие X_1 произойдет в момент времени 10, а ближайшее для объекта B событие X_2 — в момент времени 20. Если по результатам события X_1 оказалось необходимым смоделировать событие X_3 для B, время наступления которого меньше 20 (например, 15), и обработчику события X_3 нужно знать состояние объекта B именно в момент времени 15 — происходит causality error. Ведь мы моделируем событие, происходящее в момент времени 20, до того, как смоделировали событие, происходящее в момент времени 15! Поэтому, прежде чем решить, можно ли одновременно моделировать X_1 и X_2, нужно знать — будет ли порождено событие X_3, но чтобы это узнать, нужно сначала выполнить X_1.

В [96] приведен конкретный пример. Друг напротив друга стоят два танка противоборствующих сторон. Сначала стреляет первый танк, через несколько секунд — второй, для каждого танка известна вероятность поражения цели. В этой ситуации параллельно моделировать выстрелы нельзя, так как, если первый танк попал, разыгрывать попадание или промах второго танка уже нет смысла.

На преодоление таких ситуаций при параллельном моделировании и затрачиваются усилия ученых. К настоящему времени общепринятым стало разделение подходов к параллельному моделированию на консервативный (conservative) и оптимистичный (optimistic). При консервативном подходе нарушения причинно-следственной связи не допускаются в принципе путем постоянного динамического

прогнозирования (lookahead) будущих событий. При оптимистичном подходе время на то чтобы «заглянуть в будущее» не тратится, поэтому причинно-следственная ошибка возможна. При ее обнаружении задействуется механизм отката (rollback), который отменяет все изменения переменных, произошедшие после наступления ошибки, и восстанавливает состояние системы. Подробно протоколы обеих стратегий, а также экспериментальные параллельные архитектуры, их реализующие, рассмотрены в уже цитировавшемся обзоре [96]. Там же приведена сравнительная таблица характеристик обеих стратегий, которую мы воспроизведем (табл. 2.1).

Таблица 2.1. Сравнительная характеристика стратегий параллельного моделирования

Стратегия	Консервативная	Оптимистичная
Принцип	Недопущение нарушений причинно-следственной зависимости	Нарушения допускаются; при их возникновении реализуется механизм отката
Параллелизм	Выражен в меньшей степени	Выражен в максимальной степени
Синхронизация	Блокирование «опасных» процессов; возможны тупики, если не принимать предупредительных мер	Механизм отката восстанавливает предыдущее сохраненное состояние

Прогнозирование	Является «узким местом», сильно влияет на производительность. Реализуется динамически	Не влияет на производительность
Конфигурация	Требует статической конфигурации логических процессов	Может меняться динамически
Память	Требует меньше памяти, чем оптимистичная стратегия	Требует больше памяти для сохранения состояний; более сложное управление распределением памяти
Реализация	Простота реализации и структур данных	Более сложная реализация и операции над данными

Обзор имеющихся языков моделирования и библиотек, реализующих одну или обе описанные стратегии, содержится в работе [92]. В ней описаны и подвергнуты сравнению языки APOSTLE, Parsec, ModSim, YADDES, а также библиотеки CPSim, GTW, ParaSol, PSK, Simkit, SPaDES, SPEEDES, TWOS и WARPED. По результатам сравнения сделан вывод о большей предпочтительности оптимистичной стратегии.

Отметим, что при распараллеливании схемы моделирования, описанной в 2.3, causality error не возникает, так как параллелизм обработки объектов заключается в одновременном выполнении на

каждом такте основного цикла метода `run()` для всех объектов, что соответствует одному и тому же моменту модельного времени. Но это вовсе не значит, что в данном случае с параллелизмом вообще нет проблем. Здесь возникают другие проблемы, связанные с выбором порядка обхода объектов.

Как показывает опыт, при обычной последовательной реализации порядок обхода объектов внутри основного моделирующего цикла может быть произвольным и не оказывает определяющего влияния на результаты моделирования. Если по логике работы системы с объектом *A* происходит нечто, вызывающее изменения в объекте *B*, совсем необязательно, чтобы опрос *A* стоял в теле цикла раньше:

```
for(int i=0;i<MAX;i++)
{
  . . . . .
  A.run();
  B.run();
  . . . . .
}
```

Если раньше опрашивается объект *B*, искомое событие для него будет сгенерировано на следующем такте цикла. Теоретически, можно предположить следующую ситуацию: в системе действует некоторая политика, и объект *B* обязательно должен узнать о произошедшем с объектом *A* событии первым, раньше других объектов. Тогда, конечно, опросы *A* и *B* должны следовать жестко друг за другом. Но такая ситуация свойственна скорее не техническим, а человеческим системам. А их моделирование — это уже совсем другая наука.

Ситуация резко меняется, если рассмотреть кажущуюся вполне

естественной возможность параллельной реализации вызовов методов `run()` в такте основного цикла. Эту возможность легко осуществить, например, средствами Java, используя концепцию параллельных потоков. Если пустить этот параллелизм «на самотек», результаты моделирования могут быть серьезно искажены. Представим себе, что объект *A* — некая промышленная установка с флюктуирующей температурой, иногда «зашкаливающей» за предельно допустимое максимальное значение. Когда это происходит, срабатывает система аварийной защиты (объект *B*). Программно объект *B* меняет состояние путем опроса объекта *A*, который происходит в теле метода `B.run()`.

При параллельной реализации потоков порядок их взаимного выполнения друг относительно друга зависит от порядка выделения потокам программы квантов времени центрального процессора, который определяется алгоритмами диспетчеризации операционной системы. Если результат выполнения программы зависит от этого порядка, явление называется *гонкой* (race condition) и считается недопустимым. Предположим, на *i*-м такте цикла метод `B.run()` опросил состояние объекта *A* до того, как метод `A.run()` его изменил (повысил температуру так, что она стала выше максимальной). А на следующем, (*i*+1)-м такте случилось обратное — метод `A.run()` успел понизить температуру объекта *A* до того, как ее считал метод `B.run()`. При фиксированном порядке обхода объектов такого, конечно, произойти не может, а при параллельном — вполне реально. Тогда имитационный эксперимент не покажет срабатывания аварийной защиты, хотя в реальной системе оно бы произошло, так как в ней процессы в *A* и *B* являются непрерывными, а не дискретными. Возможна и обратная ситуация. На *i*-м такте вызовы

пройдут в порядке (A, B), на (i+1)-м — в порядке (B, A). Тогда в имитационном эксперименте объект B проведет в активизированном состоянии больше времени, чем провел бы на самом деле. Даже если эти случайности в итоге статистически компенсируют друг друга и общее время работы системы аварийной защиты окажется достоверным, все равно будут искажены другие важные характеристики объекта B — например, период занятости или функция распределения для потока включений.

Таким образом, «лобовая» реализация естественного параллелизма программы может привести к ее некорректному выполнению, так как нечетко определенные или плохо поддерживаемые информационные зависимости между объектами могут стать причиной непредвиденных осложнений — сами того не подозревая, мы можем решать совсем другую задачу.

Существует множество способов борьбы с «гонками», позволяющих синхронизировать и согласовывать зависимые потоки. К ним относятся, например, такие известные средства системного программирования, как сигналы и семафоры. В языке Java для этой цели предусмотрены мониторы и пара важных методов, `wait()/notify()`, применение которых требует от разработчика определенного искусства и понимания существа дела, так как некорректное использование мониторов может привести к состоянию тупика (deadly embrace).

Если говорить об объектно-ориентированных возможностях, то здесь на помощь может прийти один из паттернов проектирования, называемый `Observer` (Наблюдатель) [26]. Применительно к рассмотренному примеру объект A выступит в качестве субъекта, а

объект *B* — в качестве наблюдателя. Субъект, располагая информацией о своем наблюдателе, немедленно уведомляет его об изменении, которое может привести к рассогласованности состояний, то есть о превышении максимально допустимой температуры. Программно это реализуется в виде вызова метода `Notify()`, входящего в стандартный интерфейс класса `Subject`, внутри которого вызывается метод `Update()` для связанного с субъектом наблюдателя, входящий в стандартный интерфейс класса `Observer`. Если в методе `Update()` реализована логика реакции наблюдателя на изменение состояния субъекта, эта реакция гарантированно появится в том же такте моделирования, и произошедшее с субъектом событие не будет упущено моделирующей программой.

2.5. Сбор статистики

Имитационный эксперимент ставится с целью получения результата, в качестве которого выступают различные характеристики изучаемой системы. Поэтому изящно запрограммировать с помощью объектов то, как система «живет и дышит», хоть и крайне важно, но недостаточно — само по себе это ничего не дает. Необходимо еще каким-то образом снимать по ходу моделирования нужные показания и сохранять их для последующей обработки. Трудно дать общие рекомендации о методе вычисления, периодичности сохранения и методе статистической обработки результатов, поскольку это существенно зависит от того, что именно мы хотим исследовать. Текущие показания изучаемой величины, например, длины очереди, можно записывать либо в динамический массив, либо, если их слишком много, — в файл. После завершения эксперимента этот массив или файл можно будет подвергнуть статистическому анализу и

на основании этого получить с той или иной достоверностью моменты случайной величины и выдвинуть гипотезу о законе ее распределения, а также представить результаты в графическом виде. Если же нас интересует только среднее значение, то можно обойтись и без массивов, регулярно пересчитывая среднее на каждом такте основного цикла. В самом деле, пусть $N_{\text{ср}}^{(i)} = \frac{\sum_{j=1}^{i} a_j}{i}$ — среднее значение величины a за i тактов. Тогда

$$N_{\text{ср}}^{(i+1)} = \frac{\sum_{j=1}^{i} a_j + a_{i+1}}{i+1} = \frac{N_{\text{ср}}^{(i)} i + a_{i+1}}{i+1} = N_{\text{ср}}^{(i)} \left(1 - \frac{1}{i+1}\right) + \frac{a_{i+1}}{i+1}.$$

Если a является некоторой нетривиальной характеристикой, не входящей в переменные состояния классов, то для вычисления a_i может потребоваться написание отдельного закрытого метода класса.

Подробнее вопрос сбора статистики при проведении имитационного эксперимента рассматривается в [7].

2.6. Пример моделирования на C++ с помощью событийной схемы

Для сравнения удобства и ясности двух подходов приведем пример довольно простой моделирующей программы на C++, разработанной на основе календарно-событийной (event-driven) схемы. Эта программа детально рассмотрена в работе [63], где описывается свободно распространяемая библиотека SimPack — набор классов и служебных функций, которые могут быть использованы внутри C++-программы для реализации алгоритмов моделирования систем.

В крайне упрощенном виде моделируется циклическое выполнение

заданий, попеременно занимающих единственный процессор и один из четырех дисков. Работы разделены на два класса — 0 и 1. Работы класса 1 имеют приоритет, выражающийся в возможности вытеснения с процессора заявок класса 0. Моделируется довольно большое количество циклов работ, где под циклом понимается выполнение работы сначала на процессоре, затем на одном из дисков. Используемые распределения длительностей обслуживания — экспоненциальное и Эрланга.

Листинг 2.2. Пример реализации календарно-событийной схемы моделирования

```
#include ".../../queueing/queueing.h"   //header-
файл с описанием

//библиотечных классов и функций
#define n0 6             //число работ класса 0
#define n1 3            //число работ класса 1
#define nt n0+n1       //общее число работ
#define nd 4           //число дисков

enum {BEGIN_TOUR=1, REQUEST_CPU, RELEASE_CPU,
REQUEST_DISK, RELEASE_DISK}; //названия событий
struct token_info
//структура для описания работы
{
 int cls;             //класс (приоритет) работы
 int un; //номер диска, к которому произошло
  //текущее обращение
```

```
 double ts;  //время начала текущего цикла
} task[nt+1];
//объявление массива структур, содержащего
//информацию о работах
Token a_token;
//переменная для хранения копии работы, с которой
//связано текущее событие. Класс Token –
//библиотечный, он моделирует обслуживаемую заявку
Facility *disk[nd+1];//массив указателей на диски.
//Класс Facility – библиотечный, моделирует
//обслуживающее устройство
int nts=500;
//количество моделируемых циклов работ
//(продолжительность моделирования)
double tc[2]={10.0, 5.0 }, //среднее время
//обслуживания на ЦПУ для классов работ
td=30.0,sd=2.5; //среднее время обслуживания работ
//на диске и среднее отклонение
int main()
{
 int icount, i, j, event, n[2];
 double t, s[2], rn;
 struct token_info *p;
 n[0]=n[1]=0;
 s[0]=s[1]=0.0;
 for(i=1;i<=nt;i++)
  task[i].cls=(i>n0) ? 1:0;
//назначение приоритетов работам
```

```
 init_simpack(LINKED);
//инициализация модели
 Facility cpu(0,1);
//инициализация ЦПУ.Первый аргумент конструктора –
//id устройства
 for(i=1; i<=nd; i++)
//инициализация дисков. id устройства – порядковый
//номер диска
  disk[i]=new Facility(i,1);
 for(i=1; i<=nt; i++)
 {
  a_token=i;
  event_list.schedule(BEGIN_TOUR, 0.0, a_token);
//инициализация списка событий. event_list –
//библиотечный класс. schedule(a,b,c) – метод,
//заносящий в список событий событие a со временем
//наступления b и клиентским объектом c
 }
 icount=0;
 while(nts) //основной моделирующий цикл
 {
  icount++;
  event_list.next_event(event, a_token);
//выборка из списка ближайшего события. event и
//a_token –выходные аргументы
  i=a_token.id;
//определяем, какая именно работа участвует в
//событии
```

```
  p=task+i;
//получение информации об этой работе путем
//копирования указателя на ее информационную
//структуру в переменную p
  switch(event)
//обработка события в зависимости от его типа
  {
   case BEGIN_TOUR:       //начало цикла работы
    a_token.id=i;
    p->ts=time();
//сохранение времени начала цикла
    event_list.schedule(REQUEST_CPU, 0.0,
a_token);   //следующее событие для a_token –
//запрос к ЦПУ. Время наступления - немедленно
 update_arrivals(); //вызов функции из библиотеки
                    //SimPack
    break;
   case REQUEST_CPU:     //запрос к ЦПУ
    j=p->cls;  //какой приоритет у запрашивающей
              //работы
    a_token.id=i;
    if (cpu.preempt(a_token, j)==FREE)   //может
//ли a_token с учетом своего приоритета немедленно
//занять ЦПУ
    {
     rn=expntl(tc[j]); //да, может. Разыгрываем
//время обслуживания
     a_token.id=i;
```

```
     event_list.schedule(RELEASE_CPU, rn,
a_token);   //заносим в список очередное ближайшее
//событие – освобождение ЦПУ работой a_token,
//которое произойдет через rn единиц времени
    }
    break;
   case RELEASE_CPU:
//завершение обслуживания на ЦПУ
    a_token.id=i;
    cpu.release(a_token);
//моделирование освобождения устройства с помощью
//библиотечного метода класса Facility
    p->un=random(1, nd);
//равновероятный розыгрыш запрашиваемого диска
    event_list.schedule(REQUEST_DISK, 0.0,
a_token);   //заносим в список очередное ближайшее
//событие – запрос a_token к диску. Наступает
//немедленно
    break;
   case REQUEST_DISK:      //запрос к диску
    a_token.id=i;
    if (disk[p->un]->request(a_token, 0)==FREE)
//свободен ли выбранный диск
    {
     rn=erlang(td, sd); //розыгрыш длительности
//обслуживания на диске
     a_token.id=i;
```

```
     event_list.schedule(RELEASE_DISK, rn,
a_token);   //заносим в список очередное ближайшее
//событие для a_token – освобождение диска,которое
//произойдет через rn единиц времени
    }
    break;
   case RELEASE_DISK:
//освобождение диска
    a_token.id=i;
    disk[p->un]->release(a_token); //работа
//a_token освобождает диск p->un
    j=p->cls;   //в целях сбора статистики
//считываем приоритет ушедшей с диска работы
    t=time();
    s[j]=s[j]+t-p->ts;
//наращиваем суммарную длительность циклов работ
//класса j
    n[j]++;
//инкремент количества циклов работ класса j
    update_completions();  //вызов библиотечной
//функции Simpack
    a_token.id=i;
    event_list.schedule(BEGIN_TOUR, 0.0, a_token);
//a_token начинает новый цикл
    nts--;
//декремент общего счетчика туров
    break;
  }
 }
```

```
 report_stats();
//библиотечная функция SimPack
 printf("\n\n");
//вычисление средней длительности цикла для работ
//каждого из классов
 printf("class 0 tour time=%.2f\n", s[0]/n[0]);
 printf("class 1 tour time=%.2f\n", s[1]/n[1]);
 return 0;
}
```

В приложении 1 приведена реализация этой же модели с помощью схемы, рассмотренной в 2.3, с сохранением, где это возможно, прежних имен переменных. Объем исходного кода, включая полный протокол класса, функцию `main()` и описания внешних имен, составляет (без комментариев) около 3,5 Кбайт. Никакого другого специального кода для компиляции программы не требуется. Объем же текста приведенной календарно-событийной программы (без комментариев) — около 2 Кбайт, однако для ее компиляции необходимо еще довольно много библиотечного кода:

протокол объекта `event_list`, реализующего основную структуру данных — календарь событий;

протокол класса `Facility`;

тексты функций `update_arrival()`, `update_completions()`, `time()`.

Объем перечисленного кода, конечно же, намного больше 1,5 Кбайт.

Глава 3. Моделирование Web-кластера: статические дисциплины

3.1. О понятии Web-кластера

Стремительное разрастание Всемирной паутины и связанное с этим значительное увеличение объема трафика продолжают ставить перед специалистами новые задачи. Веб-серверы стали хранилищем не только огромного количества текстовой и графической информации, но также больших объемов видео- и аудиоинформации, средством проведения масштабных коммерческих операций. В связи с этим главной задачей команды разработчиков становится реализация обслуживания запроса клиента за приемлемое для него время, или, иными словами, самый важный вопрос, на который требуется ответить — как свести задержки к минимуму. Поиск ответа потребовал новых технических, алгоритмических, программных решений, приведших к созданию распределенного веб-сервера, называемого также веб-кластером.

Это решение возникло, когда стало ясно, что простой переход на более мощную аппаратную платформу (hardware scale-up) и добавление новых ресурсов, таких как память, диски, процессор, не решают проблемы. При весьма больших затратах эффект от такого экстенсивного решения ощущается крайне непродолжительное время, так как, согласно [95], количество активных пользователей Интернета увеличивается на 90% ежегодно. В [55] в свое время был приведен следующий факт. Для организации портала NetCenter компания Netscape приобрела мощный SGI-сервер, но это не дало ожидаемых результатов. Ситуация значительно улучшилась после замены конфигурации десятком обычных PC, объединенных в кластер.

За последние 15-20 лет веб-кластеры стали объектом пристального изучения специалистов самой различной направленности — IT-менеджеров, инженеров-сетевиков, ученых-математиков. Об этом свидетельствуют нарастающий из года в год поток публикаций (в том числе монографии [51], [80]), тематика международных конференций, появление экспериментальных систем в университетах и научных центрах, интенсивно развивающийся рынок распределителей (диспетчеров) нагрузки, на котором работают 8 крупных компаний [98], [99], [100], [101], [102], [103], [104], [105], формирование терминологии. Типичные схемы, используемые для обозначения веб-кластера, представлены на рис. 3.1 (упрощенная) и 3.2 (более подробная).

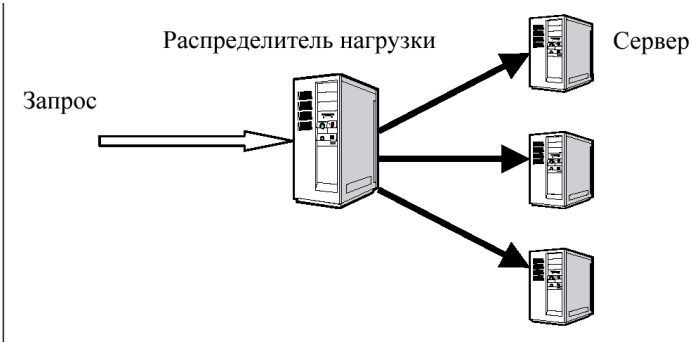

Рис. 3.1. Упрощенная схема веб-кластера

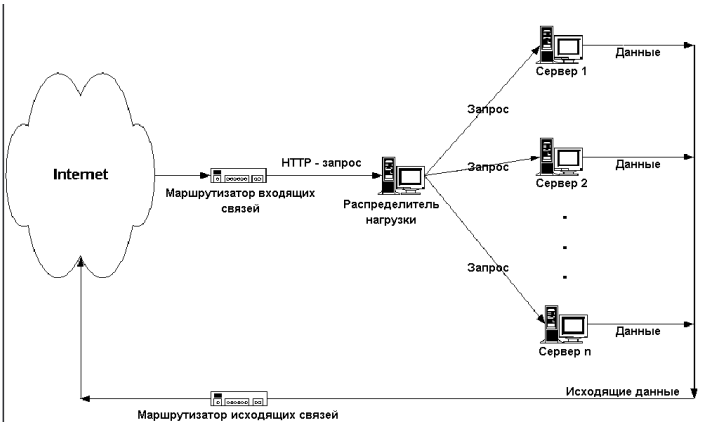

Рис. 3.2. Развернутая схема веб-кластера

Организация распределенного веб-сервера весьма сложна и требует детальной проработки множества вопросов, например, таких:

- изучение и анализ сетевого трафика и ожидаемой загрузки сервера, распределения длительности обслуживания запросов;
- выбор архитектуры распределенной системы;
- маршрутизация запросов внутри системы (requested-routing mechanism);
- функциональность распределителя нагрузки;
- дисциплина обслуживания запросов и выбора сервера (dispatching algorithm);
- распределение хранимой информации (content placement).

Подробные обзоры текущего состояния вопросов, связанных с изучением веб-кластеров, можно найти в [33] и [95].

3.2. Потоки данных в Web

Изучение характерных особенностей потоков данных и нагрузки на

сервер в Веб имеет настолько важное значение для эффективного проектирования, что давно уже оформилось в отдельный раздел теории вычислительных систем. Обширному кругу входящих в него вопросов посвящено множество публикаций, выявлены важнейшие закономерности, ставшие общепризнанными и играющие ключевую роль при разработке новых дисциплин обслуживания в веб-кластерах. Рассмотрим их несколько подробнее.

На протяжении нескольких десятилетий при анализе моделей, возникающих в теории вычислительных систем, исследователи привыкли предполагать входные потоки пуассоновскими, а распределения длин заявок — экспоненциальными. Эти предположения позволяют легко строить марковский процесс и получать наглядные аналитические результаты, которые носят если и не предсказательный (predictive), то по крайней мере объясняющий (explanatory) характер. Разумеется, когда в первой половине 90-х гг. специалисты занялись моделированием столь сложной структуры, как Всемирная паутина, не мог не возникнуть вопрос, насколько эти предположения применительно к ней близки к реальности. Основополагающей здесь явилась работа [87], с тех пор регулярно цитируемая. Обработав большой статистический материал, авторы убедительно показали, что потоки в Веб — не пуассоновские, а описываются другими законами распределения — с «тяжелым хвостом» (heavy-tailed), наиболее известными представителями которых являются степенные распределения. С этого момента появилось большое количество работ, в которых, с одной стороны, находятся все новые характеристики Сети, подчиняющиеся этому закону, с другой — строятся вероятностные модели на основе

степенных распределений.

Эти распределения подчиняются следующей зависимости: $\Pr(X > x) \sim x^{-a}$, где $0 < \alpha < 2$, $\Pr(X > x)$ — вероятность того, что значение случайной величины X превысит заданное число x.

Типичными примерами распределения с «тяжелым хвостом» являются распределение Парето

$$\Pr(X \le x) = \begin{cases} 0, & x < s; \\ 1 - s^{\alpha} x^{-\alpha}, & x \ge s, \ \alpha < 2 \end{cases}$$

и его модификация для случая ограниченного диапазона значений случайной величины

$$\Pr(X \le x) = \begin{cases} 0, & x \le 0; \\ k(1 - (x+1)^{-\alpha}, & 0 < x < \beta, \ k = 1/1 - (\beta+1)^{-\alpha}; \\ 1, & x \ge \beta. \end{cases}$$

Основными свойствами НТ-распределения являются:

с увеличением продолжительности накопленного интервала времени ожидаемое остаточное время увеличивается;

бесконечное значение дисперсии, а при $\alpha < 1$ — бесконечное математическое ожидание;

большая доля нагрузки приходится на очень малую (менее 1%) долю «длинных» интервалов времени.

Остановимся подробнее на первом свойстве. Дело в том, что случайные величины, описываемые различными законами распределения, могут вести себя по-разному не только количественно, но и качественно. В частности, одна из классификаций разделяет непрерывные функции распределения на ВФИ (с возрастающей функцией интенсивности) и УФИ (с убывающей функцией

интенсивности) [47], которые иначе называют, соответственно, *стареющими* и *молодеющими*. Пусть случайная величина (для определенности — длительность интервала ожидания) описывается функцией распределения $F(x)$. Введем в рассмотрение функцию

$$P_y(x) = \frac{F(x+y) - F(x)}{1 - F(x)}$$

— вероятность того, что интервал завершится в течение ближайших y единиц времени при условии, что с момента начала интервала прошло x единиц. Для ВФИ-распределений эта функция является возрастающей, для УФИ — убывающей. Таким образом, чем больше «возраст» ВФИ-величины,ее тем выше вероятность того, что в течение ближайшего времени она завершится. У УФИ-величины все наоборот — чем выше ее «возраст», тем вероятность ее дальнейшей «жизни» увеличивается. Разумеется, подавляющее большинство случайных величин, с которыми мы сталкиваемся в повседневной жизни (при ее нормальном течении), являются «стареющими», потому что все должно когда-нибудь заканчиваться. Наличие «молодеющей» величины обычно свидетельствует о какой-то не слишком хорошей, а зачастую и аварийной, ситуации. В соответствии с законами диалектики, эти два семейства случайных величин одновременно и далеки по свойствам друг от друга, и близки — одна при определенных обстоятельствах легко может перейти в другую. Рассмотрим примеры.

Длительность ожидания автобуса на остановке является в нормальной ситуации «стареющей» величиной — чем дольше мы простояли, тем больше вероятность того, что в следующую минуту автобус придет. Однако когда эта длительность достигает определенного довольно

большого значения, появляется подозрение — произошло нечто, нарушившее нормальный график движения, и автобус вряд ли придет. С этого момента означенная случайная величина начинает «молодеть» — чем больше мы ждем, тем уверенность, что ждать бесполезно, становится крепче, и в конце концов мы покидаем остановку.

Можно привести и другой пример. Вероятность того, что в течение ближайшего года выйдет замуж 20-летняя девушка, выше, чем та же вероятность была год назад, когда ей было 19 лет. Однако если по достижении определенного возраста (тут каждый может поставить любую цифру) замужество все еще не состоялось, эта вероятность с каждым годом уменьшается, то есть распределение остаточной длительности «холостой» жизни «молодеет».

К «стареющим» распределениям относится, например, равномерное на интервале от a до b, для которого $P_y(x) = y/(b - x)$. С ростом x эта функция возрастает. Для распределения Парето функция $P_y(x) = 1 - (1 - (y/x + y))^{\alpha}$, наоборот, убывает.

В науке, как и в жизни, «молодеющие» распределения, в частности, распределения «с тяжелым хвостом» (далее HT), представляют проблему — системы, описываемые ими, анализировать сложнее, в чем мы еще убедимся в дальнейшем.

Подробному изучению распределений с «тяжелым хвостом» посвящена монография [56]. Современные методы анализа вычислительных систем с очередями полно и всесторонне рассмотрены в [89], в частности, нагрузке, описываемой «тяжелыми хвостами», полностью посвящены главы 3 и 6. Известно [73], [74], что

для файлов, передаваемых в Интернете по протоколу http, значение α лежит в пределах от 1,1 до 1,3.

Подробная классификация статистических данных по трафику в Интернете приведена в [49] и [50]. В качестве одной из ключевых особенностей отмечается значительная неравномерность распределения различных показателей. Это относится в первую очередь к концентрации ссылок — на 10% файлов приходится до 90% запросов, которые составляют 90% трафика, и географической статистике — обращения из 10% от общего числа доменов составляют свыше 75% всех обращений.

3.3. Дисциплина маршрутизации запросов SITA-E

Главную роль в функционировании веб-кластера играет распределитель нагрузки (dispatcher, load balancer), который принимает решение о дальнейшем движении каждого запроса из входного потока. При этом весь веб-кластер представлен для внешнего наблюдателя только одним IP-адресом, называемым *виртуальным* (VIP).

В нашу задачу входит рассмотрение и сравнение дисциплин маршрутизации — тех алгоритмов, на основании которых диспетчер принимает решение о том, на какой сервер направить поступивший запрос. Повсеместное же применение степенных распределений к анализу самых разнообразных статистик в Веб объясняет тот факт, что именно это распределение предполагается исходным условием при разработке и анализе эффективности новых дисциплин обслуживания для веб-кластеров.

В этой главе мы займемся моделированием и сравнением статических

дисциплин обслуживания заявок, то есть таких, которые для принятия решения не учитывают текущую информацию о загрузке серверов. Традиционно к числу наиболее распространенных из них относятся FCFS (первым прибыл — первым обслужен) и PS (разделение процессора), описанные в главе 1. Если серверов несколько, к аббревиатурам этих дисциплин добавляют слово RANDOM, которое показывает, что сервер выбирается случайным образом с равной вероятностью. Основные закономерности, присущие дисциплинам FCFS и PS, давно и хорошо изучены. Одна из них утверждает, что при малых значениях коэффициента вариации длин заявок (менее единицы) меньшее время пребывания заявки в системе достигается при использовании FCFS. Если же коэффициент вариации больше единицы, лучшие показатели дает использование PS. Проиллюстрируем это утверждение примитивным, но акцентированным примером. Пусть длина заявки фиксирована (то есть коэффициент вариации равен нулю) и равна A. Предположим, что в начальный момент времени в очереди перед пустым сервером находятся две заявки. При дисциплине FCFS первая заявка проведет в системе A единиц времени, вторая — $2A$, то есть среднее время пребывания равно $1{,}5A$. При дисциплине PS обе заявки будут обслуживаться одновременно в течение времени $2A$, то есть среднее время пребывания равно $2A$, что в 1,33 раза больше показателя для FCFS.

Улучшить показатели производительности по сравнению с использованием обеих дисциплин можно только в том случае, если при маршрутизации принять во внимание трудоемкость и ресурсоемкость поступившего запроса. При этом предположение о

НТ-свойстве длин поступающих запросов приводит к коренному пересмотру традиционных представлений о балансировке нагрузки. Но прежде всего следует ответить на вопрос: насколько реально априорно оценить длину поступившей заявки?

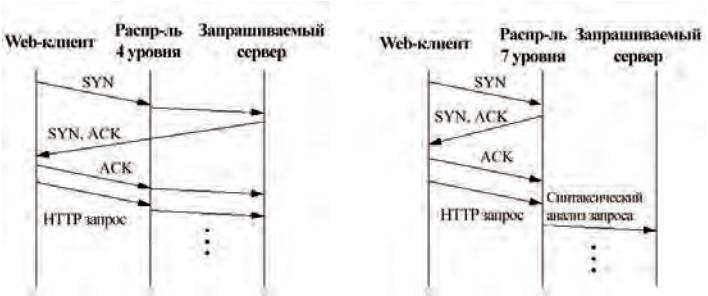

Рис. 3.3. Различие между обработкой запросов устройствами layer-4 и layer-7

Функциональность современных распределителей нагрузки позволяет это. Уже стала общепринятой их классификация на устройства уровня 4[1] (layer-4) и уровня 7[2] (layer-7) [54]. Различие между ними иллюстрирует рис. 3.3. Устройства layer-4 анализируют пакеты только на уровне TCP/IP, не доходя до уровня приложения. Они не принимают во внимание содержимое HTTP-запроса при перенаправлении пакета на сервер. Такой механизм называется *слепым к содержанию* (content-blind), или *немедленным связыванием* (immediate binding). Устройства же layer-7 принимают решение о маршрутизации запроса только при получении HTTP-пакета. При этом они могут, проведя анализ URL, принять во внимание тип запроса, оценить его трудоемкость и определить, на каком сервере

[1] Транспортный уровень согласно модели сетевых функций OSI

[2] Прикладной уровень согласно модели сетевых функций OSI

находится запрашиваемый ресурс. Этот механизм называется *content-aware*, или *отложенное связывание* (delayed binding). Уровень приложения сейчас поддерживают многие коммерческие продукты. Их полный перечень приведен в [95]. С другой стороны, применение устройств layer-7 оправдано, если его возможности используются дисциплиной обслуживания. Каким же образом можно это сделать? Здесь свое слово должны были сказать математики.

Одной из первых работ на эту тему была [55]. В ней отмечено, что стремление равномерно загрузить серверы не соответствует структуре потоков в Интернете. Авторы предложили дисциплину SITA-V (Size Interval Tasks Assignment-variable), основная идея которой состоит в том, чтобы «легкие» задачи посылать на хост с меньшей загрузкой, а «тяжелые» — с большей, где граничные точки, определяющие разделение задач на «легкие» и «тяжелые», вычисляются согласно специальному алгоритму. Так как при степенном распределении первых задач больше, а вторых — меньше, эффективность SITA-V возрастает с увеличением выраженности «heavy-tail»-свойства, то есть с уменьшением параметра α. В книге приведен пример, в котором средний коэффициент задержки (напомним, что так называется отношение времени пребывания заявки в системе к ее длине) снижается более чем в 100 раз по сравнению с дисциплиной равномерной загрузки. Однако среднее время пребывания на сервере при этом возрастает. Кроме того, предложенные методы расчета точек разбиения длин заявок на интервалы, позволяющие отличить «легкие» задачи от «тяжелых», носят весьма приближенный характер.

Следующим шагом в этом направлении явилась работа [74] тех же авторов. В ней рассмотрена новая дисциплина SITA-E (E означает

equal). Основная идея та же — путем учета длины заявки при маршрутизации снизить коэффициент вариации длин заявок на каждом из серверов, повысив таким образом конкурентоспособность FCFS. Ее отличие от SITA-V заключается в том, что для точек разбиения (cutoff points) предложена точная формула: если требуется разделить входной поток заявок между N серверами, $N-1$ точка разбиения длин заявок на интервалы находится из условия

$$\int_0^{x_1} xf(x)dx = \int_{x_1}^{x_2} xf(x)dx = \ldots = \int_{x_{N-1}}^{X} xf(x)dx,$$

где $f(x)$ — плотность распределения длин заявок. Заявки, длина которых лежит в пределах интервала $[x_{i-1}; x_i]$, направляются на i-й сервер. Интегральное соотношение означает равенство интервалов, взвешенных по вероятности.

Для практической реализации или имитационного моделирования дисциплины SITA-E необходимо предварительно вычислить точки разбиения x_i, которых в случае N серверов будет $N-1$. Удобство вычислений зависит от вида функции $f(x)$: для одних функций можно получить элементарное решение, для других — нет. Приведем несколько примеров:

Равномерное распределение на интервале $[a; b]$:

$$x_i = \sqrt{a^2 + i\left(\left(b^2 - a^2\right)/N\right)},\ i = 0, 1, \ldots, N. \qquad (3.2)$$

Распределение Парето: $x_i = s\left(1 - \frac{i}{N}\right)^{\frac{1}{1-\alpha}}$, $1 < \alpha < 2$.

Экспоненциальное распределение: аналитического решения нет. x_1 находится из численного решения уравнения

$$e^{-\mu x_1}\left(x_1 + 1/\mu\right) = (N-1)/(N\mu),$$

далее последовательно решаются уравнения

$$e^{-\mu x_{i+1}}\left(x_{i+1} + 1/\mu\right) = e^{-\mu x_i}\left(x_i + 1/\mu\right) - 1/(N\mu).$$

Чтобы получить представление о том, насколько по-разному ведут себя распределения, приведем некоторые численные результаты. Выпишем значения точек разбиения для экспоненциального и Парето-распределений при $N = 8$, $s = 0{,}5$, $\alpha = 1{,}2$, $\mu = 1/3$ (табл. 3.1).

Таблица 3.1. Точки разбиения для экспоненциального и Парето-распределений

Разбие	**Значения точек разбиения**							
ние	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Эксп-е	0	1,83	2,89	3,92	5,04	6,36	8,08	10,81
Парето	0,5	0,97	2,11	5,24	16	67,42	512	16 384

Математическое ожидание при этих значениях параметров у обоих распределений одинаково и равно 3.

Как объяснить столь разительные отличия? Представим себе какое-то очень большое число U, такое, что подавляющее большинство реализаций случайной величины, распределенной по закону $F(x)$, попадает в интервал $[0; U)$. Это подавляющее большинство может быть уравновешено ничтожным меньшинством реализаций, попавших в интервал $X > U$. Образно говоря, первые возьмут количеством, вторые — весом. Но это произойдет только в том случае, если функция распределения $F(X)$ допускает появление на интервале $X > U$ с достаточной для этого вероятностью нужного количества «тяжелых» заявок (отсюда и название — с «тяжелым хвостом»). Так вот, распределение Парето допускает такую ситуацию для гораздо

больших значений U, нежели экспоненциальное.

При тестировании ГСЧ для распределения Парето был получен 1 млн случайных чисел. Из них в интервал [0,5; 0,97] попало 551 064 (более 50%), в интервал [512; 16 382] — 232 (менее 0,03%). 232 «тяжелых» заявки загрузят свой сервер приблизительно так же, как более полумиллиона «легких».

3.4. Парадокс SITA-E

Программную реализацию SITA-E следует проводить с достаточной осторожностью, так как простое следование алгоритмическому описанию может привести к совершенно парадоксальным и необъяснимым результатам. Новое зачастую бывает коварным, скрывая свои «подводные камни» за кажущейся простотой и понятностью. Рассмотрим простой пример [39]. Пусть интенсивность входного потока равна $\lambda = 0{,}21$, а средняя длина заявки равномерно распределена на интервале [20; 40], то есть $\mu = 1/30 \approx 0{,}033$. Разумеется, один сервер не может в этом случае обеспечить стационарный режим, так как $\mu < \lambda$. Минимально необходимое количество серверов для этого определяется из условия $n>\lambda/\mu$ и равно 7, но для верности пусть их будет 8. Предположим также, что модельное время не масштабируется, то есть длина заявки генерируется как целое число от 20 до 40 с равной вероятностью 1/21 каждое. Если моделировать такую многоканальную систему при условии общей очереди, стационарный режим будет существовать. Если разделить очереди, используя дисциплину FCFS-Random, заявки в статистически равных количествах «разойдутся» по восьми очередям, опять-таки обеспечив существование стационарного

режима. А вот попытка применить SITA-E приводит к «астрономическим» значениям для счетчиков общего числа заявок и времени пребывания их в системе и, как следствие, к аварийному завершению программы по причине переполнения. В чем же дело? SITA-E никуда не годится? Не будем торопиться с выводами.

Вычислим по формуле (3.2) с точностью до двух знаков значения точек разбиения. Они равны: 20; 23,45; 26,46; 29,15; 31,62; 33,91; 36,06; 38,08; 40. Таким образом, на серверы попадают заявки следующей длины, создавая тем самым нагрузку:

1. Сервер 1 — 20, 21, 22, 23. $\lambda_1 = 0{,}21 \cdot (4/21) = 0{,}04$, $\mu_1 = 1/21{,}5 \approx 0{,}0465 > \lambda_1$.

Сервер 2 — 24, 25, 26. $\lambda_2 = 0{,}21 \cdot (3/21) = 0{,}03$, $\mu_2 = 1/25 \approx 0{,}04 > \lambda_2$.

Сервер 3 — 27, 28, 29. $\lambda_3 = 0{,}21 \cdot (3/21) = 0{,}03$, $\mu_2 = 1/28 \approx 0{,}036 > \lambda_3$.

Сервер 4 — 30, 31. $\lambda_4 = 0{,}21 \cdot (2/21) = 0{,}02$, $\mu_2 = 1/30{,}5 \approx 0{,}033 > \lambda_4$.

Сервер 5 — 32, 33. $\lambda_5 = 0{,}21 \cdot (2/21) = 0{,}02$, $\mu_5 = 1/32{,}5 \approx 0{,}031 > \lambda_5$.

Сервер 6 — 34, 35, 36. $\lambda_6 = 0{,}21 \cdot (3/21) = 0{,}03$, $\mu_6 = 1/35 \approx 0{,}0286 < \lambda_6$.

Сервер 7 — 37, 38. $\lambda_7 = 0{,}21 \cdot (2/21) = 0{,}02$, $\mu_7 = 1/37{,}5 \approx 0{,}027 > \lambda_7$.

Сервер 8 — 39, 40. $\lambda_8 = 0{,}21 \cdot (2/21) = 0{,}02$, $\mu_2 = 1/39{,}5 \approx 0{,}025 > \lambda_8$.

Причина найдена: на шестом сервере стационарный режим не обеспечивается. Если распределение длин заявок является на самом деле не дискретным, а непрерывным, для исключения подобной ситуации следует применить масштабирование времени. Уже при значении коэффициента масштабирования 10 парадокс исчезает.

3.5. Программная реализация

С точки зрения программирования, ничего нового сверх того, чем мы пользовались ранее, для моделирования распределенного веб-сервера не требуется. Поэтому не будем останавливаться на вопросах проектирования классов, а сразу приведем листинги их протоколов, комментариев к которым вполне достаточно для прояснения возникающих вопросов. Задержка заявок на распределителе нагрузки для простоты предполагается равной нулю, поэтому маршрутизация на сервер происходит мгновенно и очереди к распределителю нет.

Итак, опишем классы, необходимые для моделирования дисциплины SITA-E, при условии, что длины заявок распределены в соответствии с распределением Парето. Заявки моделируются объектами класса `Dispatcher`, серверы – объектами класса `Server`, а распределитель нагрузки – объектом класса `Dispatcher`. Дисциплина реализуется методом `Arrival()` класса `Dispatcher`.

