

С Е Р И Я



КЛАССИКА COMPUTER SCIENCE



UNIX internals: the new frontiers

Uresh Vahalia



**Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com**



Ю. ВАХАЛИЯ

UNIX изнутри



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара

Киев · Харьков · Минск

2003

ББК 32.973-018.2

УДК 681.3.066

B22

**B22 UNIX изнутри / Ю. Вахалия. — СПб.: Питер, 2003. — 844 с.: ил. —
(Серия «Классика computer science»)**

ISBN 5-94723-013-5

Эта книга показывает ядро UNIX с точки зрения разработчика систем. Для каждого компонента ядра приводится описание архитектуры и внутреннего устройства, практической реализации в каждом из описываемых вариантов операционной системы, а также преимуществ и недостатков альтернативных вариантов рассматриваемого компонента. Вы увидите описание основных коммерческих и научных реализаций операционной системы

Книга не рассчитана на начинающих и содержит знания о таких концептуальных вещах, как ядро системы, процессы или виртуальная память. Она может быть использована как профессиональное руководство или как пособие для изучения UNIX в высших учебных заведениях. Уровень изложения материала достаточен для изложения в качестве основного или дополнительного курса лекций по операционным системам.

ББК 32 973-018 2

УДК 681.3.066

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги

© 1996 by Prentice Hall, Inc.

ISBN 0-13-101908-2 (англ.)

ISBN 5-94723-013-5

© Перевод на русский язык, ЗАО Издательский дом «Питер», 2003

© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2003

Краткое содержание

От редактора английского издания	22
От издательства	24
Предисловие	25
Глава 1. Введение	30
Глава 2. Ядро и процессы	57
Глава 3. Нити и легковесные процессы	97
Глава 4. Сигналы и управление сеансами	145
Глава 5. Планирование процессов	186
Глава 6. Межпроцессное взаимодействие	237
Глава 7. Синхронизация. Многопроцессорные системы	286
Глава 8. Базовые элементы и интерфейс файловой системы	331
Глава 9. Реализации файловых систем	384
Глава 10. Распределенные файловые системы	427
Глава 11. Усовершенствованные файловые системы	493
Глава 12. Выделение памяти ядром	540
Глава 13. Виртуальная память	578
Глава 14. Архитектура VM системы SVR4	629
Глава 15. Дополнительные сведения об управлении памятью	679
Глава 16. Ввод-вывод и драйверы устройств	731
Глава 17. Подсистема STREAMS	781
Алфавитный указатель	830

Содержание

От редактора английского издания	22
От издательства	24
Предисловие	25
Изложение материала	25
Реализации UNIX	26
Для кого предназначена эта книга	26
Как организована эта книга	27
Некоторые обозначения, принятые в книге	28
Благодарности	28
Дополнительная литература	29
Глава 1. Введение	30
1.1. Введение	30
1.1.1. Краткая история	31
1.1.2. Начало	31
1.1.3. Распространение	33
1.1.4. BSD	34
1.1.5. System V	36
1.1.6. Коммерциализация	36
1.1.7. Mach	38
1.1.8. Стандарты	38
1.1.9. OSF и UI	40
1.1.10. SVR4 и ее дальнейшее развитие	41
1.2. Причины изменений системы	42
1.2.1. Функциональные возможности	43
1.2.2. Сетевая поддержка	44
1.2.3. Производительность	45
1.2.4. Изменение аппаратных платформ	45
1.2.5. Улучшение качества	46
1.2.6. Глобальные изменения	47
1.2.7. Поддержка различных приложений	48
1.2.8. Чем меньше, тем лучше	49
1.2.9. Гибкость системы	50
1.3. Оглянемся назад, посмотрим вперед	50
1.3.1. Преимущества UNIX	51
1.3.2. Недостатки UNIX	53

1.4. Границы повествования книги	54
1.5. Дополнительная литература	55
Глава 2. Ядро и процессы	57
2.1. Введение	57
2.2. Режим, пространство и контекст	61
2.3. Определение процесса	64
2.3.1. Состояние процесса	64
2.3.2. Контекст процесса	67
2.3.3. Полномочия пользователя	68
2.3.4. Область и структура прос	70
2.4. Выполнение в режиме ядра	72
2.4.1. Интерфейс системных вызовов	73
2.4.2. Обработка прерываний	73
2.5. Синхронизация	76
2.5.1. Операции блокировки	78
2.5.2. Прерывания	80
2.5.3. Многопроцессорные системы	81
2.6. Планирование процессов	81
2.7. Сигналы	83
2.8. Новые процессы и программы	84
2.8.1. Вызовы fork и exec	84
2.8.2. Создание процесса	86
2.8.3. Оптимизация вызова fork	87
2.8.4. Запуск новой программы	88
2.8.5. Завершение процесса	91
2.8.6. Ожидание завершения процесса	91
2.8.7. Процессы-зомби	93
2.9. Заключение	94
2.10. Упражнения	94
2.11. Дополнительная литература	95
Глава 3. Нити и легковесные процессы	97
3.1. Введение	97
3.1.1. Причины появления технологии нитей	98
3.1.2. Нити и процессоры	99
3.1.3. Одновременность и параллельность	102
3.2. Основные типы нитей	103
3.2.1. Нити ядра	104
3.2.2. Легковесные процессы	105
3.2.3. Прикладные нити	107
3.3. Легковесные процессы: основные проблемы	112
3.3.1. Семантика вызова fork	112
3.3.2. Другие системные вызовы	113
3.3.3. Доставка и обработка сигналов	114
3.3.4. Видимость	115
3.3.5. Рост стека	116

3.4. Нитевые библиотеки прикладного уровня	116
3.4.1. Программный интерфейс	117
3.4.2. Реализация нитевых библиотек	117
3.5. Активации планировщика	119
3.6. Многонитевость в Solaris и SVR4	121
3.6.1. Нити ядра	121
3.6.2. Реализация легковесных процессов	122
3.6.3. Прикладные нити	124
3.6.4. Реализация прикладных нитей	125
3.6.5. Обработка прерываний	126
3.6.6. Обработка системных вызовов	128
3.7. Нити в системе Mach	129
3.7.1. Задачи и нити в системе Mach	129
3.7.2. Библиотека C-threads	131
3.8. Digital UNIX	132
3.8.1. Интерфейс UNIX	132
3.8.2. Системные вызовы и сигналы	134
3.8.3. Библиотека pthreads	134
3.9. Продолжения в системе Mach	136
3.9.1. Модели выполнения программ	136
3.9.2. Использование продолжений	137
3.9.3. Оптимизация работы	139
3.9.4. Анализ производительности	140
3.10. Заключение	140
3.11. Упражнения	141
3.12. Дополнительная литература	142
Глава 4. Сигналы и управление сеансами	145
4.1. Введение	145
4.2. Генерирование и обработка сигналов	146
4.2.1. Обработка сигналов	147
4.2.2. Генерирование сигналов	150
4.2.3. Типичные примеры возникновения сигналов	152
4.2.4. Спящие процессы и сигналы	153
4.3. Ненадежные сигналы	153
4.4. Надежные сигналы	155
4.4.1. Основные возможности	156
4.4.2. Сигналы в системе SVR3	156
4.4.3. Механизм сигналов в BSD	158
4.5. Сигналы в SVR4	160
4.6. Реализация сигналов	161
4.6.1. Генерация сигналов	162
4.6.2. Доставка и обработка	162
4.7. Исключительные состояния	163
4.8. Обработка исключительных состояний в Mach	164
4.8.1. Порты исключительных состояний	166
4.8.2. Обработка ошибок	167

4.8.3. Взаимодействие с отладчиком	167
4.8.4. Анализ	168
4.9. Группы процессов и управление терминалом	169
4.9.1. Общие положения	169
4.9.2. Модель SVR3	170
4.9.3. Ограничения	173
4.9.4. Группы и терминалы в системе 4.3BSD	174
4.9.5. Недостатки модели 4.3BSD	176
4.10. Архитектура сеансов в системе SVR4	177
4.10.1. Задачи, поставленные перед разработчиками	177
4.10.2. Сеансы и группы процессов	178
4.10.3. Структуры данных	180
4.10.4. Управляющие терминалы	181
4.10.5. Реализация сеансов в 4.4BSD	182
4.11. Заключение	183
4.12. Упражнения	183
4.13. Дополнительная литература	184

Глава 5. Планирование процессов 186

5.1. Введение	186
5.2. Обработка прерываний таймера	187
5.2.1. Отложенные вызовы	189
5.2.2. Будильники	191
5.3. Цели, стоящие перед планировщиком	192
5.4. Планирование в традиционных системах UNIX	193
5.4.1. Приоритеты процессов	194
5.4.2. Реализация планировщика	197
5.4.3. Операции с очередью выполнения	198
5.4.4. Анализ	199
5.5. Планировщик в системе SVR4	200
5.5.1. Независимый от класса уровень	201
5.5.2. Интерфейс с классами планирования	203
5.5.3. Класс разделения времени	205
5.5.4. Класс реального времени	208
5.5.5. Системный вызов priocntl	209
5.5.6. Анализ	210
5.6. Расширенные возможности планирования системы Solaris 2.x	212
5.6.1. Вытесняющее ядро	212
5.6.2. Многопроцессорная поддержка	213
5.6.3. Скрытое планирование	215
5.6.4. Инверсия приоритетов	216
5.6.5. Реализация наследования приоритетов	218
5.6.6. Ограничения наследования приоритетов	220
5.6.7. Турникеты	221
5.6.8. Анализ	223
5.7. Планирование в системе Mach	223
5.7.1. Поддержка нескольких процессоров	224

5.8. Планировщик реального времени Digital UNIX	227
5.8.1. Поддержка нескольких процессоров	228
5.9. Другие реализации планирования	229
5.9.1. Планирование справедливого разделения	230
5.9.2. Планирование по крайнему сроку	230
5.9.3. Трехуровневый планировщик	231
5.10. Заключение	233
5.11. Упражнения	233
5.12. Дополнительная литература	235
Глава 6. Межпроцессное взаимодействие	237
6.1. Введение	237
6.2. Универсальные средства IPC	238
6.2.1. Сигналы	238
6.2.2. Каналы	239
6.2.3. Каналы в системе SVR4	242
6.2.4. Трассировка процессов	243
6.3. System V IPC	245
6.3.1. Общие элементы	245
6.3.2. Семафоры	247
6.3.3. Очереди сообщений	252
6.3.4. Разделяемая память	254
6.3.5. Применение механизмов IPC	257
6.4. Mach IPC	258
6.4.1. Основные концепции	259
6.5. Сообщения	261
6.5.1. Структуры данных сообщения	261
6.5.2. Интерфейс передачи сообщений	263
6.6. Порты	264
6.6.1. Диапазон имен портов	265
6.6.2. Структура данных порта	265
6.6.3. Преобразования портов	266
6.7. Передача сообщений	267
6.7.1. Передача прав порта	269
6.7.2. Внешняя память	270
6.7.3. Управление нитью	273
6.7.4. Уведомления	273
6.8. Операции порта	274
6.8.1. Удаление порта	274
6.8.2. Резервные порты	275
6.8.3. Наборы портов	275
6.8.4. Передача прав	277
6.9. Расширяемость	278
6.10. Новые возможности Mach 3.0	279
6.10.1. Права на однократную отправку	280
6.10.2. Уведомления в Mach 3.0	281
6.10.3. Подсчет прав на отправку	281
6.11. Дискуссия о средствах Mach IPC	282

6.12. Заключение	283
6.13. Упражнения	283
6.14. Дополнительная литература	284
Глава 7. Синхронизация. Многопроцессорные системы	286
7.1. Введение	286
7.2. Синхронизация в ядре традиционных реализаций UNIX	288
7.2.1. Блокировка прерываний	288
7.2.2. Приостановка и пробуждение	289
7.2.3. Ограничения традиционного ядра UNIX	290
7.3. Многопроцессорные системы	292
7.3.1. Модель памяти	293
7.3.2. Поддержка синхронизации	294
7.3.3. Программная архитектура	296
7.4. Особенности синхронизации в многопроцессорных системах	297
7.4.1. Проблема выхода из режима ожидания	298
7.4.2. Проблема быстрого роста	299
7.5. Семафоры	300
7.5.1. Семафоры как средство взаимного исключения	301
7.5.2. Семафоры и ожидание наступления событий	302
7.5.3. Семафоры и управление исчисляемыми ресурсами	302
7.5.4. Недостатки семафоров	303
7.5.5. Конвой	303
7.6. Простая блокировка	305
7.6.1. Применение простой блокировки	306
7.7. Условные переменные	307
7.7.1. Некоторые детали реализации условных переменных	309
7.7.2. События	310
7.7.3. Блокирующие объекты	311
7.8. Синхронизация чтения-записи	311
7.8.1. Задачи, стоящие перед разработчиками	312
7.8.2. Реализация синхронизации чтения-записи	313
7.9. Счетчики ссылок	315
7.10. Другие проблемы, возникающие при синхронизации	316
7.10.1. Предупреждение возникновения взаимоблокировки	316
7.10.2. Рекурсивная блокировка	318
7.10.3. Что лучше: приостановка выполнения или ожидание в цикле?	319
7.10.4. Объекты блокировки	320
7.10.5. Степень разбиения и длительность	321
7.11. Реализация объектов синхронизации в различных ОС	322
7.11.1. SVR4.2/MP	322
7.11.2. Digital UNIX	324
7.11.3. Другие реализации систем UNIX	325
7.12. Заключение	327
7.13. Упражнения	327
7.14. Дополнительная литература	329

Глава 8. Базовые элементы и интерфейс файловой системы . . .	331
8.1. Введение	331
8.2. Интерфейс доступа пользователя к файлам	332
8.2.1. Файлы и каталоги	333
8.2.2. Атрибуты файлов	335
8.2.3. Дескрипторы файлов	338
8.2.4. Файловый ввод-вывод	340
8.2.5. Ввод-вывод методом сборки-рассоединения	342
8.2.6. Блокировка файлов	343
8.3. Файловые системы	344
8.3.1. Логические диски	345
8.4. Специальные файлы	346
8.4.1. Символические ссылки	347
8.4.2. Каналы и файлы FIFO	349
8.5. Базовые файловые системы	350
8.6. Архитектура vnode/vfs	351
8.6.1. Цели, поставленные перед разработчиками	351
8.6.2. Немного о вводе-выводе устройств	352
8.6.3. Краткий обзор интерфейса vnode/vfs	355
8.7. Краткий обзор реализации	357
8.7.1. Цели, стоящие перед разработчиками	357
8.7.2. Открытые файлы и объекты vnode	358
8.7.3. Структура vnode	359
8.7.4. Счетчик ссылок vnode	360
8.7.5. Объект vfs	362
8.8. Объекты, зависящие от файловой системы	363
8.8.1. Закрытые данные каждого файла	363
8.8.2. Вектор vnodeops	364
8.8.3. Зависящая от файловой системы часть уровня vfs	365
8.9. Монтирование файловой системы	366
8.9.1. Виртуальный переключатель файловых систем	366
8.9.2. Реализация вызова mount	367
8.9.3. Действия операции VFS_MOUNT	367
8.10. Операции над файлами	368
8.10.1. Преобразование полных имен	368
8.10.2. Кэш просмотра каталогов	370
8.10.3. Операция VOP_LOOKUP	371
8.10.4. Открытие файла	372
8.10.5. Файловый ввод-вывод	373
8.10.6. Атрибуты файлов	374
8.10.7. Права пользователя	374
8.11. Анализ	375
8.11.1. Недостатки реализации в системе SVR4	376
8.11.2. Модель 4.4BSD	377
8.11.3. Средства системы OSF/1	379
8.12. Заключение	380
8.13. Упражнения	381
8.14. Дополнительная литература	382

Глава 9. Реализации файловых систем	384
9.1. Введение	384
9.2. Файловая система System V (s5fs)	386
9.2.1. Каталоги	387
9.2.2. Индексные дескрипторы	387
9.2.3. Суперблок	390
9.3. Организация ядра системы s5fs	391
9.3.1. Индексные дескрипторы в памяти	392
9.3.2. Получение индексных дескрипторов	393
9.3.3. Файловый ввод-вывод	393
9.3.4. Запрос и возврат индексных дескрипторов	395
9.4. Анализ файловой системы s5fs	397
9.5. Файловая система FFS	399
9.6. Структура жесткого диска	399
9.7. Хранение данных на диске	400
9.7.1. Блоки и фрагменты	401
9.7.2. Правила размещения	403
9.8. Новые возможности FFS	404
Длинные имена файлов	405
Символические ссылки	405
Другие возможности	406
9.9. Анализ файловой системы FFS	406
9.10. Временные файловые системы	408
9.10.1. Memory File System	409
9.10.2. Файловая система tmpfs	410
9.11. Файловые системы для специальных целей	411
9.11.1. Файловая система specfs	412
9.11.2. Файловая система /proc	412
9.11.3. Процессорная файловая система	415
9.11.4. Translucent File System	416
9.12. Буферный кэш в ранних версиях UNIX	417
9.12.1. Основные операции	418
9.12.2. Заголовки буфера	419
9.12.3. Преимущества	420
9.12.4. Недостатки	420
9.12.5. Целостность файловой системы	421
9.13. Заключение	423
9.14. Упражнения	423
9.15. Дополнительная литература	424
Глава 10. Распределенные файловые системы	427
10.1. Введение	427
10.2. Общие характеристики распределенных файловых систем	428
10.2.1. Некоторые соглашения	429
10.3. Network File System (NFS)	430
10.3.1. NFS с точки зрения пользователя	431
10.3.2. Цели, стоявшие перед разработчиками	433

10.3.3. Компоненты NFS	433
10.3.4. Сохранение состояний	435
10.4. Набор протоколов	437
10.4.1. Представление внешних данных (XDR)	437
10.4.2. Вызов удаленных процедур (RPC)	439
10.5. Реализация NFS	441
10.5.1. Управление потоком	441
10.5.2. Дескрипторы файлов	443
10.5.3. Операция монтирования	443
10.5.4. Просмотр полных имен	444
10.6. Семантика UNIX	445
10.6.2. Удаление открытых файлов	446
10.6.3. Чтение и запись	446
10.7. Производительность NFS	447
10.7.1. Узкие места	447
10.7.2. Кэширование на стороне клиента	448
10.7.3. Отложенная запись	448
10.7.4. Кэш повторных посылок	450
10.8. Специализированные серверы NFS	452
10.8.1. Архитектура функциональной многопроцессорной системы Auspex	452
10.8.2. Сервер HA-NFS фирмы IBM	454
10.9. Защита NFS	456
10.9.1. Контроль доступа в NFS	456
10.9.2. Переназначение групповых идентификаторов	457
10.9.3. Переназначение режима доступа root	458
10.10. NFS версии 3	459
10.11. Remote File Sharing (RFS)	461
10.12. Архитектура RFS	461
10.12.1. Протокол удаленных сообщений	463
10.12.2. Операции сохранения состояния	464
10.13. Реализация системы RFS	464
10.13.1. Удаленное монтирование	464
10.13.2. Серверы и клиенты RFS	466
10.13.3. Восстановление после отказа	467
10.13.4. Другие части системы	468
10.14. Кэширование на стороне клиента	469
10.14.1. Достоверность кэша	470
10.15. Andrew File System	471
10.15.1. Масштабируемая архитектура	472
10.15.2. Организация хранения и пространство имен	474
10.15.3. Семантика сеансов	475
10.16. Реализация AFS	476
10.16.1. Кэширование и достоверность данных	476
10.16.2. Просмотр имен	477
10.16.3. Безопасность	478
10.17. Недостатки файловой системы AFS	479

10.18. Распределенная файловая система DCE (DCE DFS)	480
10.18.1. Архитектура DFS	481
10.18.2. Корректность кэша	482
10.18.3. Менеджер маркеров доступа	484
10.18.4. Другие службы DFS	485
10.18.5. Анализ	485
10.19. Заключение	487
10.20. Упражнения	488
10.21. Дополнительная литература	489

Глава 11. Усовершенствованные файловые системы 493

11.1. Введение	493
11.2. Ограничения традиционных файловых систем	494
11.2.1. Разметка диска в FFS	495
11.2.2. Преобладание операций записи	497
11.2.3. Модификация метаданных	498
11.2.4. Восстановление после сбоя	499
11.3. Кластеризация файловых систем (Sun-FFS)	500
11.4. Журналы	502
11.4.1. Основные характеристики	502
11.5. Структурированные файловые системы	504
11.6. Структурированная файловая система 4.4BSD	505
11.6.1. Запись в журнал	507
11.6.2. Получение данных из журнала	508
11.6.3. Восстановление после сбоя	508
11.6.4. Процесс <i>cleaner</i>	509
11.6.5. Анализ системы <i>BSD-LFS</i>	510
11.7. Ведение журнала метаданных	512
11.7.1. Функционирование в обычном режиме	512
11.7.2. Целостность журнала	514
11.7.3. Восстановление	516
11.7.4. Анализ	517
11.8. Файловая система <i>Episode</i>	519
11.8.1. Основные понятия	519
11.8.2. Структура системы	520
11.8.3. Ведение журнала	522
11.8.4. Другие возможности	522
11.9. Процесс <i>watchdog</i>	523
11.9.1. Наблюдение за каталогами	525
11.9.2. Каналы сообщений	525
11.9.3. Приложения	527
11.10. Portal File System в 4.4BSD	527
11.10.1. Применение порталов	528
11.11. Уровни стековой файловой системы	529
11.11.1. Инфраструктура и интерфейс	531
11.11.2. Модель SunSoft	533
11.12. Интерфейс файловой системы 4.4BSD	534
11.12.1. Файловые системы <i>nullfs</i> и <i>union mount</i>	535