Листинг 3.1. Описания классов

```
#include<cstdio>
#include<cstdlib>
#include<ctime>
#include<cmath>
using namespace std;
#include "List.h" /*реализация приведена в
приложении 2*/
#define RATIO 100 /*коэффициент масштабирования
времени*/
```

```
int entered=0;  /*счетчик поступлений заявок в
систему*/
int completed=0;    //счетчик обслуженных заявок
double soj_ave=0;  /*для вычисления среднего
времени пребывания заявки в системе*/
double slow_ave=0;  /*для вычисления среднего
коэффициента задержки заявки*/
double que_ave=0; /*для вычисления средней длины
очереди*/
long int total;  /*счетчик тактов модельного
времени*/

/*Протокол класса Client. В программе класс
представлен неограниченным сверху количеством
объектов*/
class Client
{
 int id;  /*уникальный идентификатор заявки; равен
значению счетчика поступлений*/
 int time;  /*счетчик времени, проведенного в
системе*/
 int length;  /*длина заявки. Ее необходимо
хранить в объекте для вычисления  коэффициента
задержки, когда заявка завершит обслуживание*/
 public:
/*Сервер и распределитель нагрузки напрямую
работают с заявками*/
 friend class Server;
 friend class Dispatcher;
```

```
 Client(int i, double k, double alpha);
};

//Конструктор
Client::Client(int i, double k, double alpha)
{
 id=i;
 time=0;
 length=(int)(get_pareto(k,alpha)*RATIO);
 if (length==0) length=1;
}

/*Протокол класса Server. В программе класс
представлен фиксированным количеством объектов*/
class Server
{
 int id;                  //номер сервера
 ListNode<Client> *queue;  /*очередь заявок к
серверу. Моделируется связным списком*/
 int q_length;       //текущая длина очереди
 Client *serving;   /*указатель на текущую
обслуживаемую заявку*/
 int to_served;   /*время, оставшееся до
завершения обслуживания текущей заявки*/
 public:
 friend class Dispatcher;   /*распределитель
нагрузки напрямую работает с серверами*/
 Server(int i);
```

```
 ~Server();
 void Completed();
 void Arrival(Client *client);
 void run();
};

/*Конструктор. В исходном состоянии сервер
свободен, очереди нет*/
Server::Server(int i)
{
 id=i;
 queue=NULL;
 q_length=0;
 serving=NULL;
 to_served=-1;
}

/*деструктор. Удаляет объекты класса Client,
находящиеся в очереди и на обслуживании в момент
завершения моделирования*/
Server::~Server()
{
 if (serving) delete serving;
 while (queue) queue=ListDelete<Client>(queue,
queue);
}
```

```
//Завершение обслуживания заявки на сервере
void Server::Completed()
{
 completed++;   /*инкремент счетчика обслуженных
заявок*/
//Перевычисление среднего коэффициента задержки
slow_ave=slow_ave*(1-1.0/completed)
+(double)(serving->time)/((serving-> length)
*completed);
/*Перевычисление среднего времени пребывания
заявок в системе*/
soj_ave=soj_ave*(1-1.0/completed)
+(double)(serving->time)/completed;
/*Удаление объекта, соответствующего обслуженной
заявке*/
 delete serving;
 if (q_length==0)   //очередь пуста
 {
   serving=NULL;
   to_served=-1;
   return;
 }
/*Очередь не пуста. Первую заявку из очереди
ставим на обслуживание*/
 serving=queue->Data();
 to_served=serving->length;
 q_length--;
 queue=queue->Next();   //продвигаем очередь
```

```
}

//Поступление на сервер новой заявки
void Server::Arrival(Client *client)
{
 ListNode<Client> *ptr;
 if (!serving)   /*сервер свободен. Ставим новую
заявку на обслуживание */
 {
  serving=client;
  to_served=serving->length;
  return;
 }
//Сервер занят. Новую заявку ставим в очередь
 q_length++;
 ptr=new ListNode<Client>(client, NULL);
 if (q_length==1) queue=ptr;  //очередь была пуста
 else ListAdd<Client>(queue,ptr);   /*очередь
существовала*/
}

//Диспетчер
void Server::run()
{
 ListNode<Client> *ptr;
 if (to_served>0) to_served--;  /*декремент
остаточного времени обслуживания*/
```

```
 if (to_served==0) Completed();
//обслуживание завершено
 if (serving) serving->time++;   /*инкремент
времени пребывания в системе обслуживаемой
заявки*/
 ptr=queue;
//Инкремент времени пребывания заявок из очереди
 while(ptr)
 {
  ptr->Data()->time++;
  ptr=ptr->Next();
 }
}

/*Протокол класса Dispatcher. В программе
представлен единственным объектом*/
class Dispatcher
{
 int M;              //количество серверов
 Server **farm;    //массив указателей на сервера
 double s;  /*первый параметр распределения
Парето – минимальная длина заявки*/
 double alpha;  /*второй параметр распределения
Парето*/
 double input_rate;  /*средняя интенсивность
входного потока, описываемого распределением
Пуассона*/
```

```
 double *cut_points;  /*массив точек разбиения
(для реализации SITA-E)*/
 double to_arrival;  /*время, оставшееся до
прибытия следующей заявки*/

 public:
 Dispatcher(int i, double b, double c, double d);
 ~Dispatcher();
 void Arrival();
 void run();
};

/*Конструктор. Аргументы: количество серверов,
интенсивность входного потока, параметры
распределения длины заявки*/
Dispatcher::Dispatcher(int i, double b, double c,
double d)
{
 int k, p1, p2;
 M=i;
 S=C;
 alpha=d;
 input_rate=b;
 farm=new Server *[M];
 for(k=0;k<M;k++)    //инициализация серверов
  farm[k]=new Server(k+1);
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival=1;
```

```
 cut_points=new double[M];
  for(k=0;k<M;k++)   //вычисление точек разбиения
  cut_points[k]=serv_mean*pow(1-
((double)k)/M,1.0/(1-serv_offset))*RATIO;
}

//деструктор. Возвращает память
Dispatcher::~Dispatcher()
{
 delete [] cut_points;
 for(int i=0;i<M;i++)  delete farm[i];
 delete [] farm;
}

//Прибытие новой заявки
void Dispatcher::Arrival()
{
 int k, pr; Client *cl;
/*Разыгрываем новый интервал между прибытиями
заявок*/
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival=1;
 entered++;   /*инкремент счетчика поступлений
заявок в систему*/
 cl=new Client(entered, s, alpha);  /*создаем
новую заявку*/
```

```
/*Маршрутизируем заявку в соответствии с ее длиной
и дисциплиной SITA-E*/
 pr=0;
 for(k=0;k<(M-1);k++)
 {
  if ((cl->length>=cut_points[k])&&(cl-
>length<=cut_points[k+1]))
  {
   farm[k]->Arrival(cl); /*решение принято –
заявка отправляется на k-й сервер*/
   pr=1;
   break;
  }
 }
 if (pr==0)  farm[M-1]->Arrival(cl);  /*заявка
отправляется на последний сервер, соответствующий
полуоткрытому интервалу*/
}

//Диспетчер
void Dispatcher::run()
{
 int p, k, q;
 to_arrival--;
 if (to_arrival==0) Arrival();
/*Последовательный вызов диспетчера для всех
серверов*/
 for(k=0;k<M;k++)
```

```
  farm[k]->run();
/*Каждые сто единиц модельного времени – пересчет
средней длины очереди*/
 if ((total+1)%100==0)
 {
  q=0;
  for(k=0;k<M;k++)
   q+=farm[k]->q_length;
  p=(total+1)/100;
  que_ave=que_ave*(1-1.0/p)+((double)q)/p;
 }
}
```

Приведем теперь описания тех же классов для дисциплины разделения процессора PS. Поскольку нам еще не приходилось моделировать эту дисциплину, коротко опишем особенности ее реализации. Как известно [47], при дисциплине разделения процессора очередь отсутствует. Заявка сразу поступает на обслуживание к одному из серверов, и ресурс процессора равномерно делится среди всех находящихся на сервере заявок. Таким образом, если на сервере находятся N заявок, то скорость обслуживания каждой из них равна не 1, а $1/N$. Соответственно, и остаточное время обслуживания заявки в каждом такте моделирования убывает не на единицу, а на дробную величину $1/N$. Поэтому тип такого традиционного поля данных, как время, оставшееся до завершения обслуживания заявки, нужно объявлять не `int`, а `float` или `double`. Кроме того, в данной программе автору показалось удобнее сделать эту величину полем данных заявки, а не заводить

специальный массив на сервере. И последняя деталь. Если алгоритмическая реализация дисциплины SITA-E осуществлялась исключительно распределителем нагрузки, то дисциплину разделения процессора реализует по отдельности каждый сервер, поэтому класс `Server` претерпевает наибольшие изменения. У распределителя же меняется всего несколько строчек кода в методе `Arrival()`, так как теперь прибывшая заявка случайным образом пересылается на любой из серверов. Дисциплина реализуется методом `Arrival()` класса `Dispatcher` и методами класса `Server`.

Листинг 3.2. Модифицированные описания классов для дисциплины PS

```
class Client
{
 int id;
 int time;
 int length;
 double to_served;  /*остаточное время
обслуживания заявки – НЕ целое */
 public:
 friend class Server;
 friend class Dispatcher;
 Client(int i, double c, double d);
};
Client::Client(int i, double c, double d)
{
 id=i;
 time=0;
```

```
 length=(int)(get_pareto(c,d)*RATIO);
 if (length==0) length=1;
 to_served=length;   /*начальное значение
остаточного времени – длина заявки
}

class Server
{
 int id;
 ListNode<Client> *queue;  /*теперь это список не
ожидающих, а обслуживаемых заявок*/
 int q_length;
 public:
 friend class Dispatcher;
 Server(int i);
 void Completed(ListNode<Client> *client);
 void Arrival(Client *client);
 void run();
};

Server::Server(int i)
{
 id=i;
 queue=NULL;
 q_length=0;
}
```

```
void Server::Completed(ListNode<Client> *client)
{
 completed++;
 slow_ave=slow_ave*(1-
1.0/completed)+(double)(client->Data()->
time)/((client->Data()->length)*completed);
 soj_ave=soj_ave*(1-
1.0/completed)+(double)(client->Data()-
>time)/completed;
 q_length--;
/*Заявка, завершившая обслуживание, не обязательно
находится в голове списка. Поэтому просто
продвинуть очередь уже нельзя. Нужно
воспользоваться методом ListDelete*/
  queue=ListDelete<Client>(queue, client);
}

void Server::Arrival(Client *client)
{
 ListNode<Client> *ptr;
 q_length++;
 ptr=new ListNode<Client>(client, NULL);
  if (q_length==1) queue=ptr;
 else ListAdd<Client>(queue,ptr);
}

void Server::run()
{
```

```
 int z;
 ListNode<Client> *ptr;
 if (q_length==0) return;
  z=q_length;
/*Обход всех заявок на сервере. Декремент
остаточного времени и при необходимости обработка
события завершения обслуживания*/
 m1: ptr=queue;
 while(ptr!=NULL)
 {
  ptr->Data()->to_served-=1.0/z; /*«дробный»
декремент остаточного времени обслуживания*/
  if (ptr->Data()->to_served<=0)   /*обслуживание
завершено*/
  {
   Completed(ptr); /*исключаем обслуженную заявку
из связного списка и... */
   if (q_length==0) return;
   else goto m1;    /*если заявки на сервере после
этого еще остались, продолжаем их обход*/
  }
  else ptr=ptr->Next();
 }
/*После исключения обслуженных заявок производим
инкремент времени пребывания в системе у
оставшихся*/
 ptr=queue;
 while(ptr!=NULL)
```

```
 {
  ptr->Data()->time++;
  ptr=ptr->Next();
 }
}

//Для класса Dispatcher меняется только метод
Arrival
void Dispatcher::Arrival()
{
 int k; Client *cl;
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival=1;
 entered++;
 cl=new Client(entered, s, alpha);
k=rand()%M;  //сервер выбирается случайным образом
farm[k]->Arrival(cl);
}
```

3.6. Анализ результатов

Во всех проведенных на основе разработанных классов вычислительных экспериментах были заданы следующие значения параметров:

средняя интенсивность пуассоновского входного потока — 0,16;

средняя длительность обслуживания — 30 единиц времени;

коэффициент масштабирования времени — 100;

продолжительность моделирования — 1 млн тактов.

Исследовались зависимости средней длины очереди, среднего коэффициента задержки и среднего времени пребывания заявки в системе от параметров, характеризующих дисперсию длины заявки (при фиксированном количестве серверов, равном восьми), и от количества серверов (при фиксированных параметрах распределения длины заявки). Моделировались три распределения длины заявки — равномерное, экспоненциальное и Парето — в сочетаниях с тремя дисциплинами обслуживания — SITA-E, PS-Random и FCFS-Random. Прокомментируем результаты с точки зрения имеющейся у нас информации о свойствах распределений и дисциплин.

На рис. 3.4–3.6 приведены зависимости показателей системы от максимального отклонения от средней длительности обслуживания заявки. Длина заявки имеет равномерное распределение со средним 30, параметр, задающий максимальное отклонение от среднего, меняется от 1 до 29, количество серверов — 8.

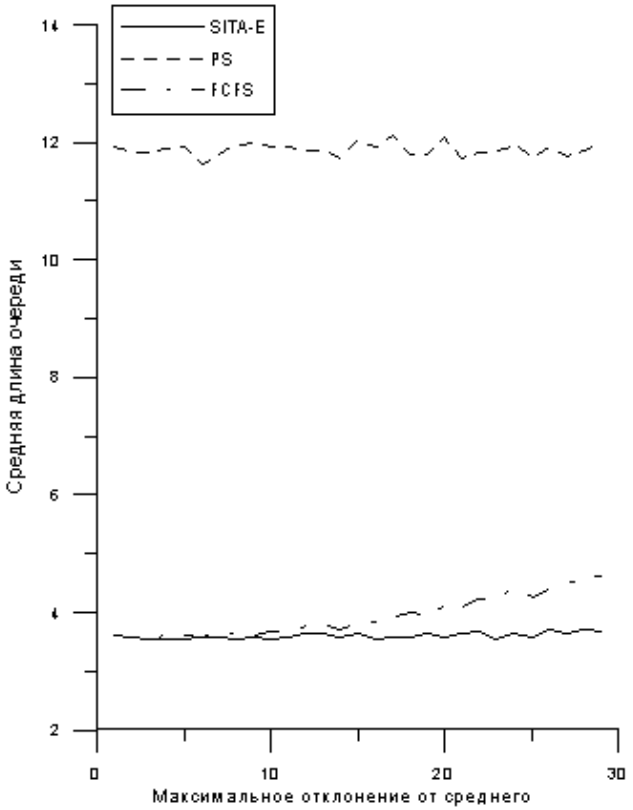

Рис. 3.4. Зависимость средней длины очереди от максимального «разброса» длины заявки. Распределение равномерное, среднее значение — 30, 8 серверов

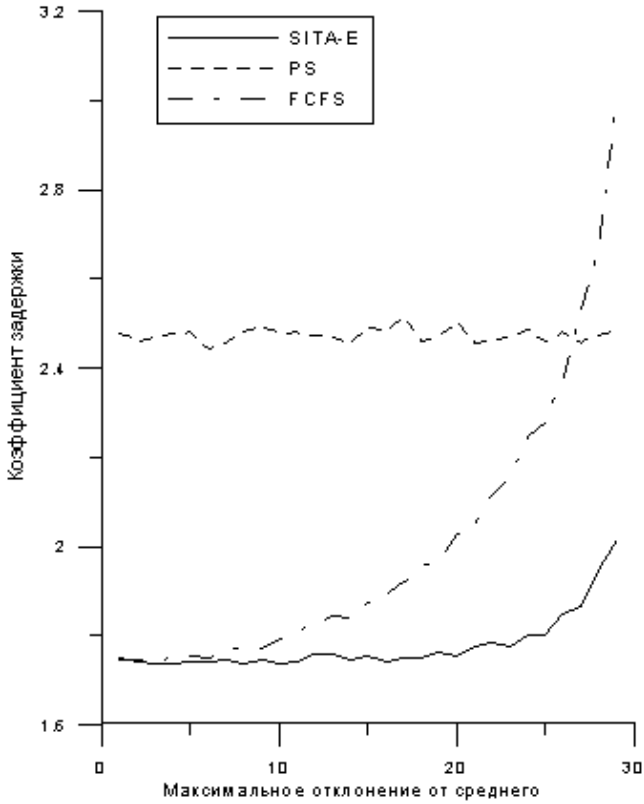

Рис. 3.5. Зависимость среднего коэффициента замедления от максимального «разброса» длины заявки. Распределение равномерное, среднее значение — 30, 8 серверов

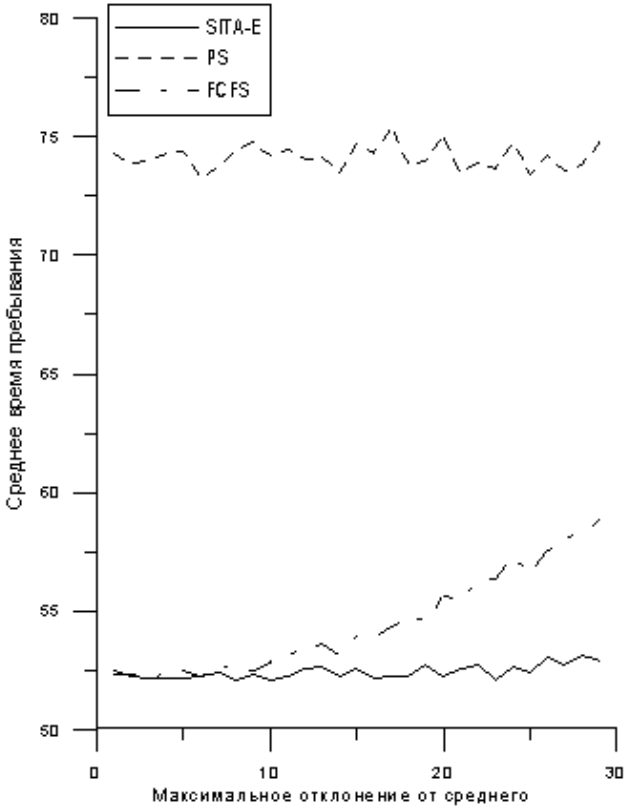

Рис. 3.6. Зависимость среднего времени пребывания заявки в системе от максимального «разброса» длины заявки. Распределение равномерное, среднее значение — 30, 8 серверов

Мы видим, что показатели дисциплины PS хуже остальных. Этот факт объясняется тем, что коэффициент вариации для равномерного распределения меньше единицы, поэтому дисциплина PS неэффективна. Все же для коэффициента задержки при больших значениях максимального отклонения преимущество получает PS.

Примерно ту же картину мы наблюдаем и для зависимостей от

количества серверов, которое на рис. 3.7–3.9 меняется от 6 до 20.

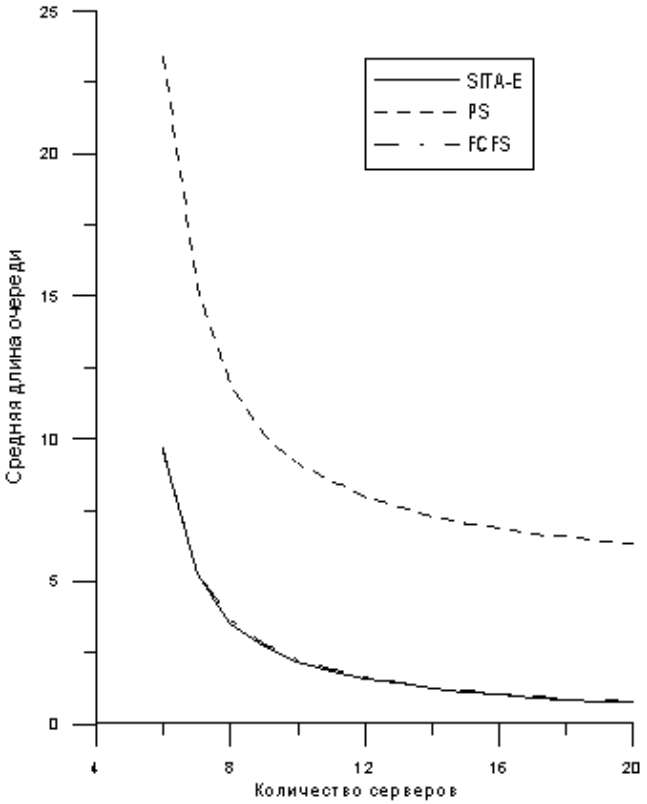

Рис. 3.7. Зависимость средней длины очереди от количества серверов. Распределение равномерное, среднее значение — 30, максимальное отклонение — 10

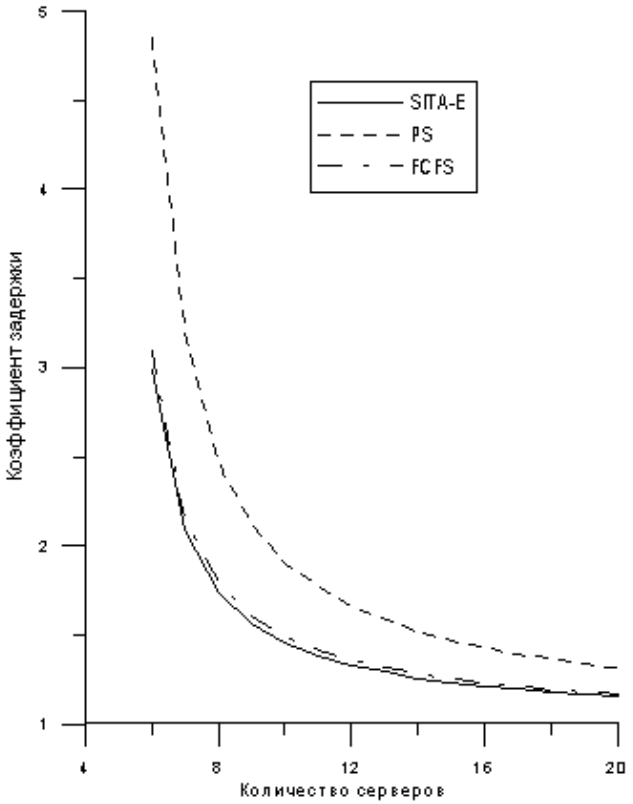

Рис. 3.8. Зависимость среднего коэффициента задержки от количества серверов. Распределение равномерное, среднее значение — 30, максимальное отклонение — 10

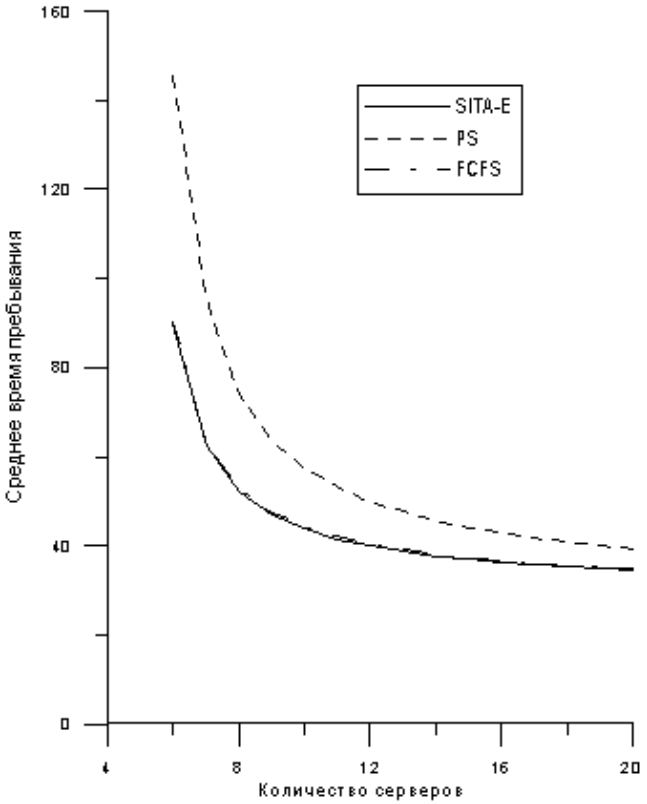

Рис. 3.9. Зависимость среднего времени пребывания заявки в системе от количества серверов. Распределение равномерное, среднее значение — 30, максимальное отклонение — 10

Экспоненциальное распределение определяется только одним параметром, поэтому для него нельзя менять дисперсию, не меняя среднего значения. По этой причине для экспоненциального распределения приведены только зависимости от количества серверов (рис. 3.10–3.12).

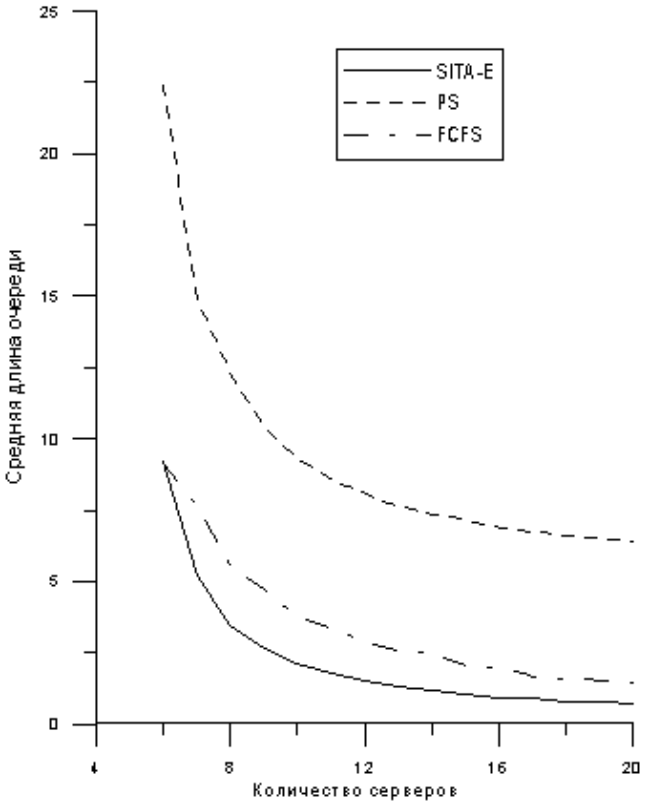

Рис. 3.10. Зависимость средней длины очереди от количества серверов. Распределение экспоненциальное, среднее значение — 30

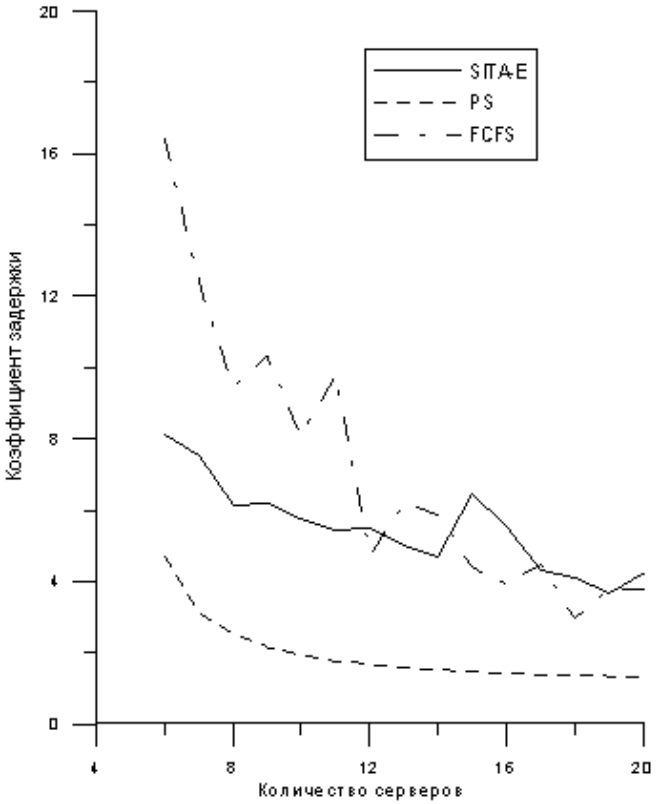

Рис. 3.11. Зависимость среднего коэффициента задержки от количества серверов. Распределение экспоненциальное, среднее значение — 30

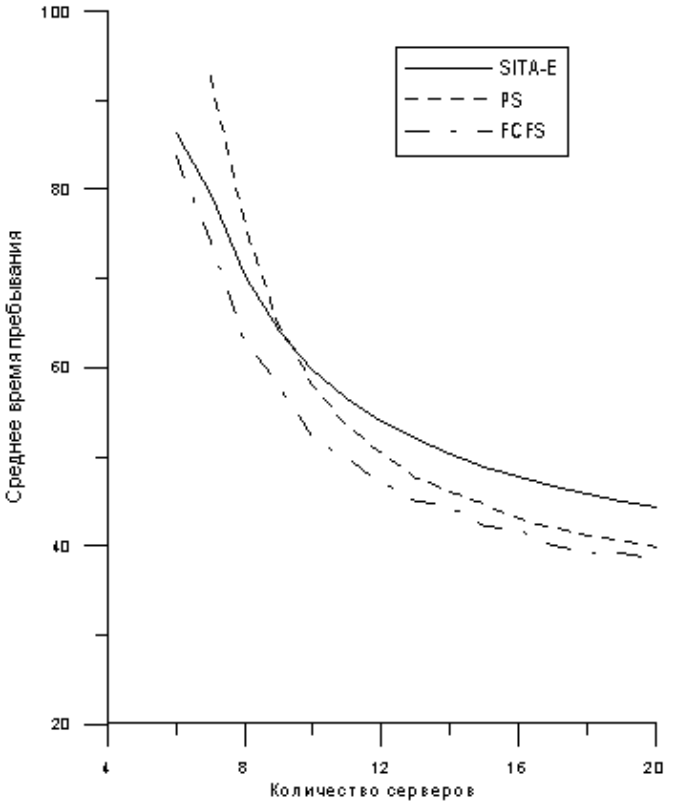

Рис. 3.12. Зависимость среднего времени пребывания заявки в системе от количества серверов. Распределение экспоненциальное, среднее значение — 30

Коэффициент вариации экспоненциального распределения имеет «граничное» значение — единицу, поэтому картина получается несколько «размытой». По критерию наименьшей средней длины очереди лучшая дисциплина маршрутизации запросов SITA-E, по критерию наименьшего среднего коэффициента задержки — PS, по критерию наименьшего времени пребывания — FCFS.

На рис. 3.13–3.15 изображены зависимости показателей системы от параметра α распределения Парето, который меняется от 1 до 2. Параметр s меняется при этом таким образом, чтобы математическое ожидание сохранялось равным 30. Напомним, что хотя дисперсия для распределения Парето и равна бесконечности при любом α, меньшие значения α интуитивно соответствуют «большей» дисперсии (более высокой скорости стремления к бесконечности несобственного интеграла, с помощью которого находится второй момент), а большие значения α — «меньшей» дисперсии.

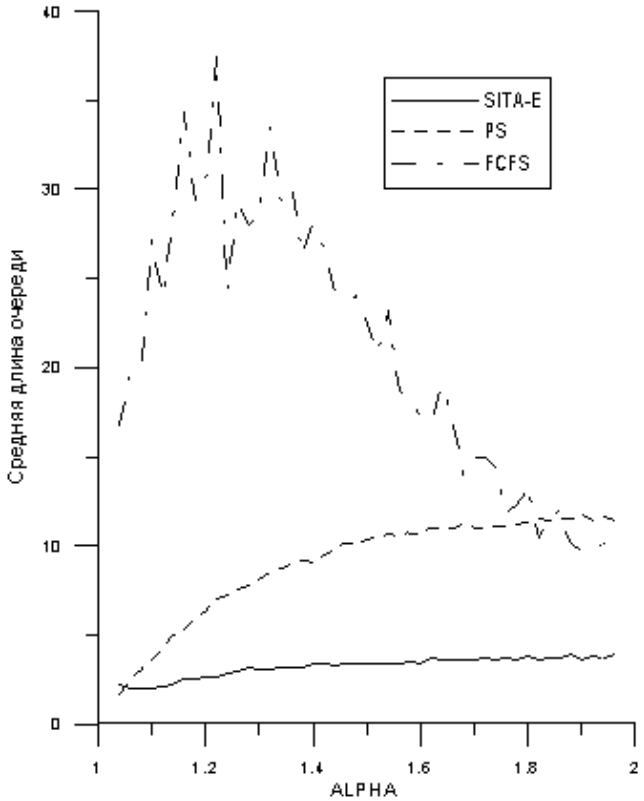

Рис. 3.13. Зависимость средней длины очереди от α. Распределение Парето, среднее значение — 30, 8 серверов

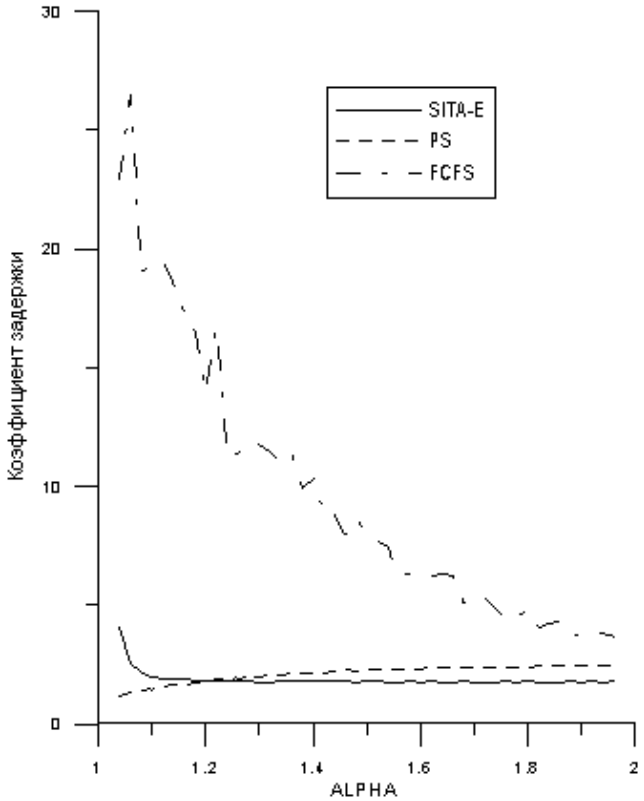

Рис. 3.14. Зависимость среднего коэффициента задержки от α.
Распределение Парето, среднее значение — 30, 8 серверов

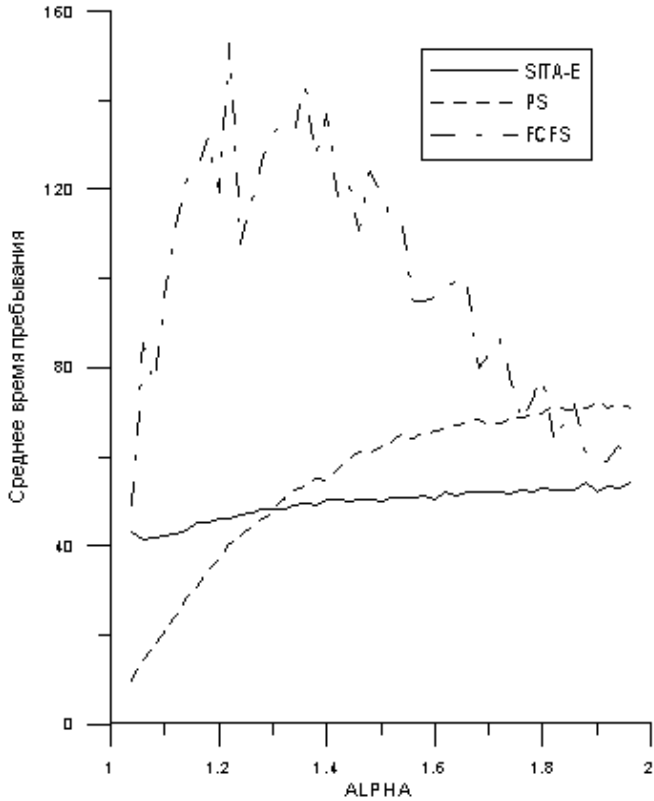

Рис. 3.15. Зависимость среднего времени пребывания заявки в системе от α. Распределение Парето, среднее значение — 30, 8 серверов

Из графиков видно, что при распределении Парето дисциплина PS ведет себя значительно лучше FCFS. Лишь при больших значениях α (близких к двум) их показатели несколько выравниваются. Взаимоотношения же между PS и SITA-E несколько сложнее. Мы видим, что практически всегда лучше ведет себя SITA-E. Но при значениях α, меньших некоторой величины (для каждого из трех

критериев она своя), наилучшие показатели имеет PS. Видимо, с увеличением дисперсии того повышения эффективности дисциплины FCFS, которое дает ей алгоритм SITA-E, становится недостаточно, чтобы перекрыть «природное» преимущество при больших дисперсиях PS над FCFS.

На рис. 3.16–3.18 изображены зависимости показателей системы от количества серверов при $s = 10$, $\alpha = 1.5$. При этом значении α преимущество SITA-E неоспоримо в любой точке.

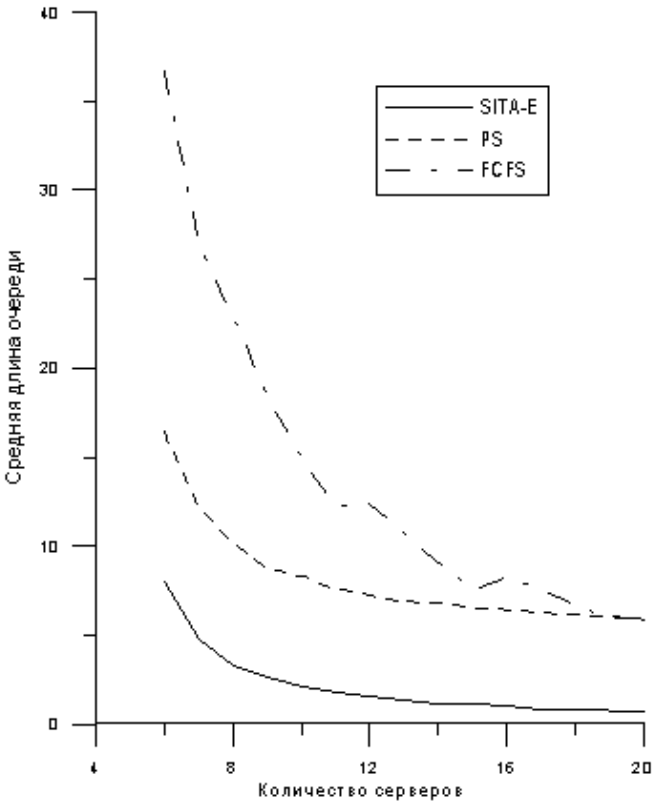

Рис. 3.16. Зависимость средней длины очереди от количества

серверов. Распределение Парето, $\alpha = 1{,}5$, среднее значение — 30

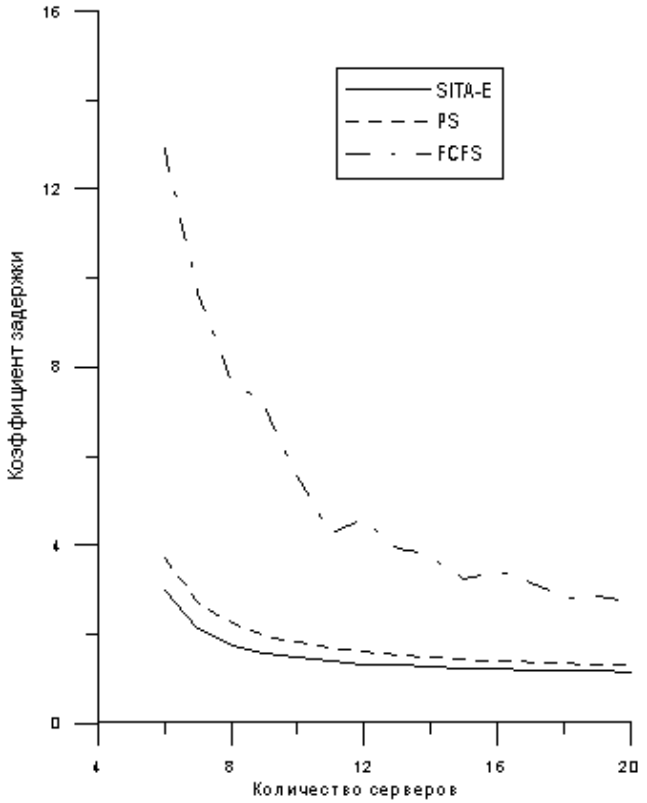

Рис. 3.17. Зависимость среднего коэффициента задержки от количества серверов. Распределение Парето, $\alpha = 1{,}5$, среднее значение — 30

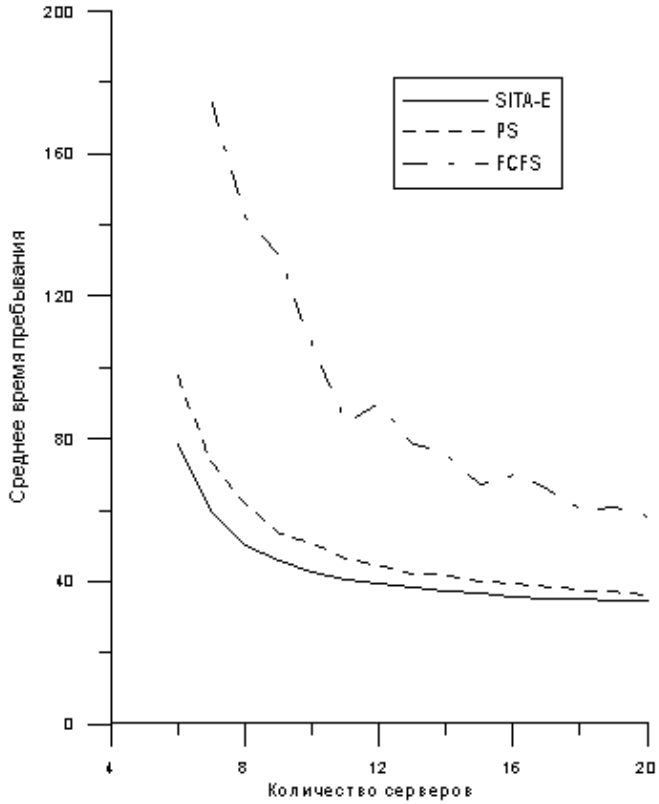

Рис. 3.18. Зависимость среднего времени пребывания заявки в системе от количества серверов. Распределение Парето, $\alpha = 1{,}5$, среднее значение — 30

Задания для самостоятельной работы

При всей своей эффективности алгоритм SITA-E имеет и существенный недостаток — применять его нельзя, если длину заявки невозможно определить заранее. В работе [72] один из авторов SITA-E предложил алгоритм TAGS (Task Assignment Based on Guessing Size), применимый для потоков заявок с неизвестной длиной. Длина

заявки определяется динамически. Сначала заявка поступает на обслуживание на первый сервер. Если s_1 единиц процессорного времени оказалось недостаточно для завершения обслуживания, процесс прерывается, заявка ставится в очередь ко второму серверу, где ее обслуживание начинается с начала и т. д. TAGS, таким образом, достигает приблизительно тех же целей, что и SITA-E, но без априорной информации о длине заявки. Для реализации этой дисциплины не требуется анализ HTTP-заголовка и наличие распределителя нагрузки.

Разработайте программу моделирования дисциплины TAGS и с ее помощью исследуйте следующие проблемы:

- Сколько необходимо серверов, чтобы значение среднего коэффициента задержки не превышало заданной величины?
- Каков оптимальный выбор последовательности s_i, $i = 1, 2, \ldots, N - 1$ (N — количество серверов), минимизирующей средний коэффициент задержки?

Глава 4. Моделирование Web-кластера: динамическая дисциплина и устаревание информации

4.1. Динамические дисциплины обслуживания

Алгоритмы диспетчеризации, применяемые распределителем нагрузки, можно разделить на *статические* и *динамические*. Главное их различие заключается в том, что в последнем случае распределитель осуществляет периодический мониторинг серверов, входящих в кластер, и выбирает для направления запроса наилучший из них по некоторому критерию. В статических алгоритмах, которые рассматривались в предыдущей главе, состояние серверов не отслеживается. Распределение длин заявок («тяжелый хвост» или что-либо другое) не предполагается при динамической дисциплине обслуживания известным заранее. Извне поступает поток заявок, а распределитель нагрузки исходит только из того, что происходит в системе в данный момент. Если какая-то длинная заявка долго занимает сервер («тяжелый хвост»!), распределитель это отследит, и на этот сервер другую заявку не направит. При статической дисциплине мы пытаемся предусмотреть модель загрузки заранее. Если прогноз распределения длин заявок оказался удачным – производительность повысится, если нет – снизится. При динамической дисциплине мы как бы констатируем, что заранее ничего предусмотреть не можем и действуем уже в процессе – по обстоятельствам. В этом случае шансы равномерно загрузить серверы возрастают, но повышаются и накладные расходы – на периодический мониторинг серверов и принятие решений.

Вопрос предпочтения одних алгоритмов другим требует детального

исследования и зависит от многих факторов, так как повышенная «интеллектуальность» динамических алгоритмов вызывает накладные расходы, связанные со сбором информации с серверов.

Наиболее естественными динамическими дисциплинами являются следующие:

Least Loaded (LL) — сервер выбирается по критерию наименьшей загрузки его ресурсов — ЦПУ, памяти, диска, количества очередей сообщений;

Least Connected (LC) — сервер выбирается по критерию наименьшего числа текущих открытых TCP/IP-соединений;

Fast Response (FR) — сервер выбирается по критерию наиболее быстрого ответа на тестовый запрос распределителя;

Weighted Round Robbin (WRR). Здесь, в отличие от статической RR, при циклическом переборе каждому серверу передается не один запрос, а подряд несколько в соответствии с весом сервера, пропорциональным, например, его текущей загрузке.

Универсальный подход к использованию динамических дисциплин реализован в IBM Net Dispatcher и описан в [83]. Авторы ввели понятие системы Load Metrics Index (LMI) и разделили динамические показатели качества работы системы на три класса:

Input — вычисляются локально диспетчером. Пример: количество соединений, установленных с сервером за последние t единиц времени;

Host — вычисляются на каждом сервере. Пример: загрузка сервера;

Forward — вычисляются путем сетевого взаимодействия диспетчера с

сервером. Эти характеристики доставляют специальные скрипты-агенты. Пример: подача HTTP-запроса "`GET /`" и измерение времени ответа.Далее по вектору LMI для каждого сервера вычисляются значения функции WCF (Weight Computation Function) и выбирается сервер с наибольшим значением. Эта функция имеет вид взвешенной суммы по элементам вектора LMI: $\Sigma a_i LMI_i$, где a_i – весовой коэффициент, назначаемый экспертом.

Программное обеспечение описанного диспетчера состоит из трех компонентов — Advisor, Manager и Executor. Advisor опрашивает серверы и собранную информацию передает компоненту Manager для вычисления WCF. Executor непосредственно маршрутизирует пакеты. Открытыми остаются вопросы динамического подстраивания весов для WCF, а также оптимального значения периодичности обновления LMI.

Выбор значения периодичности обновления весьма важен, и о нем следует сказать подробнее. Распределитель нагрузки[3] не может собирать информацию о серверах при поступлении каждого запроса, он делает это с некоторым периодом, который назовем *фазой*. Собранная информация используется при маршрутизации запросов в течение всей фазы, пока не будет обновлена. Но дело в том, что эта информация при интенсивном входном потоке может быстро устареть и оказаться не только бесполезной, но и вредной для диспетчера нагрузки. Известен так называемый *эффект стада* (herd effect). Этим термином названа пересылка «наилучшему» на данный момент

[3] Под распределителем нагрузки в данном случае понимается его программное обеспечение, т.е. Advisor, Manager и Executor в совокупности.

серверу подряд большое количество запросов, в результате чего он быстро становится наихудшим — задолго до следующего сбора информации о состоянии, когда можно будет что-то изменить. Методам предотвращения этой ситуации посвящен ряд работ, наиболее содержательными из которых являются [57], [82]. В них поставлена задача — учесть «возраст» информации о загрузке серверов. Основная идея такова: если информация свежая, следует использовать динамическую дисциплину, иначе — статическую (`FCFS-Random`). Рассмотрим возможные пути реализации, считая, что показателем загрузки сервера является текущая длина очереди запросов к нему, которую для i-го сервера обозначим L_i. Длину фазы обозначим T. В качестве критерия эффективности для всех алгоритмов будем использовать среднее время пребывания заявки в системе.

4.2. Алгоритмы учета «возраста» информации о загрузке серверов

Идея алгоритма, предложенного в [82], очень проста. Пусть при поступлении запроса из общего количества серверов N случайным образом выбирается некоторое подмножество k серверов и запрос пересылается на наименее загруженный из них. Эту дисциплину будем называть *алгоритм 1*. Нетрудно заметить, что при $k = 1$ она вырождается в полностью статическую с равновероятным выбором сервера, а при $k = N$ — в полностью динамическую. Интуитивно понятно также, что при возрастании длины фазы T наилучшее значение k должно уменьшаться, так как при длинной фазе вероятность недостоверности информации о загрузке серверов возрастает, отсюда и возрастающий «уклон» в сторону статичности

дисциплины маршрутизации. Результаты вычислительных экспериментов, приведенные в конце главы, подтверждают этот факт.

Более интересные алгоритмы предложены в [57]. Прежде всего заметим, что ожидаемое число запросов в течение фазы равно λT. Цель *алгоритма 2* — определить такие маршрутные вероятности, которые равномерно загрузили бы серверы, то есть чтобы количество запросов, равное общему числу запросов в системе в начале фазы плюс количество запросов, прибывших в течение фазы, оказалось приблизительно поровну распределено между серверами. При этом принимается упрощающее предположение о том, что интенсивность выходных потоков для всех серверов одинакова. Его можно считать вполне корректным в случае высокой загрузки, когда вероятность того, что сервер пуст, довольно мала.

Определим маршрутные вероятности в течение фазы следующим образом [57]:

$$p_i = \frac{\frac{L_{\text{tot}} + \lambda T}{N} - L_i}{\lambda T}, \; i = 1 \ldots N, \qquad (4.1)$$

при условии, что $\forall i \left(\frac{L_{\text{tot}} + \lambda T}{N} \geq L_i \right)$

где L_{tot} — общее число запросов на всех серверах в начале фазы. Условие в формуле (4.1) означает, что количества запросов, поступивших за время фазы, должно хватить для того, чтобы равномерно загрузить все серверы. В этом случае нетрудно показать, что сумма всех N вероятностей равна единице. Но если это условие не выполняется, формулой (4.1) пользоваться нельзя. Поэтому модифицируем ее следующим образом. Без потери общности перенумеруем серверы по возрастанию их нагрузки L_i на момент

начала фазы, то есть $L_i \le L_j$ при $i < j$. Определим параметр M следующим образом:

$$M = \max_m \sum_{i=0}^{m-1} (L_{m-1} - L_i) < \lambda T.$$

где m = 1,.., N. По смыслу M — это максимальное число серверов, которые могут быть приблизительно равномерно загружены за счет запросов, поступивших в течение фазы. Теперь модифицируем маршрутные вероятности:

$$p_i = \begin{cases} \dfrac{\lambda T - \sum_{j=0}^{M-1} (L_{M-1} - L_j)}{L_{M-1} - L_i + \dfrac{M}{\lambda T}}, & i < M; \\ 0 \text{ в противном случае.} \end{cases} \qquad (4.2)$$

Более акцентированным выражением той же идеи является *алгоритм 3*. В нем также в момент начала фазы N серверов упорядочены по возрастанию количества активных соединений и сервер для поступающих запросов выбирается среди первых k случайным образом. В отличие от алгоритма 1, картина здесь полностью меняется: при $k = 1$ имеем полностью динамическую дисциплину, при $k = N$ — полностью статическую со случайным выбором сервера. Фаза делится на N интервалов, и на j-м интервале длиной T_j нагрузка делится между первыми j серверами. Таким образом, по мере старения информации осуществляется постепенный переход от полностью динамической дисциплины к полностью статической. Формулы для расчета T_j в [57] предложены следующие:

$$T_j = \frac{(j+1)(L_{j+1} - L_j)}{\lambda}, j = 0, \ldots, N-1. \qquad (4.3)$$

Смысл формулы (4.3) заключается в том, что новые запросы,

поступившие в течение j-го подпериода фазы, должны в совокупности с имеющимися приблизительно равномерно загрузить $j + 1$ серверов. Отсутствие в этой формуле величины T смущать не должно, так как если при некотором j окажется, что $\sum_{k=0}^{j} T_k > T$, можно принять $T_k = 0$ для всех $k > j$. Это означает, что $N - j$ наиболее загруженных в начале фазы серверов загружены настолько, что не нуждаются в новой нагрузке. Выражение для нахождения маршрутных вероятностей для алгоритма 3 очень простое [57]:

$$p_{ij} = \frac{1}{j+1}, \quad i \le j,$$

где p_{ij} — вероятность того, что запрос, поступивший в течение j-го подпериода фазы, будет передан на обслуживание i-му серверу.

4.3. Программная реализация

Как и в предыдущей главе, будем рассматривать три класса: `Клиент`, `Сервер` и `Распределитель нагрузки`. Особенности реализации описаны в комментариях к программному коду. Входной поток предполагается пуассоновским, распределение длин заявок — экспоненциальным.

Листинг 4.1. Реализация алгоритма 1

```
#include<cstdio>
#include<cstdlib>
#include<ctime>
#include<cmath>
using namespace std;
#include "List.h"
```

```
#define RATIO 10  /*коэффициент масштабирования
времени*/
int entered+0;    //счетчик поступлений запросов
int completed=0;   //счетчик обслуженных запросов
float soj_ave=0;  /*хранение среднего времени
пребывания заявки в системе*/
long int total;  /*счетчик тактов модельного
времени*/

/*Метод формирует случайную выборку размером k из
A чисел от 1 до A и //записывает ее в массив mas.
Последовательно выбирает по одному элементу,
который тут же исключается из общей совокупности*/
void get_subset(int A, int k, int *mas) /*размер
массива mas равен k */
{
 int *arr, i, length, l, j;
 arr=(int*)malloc(A*sizeof(int));   /*массив для
хранения текущей совокупности элементов, из
которой производится выборка*/
 for(i=0;i<A;i++) arr[i]=i;
//инициализация
 length=A;
//инициализация размера общей совокупности
 for(i=0;i<k;i++)
 {
  l=rand()%length;
//равновероятный выбор очередного элемента
  mas[i]=arr[l];  //запись в результирующий массив
```

```
  for(j=l;j<(length-1);j++)  /*исключение
выбранного элемента из общей совокупности*/
   arr[j]=arr[j+1];
  length--;  //декремент размера совокупности
 }
 free(arr);
 return;
}
/*Протокол класса Client, моделирующего запрос.
Объекты класса необходимы для хранения времени,
проведенного заявками в системе*/
class Client
{
 int id;  /*уникальный идентификатор заявки. Равен
текущему значению счетчика поступлений.*/
 int time;   /*текущее значение времени,
проведенного заявкой в системе*/
 public:
/*Сервер и распределитель напрямую работают с
запросами*/
 friend class Server;
 friend class Dispatcher;
 Client(int i);
};
//Конструктор
Client::Client(int i)
{
 id=i;
```

```
 time=0;
}

/*Протокол класса Dispatcher, моделирующего
распределитель нагрузки*/
class Dispatcher
{
 int M;           //количество серверов
 int subset_number;   //значение параметра k
 int T_period;        //продолжительность фазы
 float input_rate; //интенсивность входного потока
 Server **farm;     //массив указателей на серверы
 int to_arrival;   /*время, оставшееся до прибытия
следующей заявки*/
 int to_observe;   /*время, оставшееся до
следующего опроса серверов и снятия информации об
их загрузке*/
 int *mas;  /*массив значений длин очередей на
серверах, полученный в начальный момент текущей
фазы*/
 public:
  Dispatcher(int i, int j, int q, float b, float
c);
 ~Dispatcher();
 void Arrival();
 void Observe();
 void run();
};
```

```
/*Конструктор. Аргументы:
i – количество серверов; k – значение параметра k;
q – длина фазы;
b – интенсивность входного потока;
c – интенсивность обслуживания (величина, обратная
средней длине заявки)
В начальном состоянии все серверы свободны,
очередей нет, начинается первая фаза.*/
Dispatcher::Dispatcher(int i, int k, int q, float
b, float c)
{
 int p;
 M=i;
 subset_number=k;
 T_period=q*RATIO;
 input_rate=b;
 farm=new Server *[M];
 for(p=0;p<M;p++)
  farm[p]=new Server(p+1, c);
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival+1;
 mas=new int[M];
 for(p=0;p<M;p++)
  mas[p]=0;
 to_observe=T_period;
}
```

```
//деструктор. Возвращает память
Dispatcher::~Dispatcher()
{
 delete [] mas;
 for(int i=0;i<M;i++)  delete farm[i];
 delete [] farm;
}

/*Обновление информации о загрузке серверов.
Начало новой фазы*/
void Dispatcher::Observe()
{
 for(int k=0;k<M;k++)
  mas[k]=farm[k]->q_length;
 to_observe=T_period;
}

/*Прибытие новой заявки. Ее маршрутизация в
соответствии с алгоритмом*/
void Dispatcher::Arrival()
{
 int k, mi, z, p;
Client *cl;
int *choice;
/*Массив, в который будет занесена выборка k
номеров серверов*/
```

```
 choice=(int*)malloc(subset_number*sizeof(int));
/*Разыгрываем новое значение интервала между
поступлениями заявок*/
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival+1;
 entered++;
//инкремент счетчика поступлений
 cl=new Client(entered);  /*создаем для новой
заявки объект*/
 get_subset(M,subset_number, choice);  /*делаем
случайную выборку серверов ищем среди них сервер
с минимальной загрузкой*/
 mi=32000; /*инициализируем минимум очень большим
числом*/
 for(k=0;k<subset_number;k++)
 {
  p=mas[choice[k]];
  if (p<mi)
  {
   mi=p;
   z=choice[k];
  }
 }
 farm[z]->Arrival(cl);
//пересылаем ему поступившую заявку
}

//Диспетчер
```

```
void Dispatcher::run()
{
 int p, k, q;
 to_arrival--;
 if (to_arrival==0) Arrival();
//поступление новой заявки
 to_observe--;
 if (to_observe==0) Observe();
//окончание текущей фазы
//Вызов диспетчера для всех серверов
 for(k=0;k<M;k++)
  farm[k]->run();
}
```

Реализация алгоритма 2 несколько сложнее. У распределителя нагрузки появляется новое поле данных — массив маршрутных вероятностей, которые рассчитываются в методе `Observe`. Кроме того, в этом алгоритме серверы должны упорядочиваться по возрастанию нагрузки, поэтому для каждого сервера нужно хранить еще и его номер в упорядоченной последовательности. Приведем те фрагменты кода и методы, которые претерпевают изменения по сравнению с алгоритмом 1.

Листинг 4.2. Реализация алгоритма 2

```
class Dispatcher
{
 int M;
 int T_period;
 float input_rate;
```

```
 Server **farm;
 int to_arrival;
 int to_observe;
 int *mas;
 float *probs;   //массив маршрутных вероятностей
 int l_border;  //значение параметра M алгоритма 2
 public:
  Dispatcher(int i, int q, float b, float c);
 ~Dispatcher();
 void Arrival();
 void Observe();
/*Метод, возвращающий номер сервера, на который
следует переслать новую заявку*/
 int get_server();
 void run();
};
Dispatcher::Dispatcher(int i, int q, float b,
float c)
{
 int k;
 M=i;
 T_period=q*RATIO;
 input_rate=b;
 farm=new Server *[M];
 for(k=0;k<M;k++)
  farm[k]=new Server(k+1, c);
 to_arrival=(int)(get_exp(input_rate)*RATIO);
```

```
 if (to_arrival==0) to_arrival+1;
 mas=new int[M];
 for(k=0;k<M;k++)
  mas[k]=0;
 to_observe=T_period;
 l_border=M; /*первоначально заявки распределяются
между всеми серверами*/
 probs=new float[M];
 for(k=0;k<l_border; k++)
  probs[k]=1.0/l_border;   /*начальное
распределение – равновероятное*/
}

//деструктор. Возвращает память
Dispatcher::~Dispatcher()
{
 delete [] mas;
 delete [] probs;
 for(int i=0;i<M;i++)  delete farm[i];
 delete [] farm;
}

/*Не только снятие информации о нагрузке, но и
пересчет маршрутных вероятностей, и упорядочение
серверов по возрастанию нагрузки*/
void Dispatcher::Observe()
{
```

```
 float arr_t;
 int k, s, s_old, i, p;
 int ordering[M]; /*массив для хранения номеров
серверов в упорядоченной последовательности*/
 for(k=0;k<M;k++)
 {
  mas[k]=farm[k]->q_length;
  ordering[k]=k;
 }
 //Упорядочение массива mas по возрастанию.
 for(k=0;k<(M-1);k++)
  for(i=0;i<(M-k-1); i++)
  {
   if (mas[i]>mas[i+1])
   {
    p=mas[i];
    mas[i]=mas[i+1];
    mas[i+1]=p;
    p=ordering[i];
    ordering[i]=ordering[i+1];
    ordering[i+1]=p;
   }
  }
 to_observe=T_period;
 arr_t=input_rate*T_period/RATIO;
//Расчет значения поля данных l_border
 for(k=1;k<+M;k++)
```

```
 {
  s=0;
  for(i=0;i<k;i++)
   s=s+mas[k-1]-mas[i];
  if (s>=arr_t) break;
  s_old=s;
 }
 l_border=k-1;
/*Расчет маршрутных вероятностей для каждого
сервера по формуле (4.2)*/
 for(i=0;i<l_border;i++)
  probs[ordering[i]]=(mas[l_border-1]-
mas[i]+(float)(arr_t-s_old)/l_border)/arr_t;
 for(i=l_border;i<M;i++)
  probs[ordering[i]]=0;
}

void Dispatcher::Arrival()
{
 int k, mi, z, p; Client *cl;
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival+1;
 entered++;
 cl=new Client(entered);
 z=get_server();   /*выбор сервера для поступившей
заявки*/
 farm[z]->Arrival(cl);
```

```
}

/*Сервер выбирается в соответствии с дискретным
распределением, заданным массивом probs*/
int Dispatcher::get_server()
{
  int r_num, i; float s, right;
  r_num=rand();
  right=((float)r_num)/32768;
  s=probs[0];
  for(i=0;i<M;i++)
  {
   if (right<=s) return(i);
   s=s+probs[i+1];
  }
}
```

При реализации алгоритма 3 класс `Распределитель` вновь претерпевает изменения. Массив маршрутных вероятностей уже не нужен, зато массив `ordering` теперь нужно сделать полем данных класса, так как он необходим методу `get_server()`. Кроме того, появляется еще одно новое поле данных — массив значений точек разбиения фазы на подпериоды.

Листинг 4.3. Реализация алгоритма 3

```
class Dispatcher
{
 int M;
```

```
 int T_period;
 float input_rate;
 Server **farm;
 int to_arrival;
 int to_observe;
 int *mas;
 int *sub_int;    //массив точек разбиения фазы
 int *ordering;   /*массив номеров серверов в
упорядоченной последовательности*/
 public:
  Dispatcher(int i, int q, float b, float c);
 ~Dispatcher();
 void Arrival();
 void Observe();
 int get_server();
 void run();
};
Dispatcher::Dispatcher(int i, int q, float b,
float c)
{
 int k;
 M=i;
 T_period=q*RATIO;
 input_rate=b;
 farm=new Server *[M];
 for(k=0;k<M;k++)
  farm[k]=new Server(k+1, c);
```

```
 to_arrival=(int)(get_exp(input_rate)*RATIO);
 if (to_arrival==0) to_arrival+1;
 mas=new int[M];
 for(k=0;k<M;k++)
  mas[k]=0;
 to_observe=T_period;
 sub_int=new int[M-1];
 ordering=new int[M];
 /*Первоначально Tj + 0, j < N, TN + T, то есть
заявки распределяются между всеми серверами*/
 for(k=0;k<(M-1); k++)
  sub_int[k]=0;
 for(k=0;k<M;k++)
  ordering[k]=k;
}

//деструктор. Возвращает память
Dispatcher::~Dispatcher()
{
 delete [] mas;
 delete [] sub_int;
 delete [] ordering;
 for(int i=0;i<M;i++)  delete farm[i];
 delete [] farm;
}

void Dispatcher::Observe()
```

```
{
 float tj;
 int k, s, ty, i, p;
 for(k=0;k<M;k++)
 {
  mas[k]=farm[k]->q_length;
  ordering[k]=k;
 }
 for(k=0;k<(M-1);k++)
  for(i=0;i<(M-k-1); i++)
  {
   if (mas[i]>mas[i+1])
   {
    p=mas[i];
    mas[i]=mas[i+1];
    mas[i+1]=p;
    p=ordering[i];
    ordering[i]=ordering[i+1];
    ordering[i+1]=p;
   }
  }
 to_observe=T_period;
//расчет точек деления фазы
 s=0;
 for(k=0;k<(M-1);k++)
 {
```

```
  tj=(float)((k+1)*(mas[k+1]-
mas[k])*RATIO)/input_rate;
  ty=(int)tj;
  sub_int[k]=s+ty;
  s=s+ty;
 }
}

int Dispatcher::get_server()
{
  int i, k;
 /*Вычисление номера подпериода, в котором мы
находимся в текущий момент времени*/
  for(i=0;i<(M-1);i++)
  {
   if (T_period-to_observe<sub_int[i]) break;
  }
/*Запрос должен быть распределен между первыми i+1
серверами упорядоченной последовательности*/
  k=rand()%(i+1);
  return(ordering[k]);   /*воспользовавшись
массивом ordering, получаем фактический номер
сервера*/
}
```

4.4. Анализ результатов

Для проведения вычислительных экспериментов были заданы следующие значения:

- интенсивность входного потока — 0,16;
- интенсивность обслуживания — 0,033;
- количество серверов — 8.

Исследовались зависимости среднего времени пребывания заявки от длины фазы, которая менялась от 15 (половина средней длины заявки) до 900 (30 средних длин заявок). Длительность моделирования — 100 тыс. тактов.

На рис. 4.1 показана зависимость для алгоритма 1 при различных значениях k. С ростом длины фазы оптимальное k уменьшается. При $k = 1$ зависимость становится константой, так как для дисциплины FCFS-Random размер фазы не имеет значения — информация о загрузке серверов все равно не используется. Наилучшее время пребывания для алгоритма 1 иллюстрирует рис. 4.2. При каждом значении T выбиралось наилучшее k, и значение времени пребывания для этого k отмечалось точкой на графике. Начиная с длины фазы около 270 ($9 \cdot 30$) наилучшим становится значение $k = 1$. Таким образом, при определенных обстоятельствах наличие и учет дополнительной информации только ухудшает производительность. Этот результат хорошо иллюстрирует известный тезис о том, что слишком много информации тоже иногда является бедствием.

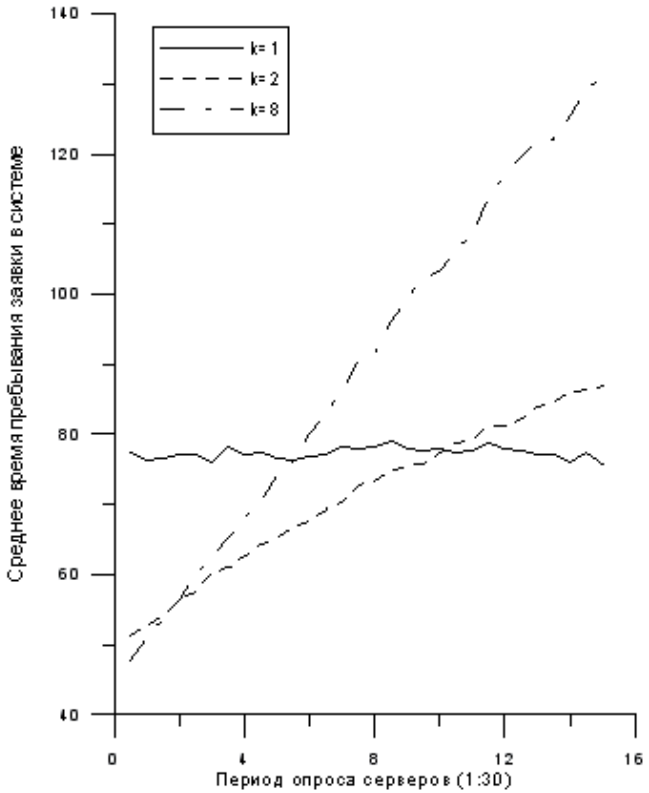

Рис. 4.1. Зависимость среднего времени пребывания заявки в системе от длины фазы для алгоритма 1 при различных k

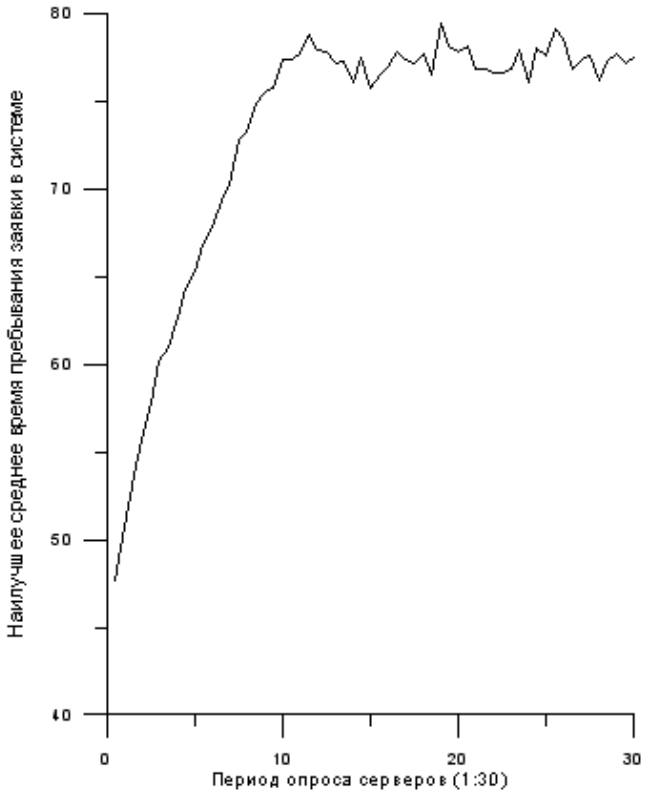

Рис. 4.2. Зависимость среднего времени пребывания заявки в системе от длины фазы для алгоритма 1 при наилучших k

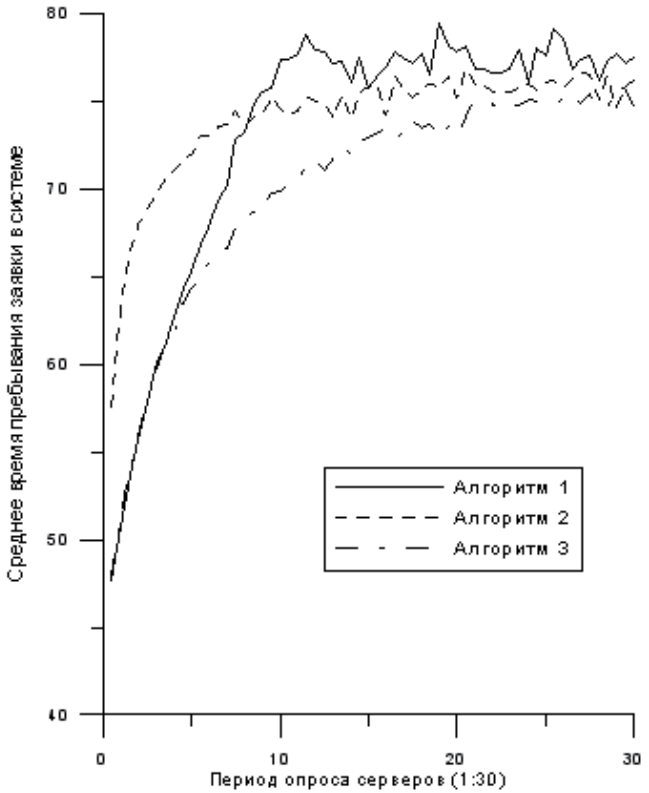

Рис. 4.3. Сравнительная характеристика трех алгоритмов

Сравнительный анализ всех трех алгоритмов дан на рис. 4.3. Мы видим, что наилучшие показатели имеет алгоритм 3, наихудшие — алгоритм 1. Показатели алгоритма 2 занимают промежуточное положение. При малых значениях T различий между тремя алгоритмами практически нет, да и при больших значениях T они не очень велики. Преимущество лучшего алгоритма измеряется не разами, а процентами (около 10%). Эти результаты согласуются с результатами, полученными в [57].

Задания для самостоятельной работы

1. Получите зависимости, аналогичные графикам на рис. 4.3, для длин заявок, подчиненных распределению Парето с сохранением среднего значения. Примите во внимание то, что в этом случае появляется еще одна зависимость — от параметра α .
2. Смоделируйте случай, когда длина фазы является не константой, а случайной величиной, подчиненной некоторому закону распределения. Рассмотрите экспоненциальное распределение и распределение Парето. Сравните результаты.

Глава 5. Моделирование гипертекстового представления результатов запросов к интернет-серверу баз данных

5.1. Описание системы

Задача, которую мы рассмотрим в этой главе, несколько отличается от ранее рассмотренных, так как не относится ни к теории очередей в ее привычном смысле, ни к сетевому планированию. В ней отсутствуют понятия очереди на обслуживание и средней длины заявки. Тем не менее, эта задача также может быть исследована как вероятностными методами, так и имитационным моделированием. Впервые она была поставлена в работе [34], следуя которой, мы кратко опишем ее суть.

Публикация баз данных в Интернете и организация удаленного доступа к ним через веб-интерфейс является в настоящее время весьма популярной областью исследований в мире информационных технологий. Каждый год издаются и переиздаются книги [27], [42], предлагаются новые архитектурные решения и средства разработки, среди которых квалифицированный программист может выбрать наиболее подходящие для реализации конкретного проекта. Вместе с тем в процессе разработки программного обеспечения для интернет-сервера баз данных возникают новые и довольно содержательные задачи аналитического характера, от правильности решения которых в значительной степени зависят производительность работы сервера и корректность результатов запросов и, как следствие, успех или неудача проекта.

Рассмотрим базу данных, основной информационной единицей которой является некоторый объект сложной структуры с уникальным идентификационным номером (или *ключом*). Каждому такому объекту

на сервере сопоставлен подкаталог, имя которого совпадает со значением ключа. В этом подкаталоге размещены файлы, хранящие в совокупности полное описание объекта, в их числе могут быть и графические стандартных форматов. С помощью специальных форм и программ файлы в подкаталогах, а также сопроводительные индексные файлы могут добавляться, модифицироваться и удаляться администратором путем веб-доступа. Типичный сервис заключается в следующем. Клиент заполняет некоторую поисковую форму и формирует запрос: перечислить объекты, обладающие совокупностью определенных свойств. Программное обеспечение на стороне сервера (назовем его частью *A*) производит поиск в базе данных и генерирует список гипертекстовых ссылок на такие объекты (если они существуют) по типу обычных поисковых серверов, а также записывает поступивший запрос в файл регистрации для сбора статистики посещений. Если клиент выбирает гипертекстовую ссылку из предложенного списка, еще одна программа на сервере (часть *B*) производит выборку данных из файлов подкаталога запрошенного объекта и формирует на их основе результирующий .html-файл (или несколько файлов, связанных ссылками) с полным отчетом о состоянии объекта, который клиент затем изучает. Основной вопрос заключается в следующем: при условии, что информация об объекте должна быть актуальной, какую ссылку следует генерировать части *A* на каждый из найденных объектов — статическую или динамическую?

Определим для объекта базы данных событие *W*: при очередном клиентском запросе объект попал в список ссылок и клиент выбрал среди прочих эту ссылку. Предположим, что отчет не только показан

в веб-браузере, но и записан на диск в подкаталог объекта в виде .htm-файла с некоторым именем (например, result.htm). Тогда при следующем попадании объекта в перечень, удовлетворяющий запросу клиента, нет необходимости в динамической ссылке и запуске программы из части *B*, если от момента последнего события *W* информация об объекте не обновлялась, — достаточно сделать ссылку на уже существующий файл. Однако, если информация была обновлена, для получения достоверного отчета необходима динамическая ссылка.

К сожалению, на этапе обновления данных не всегда возможно обновить и отчет. Если путем веб-доступа обновляются текстовые поля, программа, производящая обновление, может обновить и файл отчета. Но как быть, если администратор копирует в подкаталог объекта новый графический файл или файл апплета на языке Java и появление этих файлов должно учитываться при очередной генерации отчета? В этом случае наличие или отсутствие обновления данных в подкаталоге объекта необходимо выяснять на этапе *A*. Это можно сделать с помощью простого алгоритма:

1. Для каждого файла из подкаталога объекта определяется время его последней модификации (например, в UNIX это делается с помощью системного вызова `fstat` [46]).
2. Если самым «молодым» по этому показателю оказался файл отчета, в списке найденных по запросу объектов генерируется статическая ссылка на этот объект, в противном случае — динамическая ссылка на CGI-программу с передачей параметра, равного значению уникального ключа объекта.

Описанную стратегию назовем стратегией *M* (mixed), а безусловную

генерацию динамических ссылок — стратегией *D* (dynamic). На первый взгляд может показаться, что стратегия *M* в любом случае будет предпочтительнее стратегии *D*, однако, на самом деле это не так. Перечислим факторы, влияющие на этот выбор:

- применение стратегии *M* требует дополнительных расходов ресурса сервера, связанных с опросом всех файлов объекта, применением к ним системного вызова сбора информации и вычислением максимума или минимума времени последней модификации. В стратегии *D* эти расходы отсутствуют, однако считывание информации из всех файлов и генерация отчета, естественно, занимают значительно больше времени. В дальнейшем только эти времена будут учитываться при сравнительном анализе нагрузки на сервер;
- соотношение интенсивности потока обновлений и потока запросов к базе данных. Ясно, что, если они примерно одинаковы, стратегию *M* использовать нецелесообразно — большинство ссылок все равно придется делать динамическими и расходы на поиски самого «молодого» файла окажутся напрасными;
- количество ссылок в ответе сервера, которые открывает щелчком мыши клиент. Чем оно меньше, тем менее выгодно использование стратегии *M*. Если рассмотреть крайний случай, когда клиент вообще не открывает ссылки, а довольствуется только знанием их количества, применение *M* однозначно невыгодно.

Перечисленные факторы показывают, что ответ на вопрос об оптимальном выборе стратегии совсем не прост и требует разработки

модели, учитывающей влияние каждого из этих факторов. В [34] была при некоторых упрощающих предположениях построена вероятностная модель, определяющая аналитическое условие выбора той или иной стратегии. Однако практическое значение этого условия весьма невелико, так как оно получено не в прямом виде, а в виде преобразований Лапласа, которые могут быть доведены до элементарной формулы только в случае экспоненциальности всех входящих в исходные данные распределений. Уже при замене хотя бы одного из них распределением Эрланга выполнение преобразований требует длительных и громоздких манипуляций с функциями комплексной переменной в тригонометрической записи, а при использовании более сложных распределений может помочь только обратное преобразование Лапласа численными методами [18]. Поэтому попробуем воспользоваться имитационной моделью.

5.2. Исходные данные

Перечислим основные параметры, которыми мы будем оперировать при построении модели:

- N — общее число объектов в базе данных;
- $T_г$ — время выборки данных об объекте из файлов и генерации отчета. Примем его постоянным и одинаковым для всех объектов, хотя для имитационной модели это ограничением не является;
- $T_п$ — время, затрачиваемое на определение файла в подкаталоге объекта с самым поздним временем последней модификации. Его также примем постоянным и одинаковым;
- $f_1(t)$ — плотность распределения интервала времени между

последовательными запросами к базе данных;

- $f_2(t)$ — плотность распределения интервала времени между последовательными обновлениями объекта. Будем считать ее одинаковой для всех объектов, хотя параметры конкретной функции $f_2(t)$ можно для каждого объекта задать свои. Кроме того, примем в качестве условия, что время, затрачиваемое на само обновление, пренебрежимо мало по сравнению с этим интервалом, и поэтому процедура обновления — просто точка на временной оси. Последовательность этих точек образует обычный процесс восстановления;
- d_k ($k = 0, 1, 2, ..., N$) — дискретная плотность распределения для количества объектов, попавших в результат клиентского запроса;
- r — количество гипертекстовых ссылок, которые открывает клиент ($r < N$). Здесь приняты во внимание следующие соображения. Как правило, количество открываемых ссылок не возрастает с увеличением объема выборки. После просмотра некоторого количества ссылок складывается определенное впечатление, и интерес иссякает, получено ли в общей сумме 100 или 1000 ссылок. Любой человек может проверить это на собственном опыте, работая с поисковыми серверами. Формализация данного пункта потребовала бы задания для каждого $k = 1, 2, 3...$ отдельной плотности распределения $p_i^{(k)}$ ($i = 0, ..., k$) числа открываемых ссылок, однако ясно, что получить столь детальную статистику вряд ли представляется возможным. В модели количество открываемых («кликнутых») ссылок описывается следующим выражением:

$$\text{clicked}(k) = \min\left(k, 2r\,\text{arctg}(k)/\pi\right), \qquad (5.1)$$

где k — общее число найденных по запросу объектов. Таким образом,

$$\lim_{k\to\infty} \text{clicked}(k) = r. \qquad (5.2)$$

При желании функцию (5.1) можно заменить любой другой функцией, удовлетворяющей (5.2), а можно вообще построить ее, основываясь на другой поведенческой модели клиента.

Дополнительно, сделаем еще два предположения об однородности базы данных:

- частота попадания в выборку по запросу для каждого объекта статистически одинакова;
- вероятность выбора клиентом любой гипертекстовой ссылки из выборки также одинакова для всех объектов, попавших в интервал.

Таблица 5.1. Характеристики *M*- и *D*-стратегий

Стратегия	**Статическая ссылка**		**Динамическая ссылка**	
	Выбрана	**Не выбрана**	**Выбрана**	**Не выбрана**
Dynamic (*D*)	Не бывает		$T_г$	0
Mixed (*M*)	$T_п$	$T_п$	$T_п + T_г$	$T_п$

В качестве критерия эффективности стратегий примем время, затрачиваемое сервером на работу с одним объектом, попавшим в

выборку по запросу клиента. Под затратами времени будем понимать работу только прикладного программного обеспечения базы данных, а не HTTP-сервера (табл. 5.1).

Тогда, со статистической точки зрения, значение критерия для стратегии D равно $T_D = T_г \cdot \text{clicked} / \text{found}$, а для стратегии M — $T_M = T_п + T_г \cdot \text{clicked\_d} / \text{found}$, где clicked — общее количество «кликов»; found — общее число объектов, найденных по запросам за весь период моделирования; clicked_d — количество «кликов» на динамические ссылки.

Частный случай аналитической формы условия предпочтения стратегии M для экспоненциальных распределений $f_1(t)$ и $f_2(t)$ имеет вид [34]

$$\frac{T_г}{T_п} > \frac{z(\mu_1 R + \mu_2)}{R^2 \mu_1};\ R = \frac{\sum_{k=1}^{N} d_k \,\text{clicked}(k)}{N}. \qquad (5.3)$$

Условие означает, что стратегия M является предпочтительной только в том случае, когда левая часть неравенства не меньше некоторого «порогового» значения, и эффективность применения стратегии M возрастает с увеличением отношения $T_г / T_п$. Следовательно, чем сложнее внутренняя структура объекта и дизайн html-отчета, тем больший эффект дает избирательная генерация статических ссылок. Кроме того, из формул видно, что при увеличении N и сохранении значений всех прочих параметров правая часть неравенства должна возрастать, то есть эффективность стратегии M по сравнению с D снижается. Позднее мы убедимся в этом и по результатам имитационных экспериментов.

5.3. Программная реализация

Для программной реализации модели разработаны два класса — `Запрос` (`Request`) и `Сервер` (`Server`). Первый из этих классов представлен количеством объектов, равным числу объектов в базе данных, второй — единственным объектом. Объект класса `Request` содержит информацию, относящуюся к одному объекту базы данных, — время последнего «клика», время последнего обновления и время до следующего обновления. Программа содержит также две дополнительные функции — `get_subset()` и `get_dk()`. Функция `get_subset()` случайным образом выбирает подмножество заданного объема из некоторого множества. Эта задача возникает в двух ситуациях — при выборе из всей базы данных тех объектов, которые попадают в ответ на запрос, и далее, при выборе из этих объектов тех, по которым пользователь «кликает» для просмотра отчета об объекте. Функция `get_dk()` рассчитывает дискретное распределение объема выборки из базы данных по запросу. В модели используется биномиальное распределение, которое задается формулой $d_k = C_N^k z^k (1-z)^{N-k}$, $k = 0, 1, \ldots, N$; $0 < z < 1$, математическое ожидание — Nz.

Время генерации отчета в десять раз превышает время проверки существующего отчета на актуальность, интенсивность потока запросов на обновление объекта в 10 000 раз меньше интенсивности потока поисковых клиентских запросов (это приблизительно соответствует ситуации, когда поисковый запрос к базе данных поступает раз в минуту, а каждый объект обновляется один раз в неделю).

Листинг 5.1. Файл classes5.h с описанием протоколов классов

```
#include<cstdlib>
long int req_cl=0L;  /*счетчик клиентских запросов
к базе данных*/
long int req_adm=0L; /*счетчик запросов на
обновления объектов базы данных*/
long int found=0;  /*счетчик объектов, найденных
по поисковым запросам клиентов*/
long int large=0;  /*счетчик «кликнутых»
динамических ссылок для M-стратегии*/
long int clicked=0;   /*счетчик всех «кликнутых»
ссылок для D-стратегии*/
/*Функция случайным образом выбирает из множества
arr размера A подмножество mas в количестве k
элементов*/
void get_subset(int A, int k, int *arr, int *mas)
{
 int i, length, l, j;
 length=A;
 for(i=0;i<k;i++)
 {
 /*Выбираем из массива arr очередной элемент и
записываем его в mas*/
  l=rand()%length;
  mas[i]=arr[l];
//Удаляем выбранный элемент из arr
  for(j=l;j<(length-1);j++)
   arr[j]=arr[j+1];
```

```
  length--; /*декремент текущего размера массива
arr*/
 }
 return;
}

/*Вычисление k-го члена биномиального
распределения с параметрами N и z.
Обратите внимание на последовательность
вычислений. Они организованы так, чтобы, во-
первых, избежать умножений и делений на одно и то
же число, во-вторых, до минимума снизить
вероятность  переполнения по ходу вычислений*/
double get_dk(int N, double z, int k)
{
 int i;
 double result=1.0;
 if ((N-k)>k)
 {
  for(i=1;i<=k;i++)
   result=result*(N-k+i)*z*(1-z)/i;
  for(i=1;i<=N-2*k;i++)
   result=result*(1-z);
 }
 else
 {
  for(i=1;i<=(N-k);i++)
   result=result*(k+i)*z*(1-z)/i;
```

```
  for(i=1;i<=2*k-N;i++)
   result=result*z;
 }
 return(result);
}v

class Record //объект базы данных
{
 const static double change_rate=0.00001;
//интенсивность обновлений объекта БД
 int id;   //уникальный номер объекта в БД
 int from_click; /*время, прошедшее с момента
последнего «клика» по ссылке на объект*/
 long int from_change; /*время, прошедшее от
момента последнего обновления объекта*/
 long int to_change; /*время до следующего
обновления объекта*/
 public:
 friend class Server;
 Record(int i, Server *h);
 void Change();
 void run();
};

//Конструктор. Аргументы: i – уникальный номер
объекта БД, h –указатель на объект класса Server*/
Record::Record(int i, Server *h)
{
```

```
 double a;
 id=i;
/*Разыгрываем время, прошедшее от момента
последнего «клика» */
 a=h->cl_mean;

from_click=(int)((double)(rand())*a/(RAND_MAX+1));
/*Разыгрываем моменты предшествующего и
последующего изменений объекта*/
 to_change=(long int)(get_exp(change_rate));
 if (to_change==0L) to_change=1L;
 from_change=(long int)(get_exp(change_rate));
 if (from_change==0L) from_change=1L;
}

//Изменение информации об объекте
void Record::Change()
{
 from_change=0L;
/*Разыгрываем новый интервал между изменениями
информации об объекте*/
 to_change=(long int)(get_exp(change_rate));
 if (to_change==0L) to_change=1L;
}

/*Диспетчер. Отслеживает момент изменения объекта
и вызывает метод Change()*/
void Record::run()
```

```
{
 if (to_change>0)
 {
  to_change--;
  from_change++;
 }
 if (to_change==0L)
 {
  req_adm++;   /*инкремент счетчика запросов на
обновление объекта*/
  Change();
 }
}

class Server
{
 const static double input_rate=0.1;
//интенсивность потока клиентских запросов к БД
const static int N=10000; /*количество объектов в
БД*/
const static double z=0.02; /*параметр
биномиального распределения*/
const static int  r=20; /*максимальное число
открываемых ссылок в результатах запроса*/
 int to_request; /*время, оставшееся до следующего
запроса к БД*/
 Record **rec; //массив указателей на объекты БД
```

```
 double *dk;  /*массив, хранящий распределение
объем ответа на поисковый запрос*/
 public:
 double cl_mean; /*вычисляемое поле данных –
среднее время между «кликами» гипертекстовой
ссылки на один объект БД*/
 const static double Tg=10; /*длительность
генерации отчета при открытии динамической
ссылки*/
 const static double Tp=1; /*длительность проверки
текущего отчета на актуальность*/
 Server(double b, int c);
 ~Server();
 double ClickMean();
 int get_k();
 void Request();
 void run();
};


/*Конструктор класса Server. Аргументы: b –
параметр биномиального распределения; c –
максимальное число открываемых ссылок в
результатах одного запроса*/
Server::Server(double b, int c)
{
 int i;
 z=b; r=c;
```

```
/*Разыгрываем время до поступления клиентского
запроса*/
 to_request=(int)(get_exp(input_rate));
 if (to_request==0) to_request=1;
//Выделяем память под поля данных - массивы
 rec=new Record *[N];
 dk=new double[N+1];
//Вычисляем биномиальное распределение
 for(i=0;i<=N;i++)
   dk[i]=get_dk(N,z,i);
 cl_mean=ClickMean();
 for(i=0;i<N;i++)   //заполняем базу данных
   rec[i]=new Record(i, this);
}

//деструктор. Возвращает память
Server::~Server()
{
 for(int i=0;i<N;i++) delete rec[i];
 delete [] rec;
 delete [] dk;
}

/*Вычисляем среднее время между «кликами» одного
объекта*/
double Server::ClickMean()
{
```

```
 double s, c, mi;
 int k;
 s=0;
 for(k=1;k<=N;k++)
 {
  c=2*r*atan(k)/M_PI;
  if (k<c) mi=k; else mi=c;
  s=s+dk[k]*mi;
 }
 s=((double)N)/(s*input_rate);
 return(s);
}

//ГСЧ для дискретного  распределения dk
int Server::get_k()
{
 double s, right; int i;
/*функция rand() согласно описанию генерирует
случайное целое число в диапазоне от 0 до 32767*/
 right=(double)(rand())/(RAND_MAX+1);
 s=0;
 for(i=0;i<=N;i++)
 {
  if (right<=s+dk[i]) return(i);
  s=s+dk[i];
 }
}
```

```
//Поступление клиентского запроса к БД
void Server::Request()
{
 int *mas, *arr, *click, k, i, mi;
 double c;
 k=get_k();  /*разыгрываем количество ссылок в
ответе сервера*/
 if (k==0) return;
 found=found+k;   /*увеличиваем счетчик найденных
объектов*/
 mas=(int*)malloc(k*sizeof(int));
 arr=(int*)malloc(N*sizeof(int));
 for(i=0;i<N;i++)  arr[i]=i;
/*Разыгрываем номера объектов, удовлетворивших
условиям клиентского запроса */
 get_subset(N,k,arr,mas);
 c=2*r*atan(k)/M_PI;
 if (k<c) mi=k; else mi=(int)c;
 clicked=clicked+mi; /*увеличиваем счетчик
«кликов»*/
 click=(int*)malloc(mi*sizeof(int));
//Разыгрываем номера «кликнутых» объектов
 if (mi==k) for(i=0;i<k;i++)  click[i]=mas[i];
 else get_subset(k,mi,mas,click);
/*В случае использования M-стратегии проверяем,
была ли «кликнутому» объекту назначена
```

```
динамическая ссылка. Если была – производим
инкремент счетчика*/
 for(i=0;i<mi;i++)
 {
  if (rec[click[i]]->from_change <= rec[click[i]]-
>from_click)
     large++;
  rec[click[i]]->from_click=0;
 }
 free(click); free(mas); free(arr);
}

//Диспетчер
void Server::run()
{
 //Вызов диспетчера для всех объектов в БД
 for(int i=0;i<N;i++)
  rec[i]->run();
 if (to_request>0)  to_request--;
 if (to_request==0)
 {
  req_cl++;
  Request();
 }
}
```

При реализации функции `main()` для этой задачи возникает интересный вопрос, связанный с выбором длительности

моделирования. Из естественных соображений понятно, что она должна быть достаточной для охвата изменениями по нескольку раз всех (или, по крайней мере, подавляющего большинства) объектов базы данных. Поскольку заранее эту длительность предсказать нельзя, следует задать длительность моделирования косвенным образом – пока значение некоторого глобального счетчика не достигнет определенной величины. В данной программе было выбрано следующее условие: значение счетчика запросов на изменение базы данных `req_adm` не превышает jN, где N – количество объектов базы данных, j – натуральное число (эксперименты проводились при $j = 3$).