11.13. Заключение	536
11.14. Упражнения	537
11.15. Дополнительная литература	538
Глава 12. Выделение памяти ядром	540
12.1. Введение	540
12.2. Требования к функциональности	542
12.2.1. Критерии оценки	543
12.3. Распределитель карты ресурсов	545
12.3.1. Анализ	547
12.4. Простые списки, основанные на степени двойки	549
12.4.1. Анализ	551
12.5. Распределитель Мак-Кьюзика—Кэрелса	552
12.5.1. Анализ	554
12.6. Метод близнецов	555
12.6.1. Анализ	557
12.7. Алгоритм отложенного слияния в SVR4	558
12.7.1. Отложенное слияние	559
12.7.2. Особенности реализации алгоритма в SVR4	561
12.8. Зональный распределитель в системе Mach-OSF/1	561
12.8.1. Сбор мусора	562
12.8.2. Анализ	564
12.9. Иерархический распределитель памяти для многопроцессорных систем	565
12.9.1. Анализ	567
12.10. Составной распределитель системы Solaris 2.4	567
12.10.1. Повторное использование объектов	568
12.10.2. Применение аппаратных кэшей	569
12.10.3. Рабочая площадка распределителя	569
12.10.4. Структура и интерфейс	570
12.10.5. Реализация алгоритма	571
12.10.6. Анализ	573
12.11. Заключение	574
12.12. Упражнения	575
12.13. Дополнительная литература	577
Глава 13. Виртуальная память	578
13.1. Введение	578
13.1.1. Управление памятью в «каменном веке» вычислительной техники	580
13.2. Загрузка страниц по запросу	583
13.2.1. Требования к функциональности	583
13.2.2. Виртуальное адресное пространство	586
13.2.3. Первое обращение к странице	587
13.2.4. Область свопинга	587
13.2.5. Карты преобразования адресов	589
13.2.6. Правила замещения страниц	590
13.3. Аппаратные требования	592
13.3.1. Кэши MMU	594
13.3.2. Intel 80x86	596

13.3.3. IBM RS/6000	600
13.3.4. MIPS R3000	603
13.4. Пример реализации: система 4.3BSD	606
13.4.1. Физическая память	607
13.4.2. Адресное пространство	609
13.4.3. Где может располагаться страница памяти	611
13.4.4. Пространство свопинга	613
13.5. Операции работы с памятью 4.3BSD	614
13.5.1. Создание процесса	615
13.5.2. Обработка страничной ошибки	616
13.5.3. Список свободных страниц	619
13.5.4. Свопинг	622
13.6. Анализ	624
13.7. Упражнения	626
13.8. Дополнительная литература	627
Глава 14. Архитектура VM системы SVR4	629
14.1. Предпосылки появления новой технологии	629
14.2. Файлы, отображаемые в памяти	630
14.2.1. Системный вызов mmap	632
14.3. Основы архитектуры VM	633
14.4. Основные понятия	635
14.4.1. Физическая память	636
14.4.2. Адресное пространство	637
14.4.3. Отображение адресов	639
14.4.4. Анонимные страницы	640
14.4.5. Аппаратное преобразование адресов	641
14.5. Драйверы сегментов	643
14.5.1. Драйвер seg_vn	644
14.5.2. Драйвер seg_map	645
14.5.3. Драйвер seg_dev	646
14.5.4. Драйвер seg_kmem	646
14.5.5. Драйвер seg_kp	647
14.6. Уровень свопинга	647
14.7. Операции системы VM	649
14.7.1. Создание нового отображения	650
14.7.2. Обработка анонимных страниц	650
14.7.3. Создание процесса	652
14.7.4. Совместное использование анонимных страниц	654
14.7.5. Обработка страничных ошибок	655
14.7.6. Разделяемая память	657
14.7.7. Другие компоненты	658
14.8. Взаимодействие с подсистемой vnode	659
14.8.1. Изменения в интерфейсе vnode	659
14.8.2. Унификация доступа к файлам	661
14.8.3. Важные замечания	664
14.9. Виртуальное пространство свопинга в Solaris	665
14.9.1. Расширенное пространство свопинга	665

14.9.2. Управление виртуальным свопингом	666
14.9.3. Некоторые выводы	668
14.10. Анализ	668
14.11. Увеличение производительности	672
14.11.1. Причины большого количества исключений	672
14.11.2. Новые возможности подсистемы VM в SunOS	674
14.11.3. Итоги	675
14.12. Заключение	676
14.13. Упражнения	676
14.14. Дополнительная литература	677

Глава 15. Дополнительные сведения об управлении памятью 679

15.1. Введение	679
15.2. Структура подсистемы управления памятью Mach	679
15.2.1. Цели, стоявшие перед разработчиками	680
15.2.2. Программный интерфейс	681
15.2.3. Фундаментальные понятия	683
15.3. Средства разделения памяти	686
15.3.1. Разделение памяти на основе копирования при записи	686
15.3.2. Разделение памяти на основе чтения-записи	688
15.4. Объекты памяти и менеджеры памяти	690
15.4.1. Инициализация объектов памяти	690
15.4.2. Интерфейс взаимодействия ядра и менеджера памяти	691
15.4.3. Обмен данными между ядром и менеджером памяти	692
15.5. Внешние и внутренние менеджеры памяти	693
15.5.1. Сетевой сервер разделяемой памяти	694
15.6. Замена страниц	697
15.7. Анализ	699
15.8. Управление памятью в 4.4BSD	701
15.9. Корректность буфера ассоциативной трансляции (TLB)	703
15.9.1. Корректность TLB в однопроцессорных системах	705
15.9.2. Корректность TLB в многопроцессорных системах	706
15.10. Алгоритм синхронизации TLB системы Mach	708
15.10.1. Синхронизация и предупреждение взаимоблокировок	710
15.10.2. Некоторые итоги	711
15.11. Корректность TLB в SVR4 и SVR4.2	711
15.11.1. SVR4/MP	712
15.11.2. SVR4.2/MP	713
15.11.3. Отложенная перезагрузка	715
15.11.4. Незамедлительная перезагрузка	716
15.11.5. Некоторые итоги	718
15.12. Другие алгоритмы поддержания корректности TLB	718
15.13. Виртуально адресуемый кэш	720
15.13.1. Изменения отображений	722
15.13.2. Псевдонимы адресов	723
15.13.3. Прямой доступ к памяти	724
15.13.4. Поддержка корректности кэша	724
15.13.5. Анализ	726

15.14. Упражнения	727
15.15. Дополнительная литература	728
Глава 16. Ввод-вывод и драйверы устройств	731
16.1. Введение	731
16.2. Краткий обзор	731
16.2.1. Аппаратная часть	733
16.2.2. Прерывания устройств	735
16.3. Базовая структура драйвера устройства	738
16.3.1. Классификация устройств и их драйверов	738
16.3.2. Вызов кодов драйвера	740
16.3.3. Переключатели устройств	741
16.3.4. Входные точки драйвера	742
16.4. Подсистема ввода-вывода	744
16.4.1. Старший и младший номера устройств	745
16.4.2. Файлы устройств	747
16.4.3. Файловая система specfs	748
16.4.4. Общий объект snode	750
16.4.5. Клонирование устройств	751
16.4.6. Ввод-вывод символьных устройств	753
16.5. Системный вызов poll	753
16.5.1. Реализация вызова poll	755
16.5.2. Системный вызов select ОС 4.3BSD	757
16.6. Блочный ввод-вывод	758
16.6.1. Структура buf	759
16.6.2. Взаимодействие с объектом vnode	760
16.6.3. Способы обращения к устройствам	761
16.6.4. Неформатированный ввод-вывод блочных устройств	764
16.7. Спецификация DDI/DKI	765
16.7.1. Общие рекомендации	767
16.7.2. Функции раздела 3	768
16.7.3. Другие разделы	770
16.8. Поздние версии SVR4	771
16.8.1. Драйверы для многопроцессорных систем	771
16.8.2. Изменения в SVR4.1/ES	772
16.8.3. Динамическая загрузка	772
16.9. Перспективы	776
16.10. Заключение	778
16.11. Упражнения	778
16.12. Дополнительная литература	779
Глава 17. Подсистема STREAMS	781
17.1. Введение	781
17.2. Краткий обзор	782
17.3. Сообщения и очереди	785
17.3.1. Сообщения	786
17.3.2. Виртуальное копирование	787

17.3.3. Типы сообщений	789
17.3.4. Очереди и модули	790
17.4. Ввод-вывод потока	792
17.4.1. Диспетчер STREAMS	793
17.4.2. Уровни приоритетов	794
17.4.3. Управление потоком данных	795
17.4.4. Оконечный драйвер	797
17.4.5. Головной интерфейс потока	798
17.5. Конфигурирование и настройка	799
17.5.1. Конфигурирование модуля или драйвера	799
17.5.2. Открытие потока	801
17.5.3. Помещение модулей в поток	803
17.5.4. Клонирование устройств	804
17.6. Вызовы ioctl подсистемы STREAMS	805
17.6.1. Команда I_STR вызова ioctl	806
17.6.2. Прозрачные команды ioctl	807
17.7. Выделение памяти	808
17.7.1. Расширенные буферы STREAMS	810
17.8. Мультиплексирование	811
17.8.1. Мультиплексирование по входу	811
17.8.2. Мультиплексирование по выходу	812
17.8.3. Связывание потоков	813
17.8.4. Потоки данных	816
17.8.5. Обычные и постоянные соединения	816
17.9. FIFO и каналы	817
17.9.1. Файлы FIFO STREAMS	817
17.9.2. Каналы STREAMS	819
17.10. Сетевые интерфейсы	820
17.10.1. Интерфейс поставщиков транспорта (TPI)	821
17.10.2. Интерфейс транспортного уровня (TLI)	821
17.10.3. Сокеты	823
17.10.4. Реализация сокетов в SVR4	825
17.11. Заключение	826
17.12. Упражнения	827
17.13. Дополнительная литература	829
Алфавитный указатель	830

Эта книга посвящается Бхинне (Bhinna), память о которой навсегда останется в моем сердце, Рохану (Rohan) за его веселость и энтузиазм, а также Аркане (Archana), за ее любовь и поддержку.

От редактора английского издания

П. Х. Салюс (P. H. Salus), ведущий редактор Computing Systems

На сегодняшний день версий UNIX существует больше, чем производителей мороженого. Несмотря на подталкивание частью консорциума X/Open и его членами, единая спецификация UNIX все более удаляется от нас. Однако это и не есть главная цель. С тех пор как компания Interactive Systems представила первую коммерческую систему UNIX, а компания Whitesmiths создала первый клон UNIX, пользователи были поставлены перед фактом появления самых разнообразных вариантов системы, разработанных под разные платформы.

Система UNIX была создана в 1969 году. Не прошло и десяти лет, как ее версии начали множиться. Когда UNIX исполнилось 20 лет, уже существовали крупные консорциумы (Open Software Foundation и UNIX International) и большое количество различных реализаций ОС. Два основных потока развития системы исходили из AT&T (сейчас Novell) и Калифорнийского университета в Беркли. Описания этих вариантов UNIX, созданные Морисом Бахом (Maurice Bach) [1] и Сэмом Леффлером (Sam Leffler), Кирком МакКьюзиком (Kirk McKusick), Майком Кэрельсом (Mike Karels) и Джоном Квотерманом (John Quarterman) [2], можно легко найти.

Ни одна книга ранее не предлагала описание реализаций операционной системы UNIX с интересной для студентов точки зрения. Эту задачу выполнил Юреш Вахалия (Uresh Vahalia). Он сделал то, чего до него не создавал ни один автор, подробно обрисовал внутреннее устройство систем SVR4, 4.4BSD и Mach. Более того, книга содержит тщательно продуманное изложение компонентов систем Solaris и SunOS, Digital UNIX и HP-UX.

Он сделал прекрасное описание систем, свободное от предпочтений какого-либо одного варианта UNIX, часто не скрываемых другими авторами. Несмотря на то что уже создаются различные реализации относительно новых систем, таких как Linux, и даже варианты Berkeley значительно отличаются между собой, такие книги, как эта, показывают внутреннее устройство UNIX и заложенные в нее принципы, ставшие причиной популярности системы в экспоненциальной зависимости.

12 июня 1972 года Кен Томпсон (Ken Thompson) и Денис Ритчи (Dennis Ritchie) представили вторую редакцию своего руководства *UNIX Programmer's Manual*. В предисловии авторы заметили: «Количество инсталляций UNIX достигло десяти — более, чем мы ожидали». Они даже не могли предположить, что действительно произойдет с их системой в будущем.

Я рассмотрел появление и историю развития систем в книге [3], но Вахалия дал нам действительно оригинальный и исчерпывающий взгляд на сравнительную анатомию систем.

Ссылки:

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
3. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
4. Thompson, K., and Ritchie, D. M., «UNIX Programmer's Manual», Second Edition, Bell Telephone Laboratories, Murray Hill, NJ, 1972.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Предисловие

Со времени своего появления (начало 70-х годов) система UNIX претерпела значительные изменения: начав свое развитие с небольшой экспериментальной операционной системы, распространяемой бесплатно Bell Laboratories, на сегодняшний день UNIX представляет собой целое семейство ее преемников. За эти годы она вобрала в себя огромное количество предложений от представителей науки и промышленности, прошла через множество битв за стандартизацию и авторские права и превратилась в стабильную целостную операционную систему. Существует несколько реализаций UNIX, предназначенных как для коммерческого, так и для научного использования, которые имеют определенные сходства между собой, достаточные для причисления их к одному и тому же типу операционных систем. Программист, изучивший один из клонов UNIX, может также производительно работать с другими аппаратными платформами и версиями операционной системы без необходимости переучивания.

Разные возможности самых различных реализаций ОС UNIX описаны в сотнях изданных книг. Хотя многие из этих трудов показывают системы со стороны пользователя, рассказывая, например, о командной оболочке или программном интерфейсе, лишь малая часть книг посвящена внутреннему устройству UNIX. Изучение архитектуры UNIX подразумевает описание ядра, являющегося «сердцем» каждой операционной системы. На сегодняшний день все существующие книги по UNIX описывают только какую-либо одну реализацию системы. Например, книга М. Дж. Баха «The Design of the UNIX Operating System» [1] является наиболее ярким описанием ядра System V Release 2 (SVR2), книга С. Дж. Леффлера и др. «The Design and Implementation of the 4.3 BSD UNIX Operating System» [4] представляет собой подробное описание системы 4.3BSD от лица ее создателей, внутреннее устройство ОС System V Release 4.0 (SVR4) раскрывают на страницах своей книги «The Magic Garden Explained – The Internals of UNIX System V Release 4» Б. Гудхарт и Дж. Кокс [3].

Изложение материала

Эта книга показывает ядро UNIX с точки зрения разработчика систем. Вы увидите описание основных коммерческих и научных реализаций операционной системы. Для каждого компонента ядра приводится описание архитек-

туры и внутреннего устройства, практической реализации в каждой из описываемых версий операционной системы, а также преимуществ и недостатков альтернативных вариантов рассматриваемого компонента. Такой сравнительный подход придает книге отличительную особенность и дает возможность читателю рассматривать систему с критической точки зрения. При изучении операционной системы важно знать не только сильные, но и слабые ее стороны. Это возможно только при проведении анализа альтернативных вариантов.

Реализации UNIX

В этой книге большое внимание уделяется системе SVR4.2, однако здесь вы можете найти подробное описание 4.4BSD, Solaris 2.x, Mach и Digital UNIX. Более того, на страницах книги рассказывается и о самых интересных возможностях других вариантов UNIX, в том числе разработок, до сих пор не реализованных в коммерческих версиях ОС, проводится анализ развития UNIX, начиная от середины 80-х и заканчивая серединой 90-х годов. Для цельности повествования в материал книги включено краткое описание основных возможностей и реализаций системы UNIX. Если это необходимо, описание включает в себя исторический контекст, ведется, начиная со стандартных функций, проведения анализа недостатков и ограничений и заканчивая представлением последних разработок.

Для кого предназначена эта книга

Предлагаемая книга может быть использована как профессиональное руководство для изучения в высших учебных заведениях. Уровень изложения материала достаточен для изложения в качестве основного или дополнительного курса лекций по операционным системам. Книга не рассчитана на начинающих и содержит знания о таких концептуальных вещах, как ядро системы, процессы или виртуальная память. В конце каждой главы приводится набор вопросов, разработанных для стимулирования дальнейшего самостоятельного изучения и поиска дополнительного материала, а также для более глубокого изучения внутреннего устройства систем. Ответы на многие вопросы остаются открытыми, а для некоторых из них нужно изучение дополнительной литературы. Каждая глава завершается исчерпывающим списком материалов, которые могут быть использованы студентами для более подробного ознакомления с описываемой тематикой.

Книга также является профессиональным руководством для разработчиков операционных систем, программных приложений и для системных администраторов. Разработчики систем могут использовать ее для изучения архитектуры ядра существующих ОС, сравнения преимуществ и недостатков

различных реализаций систем, а также использовать изложенный материал для создания следующих поколений операционных систем. Программисты могут применить полученные знания внутреннего устройства систем для написания более эффективных приложений, максимально задействующих полезные возможности UNIX. Рассказ о разнообразных параметрах и описание функционирования систем при использовании их определенных комбинаций поможет системным администраторам в настройке и наладке обслуживаемых ими операционных систем.

Как организована эта книга

Первая глава («Введение») описывает эволюцию систем UNIX и анализирует факторы, ставшие причиной основных изменений, произошедших в системе. В главах 2–7 изложено функционирование подсистем. В частности, глава 2 рассказывает о возможностях традиционных систем UNIX (SVR3, 4.3BSD и более ранних реализаций), в то время как на страницах глав 3–7 вы познакомитесь с возможностями современных ОС, таких как SVR4, 4.4BSD, Solaris 2.x и Digital UNIX. В третьей главе описаны потоки и их реализация в ядре системы и программах пользователя. В главе 4 говорится о сигналах, управлении процессами и обработке сессий входа в систему. Глава 5 посвящена диспетчеру UNIX и постоянно растущей поддержке приложений, работающих в режиме реального времени. Из материала главы 6 вы узнаете о взаимодействии процессов (IPC), а также о возможностях системы под названием System V IPC. Здесь же описана архитектура системы Mach, использующей IPC как основу построения ядра. Глава 7 расскажет о синхронизации выполнения процессов, используемой в современных одно- и многопроцессорных системах.

Следующие четыре главы книги посвящены файловым системам. Глава 8 описывает интерфейс файловой системы с точки зрения ее пользователя, а также рассказывает о механизме vnode/vfs, определяющем взаимодействие между ядром и файловой системой. В главе 9 рассматриваются подробности реализаций различных файловых систем, в том числе оригинальной файловой системы ОС System V (s5fs), Berkley Fast File System (FFS), а также других, более редких специализированных файловых систем, использующих наилучшие возможности vnode/vfs. Глава 10 представляет большое количество распределенных файловых систем: Network File System (NFS) компании Sun Microsystems, Remote File Sharing (RFS) компании AT&T, Andrew File System, разработанной в университете Карнеги–Меллона, и Distributed File System (DFS), созданной корпорацией Transarc Corporation. Глава 11 содержит рассказ о расширенных файловых системах, использующих ведение журнала с целью достижения более высокого уровня работоспособности и производительности, а также о новой интегрированной среде файловой системы, построенной на наращиваемых уровнях vnode.

Главы 12–15 описывают управление памятью. В главе 12 рассказано о выделении памяти ядром и приведены некоторые интересные алгоритмы выделения памяти. В главе 13 представлено понятие виртуальной памяти, некоторые особенности ее использования проиллюстрированы на примере системы 4.3BSD. Глава 14 посвящена описанию построения виртуальной памяти в SVR4 и Solaris, глава 15 расскажет о моделях памяти в системах Mach и 4.4BSD. В этом разделе вы также увидите анализ эффективности таких аппаратных возможностей, как буфер ассоциативной трансляции и кэш-память.

Последние две главы книги затрагивают подсистему ввода-вывода. Глава 16 описывает работу драйверов устройств, взаимодействие между ядром и подсистемой ввода-вывода, а также интерфейс драйверов устройств системы SVR4. В главе 17 приведено описание STREAMS, используемых для написания сетевых протоколов, а также сетевых драйверов или драйверов терминалов.