5.4. Анализ результатов

На рис. 5.1 показаны зависимости эффективности стратегий (обозначим эту величину буквой E) от объема N базы данных. Как и было предсказано аналитическим путем, с ростом N показатели стратегий сравниваются, а при превышении N некоторого значения стратегия D становится лучше.

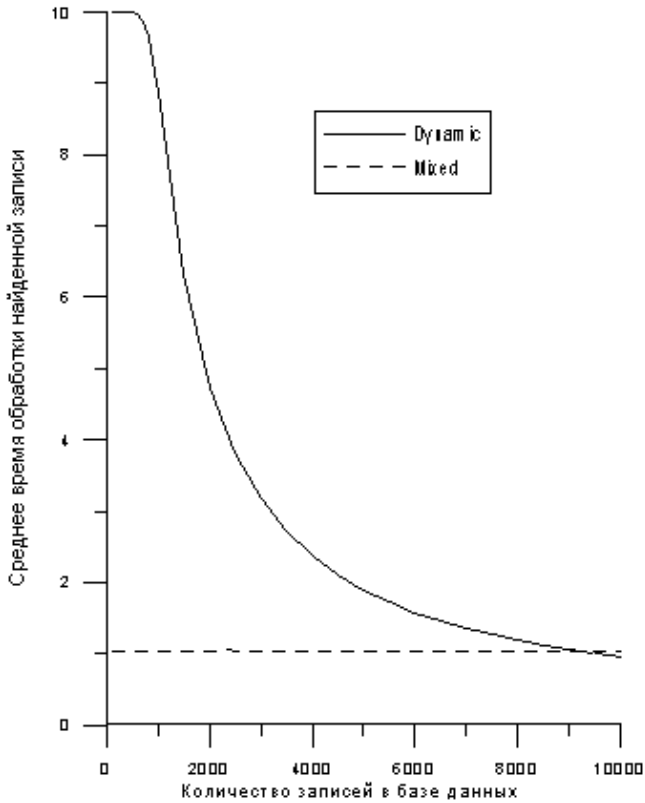

Рис. 5.1. Зависимости $E(N)$ для экспоненциальных распределений f_1 и f_2. $z = 0{,}02$, $r = 20$, $\mu_1 = 0{,}1$, $\mu_2 = 0{,}00001$, $T_г = 10$, $T_п = 1$

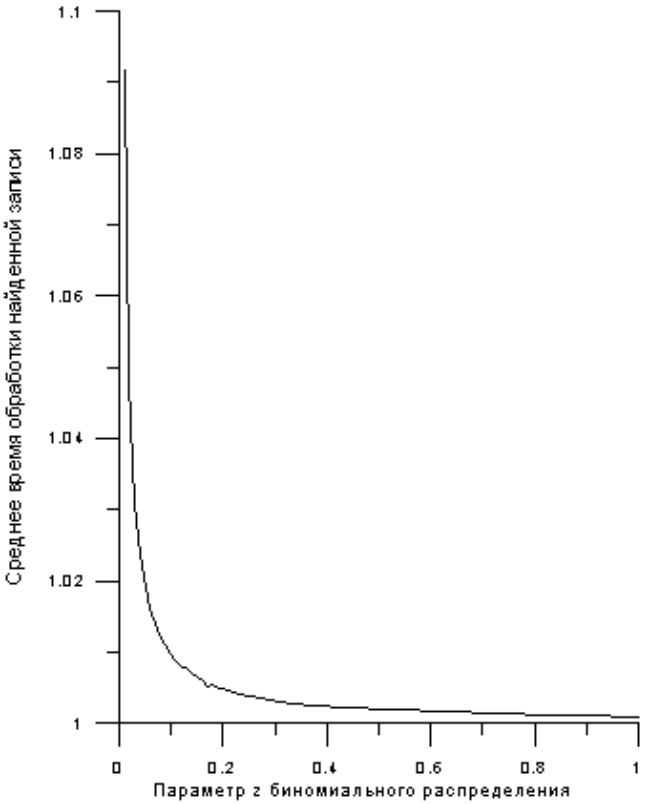

Рис. 5.2. Зависимость $E(z)$ в случае M-стратегии для экспоненциальных распределений f_1 и f_2. $z = 0{,}1$, $r = 90$, $\mu_1 = 0{,}1$, $\mu_2 = 0{,}00001$, $T_г = 10$, $T_п = 1$, $N = 1000$

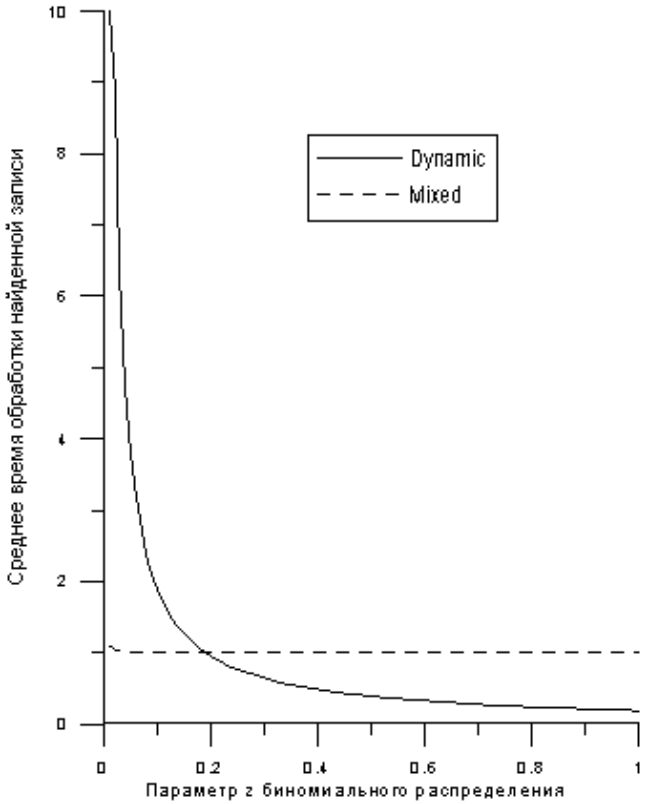

Рис. 5.3. Зависимости $E(z)$ для экспоненциальных распределений f_1 и f_2. $z = 0{,}1$, $r = 90$, $\mu_1 = 0{,}1$, $\mu_2 = 0{,}00001$, $T_г = 10$, $T_п = 1$, $N = 1000$

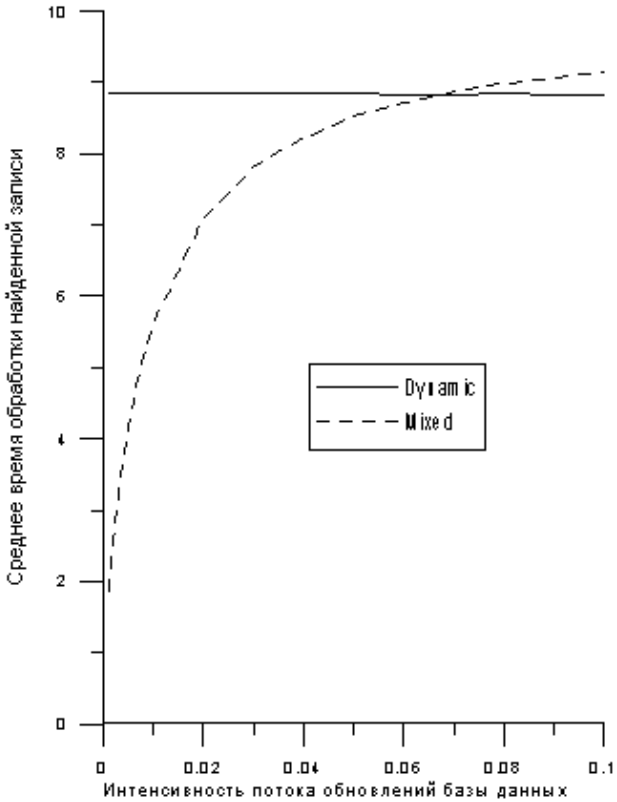

Рис. 5.4. Зависимости $E(\mu_2)$ для экспоненциальных распределений f_1 и f_2. $z = 0{,}1$, $r = 90$, $\mu_1 = 0{,}1$, $T_г = 10$, $T_п = 1$, $N = 1000$

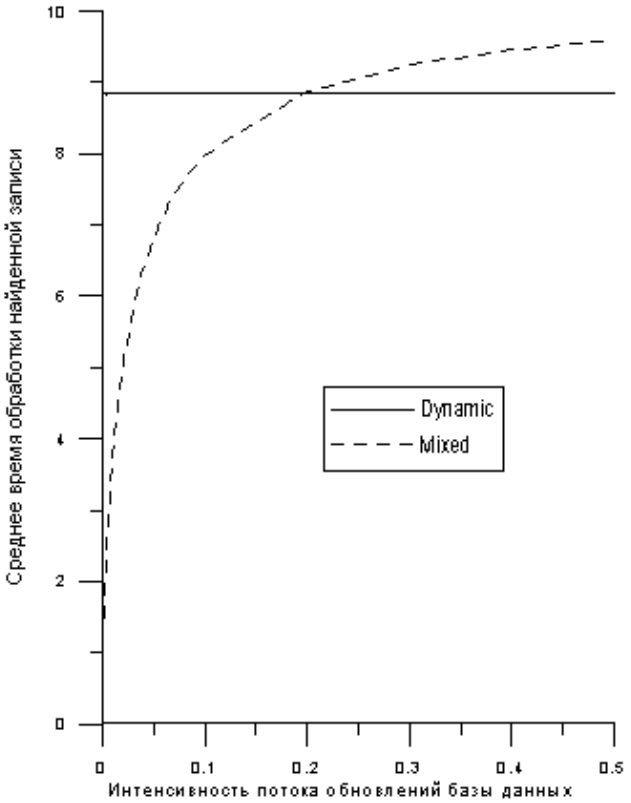

Рис. 5.5. Зависимости $E(\mu_2)$ для Парето-распределения f_1 и экспоненциального распределения f_2. $z = 0{,}1$, $r = 90$, $\mu_1 = 0{,}1$, $T_г = 10$, $T_п = 1$, $N = 1000$

На рис. 5.2 и 5.3 зависимость $E(z)$ для M-стратегии изображена, соответственно, сама по себе и вместе с зависимостью для стратегии D. Из графиков видно, насколько масштабирование может скрыть истинную картину поведения функции — на рис. 5.3 ее легко принять за константу. Графики на рис. 5.4 и 5.5 дают представление о том, как меняются эффективности стратегий в случае, когда интенсивности

потоков клиентских и административных запросов становятся сравнимыми. Заметим, что здесь картина меняется. Для *D*-стратегии безразлично, с какой интенсивностью происходят обновления, так как все ссылки все равно делаются динамическими, поэтому график для этой стратегии представляет собой константу. Эффективность же *M*-стратегии ухудшается, так как с ростом μ_2 все большее количество ссылок приходится делать динамическими.

В условии задачи приведено большое количество разнообразных параметров, поэтому дать для нее исчерпывающий анализ численных зависимостей нелегко. Некоторые задачи для дальнейшего исследования *M*- и *D*-стратегий приведены в упражнениях в конце главы.

5.5. Нечеткая стратегия

Рассмотренные стратегии генерации гипертекстовых ссылок обеспечивают доступ к информации об объекте с учетом всех сделанных обновлений. Рассмотрим теперь ситуацию, когда определенный процент устаревших отчетов допускается. В этом случае можно пренебречь проверкой отчета на новизну, а статическую ссылку разыгрывать каждый раз с некоторой вероятностью *p*. Такую стратегию генерации отчетов назовем *нечеткой*. Основной вопрос, который при этом возникает, — как связана вероятность *p* с вероятностью *g(p)* того, что при устаревшей статической ссылке произойдет событие *W*, то есть клиент получит недостоверный отчет?

При $p = 1$ (все ссылки — статические, нагрузка на сервер минимальна), $g(p)$ также в пределе равно 1, так как все отчеты в конце концов устареют. При $p = 0$ (все ссылки — динамические, нагрузка

максимальна) $g(p) = 0$. Таким образом, наилучшее значение нагрузки на сервер соответствует наихудшему g и наоборот. Отсюда есть основание полагать, что оптимальное значение критерия, учитывающего оба этих фактора, достигается при некотором значении $0 < p < 1$.

Статистически вероятность $g(p)$ можно оценить как отношение lie / clicked, где clicked — общее количество «кликов» за весь период моделирования, lie — количество «кликов», которые привели к выдаче устаревшего отчета. В [34] функция $g(p)$ была выведена аналитически и записана через преобразования Лапласа. Для случая экспоненциальных распределений она принимает следующий элементарный вид:

$$g(p) = \frac{p}{1 + \gamma(1 - p)}, \qquad (5.4)$$

где $\gamma = \frac{\mu_1 R}{\mu_2}$; R определяется формулой (5.3).

Для моделирования нечеткой стратегии в программу следует внести совсем немного изменений:

в число полей данных класса `Server` добавить еще один — `double prob_stat`, соответствующий вероятности генерации статической ссылки;

добавить еще один глобальный счетчик `lie` для подсчета количества устаревших отчетов, полученных пользователями;

изменить фрагмент кода в методе `Server::Request()`, а именно цикл, в котором производится обход всех объектов, попавших в результат запроса и выбранных пользователем для подробного просмотра.

Листинг 5.2. Модификация метода Server::Reauest() для нечеткой стратегии

```
for(i=0;i<mi;i++)   /*цикл обхода всех «кликнутых» ссылок*/
 {
   pr=(double)(rand())/(RAND_MAX+1);
//разыгрываем действительное число меньше 1
  if (pr>prob_stat) /*ссылка на объект – динамическая*/
  {
   rec[click[i]]->from_click=0;
   large++;
  }
  else  /*ссылка на объект – статическая...
  ...а отчет – устаревший*/
    if (rec[click[i]]->from_change <= rec[click[i]]->from_click)
       lie++;
 }
```

5.6. Анализ результатов по нечеткой стратегии

Моделирование производилось при значениях N 1000, $r = 90$, $z = 0{,}1$, $\mu_1 = 0{,}1$, $\mu_2 = 0{,}00001$. При этих значениях параметр γ из (5.4) равен 880. Было поставлено два комплекса экспериментов — при экспоненциальном распределении интервала между поступлениями клиентских запросов и при распределении Парето с параметрами $k = 1$, $\alpha = 1{,}11$ с сохранением средней длины интервала, равной 10. Длительность имитационного эксперимента задавалась неявно — до

тех пор, пока общее количество обновлений объектов базы данных не достигнет 50 000, то есть пока каждый из объектов не будет обновлен в среднем 50 раз. Реальная длительность эксперимента составила при этом около 5 млн тактов. Были построены зависимости $g(p)$ на интервале $p \in [0; 1]$. Результаты, полученные для экспоненциальных распределений, сравнивались с соответствующими значениями функции (5.4). Представление об этих результатах дает табл. 5.2. Так как при приближении к единице функция резко возрастает, сетка на оси абсцисс вблизи единицы сгущена, чтобы показать поведение функции подробнее.

Таблица 5.2. Результаты моделирования

P	***g*(*p*)**			***E*(*p*)**
	Имитационное моделирование, $f_1(t)$ — экспоненциальное	**Ф. (5.4), $f_1(t)$ — экспоненциальное**	**Имитационное моделирование, $f_1(t)$ — Парето**	
0,0	0,0	0,0	0,0	8,8
0,1	0,00012	0,00013	0,000078	7,96
0,2	0,000269	0,00028	0,000259	7,08
0,3	0,000467	0,00049	0,000307	6,19
0,4	0,000719	0,00076	0,000464	5,31
0,5	0,001094	0,00113	0,00072	4,42
0,6	0,001644	0,0017	0,00108	3,54
0,7	0,00256	0,00264	0,001615	2,65
0,8	0,004339	0,00452	0,002715	1,77

0,9	0,009664	0,01011	0,008987	0,88
0,95	0,020163	0,0211	0,013326	0,44
0,975	0,040901	0,04239	0,025602	0,22
0,99	0,096867	0,101	0,06398	0,09
1,0	1,0	1,0	1,0	0

Соответствие между теоретическими и экспериментальными данными вполне удовлетворительное — первые значащие цифры совпадают при всех p. Быстрый рост функции вблизи единицы объясняется большим значением γ. Среднее время обработки $E(p)$ найденного объекта одинаково для обоих распределений и убывает как линейная функция с тангенсом угла наклона –8,8.

Обратив функцию $g(p)$, можно ответить на вопрос: с какой вероятностью следует разыгрывать статическую ссылку, чтобы вероятность получения клиентом устаревшего отчета об объекте не превышала заданной величины?

Задания для самостоятельной работы

1. Проверьте, зависят ли эффективности M- и D- стратегий от каждого из параметров N и z в отдельности или только от их произведения Nz. Для этого изменяйте N и z таким образом, чтобы их произведение оставалось неизменным, и сравнивайте результаты «прогона» модели.
2. Тот же самый вопрос исследуйте для соотношения μ_1/μ_2.
3. Преобразуйте модель для случая, когда $T_г$ и $T_п$ являются случайными величинами, заданными своими законами распределения. Исследуйте влияние дисперсии этих величин на результаты.

4. Обобщите модель на случай, когда объем N базы данных является переменной величиной. Для этого рассмотрите два возможных способа:
   - введите в рассмотрение еще два потока административных запросов — на добавление и удаление объектов. Для количества добавляемых и удаляемых за один сеанс объектов задайте дискретную функцию распределения;
   - предположите, что все три действия — добавление, обновление и удаление объектов — производятся в одном запросе. Для этого необходимо дать вероятностное описание сеанса работы администратора — указать, над сколькими объектами он в течение сеанса произведет каждое из действий. Возможно, в этом случае сеанс уже некорректно рассматривать как точку на временной оси, а потребуется задать для него распределение длительности.
5. Проведите имитационные эксперименты по сравнению M- и D-стратегий для других дискретных распределений объема выборки по запросу. Так как из-за конечности объема базы данных распределения являются усеченными, константа T в каждом из случаев находится из условия нормировки $\sum_{k=1}^{N} d_k = 1$:
   - логарифмическое: $d_k = -\frac{TA^k}{k\ln(1-A)}$, $0 < A < 1$;
   - Пуассона: $d_k = T\frac{e^{-A}A^k}{k!}$, $A > 0$;
   - геометрическое: $d_k = Tp(1-p)^{k-1}$, $0 < A < 1$;
   - негативное биномиальное: $d_k = TC_{K-1}^{B-1}A^B(1-A)^{k-B}$, $0 < A < 1$, $1 \le B \le k$.

6. Для нечеткой стратегии постройте функцию $g_1(p)$, определяющую вероятность так называемой избыточной работы для объекта, то есть вероятность того, что при выборе клиентом гипертекстовой ссылки на объект текущий файл с отчетом об этом объекте окажется свежим, а ссылка — динамической.

Глава 6. Вычислительная система, управляемая потоком данных

6.1. Что такое управление потоком данных?

Как известно, архитектура первых ЭВМ основывалась на принципах, разработанных Дж. фон Нейманом. С тех пор прошло уже много лет, но и по сей день практически все применяемые в настоящее время ЭВМ являются фон-неймановскими. Напомним, что главные структурные составляющие ЭВМ при такой организации — это линейно адресуемая память, слова которой хранят команды и элементы данных, и процессор, выбирающий из памяти команды со своими операндами и записывающий в нее результаты. Каждая команда явно или неявно указывает адреса операндов, результата и следующей команды. Однако с развитием параллельных вычислений все чаще поднимался вопрос о пригодности фон-неймановской модели вычислительного процесса. Дело в том, что в системах параллельной обработки используется большое количество процессоров, и возникла необходимость в более развитой модели архитектуры и вычислений, объединяющей их в единое целое. Организация параллельных систем, основанная на модели последовательного потока команд, оказалась не вполне подходящей с точки зрения топологии соединений и программирования, и с конца 70-х — начала 80-х гг. прошлого века получила развитие другая концепция вычислительных процессов и архитектур ЭВМ — управление потоком данных (Dataflow machine). Кратко опишем, в чем она заключается.

1. Последовательность выполнения команд определяется не счетчиком адреса, который изменяет процессор, а готовностью

команды к выполнению.

2. В качестве операндов команды указываются не адреса ячеек памяти, а команды, результаты которых являются операндами данной команды.
3. Готовность команды к выполнению определяется данными: если все данные, необходимые для данной команды, вычислены, то команда исполняется. Готовую к выполнению команду, следуя [5], назовем *токен*
4. Таким образом, линейная последовательность команд естественным образом может быть распараллелена: параллельно могут выполняться все команды, аргументы которых вычислены.

Основным элементом потоковой организации вычислений является граф потока данных (Dataflow graph) — ориентированный граф, вершины (actors) которого соответствуют переменным и выражениям программы. Из вершины *A* в вершину *B* дуга (arc) ведет тогда и только тогда, когда *A* (producer) непосредственно используется при вычислении *B* (consumer). Для любой модели вычислений фундаментальными являются два механизма — вызова команд и вызова операндов. Механизм вызова команд задает порядок вызова одного вычисления другим и может быть двух типов [32]:

- вызов по готовности (data-driven) — управление сообщает о наличии аргумента, и вычисление выполняется, когда готовы все необходимые аргументы (то есть входные данные). В [24] этот принцип сформулирован в форме афоризма: *как только нечто может быть сделано, нужно это сделать*;

- вызов по запросу (demand-driven) — управление сообщает о запросе аргумента, и вычисление выполняется, когда один из генерируемых при этом выходных аргументов требуется в вызывающем вычислении.

Механизм вызова операндов задает порядок использования отдельного аргумента группой вычислений и также может быть двух типов:

- вызов операнда по значению (call by mean) — некоторый аргумент распределяется между всеми вычислениями, в которых он необходим, путем размножения на заданное число копий;
- вызов по ссылке (call by reference) — некоторый аргумент распределяется между всеми вычислениями, в которых он необходим, путем указания в них ссылки на соответствующую область памяти.

Одной из первых Dataflow-машин была машина, разработанная в Массачусетском технологическом институте (MIT) под руководством Дж. Дэнниса и названная по имени института — MIT [20], [59]. MIT представляет собой мультипроцессорную систему, являющуюся типичной схемой для статических потоковых машин и состоящую из четырех основных компонентов (рис. 6.1):

- командного блока (КБ), включающего множество блоков памяти (instruction cells), чье содержимое является в совокупности исполняемым графом потока данных. В простейшем случае блок памяти содержит код операции, порты входных данных и адреса назначения (рис. 6.2);

- селекторной сети (arbitration network), по которой готовые к выполнению команды (firable instructions) передаются в один из процессорных элементов;
- множества процессорных элементов (ПЭ), которые выполняют вычисления. ПЭ преобразует входной пакет в пакет результатов, подлежащих рассылке;
- распределительной сети (distribution network), по которой результаты расходятся по своим адресам назначения (destination addresses) в КБ.

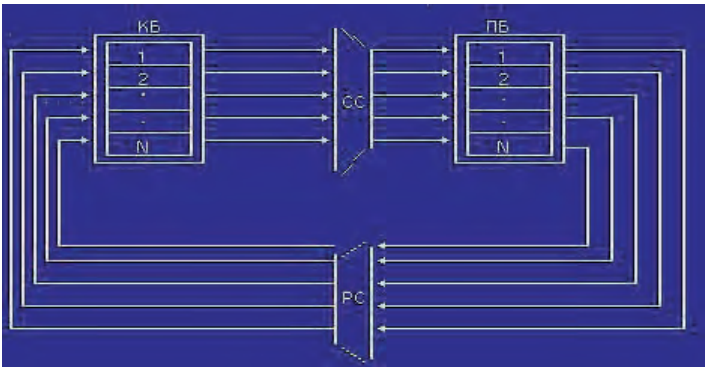

Рис. 6.1. Структура ЭВМ УПД (схема взята из источника [5])

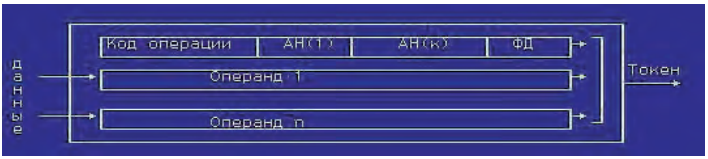

Рис. 6.2. Структура блока памяти

Отметим, что в работе [51] была выдвинута концепция организации параллельной ЭВМ на основе совмещения принципов потока данных и принципов фон Неймана. Указаны ситуации, когда выгоднее

использовать тот или иной принцип управления, требования, которым должны удовлетворять смешанная архитектура и программное обеспечение к ней. Предложены форматы команд, имеющие черты обеих составляющих новой машины. Поиск общего базиса с целью соединения их в единое целое продолжается в настоящее время.

Важная проблема, связанная с потоковыми машинами, заключается в организации обработки массивов, которая зачастую наталкивается на серьезные трудности. В [68] известный французский специалист Жан-Люк Годио сформулировал критерии, которым должен удовлетворять механизм обработки массивов:

- «Functionality of Structure Handling» — следует избегать операций, которые ослабляют действие принципа потока данных или уменьшают потенциальный параллелизм.
- «Performance» —адресация должна включать как можно меньше вычислений.
- «Hardware Overhead» — гибкая аппаратная поддержка новых операций.
- «Memory Management» — управление памятью, динамическое распределение, «сборка мусора» (garbage collection).

В [69] он же формулирует два фундаментальных принципа обработки массивов в ЭВМ с потоком операндов:

- Принцип косвенной адресации (indirect access approach). Согласно ему элементы массива считываются и записываются в память непосредственно операциями «чтение» и «запись». Массивы хранятся в специально выделяемой под них памяти.
- Принцип прямой адресации (direct access approach). Согласно

ему элементы массивов рассматриваются как обычные данные. Не требуется ни специальной памяти под массив, ни специальных команд работы с ними. Элемент массива содержит тег, который идентифицирует расположение элемента в массиве.

Методика и результаты имитационного моделирования ЭВМ УПД подробно изложены в работе [71], к настоящему времени уже несколько устаревшей. Аналитических же результатов получено очень мало по причине сложности построения вероятностных моделей для систем с нетрадиционной архитектурой. Сетевая модель одной из существующих ЭВМ УПД — манчестерской — предложена в [70], где получено решение в аналитическом виде. Команды предполагаются унарными (monadic) или бинарными (diadic). Модель, однако, имеет существенное ограничение — в ней принято постоянным значение коэффициента параллелизма, определяемого как число параллельно выполняющихся команд при неограниченном количестве процессоров (иначе, число активных процессоров). Это допущение снижает применимость модели, так как в большинстве реальных вычислительных процессов устойчивое значение коэффициента параллелизма на протяжении всего времени выполнения программы может не поддерживаться. Это ограничение снято в работах [5] и [41]. В [78] намечен алгебраический подход к моделированию итерационных алгоритмов, представленных потоком данных. Основным инструментом исследования являются степенные ряды. Особое внимание уделено связи систем потока данных с вычислениями на систолических массивах.

6.2. Описание системы

Опишем структуру исследуемой системы, следуя схеме, изображенной на рис. 6.1. Командный блок, содержащий M блоков памяти, соединен через селекторную сеть с процессорным блоком, состоящим из N процессоров. Блок памяти содержит код операции, порты входных данных, флаги данных, адреса назначения. Задание определяется программой, хранящейся в командной памяти, а токены формируются динамически во время работы программы. Результат выполнения команды, вычисляемый процессором, является операндом для i других команд и, соответственно, пересылается по распределительной сети в i блоков памяти. Для каждого i, $i = 0, 1, \ldots, h$, где h — максимальное число возможных копий результата (обычно в пределах десяти), задана вероятность q_i.

Если токен сформирован, а свободных процессоров нет, то он блокируется — не может быть передан в процессор и не может принимать новые операнды. Времена срабатывания процессора и блока памяти являются случайными величинами, заданными своими функциями распределения. В работе [5] они предполагаются распределенными экспоненциально с параметрами λ и μ соответственно, а все операции — унарными, то есть требующими только одного операнда.

Отдельно рассмотрим вопрос о том, может ли процессор переслать результат в заблокированный токен. Если предположить, что результат просто теряется, то программа будет выполняться некорректно, если же он сохраняется в специальном буфере, то описание системы значительно усложняется. Возникает также и теоретическая возможность того, что не принятые операнды будут

накапливаться бесконечно.

Отметим, что языки программирования, применяемые при управлении потоком данных, в той или иной степени используют принцип однократного присваивания (single assignment rule), который является основным при написании программ и не связан непосредственно с вычислениями. Различные варианты его формулировок приведены в работе [29], где довольно подробно рассматривается язык программирования высокого уровня LAU. Формулироваться этот принцип может, например, таким образом: *переменной можно присвоить значение в единственном операторе программы*. Использование принципа однократного присваивания и связанные с этим особенности языка позволяют исключить возможность возникновения описанной тупиковой ситуации при выполнении программы, то есть вероятность того, что результат выполнения команды не может быть направлен процессором в качестве операнда в другую команду, если соответствующий ей токен заблокирован по причине отсутствия свободного процессора.

Таким образом, рассмотрим описанную структуру системы с учетом следующего условия: результат выполнения команды может рассылаться только в незаблокированные токены.

Кроме того, предположим, что команды могут быть не только одно-, но и i-местными, $i = 1, 2, \ldots, R$, а r_i — заданная вероятность того, что случайно выбранная команда является i-местной, то есть требует для своего исполнения i вычисленных операндов. Аналитическая модель для частного случая $R = 2$ была предложена в [70]. Заметим, что вероятность r_i подразумевается вычисленной не как доля i-местных команд среди *всех* команд, записанных в программе, а как доля i-

местных команд из числа выполняющихся программой. Это означает, что при определении r_i важен не только факт появления в программе i-местной команды, но и то, входит ли эта команда в цикл и какое количество итераций проводится в этом цикле.

В работе [41] для экспоненциального распределения времен срабатывания процессора и блока памяти была построена аналитическая модель, соответствующая описанным условиям. К ее основному результату мы еще вернемся, а сейчас сосредоточим внимание на описании имитационной модели.

6.3. Программная реализация

Для реализации модели предлагается использовать три класса: `Server` — для представления процессорного блока, `Memory` — для представления командного блока и `Token` — для представления одного токена, хранящегося в командном блоке. При этом первые два класса представлены в программе единственными объектами, последний — конечным множеством объектов. Класс `Memory` является контейнерным для класса `Token`. Возникает вопрос: почему составляющая командного блока — токен — выделена в отдельный класс, а составляющая процессорного блока — один процессорный элемент — не выделена? Причина такого проектного решения заключается в объеме динамической информации, которую для поддержания своего состояния должны хранить эти сущности. Процессорный элемент либо выполняет вычисления, либо простаивает, поэтому для него требуется только одно изменяемое поле данных — время, оставшееся до окончания выполнения, которое равно –1 в случае простоя. Для хранения значений этого поля данных

нет необходимости создавать отдельный объект, достаточно завести массив из N элементов в качестве поля данных класса `Server`. Отдельный же элемент командного блока должен хранить еще и информацию о том, сколько вычисленных операндов ему осталось на данный момент получить для того, чтобы формирование токена в нем было завершено. Напрямую связываться с объектом `Server` — получать от него операнды и отсылать на выполнение токены — класс `Token` не может, обмен информацией полностью сосредоточен в классе `Memory`. Статистика ведется для двух показателей: среднего числа активных процессорных элементов, определяющего производительность системы, и среднего числа заблокированных токенов. Вероятности q_i рассчитываются в соответствии с биномиальным распределением, вероятности r_i задаются непосредственно, так как значение R хотя и является произвольным, по смыслу невелико.

Дальнейшие комментарии приведены непосредственно в коде классов.

Листинг 6.1. Описания классов

```
double que_ave=0;   /*переменная для расчета
среднего числа активных ПЭ*/
double load_ave=0;   /*переменная для расчета
среднего числа блокированных токенов*/
long int total; /*счетчик тактов модельного
времени*/
//Протокол класса Server
class Server
{
```

```
//Неизменяемые поля данных
const int N;  //количество процессорных элементов
const double lyam; //средняя производительность ПЭ
const double z;   /*параметр биномиального
распределения для числа копий результата*/
const int h;  /*максимальное число копий
результата вычислений*/
double *dk;  /*массив для хранения функции
дискретного распределения вероятностей для числа
копий результата*/
void *mem;     //указатель на объект Memory

//Изменяемые поля данных
int *to_served;  /*массив, в котором хранится
время до конца текущего  //вычисления для каждого
ПЭ*/
public:
Server(int a, double b, double c, int d);
//конструктор
~Server();
void Complete(int i); //i-й ПЭ завершил вычисление
int Receive();       //прибытие нового токена
void run();          //диспетчер
int Load();  /*вычисление текущего числа активных
ПЭ (немоделирующий метод)*/
void putMemory(void *e);
//установка связи с Memory (немоделирующий метод)
};
```

```
//Протокол класса Token
class Token
{
//Неизменяемые поля данных
int id;   //порядковый номер в командном блоке
double mu;    //средняя производительность
int arity;           //количество операндов
void *mem;     //указатель на объект Memory

//Изменяемые поля данных
int to_served; //время до завершения срабатывания
int to_set; /*сколько операндов осталось получить.
-1, если токен заблокирован*/
public:
friend class Memory; /*Memory напрямую работает с
Token*/
Token(int d, double a, int b, void *c);
//конструктор
void Complete();     //завершение срабатывания
void Receive();      //получение нового операнда
void run();          //диспетчер
};

//Протокол класса Memory
class Memory
{
```

```
//Неизменяемые поля данных
int M;                //число блоков памяти
int R;    //максимальная «местность» одной команды
double *dk; /*массив для хранения функции
дискретного распределения //вероятностей для числа
операндов в команде*/
Token **tok;  /*массив указателей на объекты
класса Token*/
void *S;        //указатель на объект Server
long int places;   /*суммарное количество мест для
операндов во всем командном блоке (вычисляемое
поле данных)*/
public:
Memory(int a, int b, double *c, double d);
//конструктор
~Memory();
void Send(int i);
//пересылка токена из i-го блока в процессор
void Receive(int i);  /*получение i-копий
результата вычислений из процессорного блока для
рассылки их блокам памяти*/
void run();           //диспетчер
long int Volume();   /*вычисление поля данных
places (немоделирующий метод)*/
void putServer(Server *s); /*установка связи с
объектом Server (немоделирующий метод)*/
int Queue();  /*вычисление текущего числа
заблокированных токенов (немоделирующий метод)*/
};
```

```
void Server::putMemory(void *e)
{
 mem=e;
}
//Конструктор. Аргументы:
//a – количество ПЭ
//b – производительность ПЭ
//c – параметр биномиального распределения
//d – максимальное число копий результата
Server::Server(int a, double b, double c, int d):
N(a),lyam(b),z(c),h(d)
{
 int i;
 to_served=(int*)malloc(N*sizeof(int));
 //Всем ПЭ задаем работу
 for(i=0;i<N;i++)
 {
  to_served[i]=(int)(get_exp(lyam));
  if (to_served[i]==0) to_served[i]=1;
 }
//Вычисление распределения числа копий
dk=(double*)malloc((h+1)*sizeof(double));
for(i=0;i<=h;i++)
dk[i]=get_dk(h,z,i);
}
```

```
//деструктор. Возвращает память
Server::~Server()
{
 free(dk);
 free(to_served);
}

//Вычисление текущего числа активных ПЭ
int Server::Load()
{
int k=0, i;
for(i=0;i<N;i++)
if (to_served[i]!=-1) k++;
return(k);
}

//i-й ПЭ завершил вычисления
void Server::Complete(int i)
{
double a,s; int j;
to_served[i]=-1;     //показываем, что он теперь
неактивен
//Розыгрыш числа копий
a=(double)(rand())/32768; //получаем случайное
число на отрезке [0;1]
s=0;
for(j=0;j<=h;j++)
```

```
 {
if (a<=s+dk[j]) break;
s=s+dk[j];
 }
//Если результат не уходит на внешние носители
(число копий не равно нулю), //пересылаем его в
командный блок
if (j>0) ((Memory*)mem)->Receive(j);
}

//Получение сформированного токена для выполнения
вычислений
//Возвращаемое значение: 1 – токен поставлен на
вычисление на один
//из имеющихся свободных ПЭ; 0 – свободных ПЭ нет
int Server::Receive()
{
int i;
for(i=0;i<N;i++)        //ищем первый свободный ПЭ
if (to_served[i]==-1)   /*найден, ставим токен на
обслуживание*/
  {
to_served[i]=(int)(get_exp(lyam));
if (to_served[i]==0) to_served[i]=1;
return(1);
  }
return(0);
}
```

```
//Метод-диспетчер
void Server::run()
{
int k, p, i;
for(i=0;i<N;i++)
 {
if (to_served[i]>0) to_served[i]--;
if (to_served[i]==0)
Complete(i);
   }
/*Каждые 10 единиц времени – сбор статистики по
среднему числу активных ПЭ*/
if ((total+1)%10==0)
 {
k=Load();
p=(total+1)/10;
load_ave=load_ave*(1-1.0/p)+((double)k)/p;
 }
}

//Конструктор класса Token. Аргументы:
//d – порядковый номер в командном блоке
//a – производительность
//b – количество операндов токена
//c – указатель на объект Memory
```

```
Token::Token(int d, double a, int b, void *c):
mu(a),arity(b),mem(c),id(d)
{
 to_served=-1;
/*Предполагаем, что первоначально каждому блоку
памяти до завершения комплектования токена не
хватает значения одного операнда*/
to_set=1;
}

//Токен готов к пересылке в процессор
void Token::Complete()
{
to_served=-1;
to_set=arity; /*до завершения формирования  –
столько операндов, какова «местность» токена*/
((Memory*)mem)->Send(id-1);  /*отправляем токен в
КБ для последующей пересылки в процессор*/
return;
}

//Получение очередного операнда
void Token::Receive()
{
if (to_set>0) to_set--; /*декремент числа
оставшихся операндов*/
if (to_set==0)        //токен сформирован
 {
```

```
to_served=(int)(get_exp(mu));
if (to_served==0) to_served=1;
 }
}

//Метод-диспетчер
void Token::run()
{
if (to_served>0)
 {
to_served--;
if (to_served==0)  Complete();
 }
else if (to_set==-1)
((Memory*)mem)->Send(id-1);  /*пробуем отправить
ранее заблокированный токен в надежде, что
появились свободные ПЭ*/
}

void Memory::putServer(Server *s)
{
S=s;
}
//Конструктор класса Memory. Аргументы:
//a – количество блоков памяти;
//R – максимальная «местность»[4] команд;
```

[4] Количество операндов

```
//c – распределение «местности» команд;
/*d – производительность токена. Это величина,
обратная среднему времени срабатывания командного
блока. Единица измерения - сформированных токенов
в единицу времени*/
Memory::Memory(int a, int b, double *c, double d)
{
double u, w; int j, i;
M=a; R=b;
dk=(double*)malloc(R*sizeof(double));
for(i=0;i<R;i++)
dk[i]=c[i];
tok=new Token *[M];
/*Создание объектов класса Token. Для каждого из
них разыгрывается «местность»*/
for(i=0;i<M;i++)
 {
w=(double)(rand())/32768; /*получаем случайное
число на отрезке [0;1]*/
u=0;
for(j=0;j<R;j++)
  {
if (w<=u+dk[j]) break;
u=u+dk[j];
} //j+1 – разыгранная «местность» очередного
токена. Четвертый аргумент //вызова конструктора –
указатель this, с помощью которого текущий объект
//(Memory) передает указатель на самого себя
```

```
tok[i]=new Token(i+1,  d, j+1, this);
  }
places=Volume();
}

//деструктор. Возвращает память
Memory::~Memory()
{
 for(int i=0;i<M;i++)  delete tok[i];
 delete [] tok;
 free(dk);
}

//Отправка токена в процессор
void Memory::Send(int i)
{
int k;
//Используя связь объектов, отправляем токен в
процессор
k=((Server*)S)->Receive();
if (k==0) tok[i]->to_set=-1;        //свободных ПЭ
нет, блокируемся
else tok[i]->to_set=tok[i]->arity;  //токен
благополучно «ушел», блок
                                    // памяти
готов к приему новых операндов
return;
}
```

```
//Получаем results копий результата от процессора
void Memory::Receive(int results)
{
int k, *pointer, *pointer1, *choice, i, j, q, l;
long int v;
k=0;
pointer=(int*)malloc(places*sizeof(int));
pointer1=(int*)malloc(places*sizeof(int));
choice=(int*)malloc(results*sizeof(int));
/*Подсчет количества мест, среди которых эти копии
можно разыграть*/
for(i=0;i<M;i++)
 {
q=tok[i]->to_set;
//Нет блокировки, нет обслуживания
if ((tok[i]->to_served==-1)&&(q>0))
  {
for(j=0;j<q;j++)
   {
pointer[k]=i;                          //k-е
свободное место – в i-м блоке
                                       //памяти
k++;
   }
  }
 }
```

```
if (k<results) l=k; else l=results;  /*всего
распределяем l операндов*/
for(i=0;i<k;i++) pointer1[i]=pointer[i];
/*Получаем в массиве choice выбранные l мест для
приема операндов. Текст метода get_subset приведен
в главе 5. Копия pointer1 требуется в связи с тем,
что в методе get_subset массив pointer
претерпевает изменения, а им далее еще нужно
воспользоваться в теле данного метода*/
get_subset(k,l,pointer1, choice);
for(i=0;i<l;i++)     //прием операндов
tok[pointer[choice[i]]]->Receive();
free(pointer); free(pointer1); free(choice);
}

//Метод-диспетчер
void Memory::run()
{
int i, k,p;
for(i=0;i<M;i++)
tok[i]->run();
/*Каждые 10 тактов – сбор статистики по числу
блокировок*/
if ((total+1)%10==0)
 {
k=Queue();
p=(total+1)/10;
que_ave=que_ave*(1-1.0/p)+((double)k)/p;
```

```
 }
}
/*Вычисление общего количества мест для операндов
во всем КБ*/
long int Memory::Volume()
{
int k, i;
k=0;
for(i=0;i<M;i++)
k+=tok[i]->arity;
return(k);
}
//Подсчет текущего числа блокировок
int Memory::Queue()
{
int k, i;
k=0;
for( i=0;i<M;i++)
if (tok[i]->to_set==-1)  k++;
return(k);
}
```

Листинг 6.2. Функция main() (R = 2, r_1 = 0,1, r_2 = 0,9, N = 5, M = 20, λ = 0,1, μ = 0,067, h = 5, z = 0,4)

```
#define G 100000  /*длительность моделирования (в
условных единицах времени)*/
int main()
```

```
{
double a1[10]; /*размерность массива назначена в
предположении, что команда не может иметь более
десяти операндов*/
a1[0]=0.1; a1[1]=0.9; /*доля одноместных и
двуместных команд в программе*/
Server s(5,0.1,0.4,5); //создаем Server
Memory m(20, 2, a1, 0.067); /*создаем Memory
И настраиваем связь между ними*/
s.putMemory(m);
m.putServer(s);
srand((unsigned)time(0)); /*инициализируем ГСЧ*/
for(total=0L;total<G;total++) /*основной
моделирующий цикл*/
{
 s.run();
 m.run();
}
//печать результатов
printf("load=%f\n", load_ave);
printf("que=%f\n", que_ave);
}
```

6.4. Условие установления

В работе [41] построена система нелинейных уравнений, переменными которой являются характеристики ЭВМ УПД (в том числе количество активных ПЭ и заблокированных токенов), и доказана теорема, согласно которой решение системы существует

тогда и только тогда, когда выполнено условие

$$\gamma = F/q < 1, \qquad (6.1)$$

где $F = \sum_{i=1}^{R} ir_i$, $q = \sum_{i=1}^{h} iq_i$. Неравенство (6.1) является в определенной степени условием существования стационарного режима в ЭВМ УПД и в некотором смысле аналогом (по виду, но не по содержанию) классического условия стационарности в системе массового обслуживания *M*/*M*/1. Несложно дать его смысловую интерпретацию. Число копий должно обеспечивать «проталкивание» команд в процессор, в противном случае система постепенно прекратит функционировать. Условие (6.1) в точности подтверждается и результатами имитационного моделирования. В этом легко убедиться, если последовательность значений, возвращаемых на каждом такте методом `Server::Load()`, выводить на экран или в файл. При невыполнении условия, какие бы аргументы мы не передали в конструкторы объектов, трасса значений в конце концов превращается в последовательность из одних нулей, то есть система перестает функционировать — активных ПЭ нет. Такое явление не наблюдается, если условие (6.1) выполнено. Причем, как показали имитационные эксперименты, справедливость (6.1) не зависит от видов распределений для времени обслуживания на ПЭ и блоке памяти, хотя аналитическое доказательство, напомним, было получено только для случая экспоненциальных распределений.