Некоторые обозначения, принятые в книге

Все системные вызовы, библиотеки подпрограмм, а также команды оболочки выделены специальным моношириным шрифтом (например, `fork`, `fopen`, `ls -l`). Имена внутренних функций ядра, переменных и примеры кода также оформлены моношириным шрифтом (например, `ufs_lookup()`). Новые термины выделяются курсивом. Имена файлов и каталогов также выделяются шрифтом (например, `/etc/passwd`). На рисунках сплошными линиями показаны прямые указатели, в то время как прерывистые линии указывают на то, что взаимосвязь между их начальной и конечной точкой является только косвенной.

Несмотря на все усилия автора возможно существование в книге некоторого количества ошибок. Присылайте все поправки, комментарии и предложения на адрес электронной почты автора vahalia@acm.org.

Благодарности

В создании книги участвовало много людей. В первую очередь я хочу поблагодарить моего сына Рохана и мою жену Аркану, чье терпение и любовь сделали написание этой книги возможным, ведь самым сложным для меня оказалось просиживание вечеров и выходных над ее созданием вместо того, чтобы провести это время со своей семьей. Они разделили вместе со мной этот тяжелый труд и постоянно поддерживали меня. Я также хочу поблагодарить моих родителей за их любовь и поддержку.

Далее я хотел бы поблагодарить моего друга Субода Бапата (Subodh Bapat), который дал мне уверенность в осуществлении этого проекта. Именно он помог мне сконцентрировать внимание на проекте и потратил большое количество своего времени на советы, консультации и поддержку. Я должен

также поблагодарить его за предоставление доступа к инструментам, шаблонам и макросам, использующимся в этой книге, за его труд «Объектно-ориентированные сети» [2], за тщательную обработку предварительных версий моего труда и за консультации по стилю изложения материала.

Для улучшения этой книги было потрачено время и использован опыт не одного рецензента. Книга имела несколько предварительных вариантов, в ходе изучения которых было получено большое количество комментариев и предложений. Я хочу поблагодарить Питера Салюса (Peter Salus) за его постоянную поддержку и консультирование, а также Бенсона Маргулиса, Терри Ламберта (Terry Lambert), Марка Эллиса (Mark Ellis) и Вильяма Балли (William Bully) за помощь в создании содержания и организации этой книги. Я также хочу поблагодарить Кейт Бостик (Keith Bostic), Еви Немет (Evi Nemeth), Пэт Парсегян (Pat Parseghian), Стивена Раго (Steven Rago), Марго Сельцер (Margo Seltser), Ричарда Стивенса (Richard Stevens) и Льва Вэйзблита (Lev Vaitzblit), написавших рецензии на отдельные части книги.

Я хочу поблагодарить моего менеджера Перси Цельника (Percy Tzelnic) за поддержку и понимание. Также я хочу выразить признательность своему издателю Аллану Апту (Alan Apt) не только за появление этой книги, но и за помочь на каждом этапе ее создания, а также остальному коллективу издательств Prentice-Hall и Spectrum Publisher Services и особенно Ширли МакГайр (Shirley McGuire), Сондре Чавес (Sondra Chavez) и Келли Риччи (Kelly Ricci) за их помощь и поддержку.

Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Bapath, S. G., «Object-Oriented Networks», Prentice-Hall, 1994.
3. Goodheart B., Cox J., «The Magic Garden Explained — The Internals of UNIX System V Release 4», An Open System Design, Prentice-Hall, 1994.
4. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.

Глава 1

Введение

1.1. Введение

В 1994 году компьютерное сообщество отметило двадцатипятилетие с момента появления операционной системы UNIX. После своего рождения в 1969 году система была перенесена на множество различных аппаратных платформ, появилось большое количество ее реализаций, созданных коммерческими компаниями, учебными заведениями и научно-исследовательскими организациями. Система UNIX начала свое развитие с небольшого набора программ и за годы переросла в гибкую ОС, использующуюся для работы огромного количества программных сред и приложений. На сегодняшний день существуют варианты UNIX для самых различных аппаратных платформ, начиная от небольших встроенных процессоров, рабочих станций и настольных систем и заканчивая высокопроизводительными многопроцессорными системами, объединяющими большое количество пользователей.

Операционная система UNIX — это среда выполнения и системные службы, под управлением которых функционируют входящие в набор ОС пользовательские программы, утилиты и библиотеки. Эта книга рассказывает о разработке и реализации самой системы, не описывая программы и утилиты, выполняющиеся под ее управлением. Система UNIX начала свою жизнь в недрах Bell Telephone Laboratories (BTL), которая и ответственна за все ее ранние реализации. Сначала система получила распространение среди нескольких компаний и учебных заведений. Именно этот факт повлиял на дальнейшее развитие различных реализаций UNIX. Все они поддерживали только набор внутренних интерфейсов, приложений и возможностей, обычно ожидаемых от стандартной «системы UNIX». Внутреннее устройство каждой было различным, отличаясь семантикой интерфейсов и набором предоставляемых «дополнительных» возможностей. В этой книге уделяется особое внимание описанию основополагающих реализаций UNIX, таких как *System V Release 4 (SVR4)* компании Novell, *Berkeley Software Distribution (4xBSD)* Калифорнийского университета и *Mach* университета Карнеги—Меллона. Здесь также обсуждается большое количество коммерческих ва-

риантов системы, таких как *SunOS* и *Solaris* компании Sun Microsystems, *Digital UNIX* компании Digital Equipment Corporation и *HP-UX* корпорации Hewlett-Packard Corporation.

Эта глава посвящена описанию развития систем UNIX. Сначала вы увидите краткий рассказ о рождении, становлении и принятии системы компьютерным сообществом. Затем будут описаны факторы, повлиявшие на ее эволюцию, и, наконец, будут отмечены возможные направления дальнейшего развития UNIX.

1.1.1. Краткая история

Перед тем как начать изучение операционной системы UNIX, полезно немного узнать о ее истории и эволюции. В следующих разделах мы проследим развитие UNIX от зарождения ее в недрах AT&T до нынешнего современного, немного хаотичного состояния в виде набора операционных систем, реализованных под различные платформы, различными авторами и существующих в самых различных вариантах. Более полное изложение истории развития UNIX можно найти в других публикациях, например в книге Питера Салюса (Peter Salus) «A Quarter Century of UNIX» [1]. Эта глава расскажет только об основных событиях, произошедших в истории системы UNIX.

1.1.2. Начало

В конце 60-х годов Bell Telephone Laboratories совместно с компанией General Electric и Массачусетским технологическим институтом организовали совместный проект, целью которого была разработка операционной системы под названием *Multics* [2]. Проект был аннулирован в марте 1969, но некоторые разработчики из BTL продолжили поиск интересных идей для последующей реализации. Один из участников проекта, Кен Томпсон (Kenneth Thompson), написал игровую программу под названием «Космическое путешествие» («Space Travel») и нашел для ее запуска малоиспользуемый в те годы компьютер *PDP-7* (созданный Digital Equipment Corporation). Однако этот компьютер не имел собственной среды разработки программ, поэтому Томпсон перенес свою программу на другую машину, Honeywell 635, работавшую под управлением ОС *GECOS*, и затем создал перфоленту со своей программой для *PDP-7*.

С целью совершенствования разработки «Космического путешествия» Томпсон совместно с Денисом Ритчи (Dennis Ritchie) начал разработку рабочей среды для *PDP-7*. Первым созданным компонентом стала простая файловая система, дальнейшее развитие которой впоследствии появилось в первых версиях UNIX и сейчас известно как *System V file system (s5fs)*. Чуть позже ими были добавлены подсистема обработки, простой командный

интерпретатор под названием *shell* (который позже развился в Bourne shell, [4]), а также небольшой набор утилит. Система стала самодостаточной и не требовала больше среды GECOS. Авторы назвали свою систему *UNIX* в честь проекта Multics¹.

В следующем году Томпсон, Ритчи и Йозеф Оссанна (Joseph Ossanna) добились того, что BTL приобрела машину Digital *PDP-11* для обработки документации в отделе патентов. Затем они экспортировали UNIX на эту машину и добавили несколько утилит обработки текста, в том числе редактор *ed* и инструмент отображения текста *gtodoff*. Томпсон также разработал новый язык программирования *B* (усовершенствовав тем самым язык *BPCL*, [5]) и написал на нем несколько первых ассемблеров и утилит. Язык *B* был интерпретируемым, вследствие чего обладал низкой производительностью. Позже Ритчи усовершенствовал свою разработку, назвав результат *C*. Язык *C* поддерживал типы и структуры данных. Успех языка *C* являлся основой успешного развития системы UNIX.

ОС UNIX становилась все более популярной внутри BTL. В ноябре 1971 года Ритчи и Томпсон под руководством Дуга Мак-Илроя (Doug McIlroy) опубликовали первую редакцию «Руководства для программиста UNIX». В дальнейшем появилось ровно 10 изданий этого руководства — по количеству версий систем UNIX, созданных в лабораториях BTL.

Первые несколько реализаций ОС использовались только внутри BTL. Третья версия, вышедшая в феврале 1973, включала в себя компилятор языка *C* под названием *cc*. В том же году система была переписана на языке *C*, в результате чего в ноябре того же года появилась версия 4. Это в высокой степени повлияло на будущий успех системы UNIX. Томпсон и Ритчи создали первую работу о UNIX под названием «The UNIX Timesharing System» [6], которая была представлена на симпозиуме по операционным системам (*ACM Symposium on Operating Systems, SOSP*) в октябре 1973 года и опубликована в июле 1974 года в *Communications of ACM*². Эта работа стала первой публикацией, возвестившей миру о UNIX.

¹ Питер Салюс в своей книге рассказывает, что этимология слова UNIX обязана своим происхождением шуткам коллег-хакеров. Multics была многопользовательской системой, а первая UNIX работала всего с двумя пользователями. Латинский корень «много» заменили на «один» («единственный»). Получилось — UNICS (Uniplexed Information and Computing Service). Название понравилось, поскольку напоминало об участии сотрудников Bell Labs в разработке Multics. Позже UNICS было изменено на UNIX. Том ван Влек, лично зная всех персонажей, «из первых рук» уточняет (<http://www.multicians.org/unix.html>), что название UNIX предложил Брайан Карнеган как «Multics without balls». История имела продолжение. Когда хакерская коалиция распалась, Ричард Столлмен решил создать систему, «совместимую с UNIX», чтобы она была переносимой и чтобы пользователи UNIX могли бы легко на нее перейти. Аббревиатура GNU была выбрана для нее в соответствии с хакерской традицией как рекурсивный акроним выражения «GNUTs Not UNIX» («GNU — это не UNIX»). — Прим. ред.

² Позже эта работа была подвергнута изменению и переиздана в виде книги [7].

1.1.3. Распространение

В 1956 году в результате антимонопольной судебной тяжбы Министерства юстиции США против компаний AT&T и Western Electric Company корпорация AT&T вынуждена была подписать согласительный документ с правительством. Это соглашение запрещало компании производить какое-либо оборудование, не относящееся к телефонам или телеграфу, а также вести дела в областях, отличных от «общих служб доставки сигнала».

В результате компания AT&T не могла заниматься продажей продукции, относящейся к вычислительной технике. С другой стороны, конференция SOSP показала наличие большого спроса на программное обеспечение UNIX. Корпорация AT&T распространяла свою систему в высших учебных заведениях для использования в образовательных и научных целях под простыми лицензионными условиями, что не противоречило подписенному соглашению. Компания не рекламировала созданную систему и не продвигала ее на рынке сбыта, а также не вела поддержку ее реализаций. Одним из первых лицензию на использование UNIX от компании AT&T получил университет Беркли (Berkeley) в Калифорнии. Это произошло в декабре 1973 года.

Такие условия дали возможность системе UNIX довольно быстро распространиться по всему миру. О широкой географии использования ОС говорит тот факт, что к 1975 году система была установлена в таких учебных заведениях, как Еврейский университет в Иерусалиме, университет Нового Южного Уэльса (Австралия) и университет Торонто (Канада). Первым переносом системы на новую аппаратную платформу стала ее реализация для машины *Interdata*, полностью выполненная университетом Уоллонгонга (Wollongong) самостоятельно в 1976 году. Годом позже аналогичный процесс был осуществлен Ритчи и Стивом Джонсоном в BTL.

Седьмая версия операционной системы UNIX, выпущенная в январе 1979 года, являлась первой настоящей переносимой ОС, что послужило большим толчком в ее дальнейшем развитии. Эта версия изначально работала на PDP-11 и Interdata 8/32. Она, с одной стороны, была устойчивее и намного функциональнее своей предшественницы, версии 6, однако, с другой стороны, работала значительно медленнее. Существовало несколько лицензий ОС, позволяющих увеличить ее производительность при использовании в различных областях. Компания AT&T позже вставила многие из этих разработок в последующие версии UNIX. Кооперация между разработчиками и пользователями системы (которая, к сожалению, перестала быть возможной после коммерческого успеха ОС) была ключевым фактором, обусловившим быстрый рост и увеличение популярности UNIX.

Вскоре система UNIX была импортирована на другие аппаратные платформы. Корпорация Microsoft совместно с Santa Cruz Operation (SCO) перенесла систему на компьютеры под управлением процессора Intel 80x86, в результате чего появилась ОС XENIX – один из первых коммерческих вариантов UNIX. В 1978 году компания Digital представила свой новый 32-разрядный компьютер VAX-11 и предложила группе разработчиков отделения BTL в Холмделе,

штат Нью-Джерси, перенести UNIX на эту машину. Так возник первый вариант системы UNIX для 32-разрядного компьютера, который был назван *UNIX/32V*. Копия этой системы была передана Калифорнийскому университету, где была переработана и стала основой ОС 3BSD, появившейся в 1979 году.

1.1.4. BSD

Калифорнийский университет в Беркли получил одну из первых лицензий на операционную систему UNIX в декабре 1974 года. За несколько лет группа выпускников университета, в состав которой входили Билл Джой (Bill Joy) и Чак Хэлей (Chuck Haley), разработала несколько утилит для этой системы, в том числе редактор *ex* (позже сопровождавшийся *vi*) и компилятор языка Паскаль. Все созданные приложения были собраны в единый пакет под названием *Berkeley Software Distribution (BSD)* и продавались весной 1978 года по цене \$50 за одну лицензию. Первые версии BSD (2BSD появилась в конце 1978 года) состояли из дополнительных приложений и утилит, сама же операционная система тогда еще не подвергалась изменению или передаче. Одной из первых разработок Джоя стала оболочка *C shell* [7], имевшая такие возможности, как управление заданиями и ведение истории команд, отсутствовавшие в то время в оболочке Bourne.

В 1978 году университет Беркли приобрел машину VAX-11/780 и операционную систему UNIX/32V, портирование которой на этот компьютер осуществила группа BTL в Холмделе, штат Нью-Джерси. Компьютер VAX имел 32-разрядную архитектуру и мог использовать до 4 Гбайт адресного пространства, однако имел всего лишь 2 Мбайт физической памяти. Примерно в то же время Озалп Бабаоглу (Ozalp Babaoglu) разработал систему страничной виртуальной памяти для VAX и добавил ее в ОС UNIX. В результате в конце 1979 года появилась новая версия ОС, 3BSD, которая стала первой операционной системой, созданной университетом Беркли.

После появления системы виртуальной памяти агентство DARPA (Defence Advanced Research Projects Agency) начало финансирование разработки систем UNIX в Беркли. Одной из главных задач, стоявших перед DARPA, являлась интеграция в создаваемой системе набора протоколов *TCP/IP* (Transmission Control Protocol/Internet Protocol). При финансовой поддержке агентства DARPA Berkeley выпустил несколько вариантов системы BSD, объединенных под общим названием *4BSD*: *4.0BSD* в 1980 году, *4.1BSD* в 1981¹, *4.2BSD* в 1983, *4.3BSD* в 1986 и *4.4BSD* в 1993.

Команде Беркли принадлежало авторство большого количества важных технических усовершенствований системы. Кроме уже упомянутых новшеств (виртуальной памяти и интеграции протоколов TCP/IP), в системе BSD UNIX была представлена файловая система *Fast File System (FFS)*, надежная реализация сигналов и технология сокетов. В 4.4BSD оригинальная разработка виртуальной памяти была заменена новой версией, базирующей-

¹ Эта версия системы, в свою очередь, имела три различных варианта: 4.1a, 4.1b и 4.1c.

ся на Mach (см. раздел 1.1.7), а также были добавлены другие возможности, например файловая система с ведением журнала.

Работа над системой UNIX производилась группой CSRG (Computer Science Research Group). Однако после выпуска 4.4BSD группа приняла решение закрыть проект и завершить разработку систем UNIX. Наиболее важными причинами этого решения были:

- ◆ уменьшение финансирования и выделения грантов;
- ◆ возможности, представленные в системе BSD, к тому времени уже были реализованы во многих коммерческих проектах;
- ◆ операционная система становилась слишком большой и сложной для разработки и поддержки силами небольшой группы программистов.

Для продвижения и продажи 4.4BSD как коммерческого продукта была создана компания Berkeley Software Design, Inc (BSDI). К тому времени почти весь код оригинальной системы UNIX был заменен разработчиками из Беркли, поэтому компания BSDI утверждала, что созданная ею версия системы *BSD/386* полностью свободна от лицензионных ограничений AT&T. Однако подразделение AT&T, UNIX System Laboratories, занимавшееся разработками UNIX, все-таки подало иск против BSDI и управляющего совета Калифорнийского университета. Компания обвиняла их в нарушении авторских прав, невыполнении условий соглашения, в незаконном перехвате коммерческих секретов, а также выступала резко против использования BSDI телефонного номера 1-800-ITS-UNIX для продажи исходных кодов своей системы. Университет подал ответный иск, вследствие чего продажи *BSD/386* были приостановлены. В результате 4 февраля 1994 года обе стороны договорились между собой вне здания суда и отменили иски друг к другу, после чего компания BSDI анонсировала новый продукт, *4.4BSD-lite*, продаваемый без исходных кодов примерно по \$1000 за пакет¹.

¹ Разработчики BSD UNIX были вынуждены вести юридическую битву с AT&T, что замедлило темпы развития системы, а успех SVR5 и альянса единой UNIX во главе с SCO к 2000 году окончательно выбил почву из-под ног университета Беркли и приверженцев коммерциализации этой ветви развития UNIX. Спецификация 4.4BSD используется во многих коммерческих ОС, но суммарный объем продаж исчезающе мал, прежде всего, по причине сильнейшей конкуренции со стороны бесплатных систем FreeBSD (www.freesbsd.org), OpenBSD (www.openbsd.org) и NetBSD (www.netbsd.org). Последние версии FreeBSD (в основе которой лежит 4.4BSD-lite) ни в чем не уступают коммерческим разработкам. FreeBSD 4.7 (октябрь 2002) улучшена относительно версии 4.6, поддерживает архитектуры Intel и ALPHA и может быть установлена из Сети с одного из множества анонимных ftp-серверов. FreeBSD 5.0 (январь 2003) явилась результатом трехлетнего труда и работает на 64-разрядных архитектурах Intel и SPARC. В ее файловой системе UFS2, следующем поколении UFS, преодолен терабайтный барьер на размер файла. Функции резервного копирования реализуются на основе Background filesystem checking (btrfsck) и моментальных снимков. Модель безопасности включает экспериментальный сервис Mandatory Access Controls (MAC) и предоставляет более гибкие возможности администрации. Работа в многопроцессорных системах обеспечивается высоким уровнем грануляции ядра. Такие новшества, как инфраструктура устройств памяти GEOM и файловая система виртуальных устройств DEVFS, облегчают управление устройствами памяти. — Прим. ред.

1.1.5. System V

Вернемся снова к истории AT&T. Судебные битвы корпорации с министерством достигли кульминации в 1982 году, с выходом разграничивающего постановления. Во исполнение его предписаний корпорация Western Electric распалась на части, AT&T лишилась своих региональных отделений, которые стали основой «детей Белла» («Baby Bells»), Bell Telephone Laboratories была отделена и переименована в AT&T Bell Laboratories. Также корпорация AT&T отныне получила возможность заниматься бизнесом в компьютерных отраслях.

Работа над системой UNIX по-прежнему велась группой разработчиков из BTL, но создание внешних реализаций системы постепенно перешло к UNIX Support Group, затем к UNIX Development Group и, наконец, к AT&T Information Systems. Этими группами были созданы *System III* в 1983 году, *System V* в 1983, *System V Release 2 (SVR2)* в 1983 и *System V Release 3 (SVR3)* в 1987. Корпорация AT&T агрессивно продвигала System V на рынок операционных систем. На основе этой ОС было разработано несколько различных коммерческих вариантов UNIX.

В System V были впервые представлены многие новые возможности и средства. Например, сегментная реализация виртуальной памяти отличалась от варианта, предлагаемого BSD. В SVR3 появилось средство взаимодействия процессов (поддерживающее совместное использование памяти, семафоров и очередей сообщений), удаленное разделение файлов, общие библиотеки, а также поддержка STREAMS для драйверов устройств и сетевых протоколов. Последняя версия операционной системы *System V Release 4 (SVR4)* будет подробнее описана в разделе 1.1.10¹.

1.1.6. Коммерциализация

Увеличивающаяся популярность UNIX вызвала интерес к этой ОС со стороны нескольких компаний, которые приняли решение начать выпуск и продажу своих собственных вариантов системы. Производители брали за основу одну из базовых реализаций UNIX от AT&T или Беркли, переносили вы-

¹ В марте 2000 года компания SCO представила новую ОС на основе *System V Release 5 (SVR5)*, усовершенствованной SVR4.2, которая включила в себя более производительную сетевую подсистему — улучшенные механизмы синхронизации процессов, планирования и управления памятью, поддержку до 32 процессоров в сервере и технологий NUMA и I2O, файлов размером до 1 Тбайт, 76 800 Тбайт внешней памяти, разбиваемой до 512 логических томов. SVR5 работает с 64-гигабайтной RAM (прямая адресация к 4 Гбайт, остальная — в режиме расширенной физической адресации (PAE)). Технология Multi-path I/O позволяет использовать несколько контроллеров ввода-вывода и удвоение дисков. Поддержана 64-разрядная журнальная отказоустойчивая файловая система VxFS компании Veritas, 64-разрядный API и системные вызовы. Дополнительно: графические Java-консоли администрирования (SCoAdmin), программная поддержка RAID, кластеризация, файл-серверы и серверы печати, совместимые с Windows, и многое другое. — Прим. ред.

бранную систему на свои машины и добавляли в ОС свои дополнительные возможности. Первой компанией, начавшей в 1977 году продажу собственного коммерческого варианта UNIX, стала Interactive Systems. Продукт назывался *IS/1* и работал на PDP-11.

В 1982 году Билл Джой покинул Беркли и стал одним из создателей корпорации Sun Microsystems, выпустившей свой вариант UNIX на основе 4.2BSD и назвавшей его *SunOS* (позже компанией была разработана еще одна версия системы, *Solaris*, базирующаяся на 4.3BSD). Компании Microsoft и SCO совместно создали систему *XENIX*. Позже SCO перенесла SVR3 на платформу 80386 и выпустила эту систему под названием SCO UNIX. В 80-х годах появилось множество различных коммерческих вариантов системы UNIX, в том числе *AIX* от IBM, *HP-UX* от Hewlett-Packard Corporation и *ULTRIX* (выпущенной вслед за *DEC OSF/1*, переименованной позже в *Digital UNIX*) от компании Digital¹.

В коммерческих вариантах UNIX были представлены многие новые возможности, некоторые из которых позже были встроены и в базовые системы. В SunOS была реализована сетевая файловая система Network File System (NFS), интерфейс vnode/vfs, поддерживающий работу с различными типами файловых систем, а также новая архитектура виртуальной памяти, адаптированная позже в SVR4. Система AIX одна из самых первых начала поддержи-

¹ Выходя за «рамки повествования книги», следует заметить, что на сегодняшний день роли на рынке коммерческих систем UNIX достаточно четко определились. Интерес к коммерческим вариантам UNIX усилился в связи с успехом Linux. Радикальные перемены произошли в 1997–2000 годах, когда компания SCO представила System V Release 5, являющуюся усовершенствованной версией SVR4.2, и на ее базе выпустила ОС UnixWare 7. В настоящее время SCO оккупировала 80% рынка ОС для серверов UNIX на платформе Intel. UnixWare 7 наследовала лучшее из UnixWare 2 и SCO OpenServer 5. UnixWare 7 поддерживает процессоры Pentium, 64-разрядный Merced и 64-разрядную RISC-архитектуру (PowerPC). SCO организовала альянс с IBM и Sequent (IBM собиралась принести в жертву AIX во благо стандартного диалекта UNIX для платформ IA-32/64, поскольку компьютеры с процессорами Intel составляют половину рынка систем UNIX), Intel и др. по разработке единой UNIX, условно названной Monterey. Результатом должно было стать объединение UnixWare, элементов AIX и Sequent Duxix. Принять новую ОС в качестве стандарта согласились Compaq, Siemens Nixdorf, Hyundai, ICL HPS, Fujitsu/ICL. Ряд производителей объявили о разработке версий своих продуктов конкретно под UnixWare, в том числе Infromix, Netscape, Sybase и Oracle. Увы, в 2001 году IBM отказалась от участия в проекте, и он был прекращен. Теперь, спустя два года, SCO пытается отсудить \$1 млрд за то, что IBM якобы использовала конфиденциальные данные, полученные от SCO, в собственных продуктах (AIX 5). А пока суд да дело, Hewlett-Packard и Sun Microsystems пошли своей дорогой, продвигая собственные решения, HP/UX и Solaris. Впрочем, эти компании не делают погоды, поскольку рынок сбыта Solaris узок, внимание компаний к платформе Intel пониженнное (более 1 млн зарегистрированных пользователей), поэтому доля Sun на платформе Intel оценивается не более чем в 10%. В результате ее маркетинговые акции а-ля «бесплатная Solaris» заканчиваются возвращением в русло коммерции. Ей также требуется перенос всех возможностей со SPARC на IA-64 (на процессорах Intel поддержан только 32-разрядный режим). HP вообще не имеет версии для Intel, что отодвигает сроки ее активного внедрения на рынок. А после нужно будет решать проблему отсутствия приложений. — Прим. ред.

вать файловую систему с ведением журналов (Journaling File System, JFS) для UNIX. ULTRIX стала одной из первых систем UNIX с поддержкой многопроцессорной архитектуры.

1.1.7. Mach

Одной из главных причин популярности системы UNIX являлись ее простота и небольшой размер, сочетающиеся со множеством полезных утилит. Но после того как система стала поддерживать все большее количество возможностей, ее ядро постепенно становилось большим, сложным и громоздким. Многие специалисты пришли к мнению, что развитие UNIX постепенно уходит от предполагаемого пути, приведшего когда-то систему к успеху.

В середине 80-х годов разработчики из университета Карнеги–Меллона [8] в Питтсбурге приступили к созданию новой операционной системы под названием *Mach*. Их целью была разработка микроядра, включающего небольшой набор утилит, служащих для реализации остальных системных функций пользовательского уровня. Архитектура Mach должна была поддерживать интерфейс программирования UNIX, работать как на однопроцессорных, так и на многопроцессорных системах, а также подходить для распределенных сред. Разработчики надеялись, что, создав новую систему «с нуля», они могут избежать множества проблем, имевшихся в текущих вариантах UNIX.

Одним из первых приближений к реализации задуманных планов стало создание микроядра, в котором были выделены некоторые основные функции, в то время как большинство возможностей системы исходило от набора внешних по отношению к ядру процессов, называемых *серверами*. Система Mach также имела еще одно существенное преимущество: она никак не зависела от лицензий AT&T, что сделало ее привлекательной для многих производителей. Самой популярной версией системы стала *Mach 2.5*, и многие коммерческие ОС, такие как *OSF/1* или *NextStep*, были созданы на ее основе. Ранние версии системы имели монолитные ядра с поддержкой интерфейса 4.4BSD UNIX на высоком уровне. Первой реализацией идеи микроядра стала система *Mach 3.0*.

1.1.8. Стандарты

Распространенность различных реализаций UNIX привела к появлению проблем совместимости. Несмотря на то что все существующие варианты на первый взгляд «похожи на UNIX», на самом деле они имеют существенные различия между собой. Существование отличий было заложено изначально, за счет наличия двух веток развития UNIX, «официальной» системы AT&T System V и альтернативного варианта BSD, создаваемого в Беркли. Появление коммерческих вариантов UNIX еще более усложнило проблему.

Системы System V и 4BSD существенно отличались. Они имели различные несовместимые между собой файловые системы, реализации поддержки сетей и архитектуры виртуальной памяти. Некоторые различия были обусловлены дизайном ядра систем, но большинство из них находились на уровне программирования интерфейса. Это привело к невозможности создания сложных приложений, работающих без внесения каких-либо изменений в обеих операционных системах.

Все коммерческие варианты UNIX строились на основе либо System V, либо BSD, к которым производители добавляли дополнительные возможности. Именно эти добавления часто оказывались непереносимыми на иные платформы. В результате создатели приложений тратили огромное количество времени и усилий, для того чтобы их программы нормально функционировали в различных реализациях UNIX.

Для решения проблемы необходимо было разработать некий стандартный набор интерфейсов, чем и занялись несколько групп энтузиастов. В результате миру предстало множество стандартов, столь же многочисленных и отличающихся друг от друга, как и существовавшие в те времена варианты UNIX. Однако большинство производителей признала только несколько из созданных стандартов, в том числе «*Определение интерфейса System V*» (System V Interface Definition, SVID) компании AT&T, спецификации организации IEEE под названием *POSIX* и «*Руководство по переносу X/Open*» (X/Open Portability Guide) консорциума X/Open.

В каждом из стандартов описывалось взаимодействие между программами и операционной системой и не затрагивался вопрос реализации самого интерфейса взаимодействия. В них определялись наборы функций и подробно приводились их конструкции. Совместимые системы должны удовлетворять требованиям, изложенным в стандартах, однако реализация необходимых функций могла быть произведена как на уровне ядра, так и на уровне библиотек пользователя.

Стандарты также определяли поднабор функций, предлагаемых большинством систем UNIX. Теоретически, если пользователь будет использовать при написании приложения только те функции, которые входят в этот набор, то созданное приложение будет переносимо на любую систему, совместимую со стандартами. Это заставляло разработчиков программ использовать дополнительные возможности конкретного варианта системы, а также производить оптимизацию своих программ под конкретную аппаратную платформу или операционную систему только в том случае, если их исходные коды легко переносимы.

Стандарт SVID представляет собой подробную спецификацию программного интерфейса System V. Корпорация AT&T выпустила три версии стандарта — *SVID1*, *SVID2* и *SVID3*, описывающие соответственно ОС SVR2, SVR3 и SVR4 [10]. AT&T предоставила возможность производителям систем называть свои продукты System V только в том случае, если они отвечают

требованиям SVID. Корпорация также выпустила пакет *System V Verification Suite (SVVS)*, который проверял операционные системы на соответствие SVID.

В 1986 году организация IEEE поручила специальному комитету разработать и опубликовать стандарты на среды операционных систем. Для их обозначения было придумано название POSIX (Portable Operating System based on UNIX, что переводится как «Переносимые операционные системы, основанные на UNIX»). Эти документы описывали компоненты ядра систем SVR3 и 4.3BSD. Стандарт *POSIX1003.1*, более известный как *POSIX.1*, был опубликован в 1990 году [11]. Многие производители приняли этот стандарт, так как он не ограничивался каким-то одним вариантом системы UNIX.

X/Open – это международный консорциум производителей компьютерной техники и программного обеспечения. Он был сформирован в 1984 году. Его целью являлась не только разработка новых стандартов, но и создание открытой среды *Common Applications Environment* (Общей программной среды, CAE), базирующейся на уже существующих стандартах. Консорциум опубликовал семитомный труд «X/Open Portability Guide» (XPG), последнее (четвертое) издание которого вышло в 1993 году [12]. Материал руководства был основан на стандарте POSIX.1, расширял его и описывал многие дополнительные области, такие как интернационализация, оконные интерфейсы и обработка данных.

1.1.10. OSF и UI

В 1987 году корпорация AT&T, осознавая непринятие общественностью ее лицензионной политики, принимает решение о закупке 20% акций Sun Microsystems. AT&T и Sun решают заняться совместной разработкой SVR4, следующей версии операционной ОС AT&T System V UNIX. Корпорация Sun объявляет, что будущая операционная система станет базироваться на SVR4, в отличие от SunOS, основу которой составляла ранее система System V.

Эти заявления вызвали бурную реакцию со стороны других производителей систем, которые поняли, что созданное объединение даст корпорации Sun огромное преимущество перед остальными производителями. В ответ группа компаний, в которую входили Digital, HP, IBM, Apollo и другие, объявила в 1988 году о создании объединения *Open Software Foundation (OSF)*. OSF финансировалась компаниями-основателями. Основной задачей организации стала разработка операционной системы, пользовательской и распределенной вычислительной среды, не зависящей от ограничений, накладываемых лицензионными соглашениями AT&T. OSF распространила среди своих членов *Request for Technology* (Запрос на технологии, RFT) и затем выбрала из полученных предложений самые лучшие независимо от того, на кого из производителей работал их автор.

В ответ корпорации AT&T и Sun совместно с другими производителями систем, основанных на System V, в срочном порядке основали свою органи-

зацию, названную *UNIX International (UI)*. Ее основной целью было продвижение системы SVR4 на рынке, а также выбор дальнейшего направления развития UNIX System V. В 1990 году организация UI выпустила труд под названием *UNIX System V Road Map*, в котором были выделены основные направления будущего развития системы UNIX.

В 1989 году OSF представила графический пользовательский интерфейс *Motif*, положительно встреченный многими пользователями. Позже организация выпустила первую версию своей операционной системы под названием *OSF/1*. Первая версия OSF/1 базировалась на Mach 2.5, имела совместимость с 4.3BSD и обладала некоторыми возможностями IBM AIX. Представленная система имела множество дополнительных возможностей, не поддерживаемых в SVR4, таких как полная поддержка многопроцессорных систем, динамическая загрузка и монтирование томов. В планах членов организации UI была дальнейшая разработка коммерческих операционных систем, базирующихся на OSF/1.

Объединения OSF и UI начинали с весьма высоких целей, но все равно очень быстро столкнулись с общими, не зависящими от них проблемами. Экономический спад начала 90-х, экспансия Microsoft Windows на рынок операционных систем резко уменьшили рост UNIX-систем. Организация UI ушла из компьютерного бизнеса в 1993 году, а объединение OSF было вынуждено расстаться с большинством амбициозных планов (в том числе и планов по созданию Распределенной среды управления, *Distributed Management Environment*). Одной из основных систем, основанных на OSF, стала DEC OSF/1, созданная компанией Digital в 1993 году. Позже компания приняла решение удалить из этой системы многие возможности, отличающие ее от своей ОС, и в 1995 году изменила имя системы на *Digital UNIX*.

1.1.10. SVR4 и ее дальнейшее развитие

В 1989 году вышла первая версия совместно разработанной корпорациями AT&T и Sun системы System V Release 4 (SVR4). Эта система объединила в себе возможности SVR3, 4BSD, SunOS и XENIX. В SVR4 также были добавлены новые функции, такие как изменение состава классов в режиме реального времени, командный интерпретатор *Korn shell* и новые возможности подсистемы STREAMS. В следующем году AT&T основала компанию UNIX Systems Laboratories для разработки и продажи систем UNIX.

В 1991 году компания Novell, Inc., создатель сетевой операционной системы NetWare для персональных компьютеров, приобрела часть акций USL и основала совместное предприятие под названием Univel. Целью новой компании стало создание версии SVR4 для настольных систем, интегрированной с ОС NetWare. Такая система была разработана в конце 1992 года и получила название *UnixWare*. После этого было выпущено еще несколько вариантов системы SVR4. Последний вариант, *SVR4.2/ES/MP*, предлагает пользователям расширенную защиту и поддержку многопроцессорных систем.

В 1993 году корпорация AT&T полностью передала USL компании Novell. В следующем году Novell получила права на торговую марку UNIX и подтверждение совместимости своих операционных систем с X/Open. В 1994 году корпорация Sun Microsystems выкупила права на использование кодов SVR4 у Novell, что освободило ее от проблем, связанных с возможным нарушением лицензионных прав и совместимости со стандартами. Система Sun, основанная на SVR4, получила название *Solaris*. Ее последняя версия — это *Solaris 2.5*. Система поддерживает многие дополнительные возможности, такие как собственное многопоточное ядро и поддержка многопроцессорных систем¹.

1.2. Причины изменений системы

Система UNIX сильно преобразилась за годы своего существования. Начав с небольшой операционной среды, использовавшейся группой людей в единственной лаборатории, на сегодняшний день система UNIX стала одной из

¹ ОС Solaris была перенесена на платформу Intel с платформы SPARC. Последние ее версии представляют собой мощные и масштабируемые системы для рабочих станций, младших серверов, корпоративных серверов и суперсерверов. Многие решения Sun не имеют аналогов, технологии компаний всегда были «на гребне» (так же Java), а ее системы отличаются высокой надежностью. Но поскольку Sun приходится вести войну на всех фронтах, в первую очередь против Linux и Microsoft, она вынуждена поспевать за конкурентами. В результате производительность Solaris 8 оказывается ниже таковой для Solaris 7, надежность страдает, а проблемам интернационализации (в том числе поддержке русского языка) уделяется недостаточное внимание. За последние 4 года вышли версии для Intel 7 (или Solaris 2.7, SunOS 5.7), 8 и 9 (начиная с 2003). Solaris 7 работает на одно- и многопроцессорных системах, поддерживает до 4 Гбайт RAM и файловые системы до 1 Тбайт. Файловая система UFS с расширениями от Sun и 64-разрядной адресацией позволяет протоколировать события (по образу журналов). Недостатки низкой производительности NFS слажены за счет кэширования информации непосредственно на локальном диске (файловая система CacheFS). Solaris 8 Sun противопоставила Windows 2000 Datacenter, а Solaris 9 — Microsoft Windows 2000 Server. Самая главная новация в Solaris 8 — Live Upgrade («горячее» обновление). Она позволяет администраторам устанавливать исправления в ядро и аппаратные средства на серверах SPARC без их перезагрузки. Встроенная кластеризация до 4-х процессоров (для SPARC) позволяет увеличить производительность веб-сервера. Solaris 8 поддерживает стандарт IPv6 и спецификацию IPSec (для IPv4), стандарт Mobile IP для мобильных пользователей. Sun объявила о поддержке сервера каталогов LDAP и об отказе от службы NIS, вошедшей в общее пользование с ее же подачи. LDAP интегрирован в Solaris 9, что повышает ее производительность в сравнении с Solaris 8 в 5 раз. В Solaris 8 встроены две графические оболочки — CDE (Common Desktop Environment) и Open Windows, в Solaris 9 — графический интерфейс пользователя Web Start и консоль администрирования Solaris Management Console (SMC). Последняя версия — Solaris 9 — включила в себя множество новых возможностей. Это: протокол сетевой аутентификации Kerberos 5, протокол Secure Shell для безопасного установления связи с UNIX-машинами, поддержка RAID-массивов, расщепление и удвоение дисков, новая организация нитей, повышающая производительность многопроцессорных систем, защита от переполнения буфера — «бич» безопасности для любых систем, основанная на блокировке выполнения стековых операций определенными приложениями. Веб-сервисы, Java-сервер приложений, выполняющий программы e-бизнеса, средства управления предприятиями на основе интернет-технологий WBEM направлены на интеграцию множества компьютеров в один огромный вычислительный комплекс. — Прим. ред.

основных операционных систем, предлагаемой в самых различных вариантах многими поставщиками. В настоящее время UNIX используется на самых различных системах, начиная от небольших встроенных контроллеров и заканчивая огромными мэйнфреймами и массивно-параллельными системами. Под управлением UNIX работают самые разнообразные приложения: в офисах UNIX используется как настольная ОС; в финансовых областях с ее помощью обрабатываются крупные базы данных, а научные лаборатории применяют эту систему для сложных математических вычислений.

Изменения и рост UNIX обусловлены, в первую очередь, появлением новых задач, стоявших перед системой. Хотя на данный момент UNIX является целостной операционной системой, это не значит, что она будет оставаться неизменной в дальнейшем. Причиной постоянно происходящих изменений никак нельзя назвать изначально неверный дизайн ОС. Напротив, простота добавления в UNIX новых возможностей по мере развития технологий убедительно доказывает обратное. Не имея точного представления о цели, формах и задачах будущей системы, ее создатели начали работу с построения простых, расширяемых базовых средств, собирая и анализируя предложения по усовершенствованию системы отовсюду, от научных учреждений, коммерческих предприятий и простых пользователей.

Проведем анализ основных факторов, повлиявших на рост и совершенствование системы. В этом разделе вы увидите не только описание самих факторов, но и соображения автора относительно предполагаемых трансформаций системы UNIX в будущем.

1.2.1. Функциональные возможности

Главной причиной, подталкивающей к изменениям, является необходимость добавления новых возможностей в систему. Изначально новые функциональные средства появлялись в UNIX только как пользовательские инструменты и утилиты. Позже, когда UNIX превратилась во вполне развитую систему, разработчики стали добавлять многие дополнительные возможности прямо в ядро ОС.

Многие из новых функций системы создавались для поддержки более сложных программ. Одним из примеров таких нововведений является набор Interprocess Communication (IPC) в ОС System V, в состав которого входят поддержка разделения памяти, семафоры и очереди сообщений. Все эти возможности позволили процессам взаимодействовать, используя совместно данные, обмениваясь сообщениями и синхронизируя свои действия. Большинство современных систем UNIX также имеют несколько уровней поддержки для создания многопоточных приложений.