По значению коэффициента γ можно судить об эффективности потоковой реализации алгоритма и проектировать алгоритм так, чтобы γ был как можно меньше. Этот вывод соответствует известному тезису [6] о том, что вычисления по принципу потока данных

эффективны лишь при довольно высоком уровне параллелизма алгоритма. Значение γ позволяет дать количественную оценку этого уровня.

Чтобы неравенство (6.1) не осталось для читателя отвлеченной математической фикцией, проиллюстрируем оценку F и q на примере некоторых распространенных задач вычислительной математики. В [60] для машины MIT предложена схема итерационного решения уравнения Лапласа (the Kernel of the Laplace Solver) на разностной сетке. Используется стандартный шаблон типа «прямой крест», расчетная формула такова:

$$u_{i,j}^{(k+1)} = \frac{u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}}{4}.$$

Соответствующий фрагмент программы на языке VAL выглядит так:

```
Y:= forall  I  in  [0, m+1]
construct
if  i=0  |  i=m+1  then X[i]
else
forall  j  in  [0, n+1]
construct
if  j=0  |  j=n+1  then  X[I,j]
else
0.25*(X[I,j-1]+X[I,j+1]+X[i-1,j]+X[i+1,j])
endif
endall
endif
endall
```

Заметим, что основная вычислительная операция требует четырех

операндов ($F = 4$), между тем как каждый посчитанный результат, то есть очередное приближение для узла (i, j), распространяется по пяти адресам. Это свойство закономерно вытекает из структуры реализованного алгоритма, так как значение u_{ij} должно поступить в ячейку, соответствующую узлу (i, j), и в ячейки, соответствующие четырем его соседям, для которых u_{ij} является одним из аргументов на следующей итерации. Таким образом, $q = 5$, и $\gamma = F/q = 0{,}8$.

Рассмотрим более сложный алгоритм — потоковую реализацию LU-разложения квадратной матрицы $n \times n$ [24], [79], [84]. Пусть n^2 процессоров расположены в виде квадратной решетки и каждый соединен с четырьмя соседями. Будем считать, во-первых, что любой процессор может переслать одно машинное слово за такое же время, какого требует одна операция над числами с плавающей точкой, и, во-вторых, пересылка и вычисления могут происходить одновременно. Суть вычислительного процесса такова.

На каждом шаге вычисляется очередной множитель, а все множители, вычисленные ранее, передаются направо, в соседние процессоры. Одновременно процессоры, имеющие необходимые данные, пересчитывают матричные элементы. Процесс как бы движется сквозь матрицу в виде диагонального волнового фронта. Поэтому такой способ организации называют LU-разложением в форме волнового фронта. Заметим, что как только модифицированное значение элемента (2, 2) определено, его можно переслать процессору (3, 2), и сразу после пересчета элемента (3, 2) можно вычислить множитель l_{32}. Это порождает новый волновой фронт, в котором производятся модификации второго шага LU-разложения. Через некоторое время сквозь матрицу будут двигаться сразу несколько волновых фронтов.

Распишем 5 начальных шагов волнового алгоритма. Размещение и передача данных на пятом шаге показаны на рис. 6.3.

Шаг 1. Переслать a_{1j} из P_{1j} в P_{2j}, $j = 1, ..., n$.

Шаг 2. Переслать a_{1j} из P_{2j} в P_{3j}, $j = 1, ..., n$. Вычислить l_{21}.

Шаг 3. Переслать a_{1j} из P_{3j} в P_{4j}, $j = 1, ..., n$. Переслать l_{21} в P_{22}. Вычислить l_{31}.

Шаг 4. Переслать a_{1j} из P_{4j} в P_{5j}, $j = 1, ..., n$. Переслать l_{21} в P_{6}. Переслать l_{31} в P_{32}. Вычислить l_{41}. Вычислить $l_{21}a_{12}$ в P_{22}.

Шаг 5. Переслать a_{1j} из P_{5j} в P_{6j}, $j = 1, ..., n$. Переслать l_{21} в P_{24}. Переслать l_{31} в P_{33}. Переслать l_{41} в P_{42}. Вычислить l_{51}. Вычислить новое значение a_{22} как $a_{22} - l_{21}a_{12}$ в P_{22}. Вычислить $l_{21}a_{13}$ в P_{6}. Вычислить $l_{31}a_{12}$ в P_{32}.

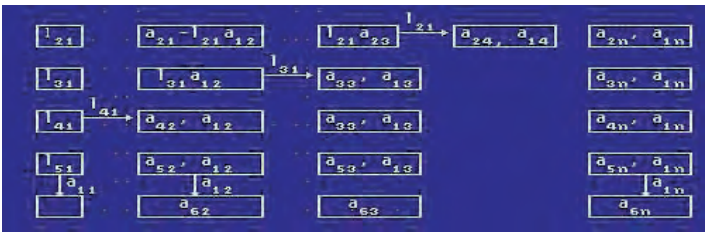

Рис. 6.3. *LU*-декомпозиция методом волнового фронта

Вычислим теперь значение F и q для $n = 4$. Пусть мы имеем 12 процессоров — ровно столько, сколько вычисляется элементов (четыре элемента первой строки матрицы U вычислять не нужно, так как они равны соответствующим элементам исходной матрицы A). Выпишем формулы, по которым вычисляются элементы матриц L и U:

$l_{21} = a_{21}/ a_{11}$; $u_{22} = a_{22} - l_{21} a_{12}$; $l_{32} = (a_{32} - l_{31} a_{12})/ u_{22}$;

$l_{31} = a_{31}/\, a_{13}$; $u_{23} = a_{23} - l_{21}\, a_{13}$; $u_{33} = a_{33} - l_{31}\, a_{13} - l_{32}u_{23}$;

$l_{41} = a_{41}/\, a_{14}$; $u_{24} = a_{24} - l_{21}\, a_{14}$; $u_{34} = a_{34} - l_{31}\, a_{14} - l_{32}u_{24}$;

$l_{42} = (a_{42} - l_{41}\, a_{12})/\, u_{22}$;

$l_{43} = (a_{43} - l_{41}\, a_{13} - l_{42}u_{23})/\, u_{33}$; $u_{44} = a_{44} - l_{41}\, a_{14} - l_{42}u_{24} - l_{43}u_{34}$.

Оценим значение F. Операции вычисления l_{21}, l_{31} и l_{41} являются в нашем смысле «0-местными», так как их операнды — константы, и результаты промежуточных вычислений не требуются. Далее местность распределяется так:

- 1-местная операция — 3 команды (u_{22}, u_{23}, u_{24});
- 2-местная операция — 2 команды (l_{32}, l_{42});
- 3-местная операция — 2 команды (u_{33}, u_{34});
- 4-местная операция — 1 команда (u_{43});
- 5-местная операция — 1 команда (u_{44}).

Таким образом, $F = 1 \cdot (3/12) + 2 \cdot (2/12) + 3 \cdot (2/12) + 4 \cdot (1/12) + 5 \cdot (1/12) \approx 1{,}83$.

Оценим теперь значение q. Каждый из вычисленных элементов матрицы L и U направляется процессором как минимум в один блок памяти команд — соответствующий команде, вычисляющей этот элемент, — а также в блоки памяти тех команд, для которых он является аргументом. Таким образом, общее число копий вычисленного результата превышает на единицу количество его адресов назначения. Запишем адреса назначения для каждого элемента:

$$l_{21} \to u_{22}, u_{23}, u_{24};$$

$$u_{33} \to l_{43};$$

$$u_{22} \rightarrow l_{32}, l_{42};$$

$$u_{34} \rightarrow u_{44};$$

$$u_{23} \rightarrow u_{33}, l_{43};$$

$$l_{41} \rightarrow l_{42}, l_{43}, u_{44};$$

$$u_{24} \rightarrow u_{34}, u_{44};$$

$$l_{42} \rightarrow L_{43}, u_{44};$$

$$l_{31} \rightarrow l_{32}, u_{32}, u_{34};$$

$$l_{43} \rightarrow u_{44};$$

$$l_{32} \rightarrow u_{33}, u_{34};$$

$u_{44} \rightarrow$ аргументом не является.

Отсюда $q = 4 \cdot (3/12) + 3 \cdot (5/12) + 2 \cdot (3/12) + 1 \cdot (1/12) \approx 2{,}83$, то есть $F/q \approx 0{,}65$, а значит, условие (6.1) выполняется.

Дадим пояснение, как получено первое слагаемое. Имеются три выражения, в которых справа от стрелки перечислены три адреса назначения. Общее число адресов для них на один больше - четыре. Значит, вероятность того, что будут созданы 4 копии результата, равна три двенадцатых. Аналогично получаем все остальные слагаемые.

6.5. Анализ результатов

На рис. 6.4 и 6.6 приведены зависимости среднего числа активных процессорных элементов и среднего числа блокированных токенов от общего количества процессорных элементов для экспоненциального и Парето-распределения длительностей работы ПЭ и блока памяти.

Расчеты проводились при следующих значениях параметров: $R = 2$, $r_1 = 0{,}1$, $r_2 = 0{,}9$, $z = 0{,}8$, $M = 20$, $\lambda = 0.1$, $\mu = 0{,}067$, $h = 5$. Параметры распределений Парето были подобраны таким образом, чтобы сохранить математические ожидания: для процессорного элемента $k = 1$, $\alpha = 10/9$, для блока памяти $k = 1$, $\alpha = 15/14$.

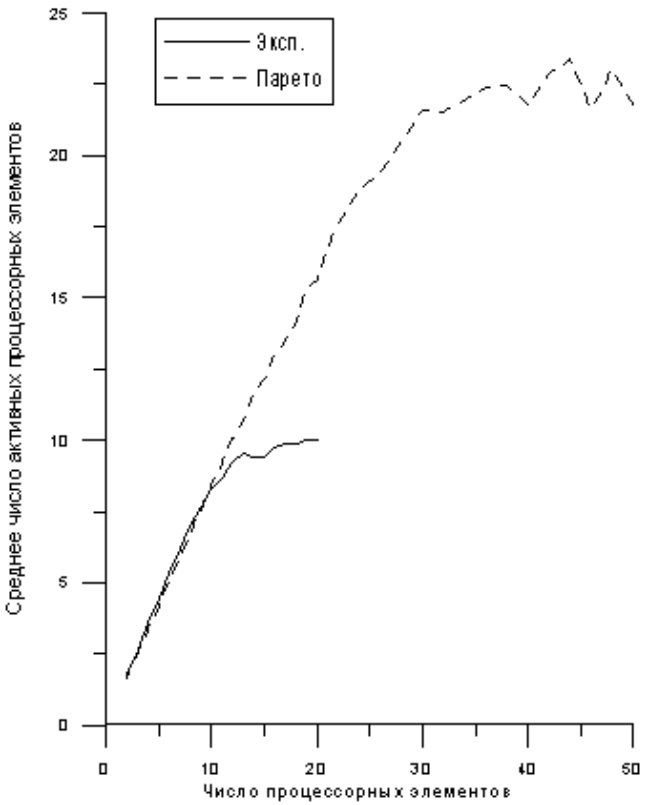

Рис. 6.4. Зависимость среднего числа активных процессорных элементов от их общего количества

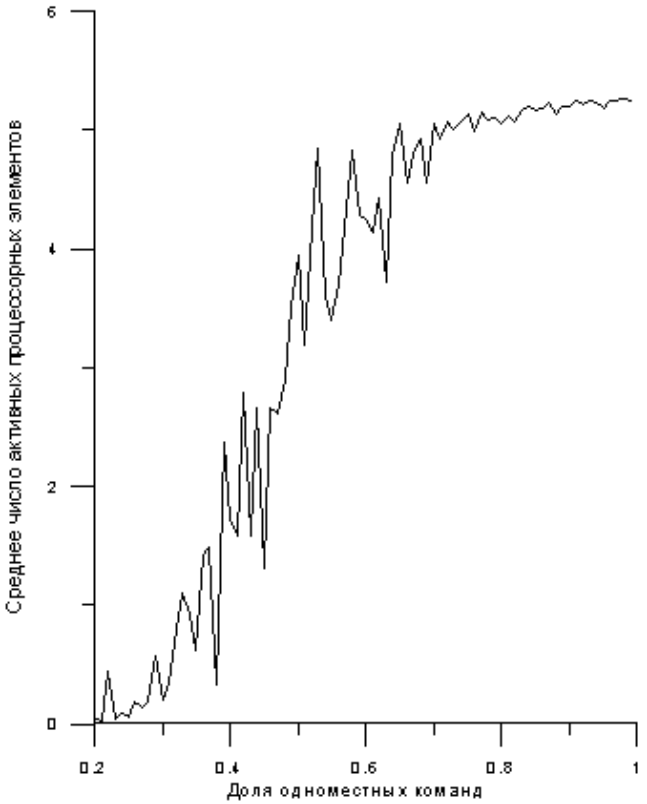

Рис. 6.5. Зависимость среднего числа активных процессорных элементов от доли одноместных команд

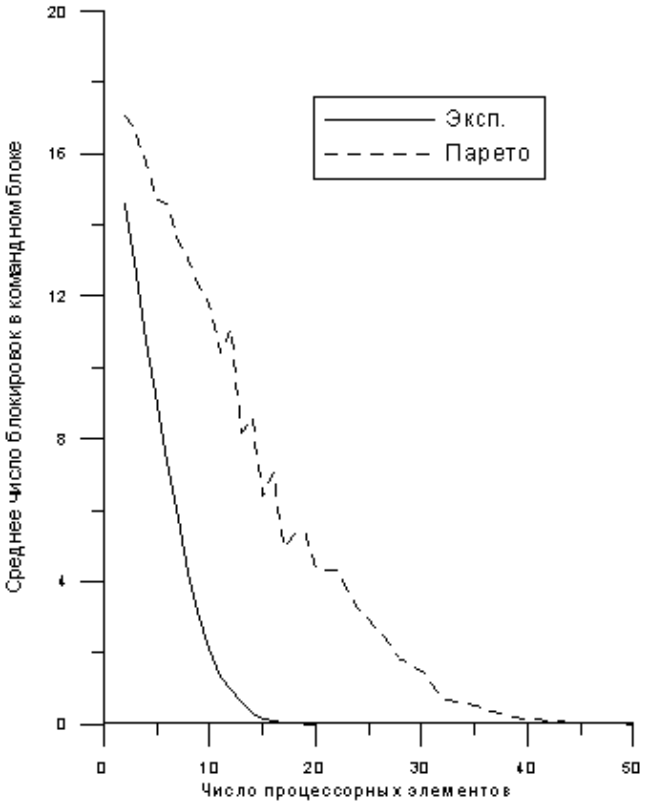

Рис. 6.6. Зависимость среднего числа блокированных токенов от количества процессорных элементов

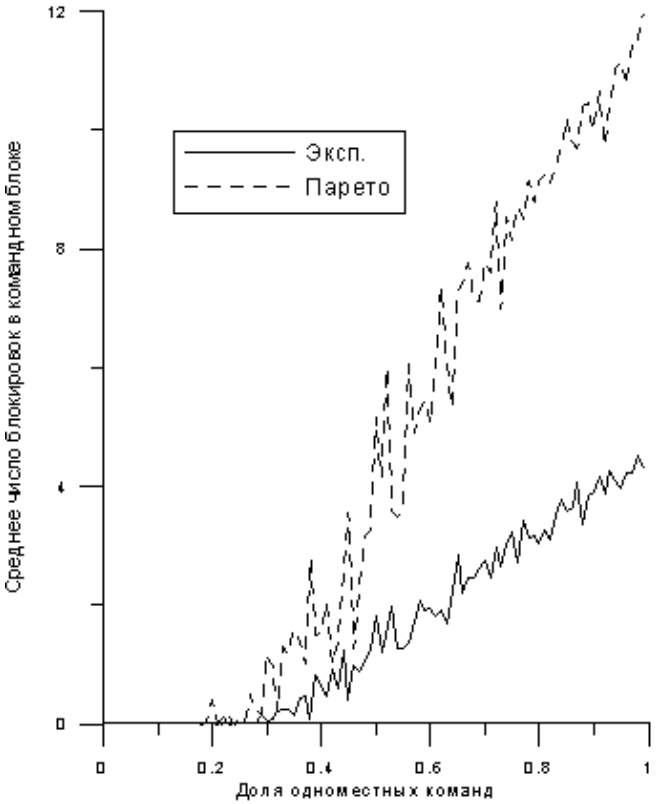

Рис. 6.7. Зависимость среднего числа блокированных токенов от доли одноместных команд

Из графиков видно, что для экспоненциального распределения предельная производительность системы равна около 9 активных ПЭ и достигается при $N = 12$. Под производительностью понимается количество выполненных команд в единицу времени, которое, в свою очередь, определяется количеством активных процессорных элементов. Для распределения Парето эти показатели равны, соответственно, 21–22 активных ПЭ при $N = 30$. Обращает на себя

внимание следующий факт. Совместный анализ рис. 6.4 и 6.6 показывает, что среднее число блокированных токенов, стремясь в пределе к нулю, становится меньше единицы именно для того N, для которого среднее число активных ПЭ достигает своего асимптотического значения. Так, на рис. 6.6 значения двух функций при $N = 12$ и $N = 30$ находятся приблизительно на одном уровне — меньше единицы.

На рис. 6.5 и 6.7 изображены соответствующие зависимости от доли одноместных команд r_1, а значит, и от γ, так как с ростом r_1 F уменьшается при неизменном q, а значит, уменьшается и γ. Здесь параметры расчетов такие: $N = 6$, $z = 0{,}36$, остальные оставлены без изменений. Значение r_1 меняется от 0,2 до 1, так как при $r_1 < 0{,}2$ не выполняется условие (6.1). Как было отмечено в 6.4, уменьшение значения γ приводит к росту числа активных ПЭ (в данном случае — до 5). От присущих изображенным на рис. 6.5 и 6.7 кривым флуктуаций, видимо, можно избавиться, если увеличить продолжительность имитационного эксперимента и количество экспериментов. Выбор этих параметров зависит от технической мощности вычислительного ресурса и наличия времени, которыми располагает исследователь. При проведении расчетов было выполнено 10 экспериментов по 100 000 тактов модельного времени каждый, при этом принципиальные особенности построенных зависимостей просматриваются уже достаточно хорошо.

Задания для самостоятельной работы

1. Построенная модель предполагает, что длительность работы процессорного элемента и блока памяти не зависит от

количества операндов в токене. Снимите это ограничение. Предположите, что параметры экспоненциальных распределений λ и μ зависят от «местности» команды и задайте для них эти зависимости, например: $\lambda_i = i\lambda$ или $\lambda_i = \lambda(1 + \ln(i))$, где i — «местность» команды. Какие изменения для этого придется внести в протоколы классов?

2. Проверьте справедливость гипотезы, согласно которой среднее число активных ПЭ не изменится при изменении λ и μ, если сохраняется их отношение $\rho = \lambda/\mu$.
3. Модифицируйте программу для случая, когда результаты вычислений могут пересылаться из ПЭ в любые блоки памяти, в том числе и с заблокированными токенами, предположив, что в последнем случае они просто теряются. Постройте зависимость среднего числа ПЭ от N. При прочих равных условиях производительность системы должна уменьшиться, так как теперь уже не все результаты вычислений «идут в дело». Проверьте это.

Глава 7. Моделирование модульной памяти

Несмотря на сравнительную молодость, наука под названием «теория вычислительных систем» уже имеет свою историю. И одной из ее страниц является задача, которую мы рассмотрим в этой главе. К настоящему времени она сохранила только учебно-методическую ценность, так как давно и хорошо проработана. Основные результаты были получены 30–40 лет назад, о чем свидетельствуют годы издания литературы по данной теме.

7.1. Что такое модульная память?

Для повышения производительности вычислительных систем в них используется оперативная память, состоящая из нескольких модулей (физических устройств). Каждый модуль представляет собой самостоятельное устройство, имеющее адресный регистр и позволяющее осуществлять обмен с ним независимо от остальных. Модульная организация памяти облегчает возможность наращивания емкости памяти и повышает эффективность обращений к ней. Для повышения производительности используется методика, называемая расслоением памяти. При расслоении модули обычно упорядочиваются так, чтобы N последовательных адресов памяти j, $j + 1, \ldots, j + N - 1$ приходились на N различных модулей. В i-м модуле $(0 \le i \le N - 1)$ находятся только слова, адреса которых имеют вид $kN + 1$, где $0 \le k \le M - 1$, M — число слов в одном модуле. Можно достичь в n раз большей скорости обмена с памятью в целом, чем у отдельного модуля, если обеспечить одновременное обращение к данным в каждом из модулей.

Если воспользоваться общепринятой классификацией параллельных

ЭВМ, впервые предложенной в фундаментальной работе [65], то следует сказать, что модульная организация памяти наиболее характерна для систем типа SIMD (один поток команд — много потоков данных), поскольку обеспечивает максимальную пропускную способность при записи и считывании массивов с единичным шагом между элементами. Примеры использования расслоенной памяти можно найти в [97] (описание структуры памяти машины TIASC с 8-кратным расслоением), [9] (высокопараллельный процессор MMP с главным формирователем — памятью большой емкости, включающей 4, 8, 16 или 32 модуля с произвольным доступом), [43] (CRAY-1 с 16-кратным и CYBER-205 с 8-кратным расслоением), [21] и [22] (суперЭВМ Fujitsu — 257-кратное расслоение). Отметим, что и память ЭВМ типа MIMD может строиться в виде модулей, доступ к которым осуществляется независимо [3], [94].

Расслоение памяти производится по принципу разделения ячеек на адресные биты низкого (low order interleaved — LOI) или высокого порядка (high order interleaved — HOI) [1], [43], [90]. Описанная структура расслоения относится к типу LOI, когда ячейки памяти, адреса которых различаются на единицу, располагаются в соседних модулях. HOI отвечает такой организации системы, когда ячейки памяти с последовательными адресами располагаются в одном модуле, а старшие разряды адреса определяют номер модуля. Например, в двумерной памяти расслоение низкого порядка обычно дает четные и нечетные адреса в разных модулях, тогда как расслоение высокого порядка дает самые младшие адреса в одном модуле и самые старшие — в другом. В этой главе мы будем моделировать память типа LOI.

Наиболее оптимистичным прогнозом повышения производительности, как уже отмечалось, является *n*-кратное ускорение обмена, достигаемое с помощью расслоения памяти. Но этот результат, конечно, является недостижимым из-за наличия конфликтов доступа или блокировок. Блокировку можно определить как «последовательное обращение к одному и тому же модулю памяти, если в нем находится общая для нескольких процессов переменная» [25]. Поэтому реальная производительность модульной памяти оказывается ниже максимально возможной. Одним из инструментов исследования эффективности использования модульной памяти является стохастическое моделирование. Наиболее важный параметр созданных с его помощью моделей — принятый исследователем механизм адресации или адресный поток обращений процессора к памяти, который зависит от равномерности расположения данных и качества программного обеспечения. Дадим краткий обзор основных из них.

Простейшим предположением для LOI является равнопериодическая адресация. Такая модель является детерминированной и исследована алгебраическими методами в [17], где сформулированы элементарные теоремы о производительности, учитывающие время цикла памяти *T*, число модулей *N* и равномерный промежуток в последовательности адресов *S*.

При моделировании модульной памяти вероятностными методами наиболее широкое распространение получили две модели обращения процессора к оперативной памяти. Согласно первой из них, любой запрос от процессора направляется к любому из *N* модулей с одинаковой вероятностью 1/*N*. Такая модель аппроксимирует

модульную организацию памяти при слабой регулярности размещения данных и обращений к ним[5]. В [1] и [77] построены дискретные и непрерывные аналитические модели для мультипроцессорных систем с модульной памятью. Память представлена как N-канальная система массового обслуживания с раздельными очередями к каждому модулю. В качестве вектора состояния используется набор длин очередей. Тактом марковского процесса является цикл памяти — изменения состояний отслеживаются только в моменты окончания цикла. Это, во-первых, влечет за собой достаточно сильное предположение о синхронности работы всех модулей, во-вторых, не позволяет учесть влияние на функционирование ОП таких важных параметров, как длительность цикла памяти и его распределение.

В [91] проанализирована обобщенная модель мультипроцессорной вычислительной системы с модульной памятью, представленная в виде замкнутой сети с очередями. Запросы в каждом узле обслуживания (модуле памяти) обслуживаются в соответствии с дисциплиной FCFS, время обслуживания — постоянное. Порядок адресации задается в общем виде — набором маршрутных вероятностей x_{ij} того, что запрос от i-го процессора поступит на j-й модуль. Чтобы обойти трудность, связанную с детерминированным временем обслуживания, введены следующие упрощающие предположения: во-первых, поступающий от процессора в очередь к модулю памяти запрос застает обслуживаемый запрос на середине

[5] Под слабой регулярностью понимается тот случай, когда последовательно считываемые данные не хранятся в последовательных адресах памяти, а расположены случайным, трудно предсказуемым образом. Типичным примером является хранение связного списка

обслуживания, во-вторых, вероятность отсутствия заявок во всех модулях является мультипликативной. Исследуемой характеристикой является среднее число загруженных модулей памяти.

В [12] и [13] применен иной подход. Здесь вычисляются вероятности p_i того, что из входного потока будет выбрано на обслуживание ровно i запросов (то есть будет загружено ровно i модулей), а $(i + 1)$-й запрос блокируется из-за обращения к уже занятому модулю. Предполагается, что к памяти обращаются несколько потоков входных заявок, упорядоченных по абсолютному приоритету. Каждый поток имеет бесконечную интенсивность — все заявки поступают на обслуживание за нулевое время. Эта модель является однократной — она описывает работу памяти лишь до первой блокировки и не дает представления о ее функционировании на большом промежутке времени в стационарном режиме.

Предположением, во многих случаях более адекватно аппроксимирующим адресный поток, является локализация обращений. При использовании этого механизма существенное значение имеет то, является ли расслоение горизонтальным (то есть LOI) или вертикальным (HOI). Для вертикального расслоения предполагается, что если k-й запрос процессора приходится на j-й модуль, то его $(k + 1)$-й запрос с вероятностью p попадает к j-му модулю, а с вероятностью $(1 - p)/(N - 1)$ — к i-му модулю $(i \neq j)$. Таким образом, процессор равновероятно обращается в любой модуль ОП, за исключением того модуля, к которому было сделано предыдущее обращение. Такая модель построена в [90]. В явном виде получены формулы для среднего числа загруженных модулей ОП, состоящей из трех модулей. Отмечено, что этот приближенный

подход дает приемлемые по точности результаты лишь при ограниченном диапазоне исходных данных. Проведенный анализ, однако, наглядно показал, что производительность HOI-памяти с локализацией хуже, чем с HOI-памяти равновероятными обращениями, что интуитивно ясно, так как повторное обращение к одному модулю увеличивает вероятность блокировки.

Локализация при LOI заключается в том, что если k-й запрос процессора приходится на j-й модуль ОП, то его $(k+1)$-й запрос обращается к $(j+1)$-му модулю с большей вероятностью, чем к остальным. Эта модель хорошо аппроксимирует модульную память команд. Такая «локализованность» обращений связана с тем, что последовательность выполнения команд представляет собой чередование линейных участков программ. На линейном участке адреса команд следуют друг за другом в естественном порядке возрастания их номеров. Если используется модульная память с горизонтальной нумерацией ячеек, то на линейном участке выполнения обеспечивается последовательное циклическое использование модулей памяти. Нарушает же эту цикличность, например, команда, замыкающая цикл или команда условного перехода. Этот порядок локализации обращений впервые был предложен в [53] и [93]. Математическая модель построена в [2], где модульная память названа многосекционной. В качестве такта принята длительность цикла памяти. Получены формулы для вероятности того, что до первой блокировки будут заполнены по крайней мере k модулей, дальнейшее развитие процесса во времени не рассматривается.

Еще раз подчеркнем, что во всех перечисленных моделях для LOI

изменения состояний системы происходят только в моменты окончаний цикла памяти либо длительность цикла памяти не рассматривается вообще. Поэтому не удается исследовать влияние на производительность распределения длительности цикла.

Отметим еще один интересный режим адресации, предложенный и исследованный для HOI в [88]. Здесь в качестве состояния марковского процесса принимается вектор «давностей» обращений к модулям. Описанный алгоритм назван автором LRUSM (least recently used stack model). Номера модулей размещены в стеке (он же — вектор состояния). Когда в очередном такте происходит обращение к некоторому модулю, его номер извлекается из стека и записывается в вершину. Вероятность обращения к модулю определяется его позицией в стеке, то есть тем, насколько давно произошло последнее обращение к нему. Вводится ключевое понятие пакета обращений: так названа совокупность номеров модулей до первого повторения — обращения к модулю, уже имеющемуся в пакете. Пример разделения последовательности по пакетам — 312 | 143 | 324 | 2... Производительность вычисляется как отношение средней длины пакета (число обращений в одном пакете) к его продолжительности (время между началами двух последовательно расположенных пакетов). К недостаткам подхода следует отнести то, что для его реализации необходимо знать множество вероятностей p_i того, что обращение произойдет к модулю, занимающему i-ю позицию в стеке. Эти вероятности можно оценить только в результате статистической обработки результатов достаточно продолжительных наблюдений за поведением программ.

Имитационное моделирование обоих режимов адресации (LOI и HOI)

позволяет снять ограничения, присущие указанным работам. Мы рассмотрим модель режима адресации LOI для равновероятных обращений, исходными данными которой являются число модулей памяти N и время цикла T, которое предполагается либо постоянной, либо случайной величиной, заданной своим законом распределения. Далее мы построим модель для модульной памяти с локализованными обращениями, в которой добавляется еще один параметр — вероятность последовательного обращения p.

Необходимо отметить, что стохастическое моделирование не является единственным подходом к изучению параллелизма работы памяти. Альтернативой ему может быть детерминированный подход. В этой связи очень интересна работа [85]. Ее авторы рассматривают память с горизонтальным m-расслоением в одно- и двухпроцессорной системе (с одним и двумя входными потоками соответственно). Моделируется выполнение векторной команды чтения или записи, причем предполагается, что компоненты вектора размещены в памяти равнопериодически с периодом d. Время обслуживания запроса — постоянная величина, равная $n_c\tau$, где τ — цикл процессора, то есть интервал между генерациями им запросов к памяти, n_c – некоторое натуральное число (параметр модели), задающее кратность времени обслуживания запроса циклу процессора. Вводятся ключевые понятия возвратного числа (return number) r и доступного множества (access set) Z, где $r = m/\mathrm{GCD}(m, d)$ — длина цикла обхода, то есть число обращений до повторного обращения к тому же модулю, а в Z входят номера модулей, которые участвуют в этом цикле, GCD — наибольший общий делитель (greatest common divisor). Например, если $\mathrm{GCD}(m, d) = 1$, то $r = m$, $Z = \{0, 1, ..., m - 1\}$. Для одного потока

анализ тривиален: ясно, что блокировок нет при $r \geq n_c$, а в случае $r < n_c$ в каждом цикле обхода длительность блокировки составляет $n_c - r$. Для анализа производительности памяти при двух потоках используются в совокупности теоретико-множественный и теоретико-числовой подходы, основанные на алгебре сравнений. Входными параметрами при этом являются периоды потоков d_1 и d_2, а также их стартовые адреса b_1 и b_2. Основной вопрос, исследуемый авторами: при каких соотношениях m, d_1, d_2, b_1, b_2 и n_c возможна безблокировочная работа памяти (conflict free (CF))? Для его решения выведены следующие формулы:

1. Критерий CF для несвязных потоков (то есть $Z_1 \cap Z_2 = \varnothing$): $f = \mathrm{GCD}(m, d_1, d_2) > 1$.
2. Критерий CF для связных потоков: $\mathrm{GCD}(m/f, (d_2 - d_1)/f) \geq 2n_c$ плюс специально подобранные при выполнении этого условия b_1 и b_2, f вычисляется по формуле, приведенной в первом пункте.

Кроме того, авторами предложена классификация видов блокировок и получены соотношения, выполнение которых влечет за собой наступление или ненаступление блокировки определенного вида. Практическая ценность доказанных теорем несколько снижается невозможностью предсказания начальных адресов b_1 и b_2. Результаты, полученные в [85], говорят о том, что при сильной регулярности расположения данных в памяти алгебраический подход может служить достаточно мощным инструментом анализа.

В исследовании, предложенном в [75], например, исходными параметрами модели являются число модулей памяти N и адресный промежуток между двумя последовательными обращениями S. Для

вычисления номера модуля, в который загружается i-й элемент вектора, предлагается функция сдвига (skewing scheme), имеющая вид

$$m(i) = \left(i + \left[\frac{i}{N} \right] \right) \bmod N .$$

С помощью алгебраического метода показано, что использование этой функции при регулярном адресном потоке позволяет снизить число блокировок по сравнению с обычным горизонтальным расслоением. К недостаткам схемы следует отнести затраты, связанные с усложненным вычислением при каждом обращении номера модуля, в котором хранятся требуемые данные.

Появившиеся в дальнейшем алгоритмы размещения данных в памяти выявили неполноту классификации моделей обращения на равновероятную и локализованную и заставили задуматься о введении новых вероятностных аппроксимаций. В качестве примера опишем подход, разработанный в [58], для размещения в N модулях элементов матрицы $N \times N$. Задача, которую можно назвать и головоломкой, ставится так: никакие два элемента одного столбца, одной строки и одной из главных диагоналей не должны храниться в одном и том же модуле. Это условие естественно, так как выборка данных из матрицы в подавляющем большинстве матричных алгоритмов происходит по строкам, столбцам и двум главным диагоналям, и выполнение поставленного условия исключает при этом возможность блокировки. Приведем без доказательства алгоритм, сопоставляющий индексам i и j элемента матрицы $A(i, j)$ номер модуля, в котором он хранится.

1.Вычислить функцию $w(i)$ по следующему правилу:

- взять представление $B(i)$ числа i в двоичной системе счисления. Если число разрядов нечетно, дополнить слева

нулем;

- левую $B_l(i)$ и правую $B_r(i)$ половины $B(i)$ сложить по mod 2: $B_+(i) = B_l(i) + B_r(i)$;
- произвести конкатенацию битовых строк $w(i) = B_r(i) \parallel B_+(i)$.

2. Вычислить $F(i, j) = w(i) + B(j)$.

3. Перевести $F(i, j)$ в десятичную систему счисления. Полученное число и есть номер модуля, хранящего $A(i, j)$.

Пример реализации алгоритма (на месте элемента $A(i, j)$ матрицы 4×4 стоит номер модуля, в котором он размещен):

$$\begin{matrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \end{matrix}.$$

Мы видим, что какую-либо тенденцию в расположении чисел уловить трудно. В то же время нельзя говорить и о равновероятном распределении, так как ясно видно, что, по крайней мере, вероятность повторного обращения подряд к одному и тому же модулю при выборке по строке или столбцу равна нулю. Поэтому в этом случае обе модели обращений нельзя признать вполне удовлетворительными.

Сделанный обзор показывает, что моделирование модульной памяти можно производить различными средствами. Выбор модели и используемый математический аппарат зависят от способа расслоения памяти и регулярности потока обращений к ней. Перейдем теперь к рассмотрению тех задач, для которых наиболее подходящим инструментом исследования является имитационное моделирование.

7.2. Имитационная модель с равновероятными обращениями

Сформулируем основные предложения о функционировании модульной оперативной памяти, положенные в основу имитационной модели с равновероятными обращениями:

- все модули работают так, что за один цикл длительности T каждый модуль может обслужить не более одной заявки, а все модули — не более N заявок одновременно;
- каждая заявка с равной вероятностью $1/N$ требует обслуживания любым из N модулей ОП;
- назначение заявке модуля памяти происходит в порядке очередности заявок во входном потоке. Из входного потока на обслуживание с интенсивностью одна в единицу времени заявки поступают до тех пор, пока не появится заявка с запросом на обслуживание уже занятым модулем. В этом случае она блокирует поступления к модулям ОП всех последующих заявок. Таким образом, в системе отсутствует понятие очереди, или, точнее, ее длина не превышает единицу. Эту единицу и составляет заявка, ожидающая освобождения требуемого ей модуля и блокирующая поступление всех последующих заявок.

Одной из наиболее естественных моделей такой системы является цепь Маркова, в которой состояние представляется вектором $t = (t_1, t_2, \ldots, t_k)$, где t_i — время, оставшееся до завершения обслуживания запроса i-м модулем. (t_i принимает целые значения от 1 до T) [40]. Однако наряду с идейной простотой эта модель обладает значительной трудоемкостью, так как количество возможных

состояний комбинаторным образом зависит от T и N. Поэтому уже для $N = 20$ число состояний будет столь велико, что расчет характеристик ОП на основе такой модели практически невозможен. Для имитационной модели это обстоятельство не является затруднительным.

Программная реализация модели состоит всего из одного класса `Memory`. Отдельные модули представлены в нем элементами целочисленного массива, хранящими остаточное время обслуживания или –1 в случае, если модуль простаивает. Параметры, по которым собирается статистика:

- среднее число активных модулей;
- средняя длительность интервала между последовательными поступлениями запросов. Если бы не было блокировок, величина равнялась бы единице. Наличие блокировок удлиняет этот интервал;
- средняя длительность одной блокировки;
- средняя длительность периода занятости — интервала времени между окончанием предыдущей блокировки и наступлением следующей.

Листинг 7.1. Файл classes7.h, содержащий описание класса Memory

```
//блок переменных для ведения статистики
double lock_ave=0;    /*для подсчета средней
длительности блокировки*/
double arr_ave=0;   /*для подсчета среднего
интервала между поступлениями*/
```

```
double busy_ave=0;   /*для подсчета средней
длительности периода занятости*/
double load_ave=0;   /*для подсчета среднего числа
активных модулей*/
int busy_counter=1;  //счетчик периодов занятости
int lock_counter=0;   /*счетчик количества
блокировок*/
int arr_counter=0;    /*счетчик количества
поступлений на обслуживание*/
long int total;  /*счетчик тактов модельного
времени*/

/*Интерфейс класса Memory. Длительность цикла ОП –
постоянная величина*/
class Memory
{
 //Неизменяемые поля данных
 const int N;          //количество модулей
 const int T;   /*длительность цикла оперативной
памяти*/

//Изменяемые поля данных
 int *served;   /*массив остаточных времен
обслуживания */
 int locking;  /*номер модуля, из-за которого
система заблокирована. –1, если блокировки нет*/

//Поля данных для сбора статистики
```

```
 int busy;   /*счетчик текущей длительности
периода занятости. –1, если текущее состояние –
блокировка*/
 int lock;   /*счетчик текущей длительности
блокировки. –1, если в текущем состоянии
блокировки нет*/
  public:
 Memory(int a, int b);   //конструктор
 ~Memory();        //деструктор
 void Arrival();   /*поступление нового запроса от
процессора*/
 void Complete(int i);   /*i-й модуль завершил
обслуживание запроса*/
 void run();           //диспетчер
 int Load();      /*подсчет текущего числа
активных модулей (немоделирующий метод)*/
};
/*Конструктор. Аргументы: a – количество модулей;
b – длительность цикла*/
Memory::Memory(int a, int b): N(a), T(b)
{
 int i;
 served=(int*)malloc(N*sizeof(int));
 for(i=0;i<N; i++) served[i]=-1;
 locking=-1;
 busy=0; lock=-1;
}
```

```
Memory::~Memory()
{
 free(served);
}

//Вычисление текущего числа активных модулей
int Memory::Load()
{
 int i, s;
 s=0;
/*Признак активности i-го модуля – положительное
значение i-го элемента массива served*/
 for(i=0;i<N;i++)
  if (served[i]>0) s++;
 return(s);
}
//Прибытие нового запроса от процессора
void Memory::Arrival()
{
 int k;
 k=rand()%N;     //требуемый адрес – в k-м модуле
 if (served[k]!=-1)   /*модуль занят. Наступает
блокировка*/
 {
  locking=k;
  lock=0;   /*включаем счетчик длительности
текущей блокировки*/
```

```
  lock_counter++;    /*инкремент счетчика
блокировок*/

/*Период занятости закончился. Пересчитываем его
среднюю длину*/
  busy_ave=busy_ave*(1-1.0/busy_counter)
+((double)busy)/busy_counter;
  busy=-1;   /*отключаем счетчик длительности
текущего периода занятости*/
 }
 else            //модуль свободен
 {
  served[k]=T;
  arr_counter++;  //инкремент счетчика поступлений
//Пересчет среднего интервала между поступлениями
  arr_ave=arr_ave*(1-1.0/arr_counter)
+1.0/arr_counter;
 }
}

//i-й модуль завершил обслуживание
void Memory::Complete(int i)
{
 if (locking!=i) served[i]=-1;   /*этот модуль не
был причиной блокировки*/
 else   /*освобождение i-го модуля означает конец
блокировки*/
 {
```

```
  served[i]=T;  /*на обслуживание сразу же
поступает заблокированный запрос*/
  locking=-1;   //отключаем признак блокировки
  arr_counter++;  //инкремент счетчика поступлений
  arr_ave=arr_ave*(1-1.0/arr_counter)
+(double)(lock+1)/arr_counter;
  busy=0;   /*включаем счетчик длительности
текущего периода занятости*/
  busy_counter++;   /*инкремент счетчика числа
периодов занятости*/

/*Блокировка закончилась. Пересчитываем ее среднюю
длительность*/
  lock_ave=lock_ave*(1-1.0/lock_counter)
+((double)lock)/lock_counter;
  lock=-1;   /*отключаем счетчик длительности
текущей блокировки*/
 }
}

//Метод-диспетчер
void Memory::run()
{
 int pr, i;
 pr=0;
 if (locking!=-1) pr=1;   /*текущее состояние –
блокировка*/
```

```
/*Если блокировка была, то в этом цикле она может
закончиться, а может и нет*/
 for(i=0;i<N;i++)
 {
  if (served[i]>0) served[i]--;
  if (served[i]==0) Complete(i);
 }
 if (pr==0) Arrival();  /*блокировки нет и не
было, можно попытаться принять новый запрос. Если
она была и прекратилась, новый запрос уже поступил
в методе Complete*/
//Пересчитываем среднее число активных модулей
 pr=Load();
 load_ave=load_ave*(1-1.0/(total+1))
+((double)pr)/(total+1);
/*Инкремент одного из счетчиков длительности
текущего состояния*/
 if (busy!=-1) busy++;
 else lock++;
}
```

7.3. Анализ результатов

На рис. 7.1 приведены зависимости среднего числа активных модулей от общего числа модулей для постоянного и экспоненциально распределенного времени цикла оперативной памяти. При этом использовались следующие значения параметров: $T = 15$, $\mu = 1/15$, длительность моделирования — 1 млн. тактов. Обращает на себя внимание довольно низкий КПД модульной памяти при

равновероятном распределении запросов — с увеличением N производительность растет крайне незначительно. Отметим также еще одно обстоятельство, касающееся не только этого, но и всех остальных рисунков. При прочих равных условиях показатели для случайного времени цикла всегда хуже, чем для постоянного. Этот факт находится в полном соответствии с типичным для теории массового обслуживания наблюдением, сделанным еще Л. Клейнроком [15], а возможно, и кем-то до него: случайное из-за наличия продолжительных флуктуаций всегда хуже детерминированного (это справедливо и в подавляющем большинстве жизненных ситуаций). Простейшей иллюстрацией этого тезиса служит отсутствие стационарного режима в системе $M/M/1$ при одинаковых интенсивностях входного потока и обслуживания. Но в детерминированном случае наличие стационарного режима совершенно очевидно: если заявки поступают строго каждые 5 мин и ровно 5 мин обслуживаются, система будет работать стабильно и очереди в ней никогда не будет.

Остается не выясненным такой вопрос: имеют ли графики на рис. 7.1 горизонтальную асимптоту или же стремятся к бесконечности? Более правдоподобным представляется первый ответ. Дело в том, что зависимости среднего интервала между поступлениями от числа модулей (рис. 7.3) по смыслу ни при каком N не могут быть меньше единицы (в случае, если блокировок нет вообще). Следовательно, эти кривые горизонтальную асимптоту имеют, а значит, и функции, изображенные на рис. 7.1, должны сходиться на бесконечности. Сходимость эта, однако, очень и очень медленная.