Возможности IPC и технологии потоков существенно помогают в разработке сложных приложений, например тех, что основаны на модели «клиент-сервер». В таких программах обычно серверная часть находится в режиме

постоянного ожидания запроса от клиентов. Если такой запрос приходит, сервер обрабатывает его и снова переходит в режим ожидания следующего. Если сервер имеет возможность обслуживания сразу несколько клиентских запросов, в этом случае предпочтительно вести их обработку параллельно. С применением технологии IPC сервер может использовать отдельный процесс для обработки каждого запроса, в то время как все выполняемые процессы могут совместно использовать одни и те же данные. Многопоточная система позволяет реализовать сервер как один процесс, имеющий несколько параллельно функционирующих потоков, использующих общее адресное пространство.

Возможно, наиболее «видимой» частью любой операционной системы является файловая система. В ОС UNIX также было добавлено множество новых возможностей, в том числе поддержка файлов *FIFO* (First In, First Out), символьных связей, а также файлов, имеющих размеры большие, чем раздел диска. Современные системы поддерживают защиту файлов, списки прав доступа, а также ограничения доступа к дискам для каждого пользователя.

1.2.2. Сетевая поддержка

За годы развития UNIX максимальным изменениям была подвергнута часть ядра, являющаяся сетевой подсистемой. Ранние версии ОС работали отдельно друг от друга и не имели возможности соединения с другими машинами. Однако распространение компьютерных сетей поставило перед разработчиками проблему необходимости их поддержки в системе UNIX. Первым занялся решением этой проблемы университет Беркли. Организация DARPA профинансировала проект встраивания поддержки TCP/IP в 4BSD. На сегодняшний день системы UNIX поддерживают большое количество сетевых интерфейсов (таких как Ethernet, FDDI и ATM), протоколов (TCP/IP, UDP/IP¹, SNA² и других) и средств (например, сокетов и STREAMS).

Появление возможности соединения с другими компьютерами во многих отношениях повлияло на операционную систему. Пользователи ОС сразу же захотели совместно использовать файлы, расположенные на соединенных между собой машинах, а также запускать приложения на удаленных узлах. Для удовлетворения этих требований развитие системы UNIX велось в трех направлениях:

- ◆ Разрабатывались распределенные файловые системы, позволявшие вести прозрачный доступ к файлам на удаленных узлах. Наиболее удачными из созданных систем оказались *Network File System (NFS)* корпорации Sun Microsystems, *Andrew File System (AFS)* университета Карнеги–Меллона и *Distributed File System (DFS)* корпорации Transarc.

¹ User Datagram Protocol/Internet Protocol, протокол пользовательских дейтаграмм/протокол Интернета.

² System Network Architecture (системная сетевая архитектура) компании IBM.

- ◆ Создавалось большое количество распределенных служб, позволяющих совместно использовать информацию по сети. Эти службы представляют собой обычные пользовательские программы, основанные на модели «клиент-сервер» и использующие удаленные вызовы процедур для активации действий на других компьютерах. Примерами таких программ являются *Network Information Service* (Сетевой информационный сервис, NIS) и *Distributed Computing Environment* (Распределенная вычислительная среда, DCE).
- ◆ Появлялись распределенные операционные системы, такие как *Mach*, *Chorus* и *Sprite*, имеющие различную степень совместимости с UNIX и продвигаемые на рынок как базовые технологии для построения будущих распределенных ОС.

1.2.3. Производительность

Постоянной движущей силой, заставляющей вносить изменения в системы, является увеличение их производительности. Конкурирующие между собой поставщики операционных систем тратят огромные усилия на демонстрацию того, что именно их ОС производительнее, чем другие. Почти все внутренние подсистемы претерпели большие изменения, выполненные с целью увеличения производительности систем.

В начале 90-х годов университет Беркли представил файловую систему *Fast File System*, благодаря интеллектуальным правилам размещения блоков на диске увеличивающую производительность системы. Позже появились более быстродействующие файловые системы, использующие внешнее размещение или технологии поддержки журналов. Увеличение вычислительных мощностей также стало основной причиной разработок в области коммуникаций между процессами, работы с памятью и многопоточных процессов. Когда для работы многих приложений оказалось недостаточно одного процессора, производители разработали многопроцессорные системы под управлением UNIX, некоторые из которых имеют сотни процессоров.

1.2.4. Изменение аппаратных платформ

Операционная система должна «шагать в ногу» с современными аппаратными технологиями. Часто это означает необходимость ее переноса на новые и более высокопроизводительные процессоры. После того как ядро UNIX было почти полностью переписано на языке C, решение этой задачи стало относительно легким. В прошлом разработчики тратили огромные усилия на отделение от общего кода программы участков, написанных для конкретного аппаратного оборудования, и включение их в отдельные модули. После проведения этих действий для переноса системы требовалось переписывать заново только отдельные модули. Обычно такие модули отвечали за обработку пре-

рываний, трансляцию виртуальных адресов, переключение контекстов и драйверы устройств.

В большинстве случаев для выживания операционной системы на рынке необходим ее постоянный перенос на вновь появляющееся оборудование. Наиболее очевидной возможностью, которую обязательно желают видеть в UNIX, является поддержка многопроцессорных систем. Ядро традиционного варианта UNIX разрабатывалось для работы на одном процессоре и не имело возможности защиты структур данных при параллельном доступе к ним одновременно нескольких процессоров. Однако позже сразу несколько производителей разработали многопроцессорные реализации системы UNIX. Большинство из них использовали традиционное ядро UNIX и добавляли собственные элементы защиты, обеспечивающие безопасность общих структур данных. Это средство называется параллелизацией. Малая часть производителей занялась построением собственного ядра на иных основах.

При более глубоком рассмотрении проблемы видно, что неравномерность развития различных аппаратных технологий оказала глубокое влияние на разработку операционных систем. С тех времен, как была создана первая версия UNIX для PDP-7, средняя скорость процессоров увеличилась примерно в 100 раз¹. Объемы памяти и дискового пространства, выделяемого для одного пользователя, возросли более чем в 20 раз. С другой стороны, скорость доступа к памяти и дискам увеличилась всего лишь в 2 раза.

В 70-х годах производительность систем UNIX ограничивалась скоростью работы процессора и размером памяти. Но вскоре в ядре системы стали использоваться такие технологии, как свопинг и, чуть позже, страничная организация памяти (технология, позволявшая выполнять большое количество процессов на малых объемах памяти). По мере развития вычислительной техники скорость доступа к памяти и процессору стала играть меньшую роль, а сама система занималась в большей степени вводом-выводом, основное время занимаясь переносом страниц между дисками и оперативной памятью. Это являлось важной причиной появления новых разработок в области файловых систем, хранения информации и архитектур виртуальной памяти, целью которых была оптимизация дисковых процедур. Именно для этого были разработаны *Redundant Arrays of Inexpensive Disks* (массивы недорогих дисков с избыточностью, RAID) и происходило быстрое распространение структурированных файловых систем.

1.2.5. Улучшение качества

Все преимущества функциональности и скорости работы можно запросто свести на нет, если в систему заложены ошибки. Разработчики вносили мно-

¹ Частота процессора Intel 4004 в первом миникомпьютере (1971 г.) составляла 1 МГц. Соответственно на 2002 год приходится говорить о росте производительности в 1000 раз. — Прим. ред.

жество изменений в дизайн систем, чтобы сделать их более устойчивыми, стараясь добиться увеличения надежности программного обеспечения.

Изначальный механизм оповещения был ненадежен и неэффективен по многим причинам. Однако позже его реализация была пересмотрена (сначала разработчиками из Беркли, затем — AT&T), и в результате появилась новая, более устойчивая система оповещения, получившая название *надежных сигналов*.

Система BSD, точно так же, как и System V, не была застрахована от возможных отказов. В системах UNIX перед записью на диск данные хранятся некоторое время в памяти. Следовательно, существует потенциальная возможность их потери в случае отказа, а также нарушения целостности файловой системы. В различных вариантах UNIX предлагается стандартная утилита `fsck(8)`, проверяющая и восстанавливающая поврежденные файловые системы. Эта операция занимает ощутимое количество времени и может длиться десятки минут на крупных серверах, имеющих диски больших объемов. Многие современные системы UNIX поддерживают файловые системы, использующие технологию поддержки журналов, что увеличивает доступность и стабильность системы и устраняет потребность в применении `fsck`.

1.2.6. Глобальные изменения

За последние три десятилетия произошли огромные изменения в принципах использования компьютеров. В 70-х годах вычислительная система представляла большой централизованный компьютер размером с комнату, поддерживающий работу пользователей, которые подключались к нему при помощи терминалов. Применялись системы разделения времени, в которых центральный компьютер разделял процессорные ресурсы среди пользователей. Терминалы представляли собой простые устройства, умеющие чуть больше, чем вывод данных в текстовом режиме.

В 80-х годах началась эра *рабочих станций*, оборудованных высокоскоростными графическими дисплеями, имеющими возможность вывода информации в нескольких окнах, в каждом из которых выполняется оболочка UNIX. Рабочие станции идеальны для интерактивного использования и имеют достаточную вычислительную мощность для работы с обычными пользовательскими приложениями. На рабочих станциях обычно в один момент времени работает один пользователь, но они могут поддерживать многих пользователей. Появление высокоскоростных сетей позволило рабочим станциям соединяться между собой, а также с другими компьютерами.

Позже появилась новая модель вычислений, называемая *клиент-серверными вычислениями*, в которой один или более мощных централизованных компьютеров, называемых *серверами*, предоставляют различные службы индивидуальным рабочим станциям, или *клиентам*. Для хранения файлов пользователей стали применяться *файловые серверы*. *Серверы приложений* — это

компьютеры, оснащенные одним или несколькими мощными процессорами, на которых пользователи могут выполнять задачи, требующие большого объема вычислений (например, решение математических уравнений). На *серверах баз данных* выполняется специальная программа, обрабатывающая запросы к базам данных, поступающие от клиентов. Обычно серверы представляют собой мощные высокопроизводительные машины с быстродействующими процессорами и большими объемами оперативной и дисковой памяти. Клиентские рабочие станции имеют меньшую производительность, размеры памяти и объемы дисков, но они, как правило, оснащаются высококачественными мониторами и предоставляют пользователю большие интерактивные возможности.

Постепенно рабочие станции становились все более производительными, различия между клиентами и серверами все больше стирались. Более того, централизованное выполнение необходимых служб на небольшом количестве серверов приводило к перегрузкам сетей и самих серверов. Результатом стало изменение подхода к построению компьютерных систем и появление новой технологии *распределенных вычислений*. При использовании этой модели компьютеры совместно предоставляют какую-либо сетевую службу. Каждый узел сети может иметь собственную файловую систему и предоставлять доступ к ней для других узлов. В результате узел функционирует как сервер по отношению к локальным файлам и как клиент по отношению к файлам, хранящимся на других узлах. Такой подход уменьшает перегрузки сети, а также количество сбоев в ее работе.

Система UNIX была адаптирована для различных моделей вычислений. Например, ранние варианты системы поддерживали только локальную файловую систему. Поддержка сетевых протоколов привела к разработке распределенных файловых систем. Некоторые из них, такие как AFS, требовали специализированных серверов. Позже файловые системы были переработаны и стали распределенными, что дало возможность каждому компьютеру в сети выступать в роли клиента и сервера.

1.2.7. Поддержка различных приложений

Система UNIX изначально разрабатывалась для применения в простых средах разделения времени, таких как исследовательские лаборатории или университеты. Системы позволяли некоторому количеству пользователей выполнять несложные программы обработки и редактирования текстов и математических вычислений. После того как UNIX обрела популярность, ею стали пользоваться для более широкого спектра приложений. К началу 90-х годов UNIX использовалась в физических и космических лабораториях, мультимедийных рабочих станциях для обработки звука и видео, а также во встроенных контроллерах, использующихся в критически важных системах.

Каждое приложение обычно имеет различные требования к системе, что подхлестнуло разработчиков систем изменять их в соответствии с этими требованиями. Мультимедийные и встроенные приложения требуют гарантий доступности ресурсов и определенной скорости отклика на запросы. Научные приложения требуют одновременной работы нескольких процессоров. Для реализации этих требований в некоторых системах UNIX появились возможности работы в режиме реального времени, такие как процессы с фиксированным приоритетом, разграничение использования процессоров и возможность сохранения данных в памяти.

1.2.8. Чем меньше, тем лучше

Одним из преимуществ оригинального дизайна системы UNIX был ее небольшой размер, простота и ограниченный набор основных функций. Изначальным подходом к системам было предоставление простых инструментов, которые можно гибко комбинировать между собой, используя такие инструменты, как конвейер (*pipe*). Однако ядро традиционного варианта системы было достаточно монолитным, следовательно, его расширение представляло собой непростую задачу. Чем больше функций добавлялось в ядро, тем оно все больше разрасталось и усложнялось, все дальше уходя от его первоначального размера, не превышающего 100 Кбайт, постепенно достигнув объема в несколько мегабайтов. Объемы памяти компьютеров стали увеличиваться, вместе с тем поставщики систем и их пользователи игнорировали этот факт. Но именно он сделал UNIX менее пригодной для использования на небольших персональных компьютерах и портативных системах.

Многие стали понимать, что такое изменение системы не предвещает ничего хорошего, так как она становится слишком большой, перенасыщенной и неорганизованной. Были потрачены большие усилия на переработку системы или на написание нового варианта, который основывался бы на оригинальной философии UNIX, но имел бы большую расширяемость и модульность. Наиболее удачной реализацией такой системы стала Mach, которая выступила основой для последующих коммерческих ОС, примерами которых являются OSF/1 и NextStep. Система Mach впоследствии использовала архитектуру микроядра (см. раздел 1.1.7), в которой небольшое ядро предоставляет средство для выполнения программ, а серверные задачи на пользовательской уровне предоставляют все остальные функции.

Не все попытки контроля размера ядра имели успех. К сожалению, микроядра не могли иметь производительность, сравнимую со скоростью работы традиционного монолитного ядра, в первую очередь, по причине издержек, накладываемых передачей сообщений. Менее амбициозные проекты, такие как модульность, ядра со страничной поддержкой и динамическая загрузка, имели больший успех, так как позволяли загружать или выгружать из памяти компоненты ядра по мере надобности.

1.2.9. Гибкость системы

В 70-х и начале 80-х годов ядро систем UNIX было недостаточно гибким. Ядро поддерживало только один тип файловой системы, набор алгоритмов планирования, а также формат выполняемых файлов (рис. 1.1). ОС имела некоторую степень гибкости только по отношению к переключателям символьных и блочных устройств, позволяя различным типам устройств иметь доступ к системе через общий интерфейс. Развитие распределенных систем в середине 80-х годов стало очевидной причиной для того, чтобы UNIX стала поддерживать как удаленные, так и локальные файловые системы. А такие возможности, как разделяемые библиотеки, потребовали поддержки различных форматов выполняемых файлов. Система UNIX стала поддерживать эти форматы, оставив традиционный *a.out* для совместимости. Одновременное сосуществование мультимедийных приложений и программ, функционирующих в режиме реального времени, потребовало поддержки планирования для различных типов приложений.

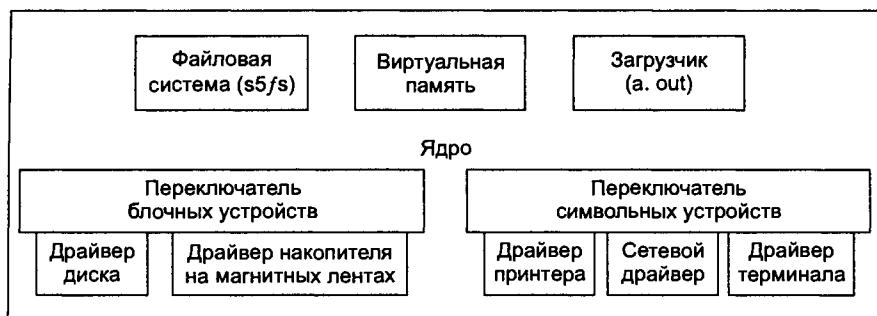


Рис. 1.1. Традиционное ядро UNIX

В итоге UNIX превращалась в более гибкую операционную систему, которая бы поддерживала несколько различных методов выполнения одной и той же задачи. Это потребовало разработки многих более гибких структур, таких как интерфейс *vnode/vfs*, системный вызов *exec*, планирование процессов, а также сегментную архитектуру памяти. Современное ядро UNIX похоже на систему, показанную на рис. 1.2. В каждом из внешних кругов представлен интерфейс, который может быть реализован различными способами.

1.3. Оглянемся назад, посмотрим вперед

UNIX заметно изменилась после своего первого представления. Несмотря на достаточно скромный «выход в свет» она завоевала большую популярность. Во второй половине 80-х годов операционную систему UNIX выбрали многие коммерческие организации, учебные заведения и научно-исследователь-

ские лаборатории. Чаще всего результатом приобретения того или иного варианта системы был осознанный выбор. Позже позиции UNIX были потеснены корпорацией Microsoft, предлагающей операционные системы семейства Windows. Постепенно ОС UNIX стала проигрывать в битве за рынок настольных систем. В этом разделе мы проведем анализ причин ее успеха и популярности, а также факты, не позволившие UNIX покорить мир.

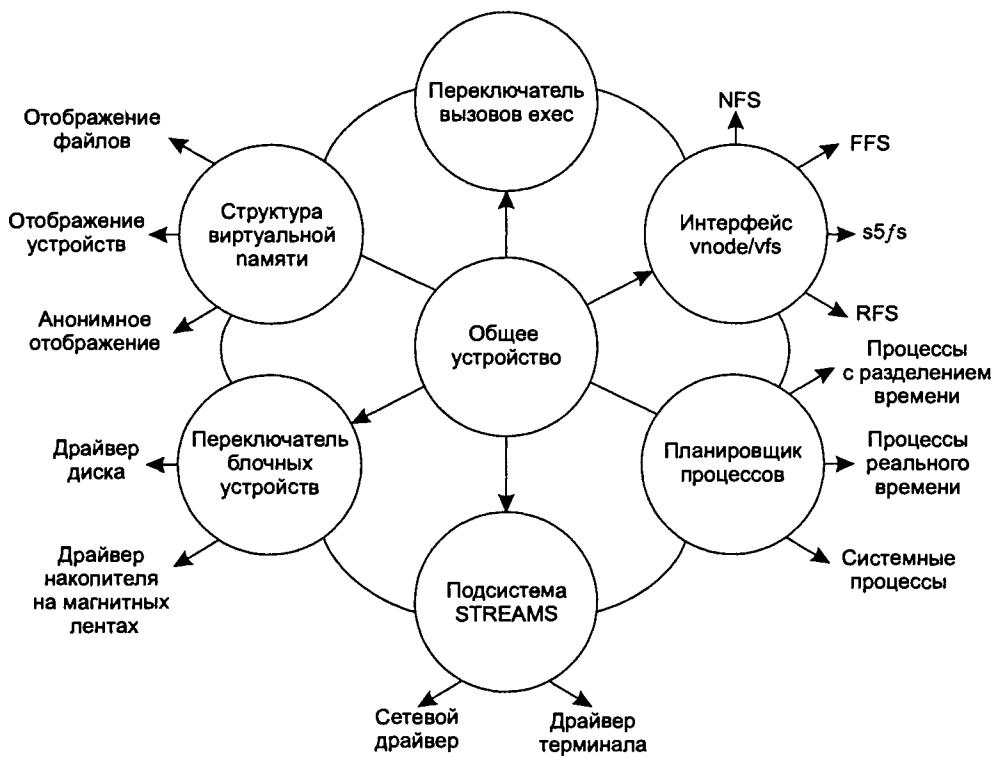


Рис. 1.2. Современное ядро UNIX

1.3.1. Преимущества UNIX

UNIX завоевала такую высокую степень популярности, на которую, возможно, даже не рассчитывали ее разработчики. Одной из главных причин успеха являлся способ распространения системы. Корпорация AT&T, ограниченная законами в своих возможностях, продавала лицензии и исходные коды системы по достаточно низкой цене, поэтому UNIX стала популярной среди многих пользователей по всему миру. Так как в комплект поставки входили и исходные коды, пользователи имели возможность экспериментировать с ними, улучшать их, а также обмениваться друг с другом созданными изменениями. Корпорация AT&T встраивала многие из новшеств в следующие версии системы.

Разработчики из Беркли также поддерживали эту традицию. Развитие системы UNIX оказалось очень открытым процессом. Добавления поступали из учебных заведений, коммерческих организаций и от хакеров-энтузиастов из разных континентов и стран мира. Даже после коммерциализации UNIX многие производители ОС поддержали концепцию открытых систем и сделали свои разработки доступными для других, создавая спецификации открытых систем, такие как NFS.

Оригинальная версия UNIX имела хороший дизайн, который являлся базисом последующего успеха более поздних реализаций и вариантов системы. Одну из сильнейших сторон системы можно охарактеризовать выражением «красота в краткости» [12]. Ядро небольшого размера имело минимальный набор основных служб. Утилиты небольшого размера производили малый объем манипуляций с данными. Механизм *конвейера* совместно с программируемой оболочкой позволял пользователям комбинировать эти утилиты различными способами, создавая мощные, производительные инструменты.

Файловая система UNIX является примером вышеописанного подхода. В отличие от других операционных систем, обладающих сложными методами доступа к файлам, такими как *Indexed Sequential Access Method* (Индексированный последовательный доступ к файлам, ISAM) или *Hierarchical Sequential Access Method* (Иерархический последовательный доступ к файлам, ISAP), система UNIX интерпретирует файлы как простую последовательность байтов. Приложения могут записывать в содержимое файлов любую структуру и использовать различные собственные методы доступа, не ставя об этом в известность операционную систему.

Многие системные приложения используют для представления своих данных символьные строки. К примеру, важнейшие базы данных, такие как `/etc/passwd`, `/etc/fstab` и `/etc/ttys`, являются обычными текстовыми файлами. Возможно, что структурированное хранение информации в двоичном формате представляется более эффективным, однако представление в виде текста позволило пользователям легко просматривать и производить манипуляции с этими файлами без применения специальных инструментов. Текст является общей, универсальной и обладающей высокой степенью переносимости формой данных, которые можно легко обрабатывать множеством различных утилит.

Еще одной особенностью системы UNIX стал простой унифицированный интерфейс с устройствами ввода-вывода. Представляя все устройства как файлы, система позволяет пользователям применять один и тот же набор команд и системных вызовов для доступа и работы с различными устройствами, равно как и для работы с файлами. Разработчики могут создавать программы, производящие ввод-вывод, не заботясь, с чем именно производится обмен, с файлом, терминалом пользователя, принтером или другим устройством. Таким образом, используемый совместно с перенаправлением данных интерфейс ввода-вывода системы UNIX — простой и одновременно мощный.

Причиной успеха и распространения UNIX стала ее высокая степень переносимости. Большая часть ядра системы написана на языке С. Это по-

зволило относительно легко адаптировать UNIX к новым аппаратным платформам. Первая реализация системы появилась на популярной тогда машине PDP-11 и затем была перенесена на VAX-11, имевшую не меньшую популярность. Многие производители «железа» после создания новых компьютеров имеют возможность просто перенести на них уже имеющуюся систему UNIX вместо того, чтобы создавать для своих разработок операционную систему заново.

1.3.2. Недостатки UNIX

Известно, что любая медаль имеет две стороны. Рассказав о преимуществах системы UNIX, необходимо привести и ее недостатки. Один из наиболее объективных обзоров UNIX был создан не кем иным, как Денисом Ритчи. В январе 1987 года на конференции USENIX в разделе «Ретроспектива UNIX» Ритчи сделал доклад, где провел анализ многих недостатков системы [13], которые кратко описаны ниже.

Хотя UNIX изначально была весьма простой системой, это вскоре закончилось. Например, AT&T добавила в стандартную библиотеку ввода-вывода буферизацию данных, что повысило ее эффективность и сделало программы переносимыми на не-UNIX-системы. Библиотека разрасталась и стала более сложной, чем системный вызов, лежащий в ее основе. Системные вызовы `read` и `write` были неделимыми операциями над файлами, в то время как буферизация библиотеки ввода-вывода уничтожила эту цельность.

UNIX сама по себе являлась замечательной операционной системой, однако большинству пользователей нужна была не сама система, а, в первую очередь, возможность выполнять определенное приложение. Пользователей не интересовала элегантность структуры файловой системы или модели вычислений. Они хотели работать с определенными программами (например, текстовыми редакторами, финансовыми пакетами или программами для создания изображений), потратив на это минимум расходов и усилий. Недостатки простого унифицированного (обычно графического) пользовательского интерфейса в первых системах UNIX были основной причиной его неприятия массами. Как сказал Ритчи: «UNIX является простой и понятной системой, но чтобы понять и принять ее простоту, требуется гений (или, как минимум, программист)».

Получилось так, что элегантность и эстетичность, свойственная UNIX, требует от пользователей, желающих эффективно работать в системе, творческого мышления и определенной изобретательности. Однако большинство пользователей предпочитают простые в изучении интегрированные многофункциональные программы, подобные тем, что применяются на персональных компьютерах.

В какой-то степени система UNIX явилась жертвой своего собственного успеха. Простота лицензионных условий и переноса на различные аппарат-

ные платформы стала причиной неконтролируемого роста и лавинообразного распространения различных реализаций ОС. Каждый пользователь имел право вносить свои собственные изменения в систему, в результате группы разработчиков часто создавали несовместимые между собой варианты. Изначально существовали две основные ветви развития UNIX, разрабатываемые компаниями AT&T и BSD. Каждая реализация имела оригинальную файловую систему, архитектуру памяти, сигналы и принципы работы с терминалами. Позже другие поставщики предложили новые варианты UNIX, стараясь привести их к некоторой степени совместимости с реализациями AT&T и BSD. Однако чем дальше, тем все менее предсказуемой становилась ситуация, а разработчикам приложений требовалось все больше усилий, чтобы приспособить свои программы ко всем различным вариантам UNIX.

Стандартизация систем стала лишь частичным решением проблемы, так как встретила определенное сопротивление. Поставщики стремились добавить в свои разработки какие-либо уникальные функции, стараясь создать продукт, имеющий отличия от остальных, и тем самым показать его преимущества среди конкурирующих вариантов.

Ричард Рашид (Richard Rashid), один из разработчиков системы Mach, предложил свою версию причин неудач UNIX. Во вступлении к курсу лекций о системе Mach (Mach Lecture Series [16]) Рашид сказал, что причиной создания ОС Mach стало наблюдение за эволюционированием системы UNIX, которая имела минимальные возможности для построения инструментов пользователя. Большие сложные инструменты создавались путем комбинирования множества простых функций. Однако такой подход не был перенесен на ядро системы.

Традиционное ядро UNIX было недостаточно гибким и расширяемым, оно имело минимальные возможности для дополнительного использования кода. Позже разработчики стали просто добавлять новые коды в ядро системы, делая его основой для новых функциональных средств. Ядро очень быстро стало раздутым, сложным и абсолютно немодульным. Разработчики Mach попытались решить эти проблемы, переписав систему заново с нуля, взяв за основу небольшое количество основных функций. В современных реализациях UNIX вышеописанная система решается различными способами, например добавляются гибкие структуры к подсистемам, как это было описано в разделе 1.2.9.

1.4. Границы повествования книги

Эта книга описывает современные системы UNIX. С целью полноты описания и передачи определенного исторического контекста приводится краткий рассказ о возможностях ранних реализаций ОС. В настоящее время существует множество вариантов системы UNIX, каждая из которых по-своему уникальна. Мы постарались разделить все системы на два типа: базовые и коммерческие

варианты. Базовые ОС включают в себя System V, 4BSD и Mach. Другие варианты происходят от одной из базовых ОС и содержат различные дополнительные возможности и расширения, созданные их разработчиками. Список таких систем включает в себя SunOS и Solaris 2.x корпорации Sun Microsystems, AIX компании IBM, HP-UX от Hewlett-Packard, а также ULTRIX и Digital UNIX от корпорации Digital.

Эта книга не специализируется на каком-то специфическом варианте или реализации системы UNIX. Вместо этого здесь проводится анализ важнейших разработок, их архитектуры и подхода к решению многих проблем. Книга уделяет внимание прежде всего системе SVR4, но вы можете найти здесь достаточно подробный рассказ о 4.3BSD, 4.4BSD и Mach. При рассказе о коммерческих вариантах ОС максимальное внимание уделяется SunOS и Solaris 2.x. Причиной повышенного акцента является не только успех систем компании Sun Microsystems на рынке ОС, но прежде всего то, что именно эта компания была разработчиком многих технических решений, интегрированных в базовые варианты систем, а также то, что компания создала большое количество книг по своим ОС.

В тексте книги часто приводятся общие ссылки на традиционный или коммерческий варианты системы UNIX. Под традиционными вариантами мы имеем в виду SVR3, 4.3BSD и их более ранние реализации. Часто в тексте книги встречаются фразы, касающиеся каких-либо возможностей традиционных систем (например, «в традиционных системах UNIX поддерживался один тип файловой системы»). Несмотря на то что между SVR3 и 4.3BSD имеются различия в каждой подсистеме, между ними есть и много общего. Общие фразы типа приведенной выше обращают внимание как раз на такие свойства систем. При рассмотрении современных ОС UNIX мы подразумеваем системы SVR4, 4.4BSD, Mach, а также реализации, основанные на них. Таким образом, общий комментарий, наподобие «современные системы UNIX поддерживают какую-либо реализацию журнальной файловой системы», означает, что такая возможность имеется во многих современных системах, но необязательно во всех из них.

1.5. Дополнительная литература

1. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M., «Mach: A New Kernel Foundation for UNIX Development», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 93–112¹.

¹ Все публикации прошлых лет в рамках конференции USENIX доступны на сайте <http://www.usenix.org> в формате PDF (кроме последних нескольких месяцев). Там же размещена полная библиография и имеется возможность поиска по автору, дате и ключевым словам. — Прим. ред.

2. Allman, E. «UNIX: The Data Forms», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 9–15.
3. American Telephone and Telegraph, «The System V Interface Definition (SV1D)», Third Edition, 1989.
4. Bostic, K., «4.4BSD Release», Vol. 18, No. 5, Sep.–Oct. 1993, pp. 29–31.
5. Bourne, S., «The UNIX Shell», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1971–1990.
6. Gerber, C. «USL Vs. Berkeley», UNIX Review, Vol. 10, No. 11, Nov. 1992, pp. 33–36.
7. Institute for Electrical and Electronic Engineers, Information Technology, «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C language]», 1003.1–1990, IEEE, Dec. 1990.
8. Joy, W. N., Fabry, R. S., Leffler, S. J., McKusick, M. K., and Karels, M. J., «An Introduction to the C Shell», UNIX User's Supplementary Documents, 4.3 Berkeley.
9. «Software Distribution», Virtual VAX-11 Version, USENIX Association, 1986, pp. 41–46.
10. Organick, E. J., «The Multics System: An Examination of Its Structure», The MIT Press, Cambridge, MA, 1972.
11. Rashid, R. F., «Mach: Technical Innovations, Key Ideas, Status», Mach 2.5 Lecture Series, OSF Research Institute, 1989.
12. Richards, M., and Whitby-Strevens, C., «BCPL: The Language and Its Compiler», Cambridge University Press, Cambridge, UK, 1982.
13. Ritchie, D. M., and Thompson, K., «The UNIX Time-Sharing System», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905–1930, Jul.–Aug. 1978.
14. Ritchie, D. M., «Unix: A Dialectic», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 29–34.
15. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
16. Thompson, K., and Ritchie, D. M., «The UNIX Time-Sharing System», Communications of the ACM, Vol. 17, No. 7, M. 1974, pp. 365–375.
17. «The X/OPEN Portability Guide (XPG)», Issue 4, Prentice-Hall, Englewood Cliffs, NJ, 1993.

Глава 2

Ядро и процессы

2.1. Введение

Основной функцией операционной системы является предоставление среды, под управлением которой могут выполняться пользовательские программы (также называемые *приложениями*). Система определяет базовую структуру для выполнения программ, а также предлагает набор различных служб, например таких, как операции работы с файлами или ввод-вывод, и предоставляет интерфейс взаимодействия с ними. Интерфейс программирования системы UNIX весьма гибок и богат возможностями [4], он может эффективно поддерживать широкий спектр приложений. В этой главе описываются основные компоненты систем UNIX, а также их взаимодействие между собой, предоставляющее пользователям мощное средство программирования.

Существует несколько вариантов системы UNIX. Часть из них — это различные реализации System V компании AT&T (на сегодняшний день последняя версия System V под названием SVR4 является собственностью корпорации Novell), реализаций системы BSD Калифорнийского университета в Беркли, OSF/1 организации Open Software Foundation, а также SunOS и Solaris, поставляемые компанией Sun Microsystems. В этой главе описывается архитектура ядра и процессов традиционных систем UNIX, то есть систем, базирующихся на SVR2 [3], SVR3 [2], 4.3BSD [5], и их более ранних версий. Современные варианты UNIX, такие как SVR4, OSF/1, 4.4BSD и Solaris 2.x, значительно отличаются от базовой модели, их архитектура будет подробно описана в следующих главах книги.

Среда приложений системы UNIX основана на фундаментальной абстракции — *процессе*. В традиционных системах процесс выполняет единую последовательность инструкций в *адресном пространстве*. Адресное пространство процесса представляет собой набор адресов памяти, к которым тот имеет доступ и на которые может ссылаться. Процесс отслеживает последовательность выполняемых инструкций при помощи *контрольной точки*, используя аппаратный регистр, обычно называемый *указателем* (*счетчиком*) команд. Более поздние варианты UNIX поддерживают сразу несколько контрольных точек и, следовательно, несколько параллельно выполняемых последовательностей инструкций в одном процессе, называемых *нитями*.

Система UNIX является многозадачной. Это означает, что в ней одновременно функционируют несколько процессов. Для этих процессов система обеспечивает некоторые свойства *виртуальной машины*. В классической архитектуре виртуальной машины операционная система создает каждому процессу иллюзию того, что он является единственной задачей, выполняемой в данное время. Программист пишет приложение так, как будто только его код будет выполняться системой. В системах UNIX каждый процесс имеет собственные регистры и память, однако для операций ввода-вывода и взаимодействия с устройствами должен полагаться на операционную систему.

Адресное пространство процесса является виртуальным¹, и обычно только часть его соответствует участкам в физической памяти. Ядро хранит содержимое адресного пространства процесса в различных объектах хранения, в том числе в физической памяти, файлах на диске, а также в специально зарезервированных *областях свопинга* (swap areas), находящихся на локальных или удаленных дисках. Подсистема управления памятью ядра переключает *страницы* (блоки фиксированного размера) памяти процесса между этими объектами по мере необходимости.

Каждый процесс также имеет набор регистров, которые соответствуют реальным аппаратным регистрам. В системе может быть одновременно активно множество процессов, но набор аппаратных регистров только один. Ядро хранит регистры процесса, выполняющегося в текущий момент времени в аппаратных регистрах, и сохраняет регистры остальных процессов в специальных структурах данных, отводимых для каждого процесса.

Процессы соперничают между собой, пытаясь захватить различные ресурсы системы, такие как процессор (также называемый *CPU* или центральным процессором), память и периферийные устройства. Операционная система должна функционировать как диспетчер ресурсов, распределяя их оптимально. Процесс, не имеющий возможности получить необходимый ресурс, должен *блокироваться* системой (его выполнение приостанавливается) до тех пор, пока ресурс снова не станет доступен. Процессор является одним из таких ресурсов, поэтому на однопроцессорной системе только один процесс может по-настоящему выполняться в данный момент времени. При этом остальные блокируются, переходя в режим ожидания освобождения процессора или иных ресурсов. Ядро системы дает иллюзию одновременной работы, предоставляя процессам возможность пользоваться процессором в течение определенного короткого промежутка времени, называемого *квантом* и составляющим обычно около 10 миллисекунд. По истечении этого времени ресурсы процессора передаются следующему процессу. Таким образом, каждый про-

¹ Существует несколько вариантов системы UNIX, не использующих виртуальную память. Это самые ранние реализации UNIX (первые ОС, поддерживающие виртуальную память, появились в конце 70-х годов, см. раздел 1.1.4) и некоторые версии, работающие в режиме реального времени. В этой книге описываются только системы, поддерживающие виртуальную память.

цесс получает часть процессорного времени, в течение которого работает. Такая модель функционирования получила название *квантования времени* (*time-slicing*).

С другой точки зрения, компьютер предоставляет пользователю различные устройства, такие как процессор, диски, терминалы и принтеры. Разработчикам приложений нет необходимости вникать в детали функционирования и архитектурные особенности этих компонентов на низком уровне. Операционная система берет на себя полное управление этими устройствами и предоставляет высокоуровневый, абстрактный программный интерфейс, которым приложения могут пользоваться для доступа к аппаратным компонентам. Система скрывает все детали, связанные с оборудованием, сильно упрощая тем самым работу программиста¹. Централизуя все управление устройствами, система также предоставляет дополнительные возможности, такие как синхронизация доступа (в тех случаях, когда два пользователя в один момент времени попытаются воспользоваться одним и тем же устройством) и устранение ошибок. Семантика любого взаимодействия между приложениями и операционной системой определяется *прикладным интерфейсом программирования (API)*.

Мы уже стали относиться к операционной системе как к некой сущности, которая *делает нечто*. Что же в точности представляет собой эта сущность? С одной стороны, операционная система — это программа (часто называемая *ядром*), которая управляет аппаратурой, создает, уничтожает все процессы и управляет ими (рис. 2.1). Если рассматривать шире, операционная система не только включает в себя ядро, но является также основой функционирования остальных программ и утилит (командных интерпретаторов, редакторов, компиляторов, программ типа date, ls, who и т. д.), составляющих вместе пригодную для работы среду. Ядро само по себе мало пригодно для использования. Пользователи, приобретающие систему UNIX, ожидают получить вместе с ней большой набор дополнительных программ. Однако ядро, тем не менее, является весьма специфичной программой по многим причинам. Оно определяет программный интерфейс системы. Ядро — это единственная программа, являющаяся необходимой, без которой ничего не будет работать. Эта книга посвящена изучению ядра системы, и когда будет упоминаться *операционная система* или UNIX, это будет означать ядро, если не оговорено иное.

Немного изменим наш предыдущий вопрос: так что же такое ядро? Есть ли это процесс или нечто отличающееся от всех других процессов? Ядро — это специальная программа, работающая непосредственно с аппаратурой.

¹ Однако в некоторых местах разработчики UNIX слегка перестарались. К примеру, интерпретация устройств работы с магнитной лентой как потоков символов весьма усложнила приложением правильную обработку ошибок и исключений. Интерфейс работы с магнитной лентой не совсем удачно вписывается в интерфейс работы с устройствами в системе UNIX [Allm 87].

Ядро находится на диске в файле, обычно имеющем название `/vmlinix` или `/uplinx` (в зависимости от производителя ОС). Когда система стартует, с диска загружается ядро при помощи специальной процедуры *начальной загрузки* (*bootstrapping*). Ядро инициализирует систему и устанавливает среду для выполнения процессов. Затем создаются несколько начальных процессов, которые в дальнейшем порождают остальные процессы. После загрузки ядро находится в памяти постоянно до тех пор, пока работа системы не будет завершена. Ядро управляет процессами и предоставляет им различные службы.

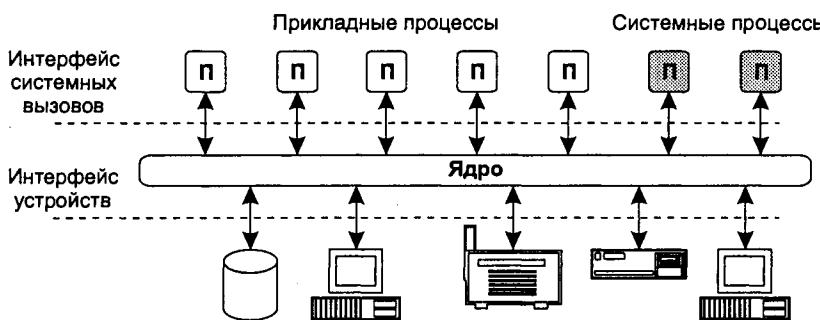


Рис. 2.1. Ядро взаимодействует с процессами и устройствами

Операционная система UNIX обеспечивает свою функциональность четырьмя различными способами:

- ◆ Прикладные процессы запрашивают от ядра необходимые службы при помощи *интерфейса системных вызовов* (см. рис. 2.1), являющегося центральным компонентом API системы UNIX. Ядро выполняет эти запросы от имени вызывающих процессов.
- ◆ Некоторые некорректные действия процесса, такие как попытки деления на ноль или переполнение стека процесса, являются причиной *аппаратных исключений*. Возникающие ошибки требуют вмешательства ядра, после чего происходит их обработка от имени процесса.
- ◆ Ядро обрабатывает аппаратные *прерывания* от периферийных устройств. Устройства используют механизм прерываний для оповещения ядра об окончании процесса ввода-вывода или изменении состояния. Ядро трактует прерывания как глобальные события, не относящиеся к какому-то одному определенному процессу.
- ◆ Набор специальных системных процессов, таких как *swapper* или *pagedaemon*, занимается выполнением обширных системных задач, таких как управление рядом активных процессов или поддержка пула свободной памяти.

В следующих разделах этой главы будут описаны вышеперечисленные механизмы и будет определено понятие контекста выполнения процесса.