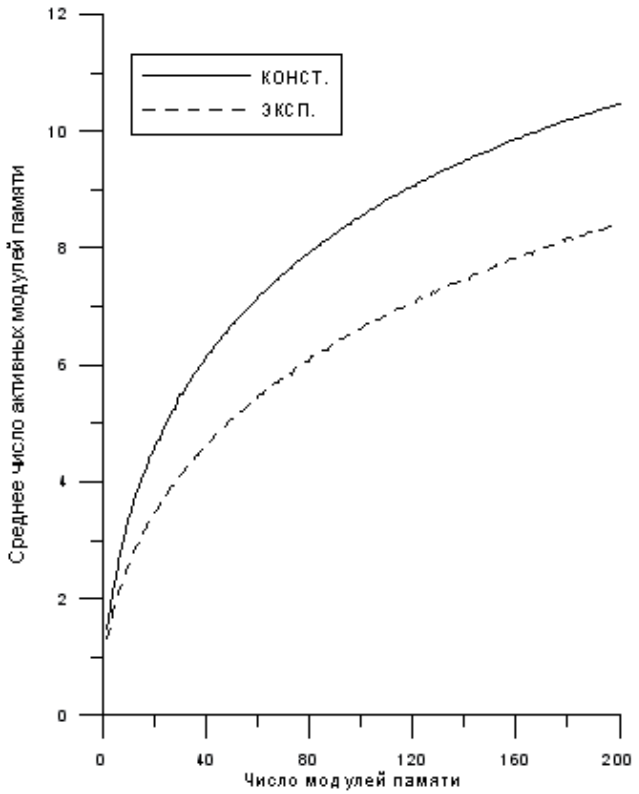

Рис. 7.1. Зависимость среднего числа активных модулей от общего количества модулей памяти, $T = 15$

Прокомментируем рис. 7.2, на котором изображена зависимость средней длительности блокировки от числа модулей. Для экспоненциального распределения это действительно будет константа, равная приблизительно среднему времени обслуживания, поскольку остаточная длительность для него распределена точно так же, как и сама величина, и поэтому не зависит ни от чего, в том числе и от N. Небольшое превышение значения 15 связано, видимо, с тем,

что мы на этот раз не промасштабировали время. Для $T = \text{const}$ картина иная, поскольку для длительности блокировки имеет значение, произошла ли она, когда модулю с повторным обращением осталось работать $T - 1$ тактов или 1 такт. А на распределение остаточного времени обслуживания в момент повторного обращения влияет количество модулей.

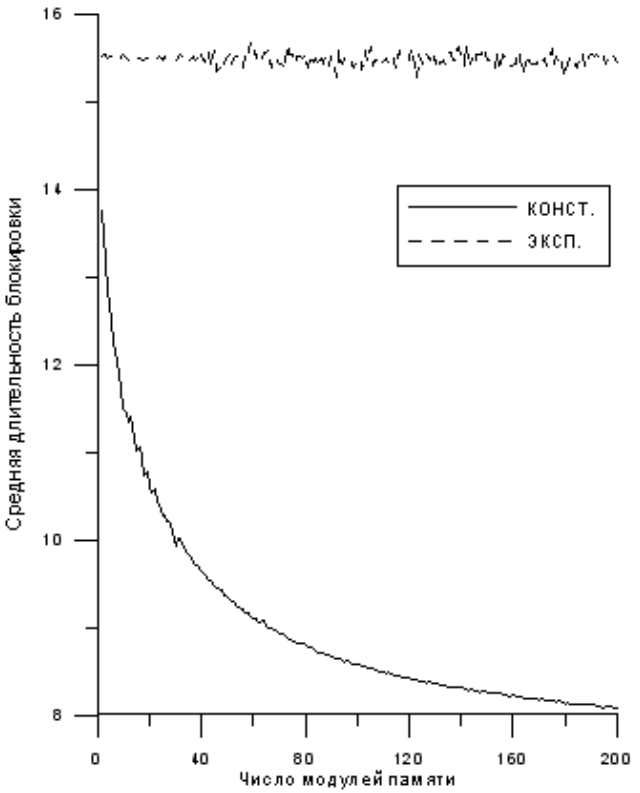

Рис. 7.2. Зависимость средней длительности блокировки от общего количества модулей памяти, $T = 15$

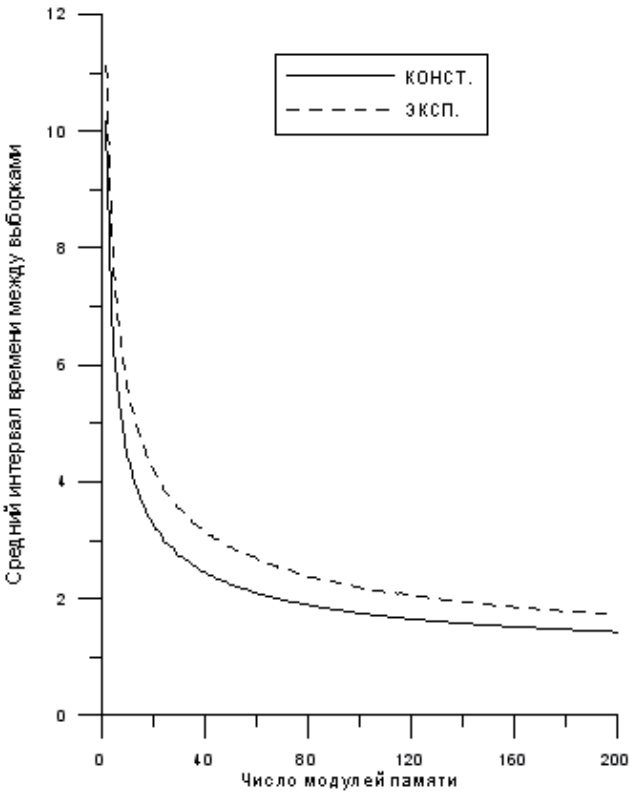

Рис. 7.3. Зависимость среднего интервала между поступлениями запросов от общего количества модулей памяти, T= 15

На рис. 7.4–7.6 построены зависимости тех же параметров от T при $N = 4$

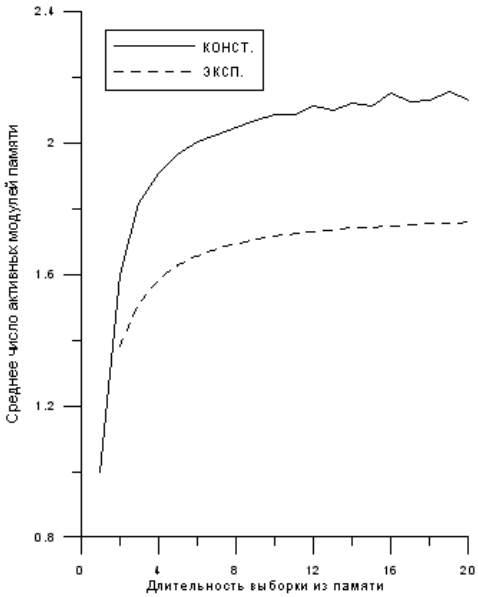

Рис. 7.4. Зависимость среднего числа активных модулей от длительности цикла памяти, $N = 4$

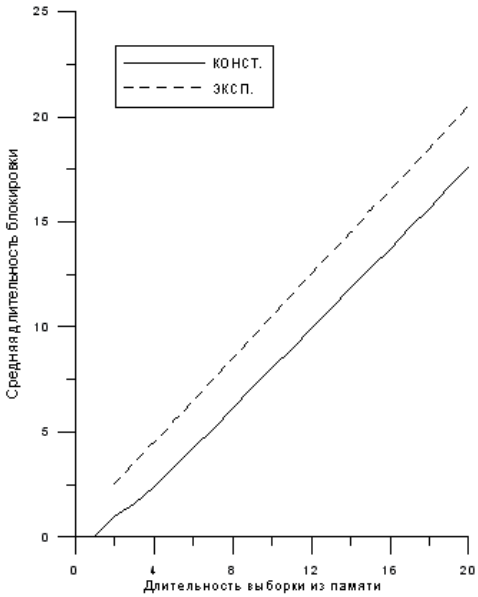

Рис. 7.5. Зависимость средней длительности блокировки от длительности цикла памяти, $N = 4$

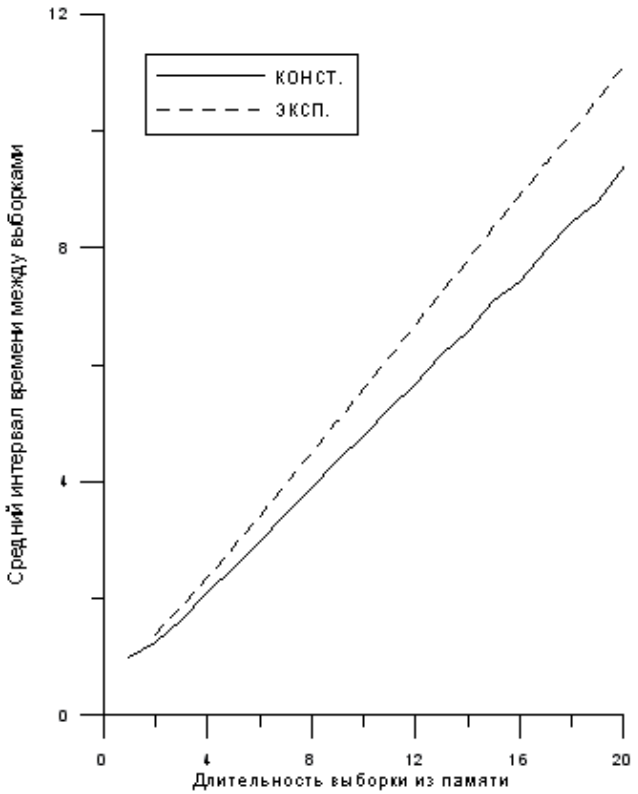

Рис. 7.6. Зависимость среднего интервала между поступлениями запросов от длительности цикла памяти, $N = 4$

7.4. Обращение с приоритетом последовательной выборки

Введем понятие «вероятность последовательного обращения». Смысл его состоит в следующем. Если в момент времени $t = k$ поступившая заявка потребовала модуль с номером i, то вероятность того, что следующая заявка потребует модуль $i + 1$, равна p (нарушения

порядка обращения не произошло!), а любой модуль (в том числе и $(i + 1)$-й) — с равной вероятностью $(1 - p)/N$. Число p зависит от количества «скачков» в тексте программы, или, иными словами, от частоты применения оператора `GOTO` и циклов. Если «скачка» нет, то последующее обращение к памяти произойдет по последующему адресу, то есть при горизонтальном расслоении памяти — к последующему модулю. Таким образом, любая поступающая заявка всегда требует какой-то из модулей с вероятностью $p + (1 - p)/N$, а остальные $N - 1$ модулей — с вероятностью $(1 - p)/N$ для каждого. Поясним, с чем связано наличие второго слагаемого $(1 - p)/N$. Дело в том, что в случае нарушения оператор `GOTO` может сослаться на адрес, содержимое которого может храниться в любом из модулей, в том числе и в следующем по порядку, только это будет обращение уже не к последующему, а какому-то другому адресу, также хранящемуся в нем.

Для моделирования этого режима адресации класс `Memory` нужно немного изменить. Прежде всего в нем появляются два новых поля данных — неизменяемое `p` и изменяемое `last`. Поле `p` хранит описанную ранее вероятность, а поле `last` — номер модуля памяти, к которому произошло последнее обращение. Среди методов наибольшему изменению подвергается `Memory::Arrival()`, так как номера модуля, на который пересылается поступивший запрос, разыгрывается по более сложному алгоритму. Приведем листинг класса `Memory` с выделенными отличиями от реализации при равновероятной адресации.

Листинг 7.2. Модификация класса Memory для новой дисциплины выборки

```
#include<cstdlib>
Class Memory
{
 const int N;
 const double mu;
 const double p;   /*вероятность последовательного обращения*/
 int *served;
 int locking;
 int last;   /*номер модуля, к которому произошло последнее обращение*/
 int busy;
 int lock;
  public:
 Memory(int a, double b, double c);
 ~Memory();
 void Arrival();
 void Complete(int i);
 void run();
 int Load();
};
/*Аргумент c – вероятность последовательного обращения*/
Memory::Memory(int a, double b, double c): N(a), mu(b), p(c)
```

```
{
 int i;
 served=(int*)malloc(N*sizeof(int));
 for(i=0;i<N; i++) served[i]=-1;
 locking=-1; last=-1;
 busy=0; lock=-1;
}

Memory::~Memory()
{
 free(served);
}

void Memory::Arrival()
{
 int k, a1; double a;
/*Если это самый первый запрос – модуль
разыгрывается случайным образом*/
 if (last==-1) k=rand()%N;
 else
 {
  a1=rand();
  a=((double)a1)/RAND_MAX;
  if (a<=p) /*нарушения порядка следования не
произошло*/
    if (last<(N-1)) k=last+1;   /*модуль last не
последний по номеру*/
```

```
    else k=0;  //модуль last – последний по номеру
  else k=rand()%N;  /*порядок следования модулей
нарушился – равновероятный розыгрыш*/
 }
 last=k;     //переназначение поля last
 if (served[k]!=-1)
 {
  locking=k;
  lock=0;
  lock_counter++;
  busy_ave=busy_ave*(1-1.0/busy_counter)
+((double)busy)/busy_counter;
  busy=-1;
 }
 else
 {
  served[k]=(int)(get_exp(mu));
  if (served[k]==0) served[k]=1;
  arr_counter++;
  arr_ave=arr_ave*(1-1.0/arr_counter)
+1.0/arr_counter;
 }
}
```

На рис. 7.7–7.12 приведены результаты работы программы. Рис. 7.7–7.9 по своему качественному содержанию идентичны рис. 7.1–7.3 и отличаются лишь числовыми данными. При моделировании значение $p = 0{,}7$. На рис. 7.10–7.12 изображены зависимости характеристик

системы от параметра p при $N = 4$, $T = 15$. Нетрудно заметить, что с увеличением p эти характеристики улучшаются, так как повышение регулярности запросов снижает вероятность блокировки. В частности, при $p = 1$, соответствующей полностью детерминированной последовательности обращений к модулям памяти, наличие или полное отсутствие блокировок зависит исключительно от того, что больше — N или T.

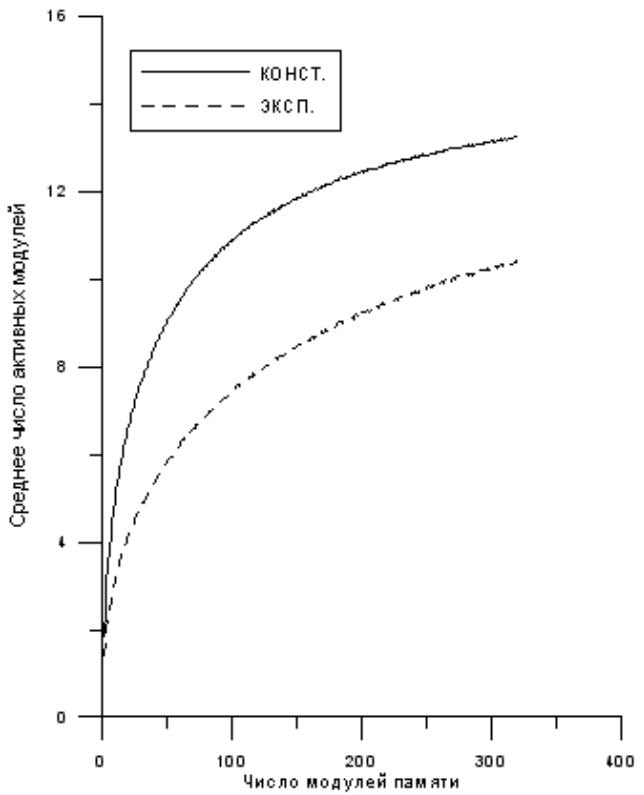

Рис. 7.7. Зависимость среднего числа активных модулей от общего количества модулей памяти, $T = 15$, $p = 0{,}7$

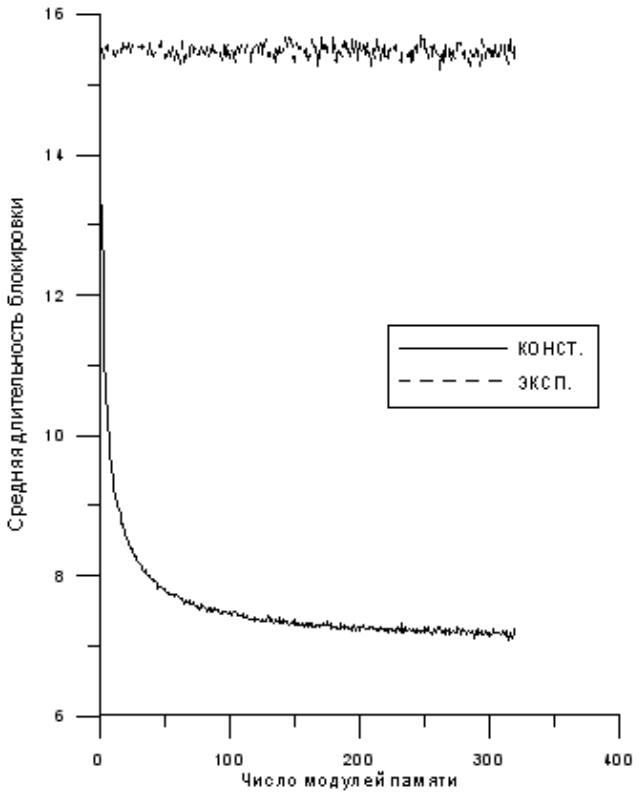

Рис. 7.8. Зависимость средней длительности блокировки от общего количества модулей памяти, $T = 15$, $p = 0{,}7$

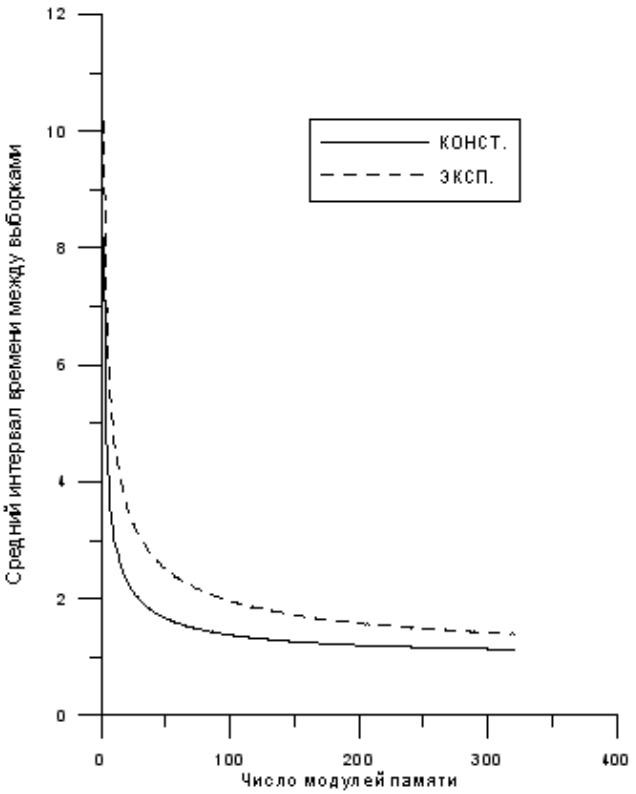

Рис. 7.9. Зависимость среднего интервала между поступлениями запросов от общего количества модулей памяти, $T = 15$, $p = 0{,}7$

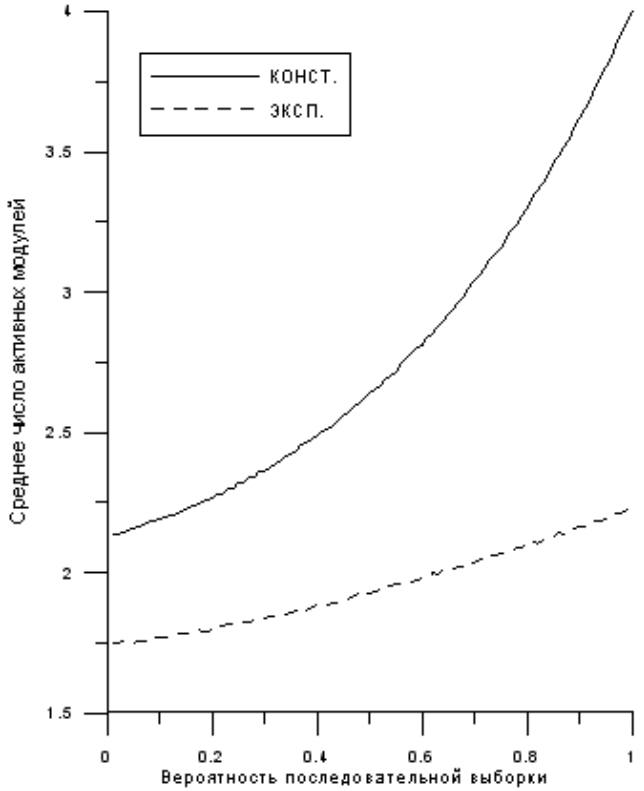

Рис. 7.10. Зависимость среднего числа активных модулей от длительности цикла памяти, $N = 4$, $T = 15$

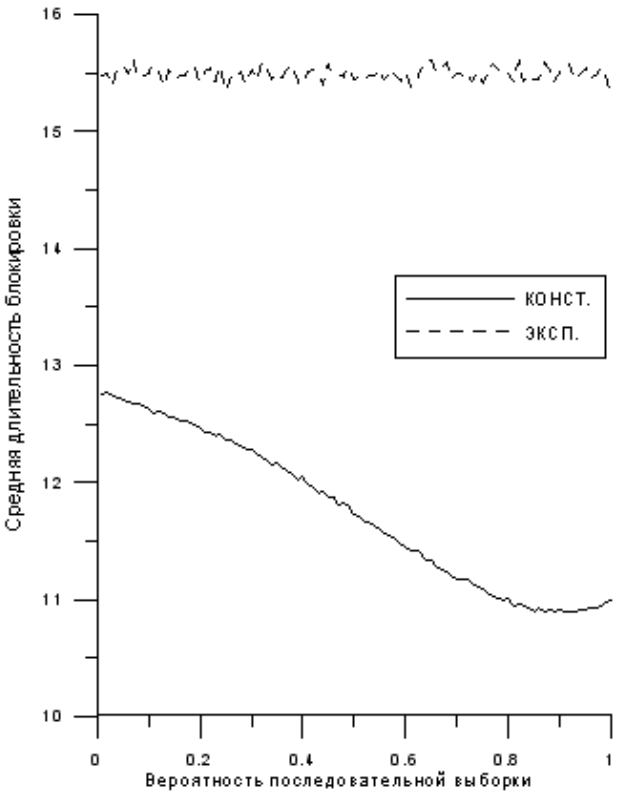

Рис. 7.11. Зависимость средней длительности блокировки от длительности цикла памяти, $N = 4$, $T = 15$

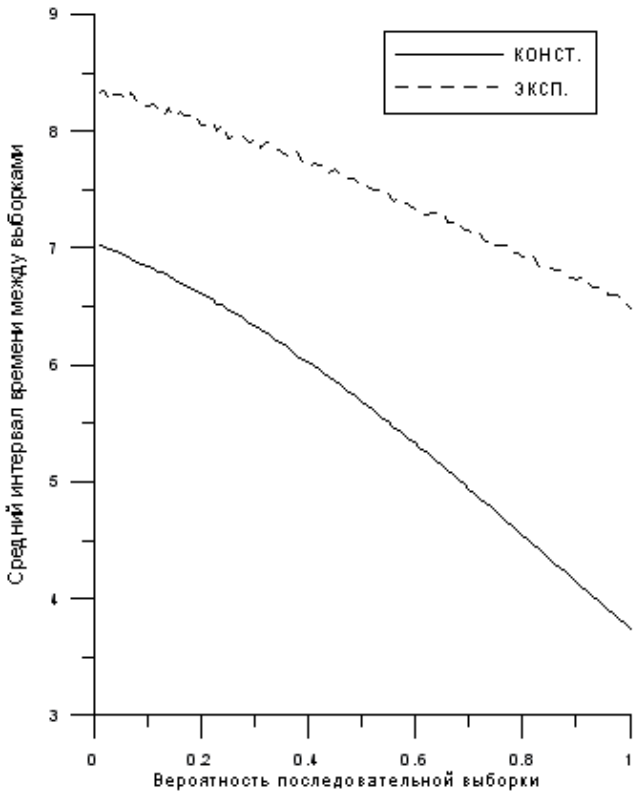

Рис. 7.12. Зависимость среднего интервала между поступлениями запросов от длительности цикла памяти, $N = 4$, $T = 15$

Задания для самостоятельной работы

1. Разработайте имитационную модель для расслоения памяти по принципу HOI. Для этого случая параметр p будет вероятностью того, что следующее обращение произойдет к тому же модулю, что и предыдущее, так как два последовательных адреса хранятся в одном модуле. С точки зрения здравого смысла, производительность такой памяти, выражаемая средним числом

активных модулей, должна оказаться хуже производительности не только LOI-расслоения, но и равновероятной адресации, а увеличение p ухудшает характеристики. Убедитесь в этом, реализовав модель.

2. Дополните рис. 7.1 или 7.7 еще одной кривой — для распределения T по закону Парето с сохранением математического ожидания, равного 15. Какое положение займет эта кривая по отношению к имеющимся?
3. Для небольшого значения T попробуйте установить, сходится ли функция зависимости среднего числа активных модулей от общего числа модулей на бесконечности, и если сходится, то к чему.

Глава 8. Моделирование программных каналов[6]

8.1. Краткие сведения о каналах

Компьютерное решение сложной задачи складывается из совокупности параллельных процессов, регулярно обменивающихся информацией при выполнении. Поэтому современные операционные системы, такие как Unix, Windows NT, OS/2, обладают развитыми встроенными средствами взаимодействия процессов и аппаратом системных вызовов. К числу таких средств явного взаимодействия относятся программные каналы (pipe), очереди сообщений и общие сегменты оперативной памяти. Их организация и построение зависят от большого количества параметров, выбор которых основывается на опыте и интуиции программиста. С другой стороны, функционирование упомянутых средств может быть описано в терминах случайных процессов, что позволит обоснованно рассчитывать показатели производительности и эффективности и, в конечном итоге, основываясь на конкретике задачи, оценивать целесообразность применения для ее решения того или иного средства взаимодействия.

Логика работы каналов описана во многих источниках [16], [46]. *Канал* — это специальный файл, в который можно записывать информацию или считывать ее из него. Особенность такого файла заключается в способе его использования. Он должен быть разделяемым среди некоторого числа процессов, открыт одновременно как для чтения, так и для записи различным процессам. Использование потоковой модели ввода-вывода при работе с каналом означает, что передаваемые через него данные им никак не

[6] Данная глава рассчитана на работу только в ОС Unix.

интерпретируются. Отделение одного сообщения от другого, определение адресата возлагается на процессы — пользователи канала. Сравнительно небольшой объем — 4096 байт — используется для обмена сообщениями между процессами-писателями и процессами-читателями в соответствии с правилами синхронизации. Эти правила таковы:

- в операции `read` указан объем порции считываемых данных меньший, чем объем информации, хранимой в текущий момент в канале. Из канала будет прочитана информация только заказанного объема. После чтения в канале остаются данные, на начало которых переустанавливается указатель считывания;
- если процесс запрашивает через `read` больший объем данных, чем находится в текущий момент в канале, то процессу будет передано только то, что имелось в канале. Канал становится пустым;
- если процесс-читатель пытается выполнить операцию `read` в момент, когда канал пуст, то процесс переводится в состояние ожидания поступления данных в канал — засыпает. Причем, если спали несколько процессов в ожидании поступления данных в пустой канал, они начнут между собой «состязаться» за возможность выполнить чтение. Если процесс выполняет `read`, то эта операция рассматривается как атомарная (неделимая) в отношении канала и не может быть прервана другими процессами;
- процесс пытается выполнить операцию `write`. При наличии свободного места для записи в канале производится запись с последующим передвижением указателя записи на следующий

свободный байт. Как и в случае чтения, операция `write` рассматривается как атомарная. Если в канал одновременно пытаются записать данные два процесса, то доступ к каналу получит только один из них, а второй будет заблокирован, пока не завершит операцию записи первый процесс;

- процесс пытается записать в канал данные, объем которых превышает объем свободного места в канале. Процесс-писатель записывает в канал столько данных, сколько возможно, и переводится в состояние ожидания момента появления свободного места.

Синтаксис вызова функции создания канала:

```
pipe(fdptr);
```

где `fdptr` — указатель на массив из двух целых переменных, в котором будут храниться два дескриптора файла: для чтения из канала и для записи в канал. Поскольку ядро реализует каналы внутри файловой системы и поскольку канал не существует до того, как его будут использовать, при создании канала ядро должно назначить ему индекс. Оно назначает для канала также пару пользовательских дескрипторов и соответствующие им записи в таблице файлов: один из дескрипторов — для чтения из канала, а другой — для записи в канал. Каналы реализованы как обычные файлы, поэтому операции ввода-вывода в канал выполняются обычными функциями чтения-записи `read`, `write` и процессам нет надобности знать, ведут ли они чтение или запись в обычный файл или в канал.

Канал следует рассматривать как очередь — процессы ведут запись на одном конце канала, а считывают данные с другого конца. Как

говорилось ранее, процессы обращаются к данным в канале в порядке их поступления в канал; это означает, что очередность, в которой данные записываются в канал, совпадает с очередностью их выборки из канала. Количество процессов, считывающих данные из канала, и количество процессов, ведущих запись в канал, совсем не обязательно должны совпадать; если одно число отличается от другого более чем на 1, процессы должны координировать свои действия по использованию канала с помощью других механизмов. Ядро обращается к данным в канале точно так же, как и к данным в обычном файле: оно сохраняет данные на устройстве канала и назначает каналу столько блоков, сколько нужно, во время выполнения вызова `write`.

Приведем пример чтения из канала и записи в канал, реализованный на языке C в ОС Unix:

Листинг 8.1. Пример использования программного канала [16]

```
char string[] = "hello";
main()
{
  char buf[1024];
  char *cp1,*cp2;
  int fds[2];
  cp1 = string;
  cp2 = buf;
  while(*cp1)
    *cp2++ = *cp1++;
   pipe(fds); //создание канала
```

```
 for (;;)
{
  write(fds[1],buf,6);
  read(fds[0],buf,6);
 }
}
```

Программа иллюстрирует искусственное использование каналов. Процесс создает канал и входит в бесконечный цикл, записывая в канал строку символов «hello» и считывая ее из канала.

Желающим более подробно изучить вопросы взаимодействия процессов можно порекомендовать имеющийся в русском переводе фундаментальный труд [30].

8.2. Обозначения и соглашения

Определим величины, являющиеся для модели входными: C — общий объем канала в каких-либо единицах измерения (например, в байтах); N_1 — число процессов, открывших канал на чтение; N_2 — число процессов, открывших канал на запись.

Будем считать, что число процессов, пишущих в канал и читающих из него, постоянно, то есть все процессы, которым нужен этот канал, уже активизированы. Промежутки времени между обращениями процессов к каналу на чтение и на запись распределены экспоненциально с параметрами λ_1 и λ_2 соответственно. Обозначим как $\beta_1(x)$ и $\beta_2(x)$ плотности распределения для заказываемых объемов на считывание и запись соответственно. В приведенном в этой главе примере распределения объемов предполагаются равномерными. Далее будем считать, что время, затрачиваемое процессами на чтение

и запись, прямо пропорционально объему считываемой либо записываемой информации с коэффициентами пропорциональности c_1 и c_2 соответственно. Условимся также, что если в какой-то момент одновременно существуют процессы (хотя бы по одному), сделавшие заказ на чтение и на запись, приоритет отдается обслуживанию процесса-читателя. Разумеется, аналогично строится и модель с приоритетом записи.

В [38], [44] и [45] построена аналитическая модель программных каналов в виде системы интегральных уравнений. Однако, во-первых, состояние канала в ней отслеживается только в моменты завершения операций чтения-записи, во-вторых, что самое главное, численное решение такой системы требует больших теоретических и вычислительных усилий, никак не меньших, чем на имитационное моделирование. К тому же, как уже не раз говорилось, имитационную модель можно безболезненно расширять дополнительными условиями; аналитическую модель в таком случае пришлось бы пересматривать, так как введение дополнительных условий было бы для нее разрушительным. Так, например, аналитическая модель упрощенно трактует ситуацию, когда запрос на чтение или запись удовлетворяется не полностью, а частично: такие запросы все же рассматриваются как удовлетворенные и покидают очередь. Учет остаточной длины неудовлетворенного запроса сразу же сделал бы модель невообразимо сложной и практически бесполезной. В имитационной модели это обстоятельство можно учесть без труда.

8.3. Программная реализация

Для моделирования программного канала в соответствии со сделанным описанием выстроена иерархия классов. Класс `Канал`

(Pipeline), представленный в модели одним объектом, обеспечивает поддержание актуальной информации о заполненности канала, принимает запросы на чтение и запись, регистрирует завершение их выполнения, управляет двумя очередями процессов — читателей и писателей, ставя их в очередь и удаляя из нее по мере необходимости. Случайный процесс изменения объема информации в канале нельзя рассматривать как простой процесс случайных блужданий с отражением на границе. Сложность вызвана тем, что запрос на чтение или на запись может оказаться не полностью удовлетворенным и поэтому возвращается в очередь с остаточной длиной запроса. Такое возвращение может происходить неоднократно. Это обстоятельство не позволяет считать ни один из потоков заявок, увеличивающих или уменьшающих наполненность канала, независимым. Кроме того, даже если бы не удовлетворенные запросы покидали очередь и больше не ждали, все равно ситуация осложнялась бы конечным числом источников запросов, так как количество процессов является конечным. Наличие не полностью удовлетворенных запросов существенно осложняет алгоритмизацию задачи.

Ввиду наличия общей функциональности у классов Читатель (Reader) и Писатель (Writer) они описаны как потомки базового абстрактного класса Client. В классе Client описаны все поля данных, которые наследуют оба потомка, конструктор, метод-диспетчер, а также две чисто виртуальные функции, которые Читатель и Писатель реализуют по-разному. Это методы Request и Complete, моделирующие, соответственно, поступление запроса к каналу и завершение операции чтения/записи. К объекту,

находящемуся в текущий момент времени на обслуживании, канал обращается через указатель на объект базового класса, так как этот объект может быть как читателем, так и писателем.

Статистика собирается по следующим семи показателям:

- средняя длина очереди на чтение;
- средняя длина очереди на запись;
- среднее время пребывания в канале процесса-читателя. Под временем пребывания понимается промежуток времени от момента поступления запроса на чтение до полного его удовлетворения и возвращения процесса в состояние дальнейшего выполнения;
- среднее время пребывания в канале процесса-писателя — аналогично;
- среднее количество раундов обслуживания запроса на чтение. Под раундом обслуживания понимается однократная операция чтения из канала. Если запрос удовлетворен не полностью, раундов будет более одного;
- среднее количество раундов обслуживания запроса на запись — аналогично;
- средняя наполненность канала.

Кроме того, представляется разумным данные о наполненности канала периодически сохранять в файле, чтобы иметь представление о характерных особенностях изменения этой случайной величины во времени.

Листинг 8.2. Файл classes8.h с описанием классов

```
#include<cstdio>
```

```
#include<cstdlib>
#include<ctime>
#include<cmath>
using namespace std;
#define RATIO 10        /*коэффициент
масштабирования времени*/
double read_ave=0;    /*переменная для расчета
средней длины очереди на чтение*/
double write_ave=0;    /*переменная для расчета
средней длины очереди на запись*/
double readT_ave=0;  /*переменная для расчета
среднего времени пребывания (обслуживания запроса
к каналу) читателя*/
double writeT_ave=0;   /*переменная для расчета
среднего времени пребывания (обслуживания запроса
к каналу) писателя*/
double readR_ave=0;  /*переменная для расчета
среднего числа раундов обслуживания читателя*/
double writeR_ave=0;   /*переменная для расчета
среднего числа раундов обслуживания писателя*/
double amount_ave=0;   /*переменная для расчета
средней наполненности канала*/
int read_enter=0;   //счетчик запросов на чтение
int write_enter=0;  //счетчик запросов на запись
FILE *trassa;        /*файл для записи данных о
текущей наполненности канала*/
long int total; /*счетчик тактов модельного
времени*/
```

```
//Базовый абстрактный класс Client
class Client
{
  /*Все поля данных описываются как защищенные
(protected), так как они должны быть доступны в
протоколах производных классов   */
  protected:
//Неизменяемые поля данных
  int alpha;   /*средний объем запроса на
чтение/запись*/
  int beta;   /*максимальное отклонение от
среднего объема запроса на чтение/запись*/
  double input_rate; /*средняя интенсивность
потока запросов на чтение/запись*/
 double serv_rate;     /*коэффициент
пропорциональности для длительности операции
чтения/записи в зависимости от объема запроса*/
  int id;          //уникальный номер процесса
  void *pipe;      //связь с объектом Канал

//Изменяемые поля данных
  int to_request;  /*время до генерации процессом
запроса на чтение/запись (-1, если процесс
находится на обслуживании в канале или в
очереди)*/
  int to_served;    /*время до завершения текущей
операции чтения/записи (-1, если процесс
выполняется)*/
```

```
 int to_satisfy; /*остаточная длина запроса – объем информации, который осталось прочитать /записать для полногоудовлетворения запроса (-1, если процесс выполняется)*/

//Поля данных для сбора статистики
 int timeInPipe;   //текущее время пребывания (-1, если процесс выполняется)*/
 int round;   //текущее количество раундов (-1, если процесс выполняется)*/
  public:
 Client(int e, double a, int b, int c, double d); //конструктор
 virtual void Request()=0;   /*генерация запроса к каналу*/
 virtual int Complete()=0;  /*завершение операции в канале (не означает полного удовлетворения запроса)*/
 void run();                  //диспетчер
 void putPipe(void *p);  /*установление связи с каналом*/
};

/*Конструктор класса Клиент. Аргументы: значения неизменяемых полей данных. */
Client::Client(int e, double a, int b, int c, double d)
{
 id=e;
```

```
 input_rate=a;
 beta=b;
 alpha=c;
 serv_rate=d;
 to_request=(int)(get_exp(input_rate)*RATIO);
 if (to_request==0) to_request=1;
 to_served=-1;
 to_satisfy=-1;
 timeInPipe=-1;
 round=-1;
}
//Диспетчер класса Клиент
void Client::run()
{
 /*Декремент и, если время истекло, – генерация
нового запроса*/
 if (to_request>0) { to_request--; if
(to_request==0) this->Request(); }
 else   /*декремент и, если время истекло,
завершение операции*/
if (to_served>0) {timeInPipe++;to_served--;if
(to_served==0) this-> Complete(); }
 else  timeInPipe++;
}

//Установление связи с Каналом
void Client::putPipe(void *p)
```

```
{
 pipe=p;
}

//Протокол класса Читатель
class Reader: public Client
{
 public:
 friend class Pipeline;   /*канал может
манипулировать процессами. На самом деле это
делает не канал, а ядро операционной системы*/
 Reader(int e, double a, int b, int c, double d);
/*В производных классах эти методы уже будут иметь
реализацию*/
 void Request();
 int Complete();
};
/*Конструктор класса Reader. Полностью замещен
конструктором базового класса*/
Reader::Reader(int e, double a, int b, int c,
double d): Client(e, a,b,c,d) {}

//Генерация процессом-читателем запроса на чтение
void Reader::Request()
{
 to_request=-1;
//Инициализация статистических полей данных
 timeInPipe=0;
```

```
 round=1;
 read_enter++;
//инкремент счетчика запросов на чтение
 to_satisfy=get_uniform(beta, alpha);
//разыгрываем длину запроса
 ((Pipeline*)pipe)->ToRead(this, 'f');
//отправляем запрос в канал.

/*Объяснение второго аргумента вызова – в
реализации класса Канал*/
}
/*Завершение процессом-читателем операции чтения.
Возвращает: 0, если запрос полностью удовлетворен,
1 – в противном случае*/
int Reader::Complete()
{
 to_served=-1;
 if (to_satisfy==0)       //запрос удовлетворен
 {
  to_request=(int)(get_exp(input_rate)*RATIO);
//разыгрываем интервал до следующего запроса
  if (to_request==0) to_request=1;
  to_satisfy=-1;
//Пересчитываем статистические показатели
  readT_ave=readT_ave*(1-1.0/read_enter)
+((double)timeInPipe)/read_enter;
  readR_ave=readR_ave*(1-1.0/read_enter)
+((double)round)/read_enter;
```

```
   ((Pipeline*)pipe)->RComplete(0); /*сообщаем
каналу о завершении операции*/
 }
 else   //запрос не удовлетворен
{
  round++;      //инкремент счетчика раундов
 ((Pipeline*)pipe)->RComplete(1);   /*сообщаем
каналу о завершении операции*/
 }
}

/*Протокол класса Писатель. Отличия от Читателя –
только в реализации чисто виртуальных функций
абстрактного класса*/
class Writer: public Client
{
 public:
 friend class Pipeline;
 Writer(int e, double a, int, int c, double d);
 void Request();
 int Complete();
};

 /*Конструктор класса Writer. Полностью замещен
конструктором базового класса*/
Writer::Writer(int e, double a, int b, int c,
double d): Client(e,a,b,c,d) {}
```

```
/*Генерация запроса на запись. Отличия от метода
Reader::Request выделены*/
void Writer::Request()
{
 to_request=-1;
 timeInPipe=0;
 round=1;
 write_enter++;
 to_satisfy=get_uniform(beta,alpha);
 if (to_satisfy==0) to_satisfy=1;
 ((Pipeline*)pipe)->ToWrite(this,'f');
}
/*Завершение операции записи. Отличия от метода
Reader::Complete выделены*/
int Writer::Complete()
{
 to_served=-1;
 if (to_satisfy==0)
 {
  to_request=(int)(get_exp(input_rate)*RATIO);
  if (to_request==0) to_request=1;
  to_satisfy=-1;
 writeT_ave=writeT_ave*(1-
1.0/write_enter)+((double)timeInPipe)/write_enter;
  writeR_ave=writeR_ave*(1-
1.0/write_enter)+((double)round)/write_enter;
  round=-1;
```

```
  ((Pipeline*)pipe)->WComplete(0);
 }
 else {  round++;
  ((Pipeline*)pipe)->WComplete(1); }
}

//Протокол класса Канал
class Pipeline
{
//Неизменяемые поля данных
 int n_reader;   //количество процессов-читателей
 int n_writer;  //количество процессов-писателей
 int pipeSize;   //размер канала в байтах

//Изменяемые поля данных
 int fill;    //текущий объем информации в канале
 Client *serving; /*указатель на обслуживаемый
процесс*/
 Reader **readers; /*очередь на чтение (массив
указателей на объекты)*/
 Writer **writers;  //очередь на запись (массив
указателей на объекты)*/

 public:
 Pipeline(int a, int N1, int N2);   //конструктор
 ~Pipeline();               //деструктор
```

```
 void ToRead(Reader *a, char c);  /*поступление
запроса на чтение*/
 void ToWrite(Writer *a, char c);   /*поступление
запроса на запись*/
 void RComplete(int k); /*завершена операция
чтения*/
 void WComplete(int k); /*завершена операция
записи*/
 void run();                //диспетчер
 int QLength(char c);  /*вычисление текущей длины
очереди на чтение/запись*/
};

/*Конструктор класса Канал. Аргументы – значения
неизменяемых полей данных*/
Pipeline::Pipeline(int a, int N1, int N2)
{
 int i;
 n_reader=N1; n_writer=N2;
 pipeSize=a;
 fill=0;
 serving=NULL;
 readers=new Reader *[N1];
 for(i=0;i<N1;i++) readers[i]=NULL;
 writers=new Writer *[N2];
 for(i=0;i<N2;i++) writers[i]=NULL;
}
```

```
/*деструктор. Количество объектов-читателей и
объектов-писателей фиксировано. Память для них
выделяется в функции main() и в ней же
возвращается до вызова деструктора класса
Pipeline*/
Pipeline::~Pipeline()
{
 delete [] readers;
 delete [] writers;
}

/*Вычисление текущей длины очереди. c='r' – на
чтение, c='w' – на запись*/
int Pipeline::QLength(char c)
{
 int i;
 if (c=='r')
 {
  for(i=0;i<n_reader;i++)
   if (readers[i]==NULL) return(i);
  return(n_reader);
 }
 else
 {
  for(i=0;i<n_writer;i++)
   if (writers[i]==NULL) return(i);
  return(n_writer);
 }
```

```
}

/*Диспетчер. Никаких моделирующих действий не
выполняет. Собирает статистику по длинам очередей
и заполненности канала. Все методы вызываются из
объектов классов Reader и Writer*/
void Pipeline::run()
{
 int a;
 if ((total+1)%RATIO==0)
 {
  a=(total+1)/RATIO;
  read_ave=read_ave*(1-1.0/a)+(double)
(QLength('r'))/a;
  write_ave=write_ave*(1-1.0/a)+(double)
(QLength('w'))/a;
  amount_ave=amount_ave*(1-1.0/a)
+((double)fill)/a;
  fprintf(trassa, "%d\n", fill);
 }
}
/*Начало новой операции чтения
c='f' – процесс «a» в момент вызова метода не
находился в очереди к каналу
c='r' – процесс «a» в момент вызова метода
находился первым в очереди к каналу*/
void Pipeline::ToRead(Reader *a, char c)
{
```

```
 int k;
 if (c=='f')   /*как бы то ни было - указатель на
объект добавляется в конец очереди, даже если она
пуста. При значении аргумента c='r' указатель на
объект уже находится в очереди*/
 {
  k=QLength('r');
  readers[k]=a;
 }
/*Если канал занят другим процессом или в нем нет
данных - возврат. Чтение откладывается, процесс
блокируется, так как его запрос не может быть
сразу удовлетворен*/
 if ((serving!=NULL)||(fill==0)) return;
 else if (fill>=a->to_satisfy)  /*операция чтения
может быть начата сразу, и запрос может быть
полностью удовлетворен*/
 {
  a->to_served=(int)((a->serv_rate)*(a-
>to_satisfy)*RATIO);   /*вычисляем длительность
операции чтения*.
  if (a->to_served==0) a->to_served=1;
/*Следующие два действия мы можем сделать уже
сейчас, поскольку операция чтения/записи не может
быть прервана*.
  fill-=(a->to_satisfy);
  a->to_satisfy=0;
  serving=a;   //процесс начинает чтение
 }
```

```
 else  /*операция чтения может быть начата сразу,
но запрос после ее завершения не будет
удовлетворен*/
 {
  a->to_satisfy-=fill;
  a->to_served=(int)((a->serv_rate)*fill*RATIO);
  if (a->to_served==0) a->to_served=1;
  fill=0;   /*после завершения операции чтения
канал будет пуст*/
  serving=a;
 }
}
/*Начало новой операции записи. Комментарии
аналогичны Pipeline::ToRead.*/
void Pipeline::ToWrite(Writer *a, char c)
{
 int k, i;
 if (c=='f')
 {
  k=QLength('w');
  writers[k]=a;
 }
 if ((serving!=NULL)||(fill==pipeSize))    return;
  else if ((pipeSize-fill)>=(a->to_satisfy))
 {
  a->to_served=(int)((a->serv_rate)*(a-
>to_satisfy)*RATIO);
  if (a->to_served==0) a->to_served=1;
```

```
  fill+=(a->to_satisfy);
  a->to_satisfy=0;
  serving=a;
 }
 else
 {
  a->to_satisfy-=(pipeSize-fill);
  a->to_served=(int)((a->serv_rate)*(pipeSize-
fill)*RATIO);
  if (a->to_served==0) a->to_served=1;
  fill=pipeSize;
  serving=a;
  }
}
/*Завершение операции чтения. Аргумент: k=0 –
запрос процесса удовлетворен, k=1 – запрос
процесса не удовлетворен*/
void Pipeline::RComplete(int k)
{
 int i;
 serving=NULL;
 if (k==0) /*запрос удовлетворен. Процесс покидает
очередь*/
 {
  for(i=0;i<(n_reader-1);i++)   /*продвижение
очереди*/
   readers[i]=readers[i+1];
```

```
  readers[n_reader-1]=NULL;
/*Если очередь на чтение не пуста и в канале есть
информация, реализуем приоритет запросов на
чтение. Выделение объяснено далее в тексте главы*/
  if ((QLength('r')>0)&&(fill!=0))
ToRead(readers[0],'r');
  else if (QLength('w')>0)
ToWrite(writers[0],'r');
 }
/*Запрос не удовлетворен, канал пуст. Пытаемся в
него что-то записать*/
 else if (QLength('w')>0)  ToWrite(writers[0],
'r');
}
/*Завершение операции записи. Комментарии
аналогичны Pipeline::RComplete*/
void Pipeline::WComplete(int k)
{
 int i;
 serving=NULL;
 if (k==0)
 {
  for(i=0;i<(n_writer-1);i++)
   writers[i]=writers[i+1];
  writers[n_writer-1]=NULL;
  if (QLength('r')>0)   ToRead(readers[0],'r');
  else if ((QLength('w')>0)&&(fill!=pipeSize))
ToWrite(writers[0],'r');
```

```
 }
 else
 {
  if (QLength('r')>0)  ToRead(readers[0],'r');
 }
}
```

Прокомментируем выделенный фрагмент в методе `Rcomplete()`. Он является прекрасной иллюстрацией того, насколько тщательно нужно подходить к переносу логики работы моделируемой системы в алгоритмическую реализацию методов классов. Предположим, что выделенное условие отсутствует. При этом может возникнуть следующая ситуация: очередь на чтение не пуста, канал пуст, очередь на запись не пуста. В этом случае метод `Pipeline::ToRead()` будет вызван, но чтения не произойдет, так как читать нечего. В этой ситуации, разумеется, должен получить доступ к каналу стоящий первым в очереди процесс-писатель по ветви `else`, но этого не произойдет, так как формально выполнилась ветвь `if`. Таким образом, имитационная модель уже сейчас поведет себя не так, как надо. Если на следующем такте сгенерирует запрос на чтение еще один процесс-читатель, этот запрос получит доступ к каналу, пройдя «поверх» запроса, находящегося в очереди.