2.2. Режим, пространство и контекст

Для возможности функционирования системы UNIX аппаратная часть компьютера должна поддерживать по крайней мере два режима выполнения: более привилегированный *режим ядра* и менее привилегированный *режим задачи*. Как и следовало ожидать, прикладные программы работают в режиме задачи, а функции ядра выполняются в режиме ядра. Ядро защищает часть адресного пространства от доступа в режиме задачи. Более того, наиболее привилегированные машинные инструкции могут выполняться только в режиме ядра.

Во многих аппаратных архитектурах поддерживается более двух режимов выполнения. Например, архитектура Intel 80x86¹ поддерживает четыре *уровня выполнения*², самым привилегированным из которых является нулевой. UNIX использует только два из них. Главной причиной появления различных режимов выполнения является безопасность. Если пользовательские процессы выполняются в менее привилегированном режиме, то они не могут случайно или специально повредить другой процесс или ядро системы. Последствия, вызванные ошибками в программе, носят локальный характер и обычно не влияют на выполнение других действий или процессов.

Большинство реализаций UNIX используют *виртуальную память*. В системе виртуальной памяти адреса, выделенные программе, не ссылаются непосредственно на физическое размещение в памяти. Каждому процессу предоставляется собственное *виртуальное адресное пространство*, а ссылки на виртуальные адреса памяти транслируются в их фактическое нахождение в физической памяти при помощи набора карт трансляции адресов. Многие системы реализуют такие карты как *таблицы страниц*, с одной записью для каждой страницы адресного пространства процесса (страница — это выделенный и защищенный блок памяти фиксированного размера). Аппаратно реализованный блок управления памятью (memory management unit, MMU) обычно обладает определенным набором регистров для определения карт трансляции адресов процесса, выполняющегося в данный момент времени (также называемого *текущим*). Когда текущий процесс уступает процессорное время другому процессу (*переключение контекста*), ядро размещает в этих регистрах указатели на карты трансляции адресов нового процесса. Регистры MMU являются привилегированными и могут быть доступны только в режиме ядра. Это дает гарантию того, что процесс будет ссылаться на адреса памяти только своего адресного пространства и не имеет доступа или возможности изменения адресного пространства другого процесса.

Определенная часть виртуального адресного пространства каждого процесса отображается на код и структуры данных ядра. Эта часть называется

¹ За исключением 8086 и 80186. — Прим. ред.

² Еще их называют уровнями привилегий или защищенности. — Прим. ред.

системным пространством или *пространством ядра* и может быть доступна только в режиме ядра. В системе может одновременно выполняться только одна копия ядра, следовательно, все процессы отображаются в единое адресное пространство ядра. В ядре содержатся глобальные структуры и информация, дающая возможность ему иметь доступ к адресному пространству любого процесса. Ядро может обращаться к адресному пространству текущего процесса напрямую, так как регистры MMU хранят всю необходимую для этого информацию. Иногда ядру требуется обратиться к адресному пространству, не являющемуся в данный момент текущим. В этом случае обращение происходит не непосредственно, а при помощи специального временного отображения.

В то время как ядро совместно используется всеми процессами, системное пространство защищается от доступа в режиме задачи. Процессы не могут напрямую обращаться к ядру и должны использовать для этого интерфейс системных вызовов. После того как процесс производит *системный вызов*, запускается специальная последовательность команд (называемая *переключателем режимов*), переводящая систему в режим ядра, а управление передается ядру, которое и обрабатывает операцию от имени процесса. После завершения обработки системного вызова ядро вызывает другую последовательность команд, возвращающую систему обратно в режим задачи (производится еще одно переключение режима), и снова передает управление текущему процессу. Интерфейс системных вызовов описан подробнее в разделе 2.4.1.

Существуют два важных для любого процесса объекта, которые управляются ядром. Они обычно реализуются как часть адресного пространства процесса. Это *область и* (u-агея, или user-agaea) и *стек ядра* (kernel stack). Область и является структурой данных, содержащей нужную ядру информацию о процессе, такую как таблицу файлов, открытых процессом, данные для идентификации, а также сохраненные значения регистров процесса, пока процесс не является текущим. Процесс не может произвольно изменять эту информацию — и, следовательно, область и является защищенной от доступа в режиме задачи (некоторые реализации ОС позволяют процессу считывать эту информацию, но не изменять ее).

Ядро системы UNIX является реентерабельным, то есть к ядру могут обращаться одновременно несколько процессов. Фактически они могут выполнять одни и те же последовательности инструкций параллельно. (Разумеется, в один момент времени выполняется только один процесс, остальные при этом заблокированы или находятся в режиме ожидания¹.) Таким образом, каждому процессу необходим собственный стек ядра для хранения данных,

¹ Реентерабельностью называется возможность единовременного обращения различных процессов к одному и тому же машинному коду, загруженному в память компьютера в единственном экземпляре. — Прим. ред.

используемых последовательностью функций ядра, инициированной вызовом из процесса. Многие реализации UNIX располагают стек ядра в адресном пространстве каждого процесса, но при этом запрещают к нему доступ в режиме задачи. Концептуально область и и стек ядра хоть и создаются для каждого процесса отдельно и хранятся в его адресном пространстве, они, тем не менее, являются *собственностью* ядра.

Еще одним важным понятием является *контекст выполнения*. Функции ядра могут выполняться только в *контексте процесса* или в *системном контексте*. В контексте процесса ядро функционирует от имени текущего процесса (например, пока обрабатывает системный вызов) и может иметь доступ и изменять адресное пространство, область и и стек ядра этого процесса. Более того, ядро может заблокировать текущий процесс, если тому необходимо ожидать освобождения ресурса или реакции устройства.

Ядро также должно выполнять глобальные задачи, такие как обслуживание прерываний устройств и пересчет приоритетов процессов. Такие задачи не зависят от какого-либо конкретного процесса и, следовательно, обрабатываются в системном контексте (также называемом *контекстом прерываний*). Если ядро функционирует в системном контексте, то оно не должно иметь доступа к адресному пространству, области и и стеку ядра текущего процесса. Ядро также не должно в этом режиме блокировать процессы, так как это приведет к блокировке «невинного» процесса.



Рис. 2.2. Режим и контекст выполнения

Мы отметили основные различия между режимами задачи и ядра, пространством процесса и системы, контекстом процесса и системы. На рис. 2.2 проиллюстрированы все эти определения. Код приложения выполняется в режиме задачи и контексте процесса и может иметь доступ только к про-

пространству процесса. Системные вызовы и исключения обрабатываются в режиме ядра, но в контексте процесса, однако эти задачи могут иметь доступ к пространству процесса и системы. Прерывания обрабатываются в режиме ядра и системном контексте и могут иметь доступ только к пространству системы.

2.3. Определение процесса

Так что же такое процесс в системе UNIX? Наиболее часто встречающийся ответ на этот вопрос таков: «Процесс – это экземпляр выполняемой программы». Однако такое определение является поверхностным, и для его более детального описания необходимо рассмотреть различные свойства процесса. Процесс – это нечто выполняющее программу и создающее среду для ее функционирования. В это понятие также входит адресное пространство и точка управления. Процесс – это основная единица расписания, так как только один процесс может в один момент времени занимать процессор. Кроме этого процесс старается перехватить ресурсы системы, такие как различные устройства или память. Он также запрашивает системные службы, которые выполняются для него и от его имени ядром.

Каждый процесс имеет определенное время жизни. Большинство процессов создаются при помощи системного вызова `fork` или `vfork` и работают до тех пор, пока не будут завершены вызовом `exit`. Во время функционирования процесс может запускать одну или несколько программ. Для запуска программ процесс использует системный вызов `exec`.

В системе UNIX процессы иерархически строго упорядочены. Каждый процесс имеет одного *родителя* (*parent*, или *родительский процесс*) и может иметь также одного или нескольких *потомков* (*child*, или *процесс-потомок*). Иерархия процессов может быть представлена как перевернутое дерево, в вершине (основании) которого находится процесс `init`. Процесс `init` (названный так в силу того, что он запускает программу `/etc/init`) является первым прикладным процессом, создаваемым во время загрузки системы. Этот процесс порождает все остальные прикладные процессы. Во время запуска системы создаются несколько различных процессов, например `swapper` или `pagedaemon` (называемого также `pageout daemon`), которые не являются потомками `init`. Если какой-либо процесс завершен и после него остаются функционирующие процессы-потомки, то они становятся «осиротевшими» (*orphan*) и наследуются процессом `init`.

2.3.1. Состояние процесса

В системе UNIX процессы всегда имели строго определенное *состояние*. Переход от одного состояния к другому осуществляется вследствие различных событий. На рис. 2.3 показаны важнейшие состояния процесса в системе UNIX, а также события, являющиеся причиной *перехода в иное состояние*.

Системный вызов `fork` создает процесс, который начинает свой жизненный цикл в *начальном* состоянии, также называемом *переходным* (*idle*). После того как создание процесса завершится, вызов `fork` переводит его в состояние *готовности к работе* (*ready to run*), в котором процесс ожидает своей очереди на обслуживание. В какой-то момент времени ядро выбирает этот процесс для выполнения и инициирует переключение контекста. Это производится вызовом процедуры ядра (обычно называемой `switch`), которая загружает аппаратный контекст процесса (см. раздел 2.3.2) в системные регистры и передает ему управление. Начиная с этого момента новый процесс ведет себя так же, как и любой другой. Последующие изменения его состояния описаны ниже.

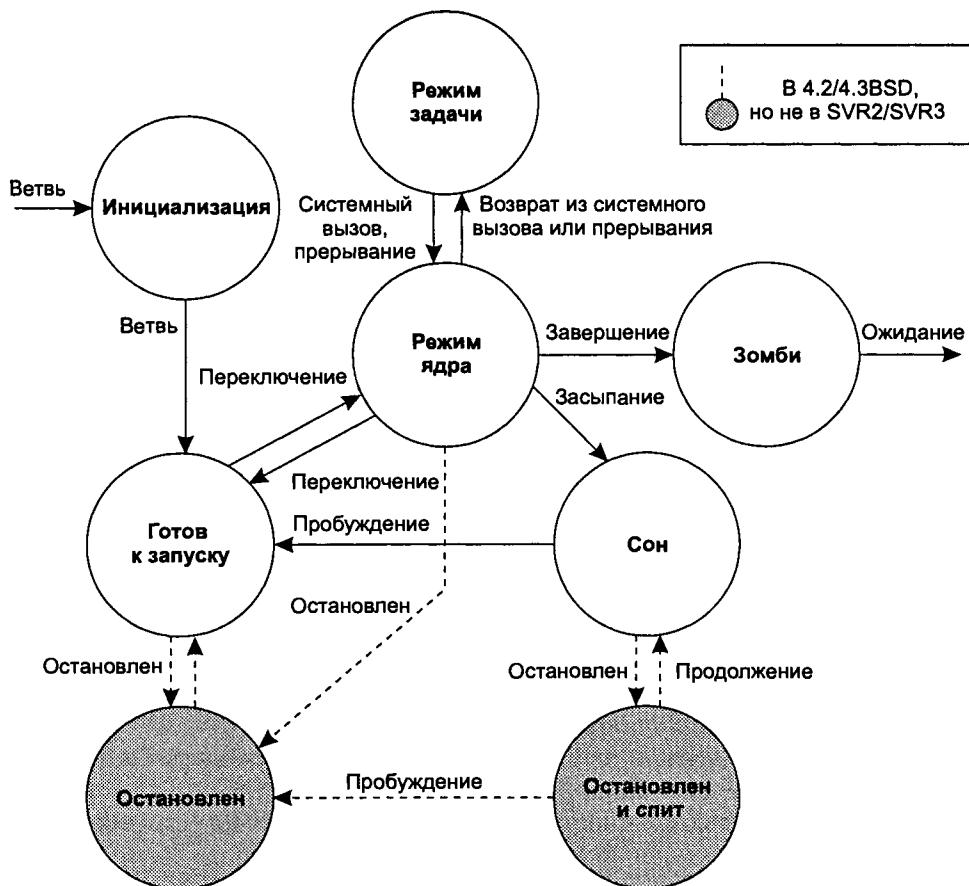


Рис. 2.3. Состояния процесса и переходы процесса в иные состояния

Процесс, функционирующий в режиме задачи, переходит в режим ядра в результате системного вызова или прерывания и возвращается обратно в ре-

жим задачи после завершения процедуры обработки¹. Пока обрабатывается системный вызов или прерывание, процессу иногда необходимо ждать некоторого события или освобождения ресурса, недоступного в данный момент времени. Такое ожидание реализуется вызовом `sleep()`, который помещает процесс в очередь «спящих» процессов и переводит его в *спящее*, или *ожидающее* (*asleep*), состояние. После того как необходимое событие произойдет или ресурс станет доступен, ядро «разбудит» процесс, который с этого момента снова станет *готов к работе* и перейдет в очередь на выполнение.

Когда процесс планируется на выполнение, он изначально выполняется в режиме ядра (*состояние выполнения ядром*), где происходит переключение его контекста. Следующее изменение состояния зависит от того, чем занимался процесс до того, как достиг своей очереди на выполнение. Если процесс был только что создан или выполнял код программы (и был вытеснен из очереди на выполнение процессом, имеющим больший приоритет), то он сразу же переводится в режим задачи. Если процесс был блокирован в ожидании ресурса во время выполнения системного вызова, то он продолжит выполнение вызова в режиме ядра.

Завершение работы процесса происходит в результате системного вызова `exit` или по *сигналу* (сигналы — это средства оповещения, используемые ядром; см. главу 4). В любом случае ядро освобождает все ресурсы завершающегося процесса (сохраняя информацию о *статусе выхода и использовании ресурсов*) и оставляет процесс в состоянии зомби (*zombie*). Процесс остается в этом состоянии до тех пор, пока его процесс-родитель не вызовет процедуру `wait` (или один из ее вариантов), которая уничтожит процесс и передаст статус выхода родительскому процессу (см. раздел 2.8.6).

В системе 4BSD определены дополнительные состояния, не поддерживающиеся в системах SVR2 или SVR3. Процесс может быть *остановлен* (*stopped*) или переведен в *состояние ожидания* при помощи сигнала *остановки* (`SIGSTOP`, `SIGTSTP`, `SIGTTIN` или `SIGTTOU`). В отличие от других сигналов, которыерабатываются только во время выполнения процесса, сигнал остановки изменяет состояние процесса немедленно. Если процесс находится в состоянии работы или готовности к работе, то его состояние изменяется на остановленное. Если процесс находился в спящем режиме, когда поступил сигнал, его состояние изменится на спящее и остановленное. Остановленный процесс может продолжить работу при помощи сигнала *продолжения* (`SIGCONT`), который возвратит его в состояние готовности к работе. Если процесс был остановлен, находясь в спящем режиме, сигнал `SIGCONT` возвратит его обратно в этот режим. Позже такие возможности были добавлены и в SVR4 (см. раздел 4.5)².

¹ Прерывания могут произойти и тогда, когда система находится в режиме ядра. В этом случае система будет оставаться в этом режиме после завершения обработки прерывания.

² Система SVR3 поддерживает состояние остановки для процесса только с целью проведения процедуры *трассировки процесса* (см. раздел 6.2.4). Если трассируемый процесс получит любой сигнал, он перейдет в состояние остановки, а его родитель будет «разбужен» ядром.

2.3.2. Контекст процесса

Каждый процесс имеет четко определенный контекст, включающий всю информацию, необходимую для его описания. Ниже перечислены компоненты контекста:

- ◆ **Адресное пространство задачи.** Обычно делится на несколько составляющих: текст программы (выполняемый код), данные, стек задачи, совместно используемые области памяти и т. д.
- ◆ **Управляющая информация.** Ядро использует для поддержки управляющей информации о процессе две основные структуры данных: область и структуру `proc`. Каждый процесс также обладает собственным стеком ядра и картами трансляции адресов.
- ◆ **Полномочия.** Включают в себя идентификаторы пользователя и группы, ассоциируемые с данным процессом (которые будут описаны в разделе 2.3.3).
- ◆ **Переменные окружения.** Это набор строк в форме:

переменная=значение

Переменные окружения наследуются от родительского процесса. Во многих вариантах UNIX такие строки хранятся в вершине стека. Стандартные пользовательские библиотеки имеют функции для добавления, удаления или изменения переменных, а также для возврата ее значения. При запуске новой программы вызывающий процесс может сообщить функции `exec` о том, что переменные окружения должны оставаться «родительскими» или вместо этого предложить новый набор переменных.

- ◆ **Аппаратный контекст.** Включает содержимое регистров общего назначения, а также набора специальных системных регистров. Системные регистры содержат:
 - ◆ **Программный счетчик** (program counter, PC). Хранит адрес следующей выполняемой инструкции¹.
 - ◆ **Указатель стека** (stack pointer, SP). Содержит адрес верхнего элемента стека².
 - ◆ **Слово состояния процессора** (processor status word, PSW). Содержит несколько битов с информацией о состоянии системы, в том числе о текущем и предыдущем режимах выполнения, текущем и предыдущем уровнях приоритетов прерываний, а также биты переполнения и переноса.

¹ Обозначение характерно для машин PDP и процессоров ALPHA, в архитектуре Intel эту роль играет регистр (E)IP — указатель команд, instruction pointer. Обозначения SP и PSW в ней сохранились. В дальнейшем мы будем чаще пользоваться вторым термином. — Прим. ред.

² Или нижнего, для машин, в которых стек растет вниз. Также на некоторых системах указатель стека может содержать адрес, по которому будет занесен следующий помещаемый в стек элемент.

- **Регистры управления памятью**, в которых отображаются таблицы трансляции адресов процесса.
- **Регистры сопроцессора** (Floating point unit, FPU).

Машинные регистры содержат аппаратный контекст текущего выполняемого процесса. Когда происходит переключение контекста, эти регистры сохраняются в специальном разделе области и текущего процесса, который называется *блоком управления процессом* (process control block, PCB). Ядро выбирает следующий процесс для выполнения и загружает его аппаратный контекст из блока PCB.

2.3.3. Полномочия пользователя

Каждый пользователь системы имеет свой уникальный номер, называемый *идентификатором пользователя* (user ID, или UID). Системный администратор обычно также создает несколько различных групп пользователей, каждая из которых обладает уникальным *идентификатором группы* (user group ID, или GID). Эти идентификаторы определяют принадлежность файлов, права доступа, а также возможность посылки сигналов другим процессам. Все перечисленные атрибуты получили единое название *полномочий*.

Система различает *привилегированного пользователя*, называемого *суперпользователем* (superuser), который обычно входит в систему под именем root. Этот пользователь имеет UID, равный 0, и GID, равный 1. Он обладает многими полномочиями, недоступными обычным пользователям. Он может иметь доступ к чужим файлам независимо от установок их защиты, а также выполнять ряд привилегированных системных вызовов (например, mknod, используемый для создания специальных файлов устройств). Многие современные системы UNIX, такие как SVR4.1/ES, поддерживают *расширенные механизмы защиты* [8]. В этих системах поддержка суперпользователя заменена разделением привилегий при проведении различных операций.

Каждый процесс обладает двумя идентификаторами пользователя — *реальным* (real) и *действительным* (effective)¹. После того как пользователь входит в систему, программа входа в систему выставляет обеим парам UID и GID значения, определенные в базе паролей (то есть в файле /etc/passwd или в некоем распределенном механизме, например службе NIS корпорации Sun Microsystems). Когда процесс создается при помощи fork, потомок наследует полномочия от своего прародителя.

Эффективный UID и эффективный GID влияют на создание файлов и доступ к ним. Во время создания файла ядро устанавливает ему атрибуты владельца файла как эффективные UID и GID процесса, из которого был сделан вызов на создание файла. Во время доступа процесса к файлу ядро использует

¹ Есть еще третий — сохраненный (saved). — Прим. ред.

эффективные идентификаторы процесса для определения, имеет ли он право обращаться к этому файлу (подробнее см. раздел 8.2). Реальный UID и реальный GID идентифицируют владельца процесса и влияют на право отправки сигналов. Процесс, не имеющий привилегий суперпользователя, может передавать сигналы другому процессу только в том случае, если реальный или эффективный UID отправителя совпадает с реальным UID получателя.

Существуют три различных системных вызова, которые могут переопределять полномочия. Если процесс вызывает `exec` для выполнения программы, установленной в *режиме uid* (см. раздел 8.2.2), то ядро изменяет эффективный UID процесса на UID, соответствующий владельцу файла программы. Точно так же, если программа установлена в *режиме sgid*, ядро изменяет GID вызываемого процесса.

Система UNIX предлагает описанную возможность с целью предоставления специальных прав пользователям для определенных целей. Классическим примером такого подхода является программа `passwd`, позволяющая пользователям изменять свои пароли. Этой программе необходимо записать результат в базу данных паролей, которая обычно недоступна пользователям для прямых изменений (с целью защиты от модификаций записей других пользователей). Таким образом, владельцем `passwd` является суперпользователь, но программа имеет установленный бит SUID. Это дает возможность обычному пользователю получить привилегии суперпользователя на время и в рамках выполнения программы `passwd`.