Листинг 8.3. Функция main()

```
#include "classes8.h"
#define G 1000000     /*общая продолжительность
моделирования (в условных единицах времени)*/
int main()
```

```
{
 Reader **r;
 Writer **w; int RR, WW, i;
 trassa=fopen("trassa", "wt");
 RR=2; WW=2;
 srand((unsigned)time(0));
 Pipeline p(4096, RR, WW);
 r=new Reader *[RR];
 w=new Writer *[WW];
 for(i=0;i<RR;i++)
 {
  r[i]=new Reader(i+1, 0.05, 55, 45, 0.04);
  r[i]->putPipe(p); /*устанавливаем связь
процессов-читателей  с каналом*/
 }
 for(i=0;i<WW;i++)
 {
  w[i]=new Writer(i+1, 0.04, 55, 45, 0.08);
  w[i]->putPipe(p); /*устанавливаем связь
процессов-писателей с каналом*/
 }
  for(total=0L;total<G;total++)
  {
   /*первыми опрашиваются процессы-читатели. Если
запросы от читателя и писателя будут выданы в
одном такте модельного времени, приоритет будет
поэтому отдан читателю, что и должно быть по
условию*/
```

```
    for(i=0;i<RR;i++)
    r[i]->run();
   for(i=0;i<WW;i++)
    w[i]->run();
   p.run();
  }
/*Выдача на печать результатов моделирования*/
.............................
/*освобождение памяти*/
 for(i=0;i<RR;i++)  delete r[i];
 for(i=0;i<WW;i++)  delete w[i];
 delete [] r;
 delete [] w;
//закрытие файлов
.....
}
```

8.4. Анализ результатов

Расчет проводился при следующих значениях входных данных: число процессов-читателей — 2, число процессов-писателей — 2, объем канала — 4096, $\lambda_1 = 0{,}05$, $\lambda_2 = 0{,}04$, $c_1 = 0{,}04$, $c_2 = 0{,}08$, длины запросов на чтение и на запись распределены равномерно на интервале от 10 до 90. Усредненные результаты 1000 прогонов таковы:

- средняя длина очереди на чтение — 0,675;
- средняя длина очереди на запись — 0,345;

- среднее время пребывания в системе запроса на чтение — 10,182 единиц времени;
- среднее время пребывания в системе запроса на запись — 5,2 единиц времени;
- среднее число раундов чтения — 1,325;
- среднее число раундов записи — 1;
- средняя наполненность канала — 139.

Трасса наполненности канала и зависимости характеристик функционирования от числа процессов-читателей (при двух процессах-писателях) и числа процессов-писателей (при пяти процессах-читателях) изображены на рис. 8.1–8.9.

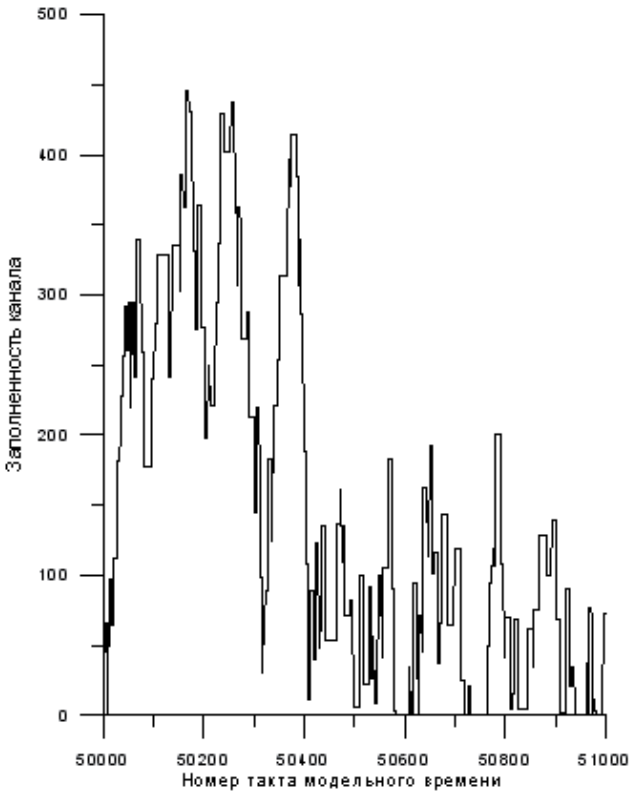

Рис. 8.1. Трасса наполненности канала на промежутке от 50 до 51 тыс. единиц модельного времени

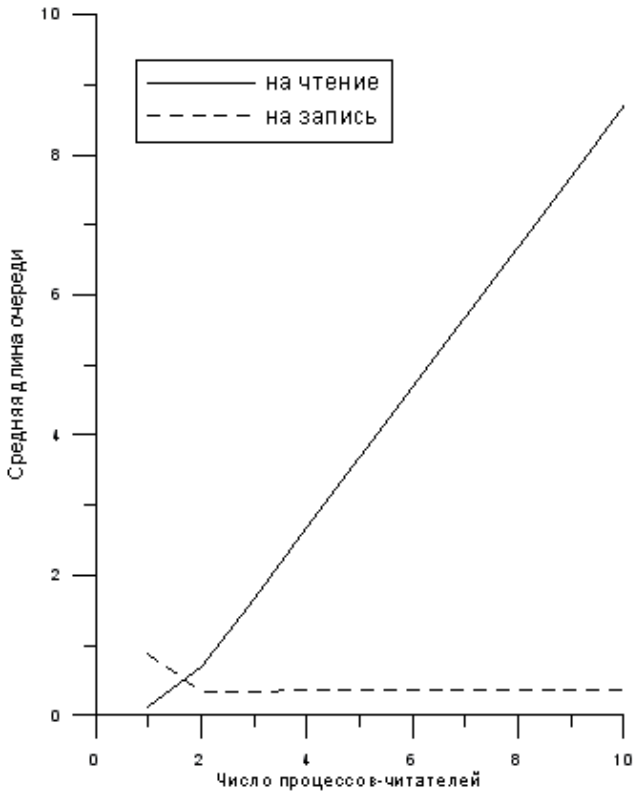

Рис. 8.2. Зависимости средней длины очереди на чтение/запись от числа процессов-читателей, 2 процесса-писателя

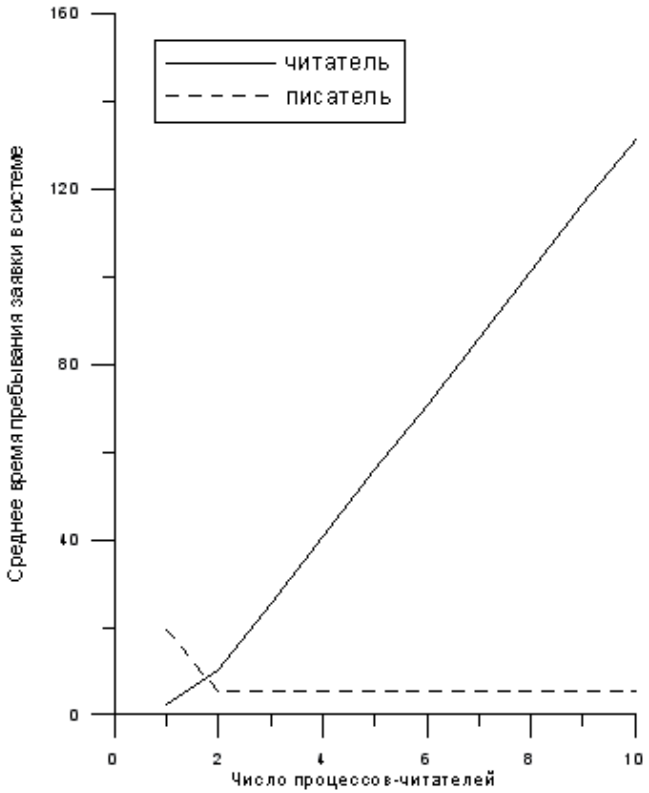

Рис. 8.3. Зависимости среднего времени пребывания в системе запроса на чтение/запись от числа процессов-читателей, 2 процесса-писателя

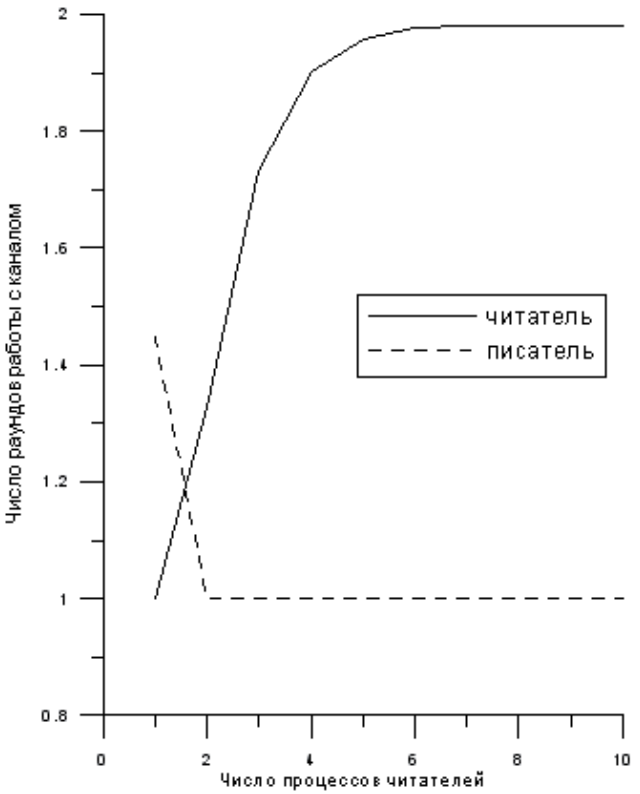

Рис. 8.4. Зависимости среднего числа раундов чтения/записи от числа процессов-читателей, 2 процесса-писателя

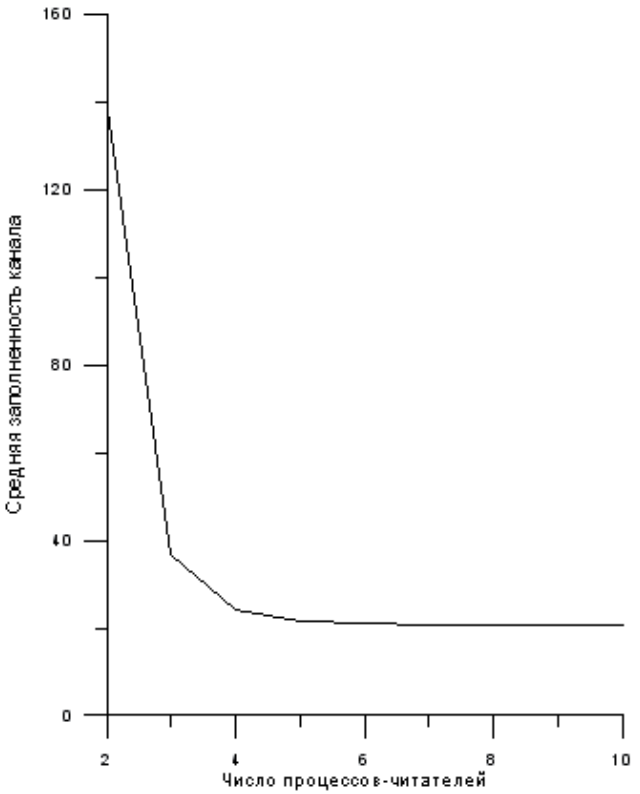

Рис. 8.5. Зависимость средней наполненности канала от числа процессов-читателей, 2 процесса-писателя

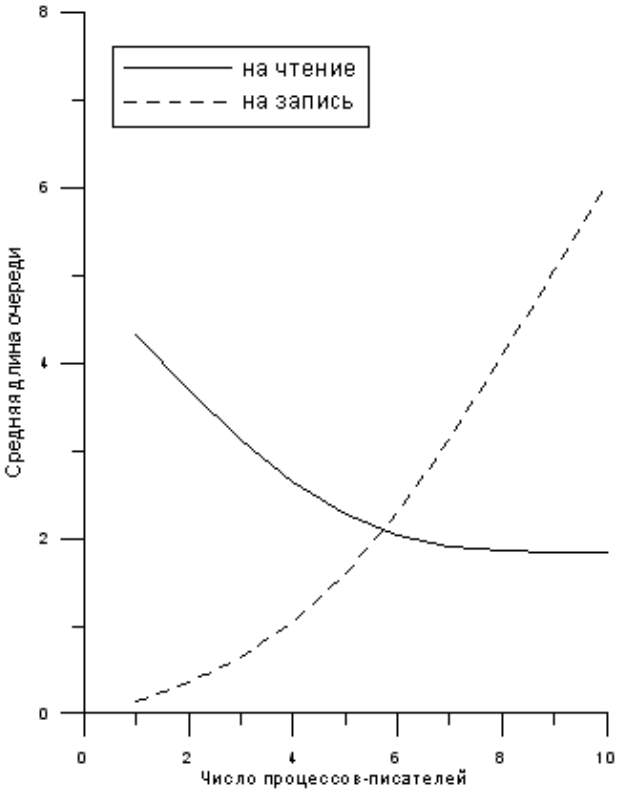

Рис. 8.6. Зависимости средней длины очереди на чтение/запись от числа процессов-писателей, 5 процессов-читателей

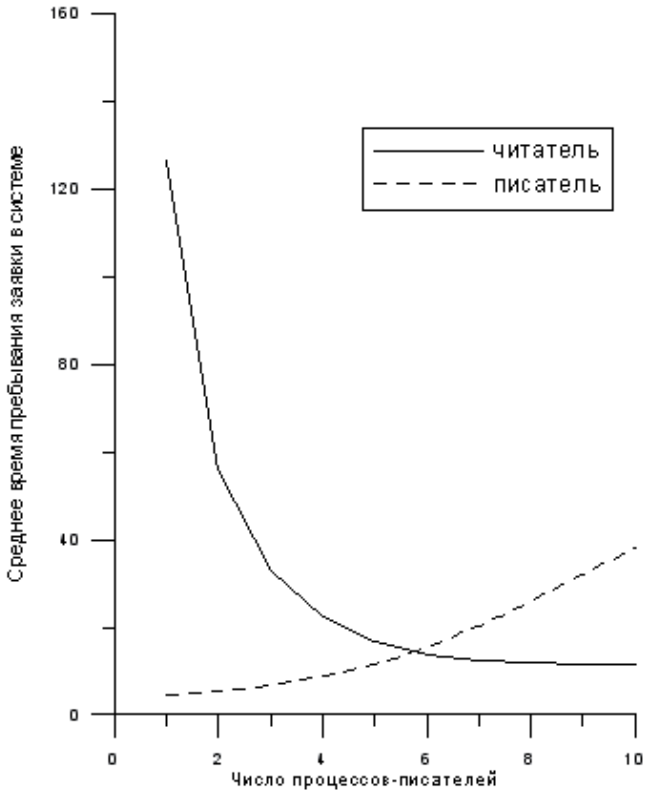

Рис. 8.7. Зависимости среднего времени пребывания в системе запроса на чтение/запись от числа процессов-писателей, 5 процессов-читателей

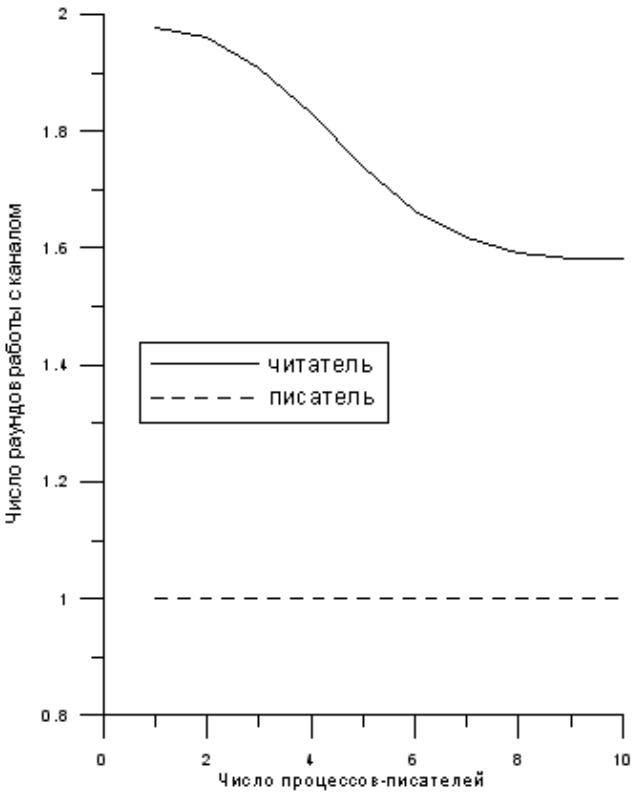

Рис. 8.8. Зависимости среднего числа раундов чтения/записи от числа процессов-писателей, 5 процессов-читателей

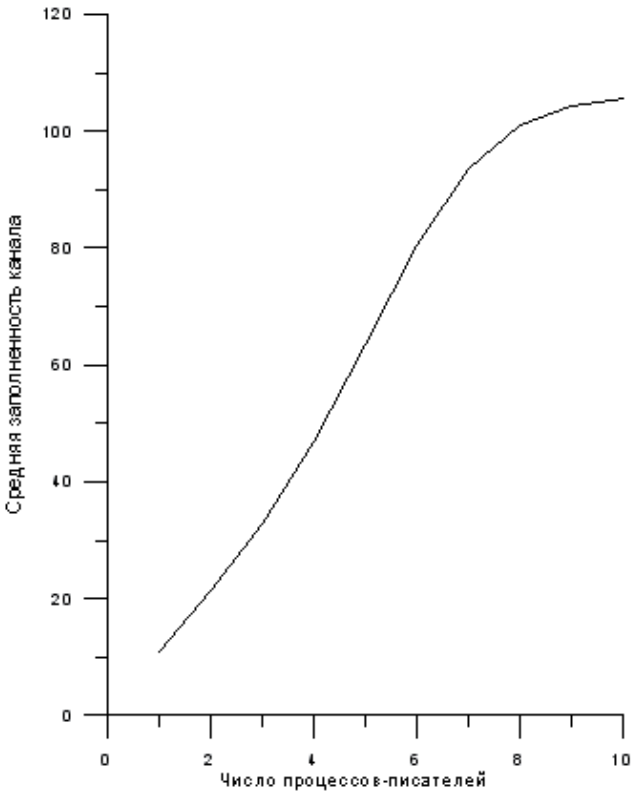

Рис. 8.9. Зависимость средней наполненности канала от числа процессов-писателей, 5 процессов-читателей

Задания для самостоятельной работы

1. Модифицируйте модель для случая, когда приоритетной операцией является запись. Запустите ее со значениями входных данных, перечисленными в п.8.4, и сравните результаты.
2. Модифицируйте модель для случая, когда не полностью удовлетворенный запрос не остается в очереди в ожидании

повторного раунда чтения/записи, а покидает ее, как будто он удовлетворен полностью. Сравните результаты, оценив таким образом погрешность аналитической модели.

3. Измените функцию распределения длины заявки на чтение/запись. Задайте вместо равномерного распределения ограниченное распределение Парето:

$$\Pr(X \le x) = \begin{cases} 0, & x \le 0; \\ k\left(1-(x+1)^{-\alpha}\right), & 0 < x < \beta, \; k = \dfrac{1}{1-(\beta+1)^{-\alpha}}; \\ 1, & x \ge \beta, \end{cases}$$

частный случай которого при $\alpha = -1$ соответствует равномерному распределению.

Математическое ожидание равно $\frac{(\beta+1)^{-\alpha}(\alpha\beta+1)-1}{(1-\alpha)\left(1-(\beta+1)^{-\alpha}\right)}$. Параметры $\beta = 100$, $\alpha = -1{,}235$ сохраняют максимальную длину равной 100 и среднюю длину заявки равной 55. Напишите ГСЧ для этого распределения и сравните результаты моделирования с результатами для равномерного распределения.

Глава 9. Явление синхронизации в однородных системах со слабой связью

9.1. Предварительные сведения

Тенденция динамических систем к синхронизации при наличии слабой связи между ними выявлена очень давно. Еще в середине XVII в. знаменитый механик и астроном Х. Гюйгенс заметил, что двое маятниковых часов синхронизируют свои колебания, если висят на одной стене, видимо, под влиянием ее едва уловимой вибрации. Явление синхронизации отмечено в электронных схемах, многих механических объектах и биологических системах, таких как клеточные популяции и сообщества светлячков. Большинство этих систем проявляют указанную тенденцию независимо от значений физических констант и начальных условий [4]. Эти факты наводят на мысль о том, что сложносвязанные системы, такие как современные компьютерные сети, могут сами по себе приходить к состоянию упорядоченности и синхронизации. Там, где синхронизация вредна, например, при сильно коррелированном всплескообразном сетевом трафике, усилия сетевых инженеров и разработчиков протоколов направлены на ее недопущение, то есть «ломку» естественного механизма ее возникновения.

Значительная доля трафика в компьютерных сетях поступает из периодических источников, и наиболее характерным их примером, который мы далее и будем рассматривать, является обмен сообщениями между маршрутизаторами. Можно предположить, что независимость источников периодического трафика вызывает независимость и некоррелированность суммарного трафика. Даже если каждый маршрутизатор будет генерировать пакеты с

интервалом, к примеру, ровно в 30 с, итоговый трафик, измеренный в любой точке сети, должен быть гладким и равномерным, так как источники находятся на разных узлах сети и стартовали в случайные моменты времени. Однако на самом деле общий трафик является не равномерно распределенным, а наоборот, сильно синхронизированным. Интуитивное предположение о некоррелированности суммарного трафика от независимых источников оказывается неверным.

Наиболее известным и давно применяемым сетевым протоколом, использующим периодический обмен сообщениями, является RIP (Routing Information Protocol), который до сих пор является основным в сетях небольших размеров с простой топологией. Полная документация по протоколу содержится в [76], разумеется, описан он и в книгах отечественных авторов [19], [23]. RIP был разработан для универсального протокола PARC Xerox и использовался в комплекте протоколов XNS. Этот протокол начали связывать как с Unix, так и с TCP/IP в 1982 г., когда версию Unix, называемую Berkeley Standard Distribution (BSD), начали отгружать с одной из реализаций RIP, названной routed. RIP был повсеместно принят производителями персональных компьютеров для использования в системах передачи данных по сети. Например, протокол маршрутизации AppleTalk является модернизированной версией RIP. RIP также явился базисом для протоколов Novell, 3Com, Ungermann-Bass и Banyan.

Протокол RIP является дистанционно-векторным протоколом внутренней маршрутизации. В протоколах этого типа каждый маршрутизатор периодически и широковещательно рассылает по сети вектор, компонентами которого являются расстояния от данного

маршрутизатора до всех известных ему сетей. Под расстоянием здесь понимается целое число, равное количеству непосредственных передач между маршрутизаторами-соседями (еще говорят «число *хопов»)*. Иными словами, это количество ребер графа, расположенных на пути от одной вершины к другой. Возможна и другая метрика, учитывающая не только число промежуточных маршрутизаторов, но и время прохождения пакетов по сети между соседними маршрутизаторами. При получении вектора от соседа маршрутизатор наращивает расстояния до указанных в векторе сетей на расстояние до данного соседа. Получив вектор, каждый маршрутизатор добавляет к нему информацию об известных ему других сетях, о которых он узнал непосредственно (если они подключены к его портам) или из аналогичных объявлений других маршрутизаторов, а затем рассылает новое значение вектора по сети. Результатом работы протокола на конкретном маршрутизаторе является таблица, где указаны расстояние до каждой сети данной RIP-системы (в хопах) и адрес следующего маршрутизатора.

Протокол RIP очень прост, но имеет недостатки, которые не позволяют применять его в обширных и сложных сетях. Перечислим основные из них:

- эффект «счета до бесконечности» ограничивает размер RIP-системы четырнадцатью промежуточными маршрутизаторами [19] в любом направлении. Если расстояние доходит до шестнадцати (т.е. 14 промежуточных маршрутизаторов), дальнейшие попытки переслать пакет прекращаются, и адресуемая сеть считается недостижимой. Это значение было выбрано разработчиками протокола и входит в его

спецификацию. По этой же причине весьма затруднительно использовать сложные метрики;

- периодическая широковещательная рассылка векторов расстояний ухудшает пропускную способность сети;
- длительность последовательного формирования таблиц маршрутизации довольно велика, а сам результат не всегда оптимален с точки зрения равномерного распределения нагрузки (подробнее это вопрос рассмотрен в [37]);
- несмотря на то что каждый маршрутизатор начинает периодическую рассылку своих векторов, вообще говоря, в случайный момент времени, через некоторое время в системе наблюдается уже известный нам эффект синхронизации маршрутизаторов, сходный с эффектом синхронизации аплодисментов. Все или большая часть маршрутизаторов рассылают свои вектора в один и тот же момент времени, что вызывает большие пики трафика и отказы в маршрутизации дейтаграмм во время обработки большого количества одновременно полученных векторов.

9.2. Модель периодической рассылки

Дадим формальное описание модели периодической рассылки сообщений, которой с теми или иными вариациями следуют протоколы EGP (Exterior Gateway Protocol), Hello, IGRP и RIP. В этих протоколах каждый маршрутизатор периодически рассылает свою таблицу маршрутов в сети. Это гарантирует, что маршрутные таблицы всегда будут содержать актуальную информацию, обеспечивающую прохождение пакета от источника к адресату по кратчайшему пути.

1. Маршрутизатор готовит и рассылает сообщение. При

отсутствии входящих сообщений переустанавливает таймер спустя T_c секунд после начала шага 1. Остальные маршрутизаторы получают первый пакет сообщения спустя T_d секунд после завершения подготовки сообщения.

2. Если маршрутизатор получает входящее сообщение во время подготовки собственного исходящего, то он обрабатывает это сообщение, для чего ему требуется время T_{c2}.
3. После завершения шагов 1 и 2, маршрутизатор устанавливает значение своего таймера. Время, по окончании которого маршрутизатор вновь перейдет к шагу 1 (период рассылки), равномерно распределено на интервале $[T_p - T_r,\ T_p + T_r]$ секунд, где T_p — средняя величина периода, T_r — случайная модификация для моделирования возможной задержки операционной системы. По истечении этого интервала маршрутизатор переходит к выполнению шага 1.
4. Если маршрутизатор получает входящее сообщение после установки таймера, он немедленно приступает к его обработке. Если в ее результате изменилось содержание маршрутной таблицы (например, в сообщении содержалась информация о разрыве какой-то связи), маршрутизатор немедленно приступает к выполнению шага 1. Подобная ситуация носит название triggered update.

Таким образом, T_p — это постоянная составляющая периода рассылки сообщений, T_r — максимальное значение случайной составляющей. Каждый маршрутизатор затрачивает T_c секунд на подготовку исходящего сообщения и T_{c2} — на обработку входящего. Далее всюду предполагается, что эти две величины имеют одинаковые значения.

Через T_d секунд после завершения подготовки исходящего сообщения маршрутизаторы, непосредственно связанные с данным, получают первый пакет.

Единственной известной автору работой, в которой синхронизация периодических рассылок исследуется аналитическими методами, является [64]. В ней построена марковская модель, с помощью которой можно вычислить ожидаемое время перехода системы из несинхронного состояния в синхронное и наоборот. Эта модель отражает наиболее характерные особенности поведения системы, но использует ряд упрощающих предположений, которые в реальности могут не иметь места. Основным понятием работы (мы также будем им пользоваться) является термин *кластер* — так называется количество синхронизированных маршрутизаторов. В процессе функционирования сеть, состоящая из N маршрутизаторов, может оказаться разделенной на несколько кластеров разных размеров. Предельные случаи: 1 кластер размером N — соответствует полной синхронизации; N кластеров размером 1 — соответствует полностью несинхронизированной сети. В дальнейшем, говоря о кластере, мы будем подразумевать кластер максимального размера, так как именно он характеризует, в какой степени система синхронизирована в данный момент. Размер максимального кластера принят в [64] в качестве состояния марковского случайного процесса.

Упрощающие предположения, на которых базируется марковская модель, таковы:

- дальнейшее поведение системы полностью определяется ее текущим состоянием и не зависит от предыстории. Это предположение, без которого невозможно построение цепи

Маркова, все же не вполне справедливо для модели периодической рассылки, так как ее будущее поведение зависит не только от размера кластера, но и от времени пересылки других маршрутных сообщений;

- марковский процесс является процессом размножения и гибели, то есть размер кластера может измениться только на единицу при переходе в следующий раунд (под раундом понимается интервал между двумя последовательными установками таймера). На самом деле это не всегда так, особенно при больших N (что подтверждается дальнейшими результатами имитационного моделирования). Два кластера размерами i и 2, где i значительно больше двух, могут объединиться (точнее, второй будет поглощен первым), сформировав таким образом кластер размером $i + 2$;
- при вычислении вероятностей переходов сделано предположение о том, что все кластеры, кроме максимального, имеют размер 1, то есть являются обособленными маршрутизаторами.

И наконец, разработанная в [64] вероятностная модель годится только для полносвязных сетей, в которых каждый маршрутизатор связан с каждым. Сделать ее настраиваемой на конкретную топологию сети невозможно. Для имитационной модели, в отличие от аналитической, это не вызывает никаких затруднений.

9.3. Имитационная модель

Прежде чем перейти к описанию программной реализации, уточним, как мы будем интерпретировать шаг 4 алгоритма периодической рассылки сообщений, описанного в п.9.2. Если входящее сообщение

не несет в себе информацию об изменении топологии сети и не приводит к модификации локальной таблицы маршрутизатора, оно просто обрабатывается в течение времени $T_{c2} = T_c$, и больше ничего не происходит. Если же происходит событие triggered update, ситуация усложняется. Возникает вопрос: каким образом моделировать — повлекло входящее сообщение событие triggered update или нет? Примем простейшее решение — в числе параметров модели зададим еще одну константу Z — вероятность того, что входящее сообщение повлекло за собой событие triggered update и требует досрочной переустановки таймера.

Так как сеть — это граф, то ее удобно представлять с помощью матрицы смежности. *i*-я строка этой матрицы является набором нулей и единиц, в котором единицы соответствуют маршрутизатором, непосредственно связанным с *i*-м.

В программе описаны два класса — `Observer`, представленный одним объектом, и `Router`, представленный N объектами. Всю непосредственную работу по моделированию выполняют объекты класса `Router`. Класс `Observer` служит для управления процессом моделирования сети в целом и выполняет следующие функции:

- организация взаимодействия между функцией `main()` и классом `Router`. Все параметры маршрутизаторов передаются сначала единственному объекту `Observer`, который и создает объекты класса `Router`, передавая эти параметры его конструктору. Функция `main()` с классом `Router` не работает;
- хранение информации, относящейся ко всей сети в целом, — о количестве маршрутизаторов и топологии (матрице смежности);

- сбор статистики по размеру кластера.

Так как `Observer` напрямую работает с объектами класса `Router`, он объявляется дружественным этому классу. Так как общее количество маршрутизаторов фиксировано, их совокупность моделируется массивом указателей. Начальный адрес этого массива входит в качестве поля данных как в класс `Router`, так и в `Observer`. Обратная связь не требуется, так как `Router` не посылает никаких сообщений классу `Observer`.

Собираемая программой статистика включает следующие данные:

- файл, в который с заданной периодичностью записывается текущий размер кластера;
- *N* файлов, в каждый из которых с заданной периодичностью записывается текущее значение таймера для соответствующего маршрутизатора;
- средний размер кластера за весь период моделирования

Топология сети вводится из матрицы смежности, хранящейся в обычном текстовом файле. Имя этого файла передается конструктору класса `Observer` в качестве одного из аргументов.

Листинг 9.1. Файл classes9.h с описанием классов Observer и Router

```
#include<cstdio>
#include<cstdlib>
#include<ctime>
#include<cmath>
#include<cstring>
#include<algorithm>
```

```
using namespace std;
#define RATIO 100 /*коэффициент масштабирования времени*/
double cl_ave=0;    /*переменная для подсчета среднего размера кластера*/
long int total;//счетчик тактов модельного времени
FILE *cl_size; /*файл для записи текущего размера кластера*/
/*Первым должен быть описан класс Router, так как класс Observer использует указатель на него в качестве одного из своих полей данных*/
class Router
{
//Неизменяемые поля данных
 int id;         //уникальный номер маршрутизатора
 int N;         //количество маршрутизаторов в сети
 double Tp;
 double Tr;
 double Tc;
 double Tc2;
 double Td;
 const int Z;    //целое число от 0 до 100
 Router **network;  /*массив указателей на маршрутизаторы. Необходим при вычислении адреса объекта для пересылки периодических сообщений*/
 int *neighbours;  /*список маршрутизаторов-соседей. Является копией соответствующей строки матрицы смежности*/
```

```
//Изменяемые поля данных
 int timer; /*время, прошедшее с момента установки
таймера. Равенство этого показателя у двух
объектов есть признак того, что они
синхронизированы*/
 int to_TP;  /*время, оставшееся до окончания
периода*/
 int to_outgo;  /*время, оставшееся до завершения
пары шагов 1 и 2*/
 int to_register;  /*время, оставшееся до
завершения обработки входящих сообщений*/
 int *to_TD;  /*массив. i-й элемент - время,
оставшееся до получения входящего сообщения от i-
го маршрутизатора*/
 public:
 friend class Observer;   /*класс Observer -
дружественный классу Router*/
 Router(int a,int b,double c,double d,double
e,double f,double g, Router **h, int *i, int k);
~Router();
 void Attention(int i);
 void Arrival(int i);
 void Register();
 void Expires();
 void Send();
 void run();
};
```

```
/*Конструктор. Вызывается из конструктора класса
Observer. Аргументы: a, b, c, d, e, f, g, h, i,
k – соответственно, значения id, N, Tp, Tr, Tc,
Tc2, Td, network, neighbours, Z*/
Router::Router(int a,int b,double c,double
d,double e,double f,double g, Router **h, int *i
int k)
{
 int j;
 id=a; N=b; Tp=c; Tr=d; Tc=e; Tc2=f; Td=g; Z=k;
 network=h; neighbours=i;
 to_outgo=-1; to_register=-1;
 to_TD=(int*)malloc(N*sizeof(int));
 for(j=0;j<N;j++)
  to_TD[j]=-1;
/*Первоначально маршрутизаторы несинхронизированы
- для каждого разыгрываем значение таймера
независимым случайным образом*/
 to_TP=get_uniform((int)(Tp*RATIO/2),
(int)(Tp*RATIO/2));
 timer=(int)(Tp*RATIO)-to_TP;
}
Router::~Router() { free(to_TD); }
/*Вызывается при рассылке исходящего сообщения
соседям. i – номер маршрутизатора-адресата,
Включает счетчик времени, через которое нужно
начинать обработку входящего сообщения*/
void Router::Attention(int i)
{
```

```
 to_TD[i]=(int)(Td*RATIO);
}
/*Прибытие входящего сообщения от маршрутизатора с
номером i*/
void Router::Arrival(int i)
{
 if (to_register==-1)
to_register=(int)(Tc2*RATIO);   //ранее прибывшее

/*входящее сообщение в этот момент не
обрабатывается*/
 else to_register+=(int)(Tc2*RATIO);
/*обрабатывается другое, ранее прибывшее входящее
сообщение*/
 if (to_outgo!=-1) to_outgo+=(int)(Tc2*RATIO);
//в это время выполняется шаг 1
 to_TD[i]=-1;
//выключаем счетчик прибытия
}
//Обработка входящего сообщения завершена
void Router::Register()
{
 int k;
 to_register=-1;
 if (to_outgo==-1)   //не на шаге 1
 {
  k=rand()%100;
  if (k<Z)
```

```
  {
   to_TP=get_uniform((int)(Tp*RATIO),
(int)(Tr*RATIO));
   timer=0;
  }
 }
}
//Окончание периода. Переход к шагу 1
void Router::Expires()
{
 to_TP=-1;
 if (to_register==-1) to_outgo=(int)(Tc*RATIO);
/*входящее сообщение в этот момент не
обрабатывается*/
 else  to_outgo=to_register+(int)(Tc*RATIO);
//обрабатывается прибывшее ранее входное сообщение
}
//Завершение шага 1. Рассылка сообщения соседям
void Router::Send()
{
 int i;
//Переустановка таймера
 to_TP=get_uniform((int)(Tp*RATIO),
(int)(Tr*RATIO));
 timer=0;
 to_outgo=-1;
 for(i=0;i<N;i++)
 {
```

```
/*Если i-й маршрутизатор – сосед, отправляем ему
сообщение*/
  if (neighbours[i]==1)
   network[i]->Attention(id-1);
 }
}

//Метод-диспетчер
void Router::run()
{
 int i;
 timer++;             //инкремент таймера
 for(i=0;i<N;i++)  /*моделирование прибытия
входящих сообщений*/
 {
  if (to_TD[i]>0) to_TD[i]--;
  if (to_TD[i]==0) Arrival(i);
 }
/*Декременты изменяемых полей данных и вызовы
методов. Важное замечание: метод Register()
обязательно должен вызываться раньше //метода
Send(), иначе моделирование будет некорректным*/
 if (to_register>0) to_register--;
 if (to_register==0)  Register();
 if (to_outgo>0) to_outgo--;
 if (to_outgo==0)  Send();
 if (to_TP>0) to_TP--;
 if (to_TP==0)  Expires();
```

```
}

//Протокол класса Observer
class Observer
{
 int N;       //количество маршрутизаторов в сети
 int *topology;      //топология сети
 Router **network;   /*массив указателей на
объекты Router*/
 FILE **trassa;    /*массив указателей на файлы
трассирования поля данных timer*/
 public:
 Observer(int a, char *b, double c, double d,
double e, double f, double g, int h);
 int Cluster();
 void run();
 ~Observer();
};

/*Конструктор. Аргумент b – имя файла с матрицей
смежности, остальные аргументы – значения
неизменяемых полей данных класса Router*/
Observer::Observer(int a, char *b, double c,
double d, double e, double f, double g, int h)
{
 FILE *data;
 int i,j,k;
 char str[10], str1[10];
```

```
 N=a;
/*Читаем из файла данные и заполняем матрицу
смежности*/
 data=fopen(b, "rt");
 topology=(int*)malloc(N*N*sizeof(int));
 for(i=0;i<N;i++)
  for(j=0;j<N;j++)
  {
   fscanf(data,"%d", &k);
   topology[N*i+j]=k;
  }
 fclose(data);
 trassa=(FILE**)malloc(N*sizeof(FILE*));
 strcpy(str,"router");
//Открываем на запись файлы с именами router1,
router2, ..., routerN
 for(i=0;i<N;i++)
 {
  str1=itoa(i+1,str1,10);   /*преобразование
целого числа в строку*/
  strcat(str, str1);    //формирование имени файла
  trassa[i]=fopen(str,"wt");
  strcpy(str,"router");
 }
 network=new Router *[N];
/*Создаем объекты Router. Topology+N*i – адрес i-й
строки матрицы  смежности, содержащей связи i-го
маршрутизатора*/
```

```
 for(i=0;i<N;i++)
  network[i]=new
Router(i+1,N,c,d,e,f,g,network,topology+N*i, h);
}

//Деструктор. Освобождает память, закрывает файлы
Observer::~Observer()
{
 int i;
 free(topology);
 for(i=0;i<N;i++)
  fclose(trassa[i]);
 free(trassa);
 for(i=0;i<N;i++)
  delete network[i];
 delete(network);
}

/*Вычисление текущего максимального размера
кластера  */
int Observer::Cluster()
{
 int mas[N], maxi, i, j, k, s;
//копируем значения таймеров
 for(i=0;i<N;i++)
  mas[i]=network[i]->timer;
```

```
/*Упорядочиваем значения таймеров по возрастанию.
Одинаковые значения в результате группируются.
Используем функцию sort() из библиотеки STL
sort(mas,mas+N);*/
/*Выделяем группу рядом стоящих одинаковых
значений максимальной длины*/
 maxi=-1; i=1; s=1;
 while(i<N)
 {
  if ((s==1)||(mas[i]==mas[i-1]))
  {
   s++;
   i++;
  }
  else
  {
   if (s>maxi) maxi=s;
   s=1; i++;
  }
 }
 if (s>maxi) maxi=s;
 return(maxi);
}

//Диспетчер
void Observer::run()
{
```

```
 int i,l, j;
//Моделирование поведения маршрутизаторов
 for(i=0;i<N;i++)
  network[i]->run();
//Регистрация статистики на каждом сотом такте
 if ((total+1)%RATIO==0) {
 l=(total+1)/RATIO;
 j=Cluster();
 fprintf(cl_size, "%d\n", j;
 for(i=0;i<N;i++)
  fprintf(trassa[i], "%f\n", (double)(network[i]-
>timer)/RATIO);
 cl_ave=cl_ave*(1-1.0/l)+((double)j)/l;
 }
}
```

Листинг 9.2. Функция main()

```
#include "classes9.h"
#define G 10000000
int main()
{
 Observer o=new Observer(20, "matr", 121, 0.1,
0.11, 0.11, 0, 50);
 cl_size=fopen("cluster", "wt");
 srand((unsigned)time(0));
 for(total=0; total<G; total++)
    o.run();
```

```
  printf("Средний размер синхронизированного
кластера -  %f\n", cl_ave);
  fclose(cl_size);
 }
```

9.4. Анализ результатов

Для моделирования в качестве базовых взяты следующие значения параметров [59]: $N = 20$, $T_p = 121$, $T_r = 0{,}1$, $T_c = T_{c2} = 0{,}11$, $T_d = 0$, $Z = 0{,}5$. Модельное время промасштабировано с коэффициентом 100. Общее количество тактов модельного времени — 10 млн.

Исследовались три топологии: полносвязная (каждый маршрутизатор непосредственно связан с каждым), «кольцо» (рис. 9.1) и «звезда» (рис. 9.2).

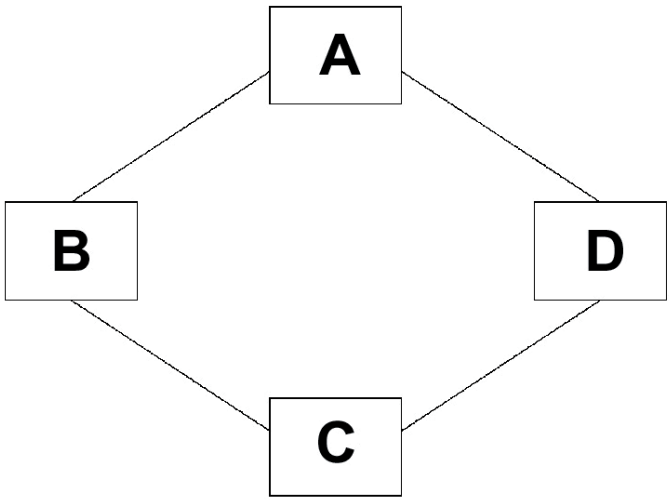

Рис. 9.1. Сетевая топология «кольцо»

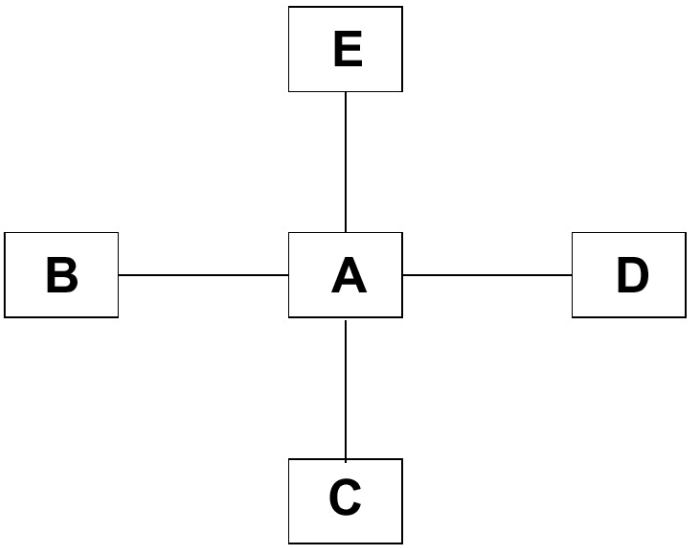

Рис. 9.2. Сетевая топология «звезда»

На рис. 9.3 показана зависимость среднего размера кластера от размера сети *N*. Из графиков видно, что наибольшую тенденцию к синхронизации (линейный рост) имеет полносвязная топология, для остальных данное явление нехарактерно. При увеличении значения T_c (рис. 9.4) размер кластера для всех трех топологий стремится к двум, то есть синхронизация исчезает. Весьма интересны всплески синхронизации для топологий «кольцо» и «звезда». Особенно загадочен резкий пик синхронизации для топологии «звезда» при $T_c = 6$. Он не случаен — при многочисленных экспериментах с различным шагом изменения T_c обнаруживает устойчивую тенденций к повторению, достигая значений от 12 до 13. Объяснения этого явления у автора (так же, видимо, как и у других исследователей) пока нет.

Для всех трех топологий характерны довольно резкие изменения размера кластера при небольших увеличениях T_c. Зависимости среднего объема кластера от вероятности Z (рис. 9.5) обнаруживают те же особенности, что и зависимости от N. Мы видим, что при $Z = 100$ (вероятность — единица) в среднем синхронизируются 15 маршрутизаторов из 20.

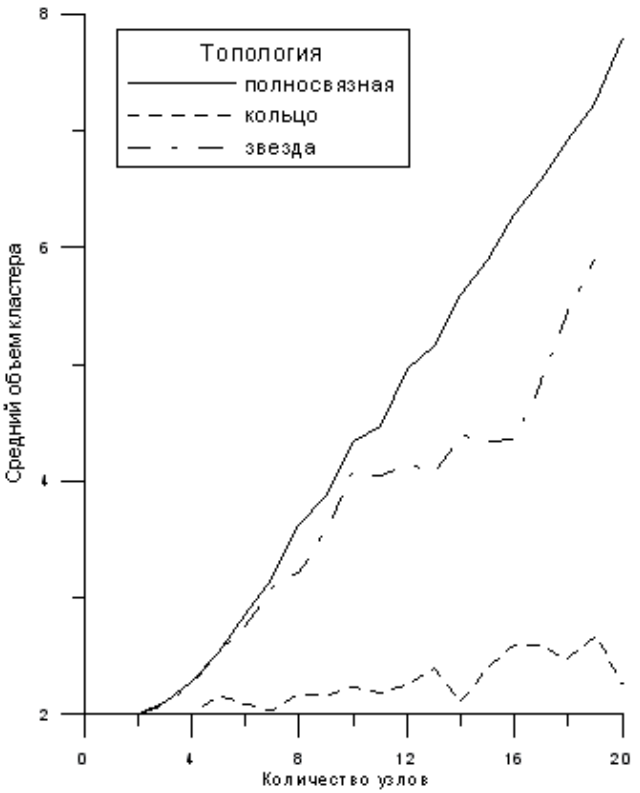

Рис. 9.3. Зависимость среднего размера кластера от количества маршрутизаторов

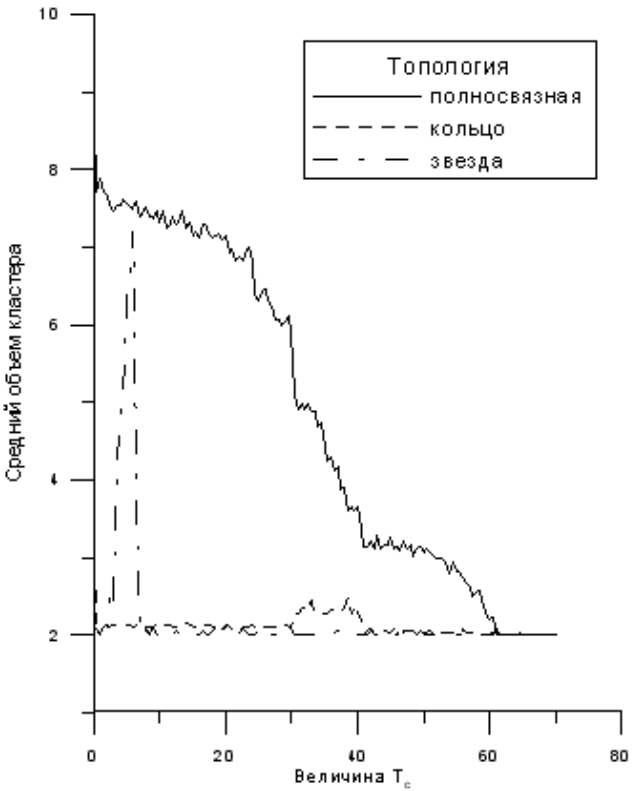

Рис. 9.4. Зависимость среднего размера кластера от длительности подготовки исходящего и обработки входящего сообщений

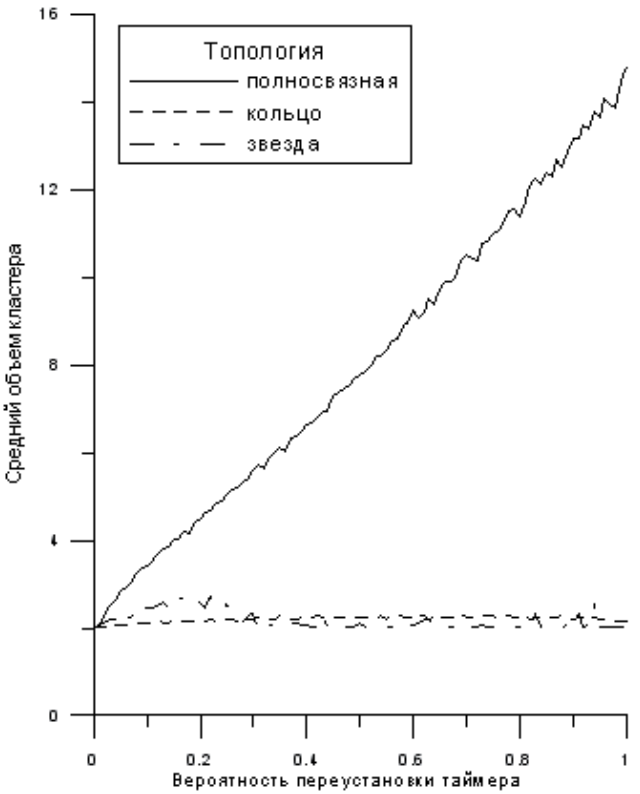

Рис. 9.5. Зависимость среднего размера кластера от вероятности события triggered update

Таким образом, можно сделать интуитивно понятный вывод о том, что явление синхронизации выражено тем сильнее, чем больше степень связности сети, то есть чем больше непосредственно связанных соседей имеет в среднем каждый маршрутизатор. При $N = 20$ это значение составляет: для полносвязной топологии — 19, для топологии «кольцо» — 2, для топологии «звезда» — 1.9.

Задания для самостоятельной работы

1. Перепишите код конструктора класса `Router` таким образом, чтобы в начальном состоянии все маршрутизаторы были синхронизированы. Проведите эксперименты при различных значениях параметров и ответьте на вопрос: всегда ли синхронизация «разбивается» на более мелкие кластеры или система может так и оставаться синхронизированной. Влияние каких параметров на этот процесс является определяющим?
2. Поэкспериментируйте с зависимостями размера кластера от параметра T_c для топологии «звезда» при различных значениях N. Постройте зависимость точки всплеска синхронизации от N, попытайтесь найти закономерность и выдвинуть собственную гипотезу, объясняющую это явление.

Приложение 1. Реализация time-driven подхода для календарно-событийной программы из главы 2

```
#define n0 6
#define n1 3
#define nt n0+n1
#define nd 4
int n[2];
int s[2];
double tc[2]={10.0, 5.0 }, td=30.0, sd=2.5;
struct token_info
{
 int cls;
 double ts;
};
int nts=500;

/*Класс Facility описывает многоканальный прибор с
общей очередью и дисциплиной обслуживания в
порядке поступления*/
class Facility
{
 int id;               //номер прибора
 int volume;          //количество каналов
 int *to_complete;  /*время, оставшееся до
завершения обслуживания на каждом канале (-1, если
канал свободен)*/
```

```
 struct token_info **serving;  /*массив указателей
на обслуживаемые заявки*/
 struct token_info **queue; /*массив указателей на
заявки, находящиеся в очереди*/
 Facility *other;     /*прибор, на обслуживание к
которому далее переходят заявки*/
 public:
//Конструктор
 Facility(int a,int b,struct token_info **c,
struct token_info **d, int *e);
void Complete(int i); /*i-ый канал завершил
обслуживание*/
 void Arrival(struct token_info *h);   /*прибытие
новой заявки*/
 void run(); //диспетчер
 void putOther(Facility *a);    /*установление
связи со следующим прибором*/
};
Facility::Facility(int a,int b,struct token_info
**c, struct token_info **d, int *e)
{
 int i;
 id=a;
 volume=b;
 to_complete=(int*)malloc(volume*sizeof(int));
 serving=(struct
token**)malloc(volume*sizeof(struct token*));
```

```
 queue=(struct token**)malloc(nt*sizeof(struct
token*));
 for (i=0;i<volume;i++)
 {
  to_complete[i]=e[i];
  serving[i]=c[i];
 }
 for (i=0;i<nt;i++)
  queue[i]=d[i];
}
void Facility::putOther(Facility *a)
{
 other=a;
}
void Fuller::Arrival(struct token_info *h)
{
 int i, choice;
 choice=-1;
//определяем первый свободный канал
 for (i=0;i<volume;i++)
 {
  if (to_complete[i]==-1)
  {
   choice=i;
   break;
  }
 }
```

```
//Свободный канал найден
 if (choice>=0)
 {
  if (id==0)
//...если прибор - ЦПУ
   to_complete[choice]=expnt(tc[h->cls]);
  else to_complete[choice]=erlang(td, sd);
//...если прибор - диски
  serving[choice]=h;
//...ставим прибывшую заявку на обслуживание
 }
/*Прибор – ЦПУ, он занят, но прибывшая заявка
имеет более высокий приоритет*/
 else if ((id==0)&&(h->cls==1)&&(serving[0]-
>cls==0))
 {
 /*Обслуживаемая заявка возвращается в голову
очереди*/
  for(i=nt-1;i>0;i--) queue[i]=queue[i-1];
  queue[0]=serving[0];
//Прибывшая заявка ставится на обслуживание на ЦПУ
  to_complete[0]=expnt(tc[1]);
  serving[0]=h;
 }
/*Прибывшая заявка ставится в конец очереди из-за
отсутствия каналов*/
 else
 {
```

```
  for (i=0;i<nt;i++)
  {
   if (queue[i]==NULL)
   {
    queue[i]=h;
    break;
   }
  }
 }
}

void Facility::Complete(int i)
{
 int i;
 if (id==1)   /*заявка завершила обслуживание на
одном из дисков*/
 {
 //Фиксация статистики по завершившемуся циклу
  s[serving[i]->cls]+=serving[i]->ts;
  n[serving[i]->cls]++;
  serving[i]->ts=0;
 }
/*Генерируем событие – поступление этой заявки к
следующему устройству*/
 other->Arrival(serving[i]);
/*Если очередь не пуста, ставим на обслуживание
новую заявку и сдвигаем  очередь*/
```

```
 if (queue[0]!=NULL)
 {
  if (id==0)
   to_complete[i]=expnt(tc[queue[0]->cls]);
  else to_complete[i]=erlang(td, sd);
  serving[i]=queue[0];
  for(i=0;i<nt-1;i++)
   queue[i]=queue[i+1];
  queue[nt-1]=NULL;
 }
 else //очередь пуста, канал объявляется свободным
{
 to_complete[i]=-1; }
 serving[i]=NULL;
}

void Facility::run()
{
 int i;
/*Для всех заявок на устройстве (обслуживаемых и
стоящих в очереди) выполняем инкремент
длительности текущего цикла*/
 for(i=0;i<volume;i++)
  if (to_complete[i]>0) serving[i]->ts++;
 for(i=0;i<nt;i++)
 {
  if (queue[i]==NULL) break;
```

```
  queue[i]->ts++;
 }
/*Выполняем декремент счетчика времени
обслуживания, в случае завершения – генерируем
событие*/
 for(i=0;i<volume;i++)
 {
  if (to_complete[i]>0) to_complete[i]--;
  if (to_complete[i]==0) Complete(i);
 }
}

int main()
{
 struct token_info task[nt], **serv1, **serv2,
**que1, **que2;
 int *mas1, *mas2, i;
 n[0]=n[1]=0;
 s[0]=s[1]=0.0;
//Инициализация заявок
 for(i=0;i<nt;i++)
 {
  task[i].cls=(i>=n0) ? 1:0;
  task[i].ts=0;
 }
/*Инициализация объектов. В начальном состоянии
первая заявка начинает обслуживаться на ЦПУ,
```

```
остальные стоят в очереди к ЦПУ. Очередь к дискам
пуста*/
 mas1=(int*)malloc(sizeof(int));
 mas1[0]=expnt(tc[task[0].cls]);
 mas2=(int*)malloc(nd*sizeof(int));
 for(i=0;i<nd;i++)
  mas2[i]=-1;
 serv1=(struct token_info**)malloc(sizeof(struct
token_info*));
 serv1[0]=task[0];
 serv2=(struct
token_info**)malloc(nd*sizeof(struct
token_info*));
 for(i=0;i<nd;i++)
  serv2[i]=NULL;
 que1=(struct token_info**)malloc(nt*sizeof(struct
token_info*));
 for(i=0;i<nt-1;i++)
  que1[i]=task[i+1];
 que1[nt-1]=NULL;
 que2=(struct token_info**)malloc(nt*sizeof(struct
token_info*));
 for(i=0;i<nt;i++)
  que2[i]=NULL;
 Facility cpu(0,1,serv1,que1,mas1);
 Facility disks(1,4,serv2,que2,mas2);
//Взаимная настройка связи между ЦПУ и дисками
 cpu.putOther(&disks);
```

```
 disks.putOther(cpu);
//Основной цикл
 while(n[0]+n[1]<=nts)
 {
  cpu->run();
  disks->run();
 }
 printf("class 0 tour time=%.2f\n",
((float)s[0])/n[0]);
 printf("class 1 tour time=%.2f\n",
((float)s[1])/n[1]);
}
```

Приложение 2. Файл List.h. Шаблон связного списка и алгоритмы его обработки

```
template <class Type> /*это постоянная «заставка»
к классам и функциям с параметризованным типом*/
class ListNode  {
 private:
  ListNode<Type> *next; /*указатель на следующий
элемент списка*/
  Type *data;  /*указатель на данные, хранящиеся в
элементе списка*/
 public:
   ListNode(Type *d, ListNode<Type> *n);
//конструктор
  ~ListNode();
//деструктор
  Type *Data();   //метод для чтения данных
  ListNode<Type> *Next();  /*метод для чтения
указателя на следующий элемент*/
  void  PutNext(ListNode<Type> *n); /*метод для
записи указателя на следующий элемент.Альтернатива
```

```
методу – объявление функций ListAdd и ListDelete
дружественными классу ListNode*/
  void Print();  /*печать содержимого элемента
списка*/
};
template <class Type>
ListNode<Type>::ListNode(Type *d, ListNode<Type>
*n) : next(n), data(d)
{
}
template <class Type>
ListNode<Type>::~ListNode()
{
 delete data;
}
template <class Type>
Type *ListNode<Type>::Data()
{
 return data;
}
template <class Type>
ListNode<Type> *ListNode<Type>::Next()
{
 return next;
}
template <class Type>
void ListNode<Type>::PutNext(ListNode<Type> *n)
{
 next=n;
}
template <class Type>
void ListNode<Type>::Print()
{
 data->Print();   /*предполагается наличие метода
Print() для класса, имя которого будет подставлено
в пользовательском коде*/
}
```

```
/*Описание класса-шаблона завершено, далее идут
функции-шаблоны, работающие не с отдельным
элементом, а со всем списком*/
template <class Type>
void ListAdd(ListNode<Type> *head, ListNode<Type>
*li)
/*Добавление нового элемента li в хвост списка с
головой head*/
{
 ListNode<Type> *old_tail, *v;
// ищем нынешний хвост списка
 for (v=head; v!=NULL; v=v->Next())
  old_tail=v;
 old_tail->PutNext(li);   /*добавляем вслед за
найденным хвостом новый элемент*/
}
template <class Type>
ListNode<Type> *ListDelete(ListNode<Type> *head,
ListNode<Type> *li)
/*Удаление элемента li из списка с головой head.
Функция возвращает указатель на голову нового
списка*/
{
 int j;
 ListNode<Type> *old, *o1;
 if (li==head)   /*удаляемый элемент может быть
головой списка,в этом случае голова у  списка
меняется*/
 {
  o1=head->Next();
  delete li;
  return o1;
 }
/*Удаляемый элемент не является головой списка.
Голова остается прежняя*/
 for (ListNode<Type>* v=head; v!=li; v=v->Next())
/*поиск элемента, предшествующего удаляемому*/
  old=v;
 o1=li->Next();
```

```
 old->PutNext(o1);   /*предшествующий элемент
теперь «видит» элемент, стоявший в списке вслед за
удаляемым*/
  delete li;
 return head;
}
//Печать всех элементов списка с головой head
template <class Type>
void ListPrint(ListNode<Type> **head)
{
 for (ListNode<Type>* v=head; v!=NULL; v=v->
Next())
    v->Print();
 }
/*Подсчет количества элементов в списке с головой
head*/
template <class Type>
int ListCount(ListNode<Type> *head)
{
 int i; i=0;
 for (ListNode<Type>* v=head; v!=NULL; v=v->
Next())
   i++;
 return i;
}
```

Например, элемент списка объектов класса `Field` можно создать с помощью описанного шаблона так:

```
Field *p = new Field('a',1);
ListNode<Field> *st=new ListNode<Field>(p, NULL);
```

Список литературы

1. Авен О. И., Гурин Н. Н., Коган Я. А. Оценка качества и оптимизация вычислительных систем. — М.: Наука, 1982.
2. Артамонов Г. Т., Брехов О. М. Аналитические вероятностные модели функционирования ЭВМ. — М.: Энергия, 1978.
3. Барлоу Р. Х. Оценка производительности параллельных алгоритмов // Системы параллельной обработки / Под ред. Д. Ивенса. — М.: Мир, 1985.
4. Блехман И.И. Синхронизация в природе и технике. — Москва, Наука, 1981.
5. Брехов О., Морару В. Определение граничной производительности ЭВМ, управляемой потоком данных // Автоматика и телемеханика — 1992. — № 6. — С. 173-181
6. Бэбб Р. Параллельная обработка крупноструктурированных потоков данных на CRAY X-MP // СуперЭВМ: аппаратная и программная реализация / Под ред. С. Фернбаха. М.: Радио и связь, 1991. — С. 290–304.
7. Вагнер Г. Основы исследования операций: В 3 т. Т. 3. — М.: Мир, 1973.
8. Васильев Ф. П. Численные методы решения экстремальных задач. — М.: Наука, 1980.
9. Гилмор П. Высокопараллельный процессор MPP // СуперЭВМ: аппаратная и программная организация / Под ред. С. Фернбаха. — М.: Радио и связь, 1991. — С. 215–266.
10. Голованов О. В., Дуванов В. Г., Смирнов В. Н. Моделирование сложных дискретных систем на ЭВМ третьего поколения (опыт применения GPSS). — М.: Энергия, 1978.

11. Грэхем Р., Кнут Д., Паташник О. Конкретная математика: основание информатики. — М.: Мир, 1998.
12. Дегтярев Е. К. Оценка средней скорости многосекционной памяти ЦВМ // Изв. АН СССР. Сер. Техническая кибернетика — 1970. — № 2. — С. 128-133.
13. Игнатущенко В. В. Организация структур управляющих многопроцессорных вычислительных систем. — М.: Энергия, 1984.
14. Кельтон В., Лоу А. Имитационное моделирование. — СПб.: Питер, 2004. (Сер. Классика CS.)
15. Клейнрок Л. Теория массового обслуживания. — Л.: Машиностроение, 1979.
16. Корнильев В., Шидяев К. Unix — коммуникационные возможности и средства // Технологии электронных коммуникаций. — 1992. — Т. 17
17. Коуги П. Архитектура конвейерных ЭВМ. — М.: Радио и связь, 1985.
18. Крылов В. И., Скобля Н. С. Методы приближенного преобразования Фурье и обращения преобразования Лапласа. — М.: Наука, 1974.
19. Кульгин М. Технологии корпоративных сетей. — СПб.: Питер, 2000.
20. Майерс Г. Архитектура современных ЭВМ: В 2 т. Т. 2. — М.: Мир, 1982. 38
21. Миура К. СуперЭВМ фирмы Fujitsu: векторная система FACOM // СуперЭВМ: аппаратная и программная организация / Под ред. С. Фернбаха. — М.: Радио и связь, 1991. — С. 166–184.
22. Миура К., Учида К. Векторный процессор FACOM //

Высокоскоростные вычисления. Архитектура, производительность, прикладные алгоритмы и программы суперЭВМ / Под ред. Я. Ковалика. — М.: Мир, 1988. — С.112-121.

23. Олифер В. Г., Олифер Н. А. Компьютерные сети: принципы, технологии, протоколы. — СПб.: Питер, 2000.
24. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем. — М.: Мир, 1991.
25. Остланд Н. С., Хиббэрд П. Г., Вайтсайд Р. А. О применении сильносвязанных многопроцессорных систем для научных вычислений // Параллельные вычисления / Под ред. Г. Родрига. — М.: Наука, 1986. — С.295-340.
26. Приемы объектно-ориентированного проектирования: паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. — СПб.: Питер, 2001.
27. Риккарди Г. Системы баз данных: теория и практика использования в Internet и среде Java. — М.: Вильямс, 2001.
28. Семишин Ю. А., Гуржий В. П., Литвинова О. К. Моделирование дискретных систем на ДАСИМ. — М.: Моя Москва, 1995.
29. Сир Ж.-К. Метод потока операндов в многопроцессорных системах типа MIMD // Системы параллельной обработки / Под ред. Д. Ивенса. М.: Мир, 1985. — С.240-276.
30. Стивенс У. UNIX: взаимодействие процессов. — СПб.: Питер, 2002.
31. СУБД Cache: объектно-ориентированная разработка приложений / В. Кирстен, М. Ирингер, Б. Рериг, П. Шульте. — СПб.: Питер, 2001.
32. Триливен Ф. К. Модели параллельных вычислений // Системы

параллельной обработки / Под ред. Д. Ивенса. М.: Мир, 1985. — С.277-284.

33. Труб И. Алгоритмическое обеспечение распределенных WEB-серверов // Открытые системы. — 2003. — № 5. — С.49-54.
34. Труб И. И. Об оптимальной стратегии генерирования результатов запросов к Internet-серверу баз данных // Автоматика и телемеханика — 2003. —№ 6 — С.95-103.
35. Труб И.И. Объектно-ориентированное моделирование на C++. Учебный курс – СПб, Питер, 2006. – 416 с.
36. Труб И.И. СУБД Cache: работа с объектами. – М., Диалог-МИФИ , 2006 – 480 с.
37. Труб И. И., Белова С. А. Об оптимальной последовательности включения интерфейсов в сетях, управляемых дистанционно-векторным протоколом маршрутизации // Сб. науч. тр. Сургут. гос. ун-та. — Вып. № 11. — Сургут: Изд-во Сургут. гос. ун-та, 2002. — С. 172–180.
38. Труб И. И., Цуканова О. Г. Вероятностная модель функционирования программного канала как средства взаимодействия параллельных процессов // Сб. науч. тр. Сургут. гос. ун-та. — Вып. № 11. — Сургут: Изд-во Сургут. гос. ун-та, 2002. — С. 181–190.
39. Труб Н.В. О парадоксе нарушения стационарного режима при моделировании web-кластера с дисциплиной обслуживания sita-e. – Вестник Московского государственного гуманитарно-экономического института. - № 2(18), 2014. – с. 98-102.
40. Фельдман Л. П., Дедищев В. А. Математическое обеспечение САПР: моделирование вычислительных и управляющих систем. — Киев: УМК ВО, 1992.

41. Фельдман Л. П., Труб И. И. Определение и анализ характеристик вычислительной системы, управляемой потоком операндов // Электронное моделирование. — 1995. — № 1. — С. 61-65.
42. Фролов А., Фролов Г. Базы данных в Интернете. — М.: Русская редакция, 2000.
43. Хокни Р., Джессхоуп К. Параллельные ЭВМ: архитектура, программирование и алгоритмы. — М.: Радио и связь, 1986.
44. Цуканова О. Г. Математическая модель функционирования программных каналов // Науч. тр. Донецк. гос. техн. ун-та. Сер. Информатика, кибернетика и вычислительная техника (ИКВТ-99). — Вып. 6. — Донецк: Изд-во Донецк. гос. техн. ун-та, 1999. — С. 85–90.
45. Цуканова О.Г. Организация межпроцессного информационного обмена с помощью очереди сообщений в технологии «клиент-сервер» // Науч. тр. Донецк. гос. технич. ун-та. Сер. Проблемы моделирования и автоматизации проектирования динамических систем. — Вып. 10. — Донецк: Изд-во Донецк. гос. техн. ун-та, 1999. — С. 83-88.
46. Чан Т. Системное программирование на C++ под UNIX. — Киев: BHV, 1997.
47. Яшков С. Ф. Анализ очередей в ЭВМ. — М.: Радио и связь, 1989.
48. Abrams M. Parallel discrete event simulation: fact or fiction? // ORSA Journal on Computing. — 1993. — № 5(3). — P. 231–233.
49. Arlitt M. F., Williamson C. L. Internet Web servers: Workload characterization and implications // IEEE/ACM Trans. on Networking. — 1997. — № 5(5). — P. 631–644.

50. Arlitt M. F., Williamson C. L. Web Server Workload Characterization: The Search for Invariants // Proc. of the ACM SIGMETRICS '96 Conference. — Philadelphia, PA, 1996.
51. Bourke T. Server Load Balancing. — O'Reilly, 2001.
52. Buehrer R., Ekanadham K. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution // IEEE Transactionson Computers. — 1987. — № 12. — P. 1515-1522.
53. Burnett G. J., Coffman E. J. Analysis of Interleaved Memory Systems Using Blockage Buffers // Commun. Ass. Comput. Mach. — 1975. — № 2. — P. 91-95.
54. Cardellini V., Casalicchio E., Colajanni M. A Performance Study of Distributed Architectures for the Quality of Web Services // Proc. 34th Hawaii International Conference on System Sciences (HICSS34). — Maui, HI, 2001.
55. Crovella M. E., Harchol-Balter M., Murta C. D Task assignment in a distributed system: Improving performance by unbalancing load // Proc. of the Joint international conference on measurement and modeling of computer systems. — Madison, WI, USA, 1998. — P. 268–269.
56. Crovella M. E., Taqqu M. S., Bestavros A. Heavy-Tailed Probability Distributions in the World Wide Web // A Practical Guide To Heavy Tails. Ch. 1. — New York: Chapman & Hall, 1998. — P. 3–26.
57. Dahlin M. Interpreting stale load information // The 19th IEEE International Conference on Distributed Computing Systems (ICDCS). — Austin, TX, 1999. — P. 285–296.
58. De-Lei L. Architecture of an Array Processor Using a Nonlinear Skewing Scheme // IEEE Transactions on Computers. — 1992. — № 4. — P. 499–505.

59. Dennis J. B. Data Flow Supercomputer // IEEE Computer. — 1980. — № 11. — P. 48–56.
60. Dennis J. B., Gao C.-R., Todd K. W. Modeling the Weather with a Data Flow Supercomputer // IEEE Transactions on Computers. — 1984. — № 7. — P. 592–603.
61. Discrete-Event Simulation. 3rd ed. / J. Banks, J. S. Carson, B. L. Nelson, D. M. Nicol. — Prentice Hall: Englewood Cliffs, NS, 2000.
62. Feldmann A., Whitt W. Fitting Mixtures of Exponentials to Long-Tail Distributions to Analyze Network Performance Models // Performance Evaluation. — 1998. — № 31. — P. 245–254.
63. Fishwick P. A. SimPack: getting started with simulation programming in C and C++ // Technical Report TR 92-022, Computer and Information Sciences / University of Florida. — Gainesville, Florida, 1992.
64. Floyd S., Jacobson V. The Synchronization of Periodic Routing Messages // IEEE/ACM Transactions on Networking. — 1994. — Vol. 2, № 2. — P. 122–136.
65. Flynn M. J. Some Computer Organizations and Their Effectiveness // IEEE Transactions on Computers. — 1972. — № 9. — P. 948–960.
66. Fujimoto R. M. Parallel discrete event simulation // Communications of the ACM. — 1990. — № 33(10). — P. 30–59.
67. Fujimoto R. M. Parallel discrete event simulation: will the field survive? // ORSA Journal on Computing. — 1993. — № 5(3). — P. 213–230.
68. Gaudiot J.-L. Structure Handling in Data-Flow System // IEEE Transactions on Computers. — 1986. — № 6. — P. 489–502.

69. Gaudiot J.-L., Wei Y.-H. Token Relabeling in Tagged Token Data-Flow Architecture // IEEE Transactions on Computers. — 1989. — № 9. — P. 1225–1239.
70. Ghosal D., Bhuyan L. N. Performance Evaluation of Dataflow Architecture // IEEE Transactions in Computers. — 1990. — № 5. — P. 615–627.
71. Gostelow K. P., Thomas R. E. Performance of a Simulated DataFlow Computer // IEEE Transactions on Computers. — 1980. — № 10. — P. 905–919.
72. Harchol-Balter M. Task assignment with unknown duration // Journal of the ACM, vol.49, No.2, March, 2002. — P.260–288.
73. Harchol-Balter M., Crovella M. E., Murta C. D. On Choosing a Task Assignment Policy for a Distributed Server System // Journ. of Parallel and Distributed Computing, Special Issue on Software Support for Distributed Computing, vol.59, No.2, September, 1999. — P. 204–228.
74. Harchol-Balter M., Crovella M., Murta C. To queue or not to queue?: When FCFS is better than PS in a distributed system // Technical Report, BUCS-TR-1997-017, October, 31, 1997.
75. Harper D. T., Jump J. R. Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme // IEEE Transactions on Computers. — 1987. — № 12. — P. 1440–1449.
76. Hedrick C. Routing Information Protocol // Request for Comments (RFC). — 1988. — № 1058.
77. Hoogendorn C. H. A General Model for Memory Interference in Multiprocessors // IEEE Transactions on Computers. — 1977. — № 10. — P. 998–1005.
78. Jen C. W., Kwai D. M. Data Flow Representation of

IterativeAlgorithms for Systolic Arrays // IEEE Transactions on Computers. — 1992. — № 3. — P. 351–355.

79. Jess A. G., Keesh G. M. A Data Structurefor Parallel L/U decomposition // IEEE Transactions on Computers. — 1982. — № 3. — P. 231–239.

80. Kopparapu C. Load Balancing Servers, Firewalls and Caches. — New York: John Wiley&Sons, 2002.

81. Misra J. Distributed discrete event simulation // ACM Computing Surveys. — 1986. — № 18(1). — P. 39–65.

82. Mitzenmacher M. How Useful is Old Information // IEEE Transactions on Parallel and Distributed Systems. — 2000. — № 11(1). — P. 6–20.

83. Network Dispatcher: a connection router for scalable Internet service / G. Hunt, G. Goldszmidt, R. King, R. Mukherjee // Computer Networks and ISDN Systems. — 1998. — Vol. 30. — P. 347–357.

84. O'Leary D., Stewart G. Data-Flow Algorithms for Parallel Matrix Computations // Commun ACM. — 1985. — № 6. — P. 840–853.

85. Oed W., Lange O. On theEffective Bandwidth of Interleaved Memories in Vector Processor Systems // IEEE Transactions on Computers. — 1985. — № 10. — P. 949–957.

86. Overeinder B. J., Hertzberger L. O., Sloot P. M. A. Parallel Discrete Event Simulation // The Third Workshop Computer Systems. — May, 1991. — Eindhoven, The Netherlands. — P. 19–30.

87. Paxson V., Floyd S. Wide-area Traffic: The Failure of Poisson Modeling // IEEE/ACM Transactions on Networking. — 1995. — June. — P. 226–244.

88. Rau B. R. Program Behavior and the Performance of Interleaved Memories // IEEE Transactions on Computers. — 1979. — № 3. —

P. 191–199.

89. Self-Similar Network Traffic and Performance Evaluation / Ed. by K. Park, W. Wilinger. — New York: John Wiley&Sons, 2000.

90. Sethi A. S., Deo N. Interference in a Multiprocessor System with Localized Memory Access Probabilities // IEEE Transactions on Computers. — 1979. —№ 2. — P. 157–163.

91. Smilauer B. General Model for Memory Interference in Multiprocessors and Mean Value Analysis // IEEE Transactions on Computers — 1985. — № 8. — P. 744–751.

92. Survey of languages and runtime libraries for parallel discrete-event simulation / Y.-H. Low, Ch-C. Lim, W. Cai, at al. // Simulation. — 1999. — № 72(3). — P. 170–186.

93. Terman F. W. A Study of Interleaved Memory Systems by Trace Driven Simulation // Proceedings of the 4th Symposium on Simulation of Computer System. — Boulder, Colorado, United States, 1976. — P. 3–9.

94. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Machine / A. Gottlieb, R. Grishman, C. P. Kruskal, at al. // IEEE Transactions on Computers. — 1983. — № 2. — P. 175–189.

95. The State of the Art in Locally Distributed Web-server Systems / V. Cardellini, E. Casalicchio, M. Colajanni, P. S. Yu // ACM Computing Surveys, vol.34, No.2, June, 2002. — P. 263–311.

96. Vee V.-Y., Hsu W.-J. Parallel discrete event simulation: a survey // Technical Report, Centre for Advanced Information Systems / Nanyang Technological University. — Singapore, 1999.

97. Watson W. J. The TIASC — A Highly Modular and Flexible SuperComputer Architecture // Proc. AFIPS Fall Joint Computer Conf., vol.41 . — AFIPS Press, Montvale, NJ, December, 1972. —

P. 2221–228.

98. www.ciscosystems.com

99. www.fortinet.com/products/fortiadc/index.html

100. f5.com

101. www.foundrynetworks.com

102. www.ibm.com

103. www.nortelnetworks.com

104. www.radware.com

105. www.resonate.com

Список рекомендованной литературы

1. Адлер Ю. П. Статистические методы в имитационном моделировании. — М.: Мир, 1990.
2. Башарин Г. П., Бочаров П. П., Коган Я. А. Анализ очередей в вычислительных сетях. — М.: Наука, 1989.
3. Бенькович Е., Колесов Ю., Сениченков Ю. Практическое моделирование динамических систем. — Санкт-Петербург: BHV, 2002.
4. Гнеденко Б. В., Коваленко И. Н. Введение в теорию массового обслуживания. — М.: Наука, 1987.
5. Гультяев А. Имитационное моделирование в среде Windows. — СПб.: Корона принт, 1999.
6. Емельянов А. А., Власова Е. А., Дума Р. В. Имитационное моделирование экономических процессов. — М.: Финансы и статистика, 2002.
7. Закс Ш. Теория статистических выводов. — М.: Мир, 1975.
8. Калашников В. В. Организация моделирования сложных систем. — М.: Высш. шк., 1990.
9. Кендалл М. Дж., Стьюарт А. Статистические выводы и связи. — М.: Наука, 1973.
10. Клейнрок Л. Вычислительные системы с очередями. — М.: Мир, 1979.
11. Кобелев Н. Б. Основы имитационного моделирования сложных экономических систем. — Москва: ОЗОН, 2003.
12. Колесов Ю. Б., Сениченков Ю. Б. Визуальное моделирование — СПб.: Мир-Семья Интерлайн, 2000.
13. Конвей Р. В., Максвелл В. Л., Миллер Л. В. Теория расписаний. — М.: Наука, 1975.

14. Максимей И. В. Имитационное моделирование на ЭВМ. — М.: Радио и связь, 1988.
15. Матвеев В. Ф., Ушаков В. Г. Системы массового обслуживания. — М.: Изд-во Моск. гос. ун-та, 1984.
16. Новиков О. А., Петухов С. И. Прикладные вопросы теории массового обслуживания. — М.: Сов. радио, 1969.
17. Овчаров Л. А. Прикладные задачи теории массового обслуживания. — М.: Машиностроение, 1969.
18. Павловский Ю. Н. Имитационные модели и системы. — М.: Высш. шк., 1990.
19. Поллард Дж. Справочник по вычислительным методам статистики. — М.: Финансы и статистика, 1982.
20. Саати Т. Л. Элементы теории массового обслуживания и ее приложения. — М.: Сов. радио, 1971.
21. Советов Б. Я., Яковлев С. А. Моделирование систем. — М.: Высш. шк., 1998.
22. Тараканов К. В., Овчаров Л. А., Тырышкин А. Н. Аналитические методы исследования систем. — М.: Сов. радио, 1974.
23. Томашевский В. Н. Имитационное моделирование систем и процессов. — Киев: ВИПОЛ, 1994.
24. Шеннон Р. Имитационное моделирование систем — искусство и наука. — М.: Наука, 1978.
25. Шерр А. Анализ вычислительных систем с разделением времени. — М.: Мир, 1970.
26. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. — Киев: Диалектика, 1993.
27. Allen A. O. Probability, statistics and queueing theory. With

computer science applications. — Boston, MA: Academic Press, 1990.

28. ANSI/ISO C++ Professional Programmer's Handbook. — Indianapolis, Macmillan Computer Publishing, 1999.
29. Carmichael D. Engineering Queues in Construction and Mining. — Chichester: Ellis Horwood, 1987.
30. Cooper R. B. Introduction to queueing theory. — London: North-Holland(Elsevier), 1981.
31. Daigle J. N. Queueing theory for telecommunications. — Reading, MA: Addison-Wesley, 1992.
32. Fishman G. S. Principles of Discrete Event Simulation. — New York: Wiley&Sons, 1978.
33. German R. Performance analysis of communication systems. Modeling with non-Markovian stochastic Petri nets. — Chichester: John Wilet&Sons, 2000.
34. Gordon G. System Simulation. — Englewood Cliffs, NJ: Prentice Hall, 1978.
35. Gross D., Harris C. M. Fundamentals of queueing theory. — New York: John Wiley&Sons, 1985.
36. Harrell C. R., Tumay K. Simulation Made Easy: A Manager's Guide. — Industrial Engineering Press, 1995.
37. Haverkort B. Performance of Computer Communication Systems: A Model-Based Approach. — New York: John Wiley&Sons, 1998.
38. Hoover S. V., Perry R. F. Simulation: A Problem Solving Approach. — Reading, MS: Addison-Wesley, 1990.
39. King J. Computer and Communication Systems Performance Modeling. — Englewood Cliffs, NJ: Prentice Hall, 1990.
40. Kneppel P. L., Arangno D. C. Simulation Validation: A Confidence

Assessment Methodology. — IEEE Computer Society Press, 1994.
41. Lindemann C. Performance Modeling with Deterministic and Stochastic Petri Nets. — New York: John Wiley&Sons, 1998.
42. Law A. M. Designing and Analysis Simulation Experiments // Industrial Engineering. — 1991. — № 3 — P. 20–23.
43. Naine P. Basic Elements of Queueing Theory: Applications to the Modelling of Computer Systems: Lecture notes. — //www-net.cs.umass.edu/pe2000/nain.pdf
44. Neelamkavil F. Computer Simulation and Modeling. — New York: John Wiley&Sons, 1987.
45. Quantitative System Performance: Computer system analysis using queueing network models // E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik. — Englewood Cliffs, NJ: Prentice Hall, 1984.
46. Queueing Networks and Markov Chains / G. Bolch, S. Greiner, H. de Meer, K. Trivedi. — New York: Wiley&Sons, 1998.
47. Rumbaugh P. Object Oriented Modeling and Design. — Englewood Cliffs, NJ: Prentice Hall, 1991.
48. Thesen A., Travis L. E. Simulation for Decision Making. — St. Paul: West Publishing Company, 1992.
49. Tijms H. Stochastic Modeling and Analysis: A Computational Approach. — New York: John Wiley&Sons, 1986.
50. Trivedi K. Probability and Statistics with Reliability. Queueing and Computer Science Applications. — Englewood Cliffs, NJ: Prentice Hall, 1982.
51. Williams T. Modelling Complex Projects. — New York: John Wiley&Sons, 2002.