Пользователь также может настраивать свои полномочия при помощи системных вызовов `setuid` и `setgid`. Суперпользователю позволено при помощи этих вызовов изменять как реальные, так и эффективные идентификаторы UID и GID. Обычные пользователи могут обращаться к этим вызовам только для изменения своих эффективных идентификаторов UID или GID на реальные.

Существуют некоторые различия в интерпретации полномочий в ОС System V и BSD UNIX. В системе SVR3 поддерживаются *хранимые (saved)* UID и GID, которые имеют значения эффективных UID и GID перед вызовом `exec`. Системные вызовы `setuid` и `setgid` могут восстановить эффективные идентификаторы из хранимых значений. Хотя в системе 4.3BSD не поддерживается описанная возможность, пользователь ОС может войти в состав *дополнительных групп* (используя вызов `setgroups`). В то время как файлы, созданные пользователем, принадлежат его основной группе, он может иметь доступ к файлам, принадлежащим к его как основной, так и дополнительной группе (в зависимости от установок прав доступа к файлам для членов группы владельца).

В ОС SVR4 встроены все перечисленные возможности. Система поддерживает дополнительные группы, а также поддерживает хранимые идентификаторы UID и GID через вызов `exec`.

2.3.4. Область и и структура proc

Управляющая информация о процессе поддерживается с помощью двух структур данных, имеющихся у каждого процесса: области и и структуры proc. В различных реализациях UNIX ядро имеет массив фиксированного размера, состоящий из структур proc и называемый *таблицей процессов*. Размер этого массива зависит от максимального количества процессов, которые одновременно могут быть запущены в системе. Современные варианты UNIX, такие как SVR4, поддерживают динамическое размещение структур proc, но массив указателей на них также имеет фиксированный размер. Так как структура proc находится в системном пространстве, она всегда видна ядру в любой момент времени, даже когда процесс не выполняется.

Область и является частью пространства процесса. Это означает, что она отображаема и видима только в тот период времени, когда процесс выполняется. Во многих реализациях UNIX область и всегда отображается в один и тот же виртуальный адрес для каждого процесса, на который ядро ссылается через переменную и. Одной из задач переключателя контекста является сброс этого отображения, с тем чтобы ядро через переменную и «добралось» до физического расположения новой области и.

Иногда ядру системы необходимо получить доступ к области и процесса, не являющегося текущим. Такое возможно, но действие должно производиться не напрямую, а при помощи специального набора отображений. Различия в особенностях доступа обусловлены особенностями информации, хранящейся в структуре proc и области и. Область и содержит данные, необходимые только в период выполнения процесса. Структура proc включает в себя такую информацию, которая может потребоваться даже в том случае, если процесс не выполняется.

Основные поля области и перечислены ниже:

- ◆ блок управления процессом используется для хранения аппаратного контекста в то время, когда процесс не выполняется;
- ◆ указатель на структуру proc для этого процесса;
- ◆ реальные и эффективные UID и GID¹;
- ◆ входные аргументы и возвращаемые значения (или коды ошибок) от текущего системного вызова;
- ◆ обработчики сигналов и информация, связанная с ними (см. главу 4);
- ◆ информация из заголовка программы, в том числе размеры текста, данных и стека, а также иная информация по управлению памятью;

¹ В современных системах UNIX, таких как SVR4, пользовательские полномочия хранятся в структуре данных, располагаемой динамически, указатель на которую находится в структуре proc. Более подробное описание см. в разделе 8.10.7.

- ◆ таблица дескрипторов открытых файлов (см. раздел 8.2.3). Современные системы UNIX, такие как SVR4, расширяют эту таблицу динамически по мере необходимости;
- ◆ указатели на *vnode*. Объекты *vnode* представляют собой объекты файловой системы и будут подробнее описаны в разделе 8.7;
- ◆ статистика использования процессора, информация о профиле процесса, дисковых квотах и ресурсах;
- ◆ во многих реализациях UNIX стек ядра процесса является частью области и этого процесса.

Основные поля структуры *proc* охватывают:

- ◆ идентификацию: каждый процесс обладает уникальным *идентификатором процесса* (process ID, или *PID*) и относится к определенной *группе процессов*. В современных версиях системы каждому процессу также присваивается *идентификатор сеанса* (session ID);
- ◆ расположение карты адресов ядра для области и данного процесса;
- ◆ текущее состояние процесса;
- ◆ предыдущий и следующий указатели, связывающие процесс с очередью планировщика (или очередью приостановленных процессов, если данный процесс был заблокирован);
- ◆ канал «сна» для заблокированных процессов (см. раздел 7.2.3);
- ◆ приоритеты планирования задач и связанную информацию (см. главу 5);
- ◆ информацию об обработке сигналов: маски игнорируемых, блокируемых, передаваемых и обрабатываемых сигналов (см. главу 4);
- ◆ информацию по управлению памятью;
- ◆ указатели, связывающие эту структуру со списками активных, свободных или завершенных процессов (зомби);
- ◆ различные флаги;
- ◆ указатели на расположение структуры в *очереди хэша*, основанной на *PID*;
- ◆ информация об иерархии, описывающая взаимосвязь данного процесса с другими.

На рис. 2.4 продемонстрированы взаимосвязи процессов в системе 4.3BSD UNIX. На схеме представлены поля, описывающие иерархию процессов. Это – *p_pid* (идентификатор процесса), *p_ppid* (идентификатор родительского процесса), *p_pptr* (указатель на структуру *proc* родителя), *p_cptr* (указатель на старшего потомка), *p_ysptr* (указатель на следующий младший процесс того же уровня), *p_osptr* (указатель на следующий старший процесс того же уровня).

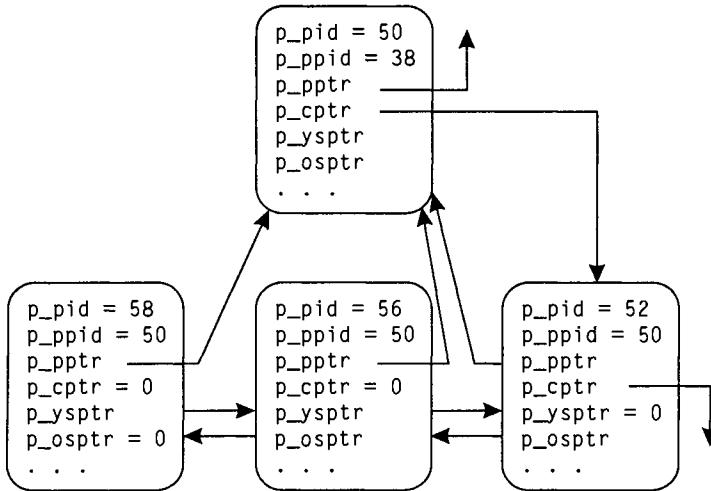


Рис. 2.4. Пример иерархии процессов в системе 4.3BSD UNIX

Во многих современных системах UNIX абстракция процесса была дополнена поддержкой нескольких нитей. Подробнее об этом будет сказано в главе 3.

2.4. Выполнение в режиме ядра

Существуют три различных типа событий, которые могут перевести систему в режим ядра. Это — прерывания устройств (interrupts), исключительные ситуации или просто исключения (exceptions), а также ловушки (traps) или программные прерывания (software interrupts). Каждый раз, когда ядру возвращается управление, оно обращается к таблице *диспетчеризации* (dispatch table), содержащей адреса низкоуровневых процедур обработки событий. Перед вызовом соответствующей процедуры ядро частично сохраняет состояние прерванного процесса (например, указатель команд и слово состояния процессора) в стеке ядра для этого процесса. После завершения работы процедуры ядро восстанавливает состояние процесса и изменяет режим его выполнения на прежнее значение. Прерывание может произойти и в тот момент, когда система уже находится в режиме ядра, в таком случае она останется в нем и после обработки прерывания.

Важно понимать разницу между прерываниями и исключительными состояниями. Прерывания — это асинхронные события, происходящие в периферийных устройствах, таких как диски, терминалы или аппаратный таймер. Так как прерывания не зависят от текущего процесса, они должны обрабатываться в системном контексте, при этом доступ в адресное пространство или область и процесса им не требуется. По этой же причине прерывания не должны производить блокировку, так как они могут заблокировать произвольный процесс. Исключительные состояния возникают по ходу работы

процесса по причинам, зависящим от него самого, например при попытке деления на ноль или обращения по несуществующему адресу. Обработчик исключительных состояний работает в контексте процесса и может обращаться к адресному пространству или области и процесса, а также блокировать процесс, если это необходимо. Программные либо системные прерывания (traps или ловушки) происходят во время выполнения процессом особых инструкций, например в процессе перехода в системные вызовы, и обрабатываются синхронно в контексте процесса.

2.4.1. Интерфейс системных вызовов

Программный интерфейс ОС определяется набором системных вызовов, предоставляемых ядром пользовательским процессам. Стандартная библиотека C, подключаемая по умолчанию ко всем программам пользователя, содержит *процедуру встраивания* для каждого системного вызова. Когда программа делает системный вызов, вызывается и соответствующая ему процедура встраивания. Она передает номер системного вызова (идентифицирующего каждый вызов ядра) в пользовательский стек и затем вызывает специальную инструкцию *системного прерывания*. Имя инструкции зависит от конкретной машины (например, это — `syscall` для MIPS R3000, `chmk` для VAX-11 или `trap` для Motorola 680x0). Функция этой инструкции заключается в изменении режима выполнения на режим ядра и передачи управления обработчику системного вызова, определенному по таблице диспетчеризации. Этот обработчик, обычно имеющий название `syscall()`, является точкой старта обработки любого системного вызова ядром.

Системные вызовы выполняются в режиме ядра, но в контексте процесса. Следовательно, они имеют возможность доступа к адресному пространству и области и вызывающего процесса. С другой стороны, они могут обращаться и к стеку ядра этого процесса. Обработчик `syscall()` копирует входные аргументы системного вызова из пользовательского стека в область `u`, а также сохраняет аппаратный контекст процесса в стеке ядра. Затем он использует номер системного вызова для доступа к его вектору (обычно называемому `sysent[]`), чтобы определить, какую именно системную функцию необходимо вызвать для обработки поступившего системного вызова. После завершения работы необходимой функции обработчик `syscall()` устанавливает возращенные ею значения или код ошибки в соответствующие регистры, восстанавливает аппаратный контекст процесса, возвращает систему в режим задачи, передавая управление обратно библиотечной функции.

2.4.2. Обработка прерываний

Основная функция прерываний в компьютере заключается в том, чтобы позволить взаимодействовать периферийным устройствам с процессором, информируя его о завершении работы задачи, ошибочных состояниях и других событиях,

требующих немедленного внимания. Прерывания происходят независимо от текущих действий системы или какого-либо процесса. Это означает, что система не знает заранее, в каком месте выполнения потока инструкций может произойти прерывание. Функция, запускаемая для обслуживания прерывания, называется *обработчиком прерывания*, или *процедурой обслуживания прерывания*. Обработчик работает в режиме ядра и системном контексте. Так как прерванный процесс обычно не имеет никакого отношения к произошедшему в системе прерыванию, обработчик не должен обращаться к контексту процесса. По той же причине он также не обладает правом блокировки.

Однако прерывание оказывает некоторое влияние на выполнение текущего процесса. Время, потраченное на обработку прерывания, является частью кванта времени, отведенного процессу, даже если производимые действия не имеют ни малейшего к нему отношения. Так, обработчик прерываний системного таймера использует тики (промежутки времени между двумя прерываниями таймера) текущего процесса и потому нуждается в доступе к его структуре `proc`. Важно отметить, что контекст процесса защищен от доступа обработчиками прерываний не полностью. Неверно написанный обработчик может причинить вред любой части адресного пространства процесса.

Ядро также поддерживает программные или системные прерывания, которые могут генерироваться при выполнении специальных инструкций. Такие прерывания могут использоваться, к примеру, в переключателе контекстов или в планировании задач, функционирующих в режиме разделения времени и имеющих низкий приоритет. Несмотря на то что описанные прерывания происходят синхронно с нормальной работой системы, они обрабатываются точно так же, как и обычные.

Вследствие того что причиной возникновения прерываний может стать множество различных событий, представима и такая ситуация, когда прерывание происходит во время того, как обрабатывается другое. Перед разработчиками системы UNIX встало необходимость поддержки различных уровней приоритетов, для того чтобы прерывания более высокого уровня обслуживались раньше, чем прерывания более низких уровней. К примеру, прерывание аппаратного таймера должно обслуживаться раньше прерывания сети, так как последнее может потребовать больших объемов вычислений в течение нескольких тиков системного таймера.

В системах UNIX каждому типу прерывания принято назначать *уровень приоритета прерывания* (interrupt priority level, `ipl`). В первых реализациях системы уровень `ipl` находился в пределах от 0 до 7. В ОС BSD значение `ipl` возросло до 0–31. *Регистр состояния процессора* обычно содержит битовые поля, в которых хранится текущий (а иногда и предыдущий) `ipl`¹. Номера приоритетов прерываний не одинаковы, так как зависят не только от конкретной реализации системы UNIX, но и от различия в архитектуре оборудо-

¹ Некоторые процессоры, например Intel 80x86, не поддерживают приоритеты прерываний на аппаратном уровне. В таких системах необходимо реализовывать уровни `ipl` программно. Эта проблема будет затронута в дальнейшем в упражнениях.

дования. В некоторых системах `ipl 0` означает низший уровень приоритета, тогда как в иных это значение может оказаться наивысшим. Для облегчения создания процедур обработки прерываний и драйверов устройств в системах UNIX представлен набор триггеров для блокировки и разблокирования прерываний. Однако в различных реализациях системы для одних и тех же целей используются различные триггеры. В табл. 2.1 показаны некоторые триггеры, применяемые в 4.3BSD и SVR4.

Таблица 2.1. Установка уровней приоритетов прерываний в системах 4.3BSD и SVR4

4.3BSD	SVR4	Назначение
<code>spl0</code>	<code>spl0</code> или <code>splbase</code>	Разрешить все прерывания
<code>splsoftclock</code>	<code>spltimeout</code>	Блокировать функции, планируемые таймерами
<code>Splnet</code>	<code>splstr</code>	Блокировать функционирование сетевых протоколов
<code>spltty</code>	<code>spltty</code>	Блокировать прерывания терминала
<code>splbio</code>	<code>spldisk</code>	Блокировать дисковые прерывания
<code>splimp</code>		Блокировать прерывания сетевых устройств
<code>splclock</code>		Блокировать прерывание аппаратного таймера
<code>splhigh</code>	<code>spl7</code> или <code>splhi</code>	Запретить обработку всех прерываний
<code>splx</code>	<code>splx</code>	Восстановить <code>ipl</code> в предыдущее сохраненное значение

После того как в системе происходит прерывание, дальнейшие действия зависят от его уровня: если уровень `ipl` окажется выше текущего, то действия приостанавливаются, и запускается обработчик уже нового прерывания. Обработчик начинает свою работу на новом уровне `ipl`. После завершения процедуры обслуживания прерывания уровень `ipl` понижается до предыдущего значения (которое хранится в предыдущем слове состояния процессора в стеке прерываний), и ядро продолжает обработку текущего прерванного процесса. Если ядро получает прерывание, имеющее уровень более низкий или равный текущему значению `ipl`, то такое прерывание не будет обрабатываться немедленно. Оно будет сохранено в регистре прерываний и обработано после соответствующего изменения уровня `ipl`. Алгоритм обработки прерываний показан на рис. 2.5.

Уровни `ipl` сравниваются и устанавливаются на аппаратном уровне в зависимости от архитектуры конкретного компьютера. Ядро системы UNIX имеет механизмы четкого определения или установки уровней `ipl`. К примеру, ядро может повысить уровень `ipl` с целью блокировки прерываний на время выполнения некоторых критичных инструкций. Подробнее об этой возможности см. в разделе 2.5.2.

На некоторых аппаратных платформах поддерживается возможность организации отдельного глобального *стека прерываний*, используемого всеми обработчиками. На платформах, не имеющих подобного стека, обработчики

задействуют стек ядра текущего процесса. В таком случае должен быть обеспечен механизм изоляции остальной части стека ядра от обработчика. Для этого ядро помещает в свой стек *уровень контекста* перед вызовом обработчика. Этот уровень контекста, подобно кадру стека, содержит в себе информацию, необходимую для восстановления контекста выполнения, предшествующего вызову обработчика прерывания.

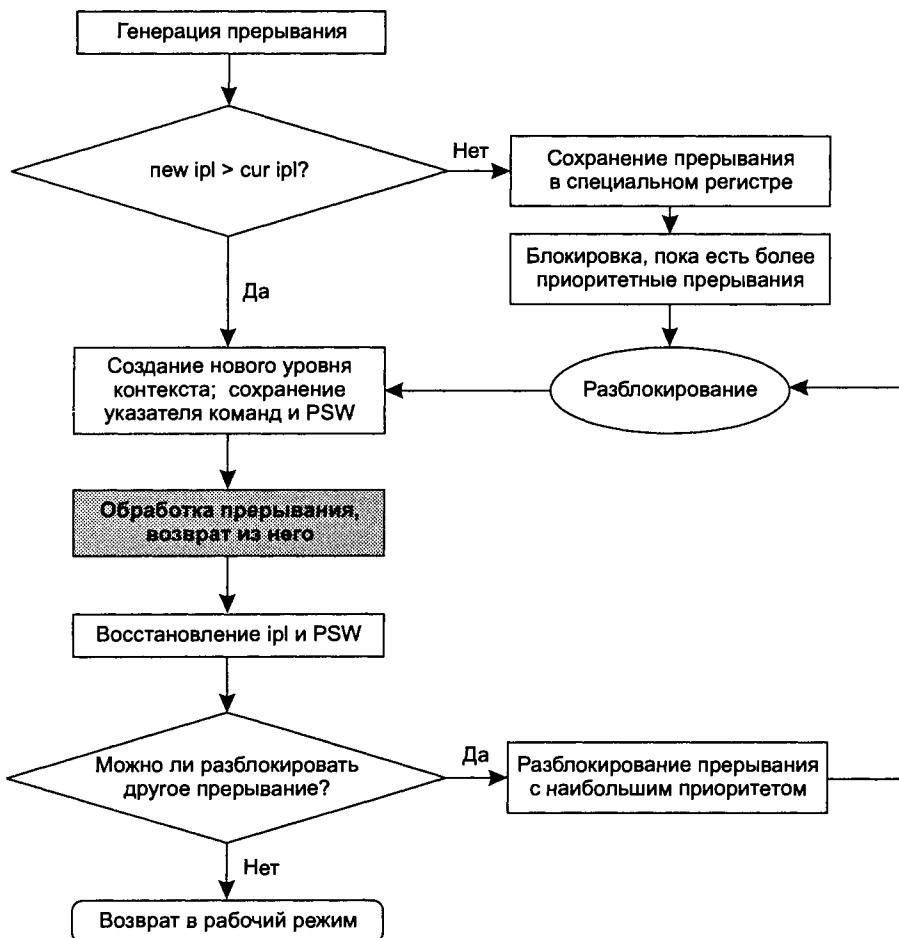


Рис. 2.5. Алгоритм обработки прерываний

2.5. Синхронизация

Ядро системы UNIX является реентерабельным. В любой момент времени в ядре могут быть активны сразу несколько процессов. Конечно, на однопроцессорных системах только один из них окажется текущим, в то время как

остальные будут блокированы, находясь в режиме ожидания освобождения процессора или иного системного ресурса. Так как все эти процессы используют одну и ту же копию структур данных ядра, необходимо обеспечивать некоторую форму синхронизации для предотвращения порчи ядра.

На рис. 2.6 показан пример, который показывает, к чему может привести отсутствие синхронизации. Представьте, что процесс пытается удалить элемент Б из связанного списка. После выполнения первой строки кода происходит прерывание, разрешающее другому процессу начать работу. Если второй процесс попытается получить доступ к тому же списку, то обнаружит его в противоречивом состоянии, как это показано на рис. 2.6, б. Становится очевидным, что необходимо использовать некий механизм, защищающий от возникновения подобных проблем.

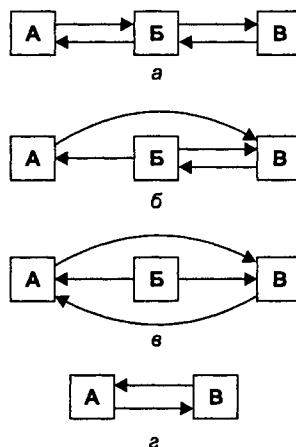


Рис. 2.6. Пример удаления элемента из связанного списка: а — начальное положение; б — после $B->prev->next = B->next$; в — после $B->next->prev = B->prev$; г — после $free(B)$

В системе UNIX применяется несколько различных технологий синхронизации. Система изначально создана невытесняющей. Это означает, что процесс, выполняющийся в режиме ядра, не может быть вытеснен другим процессом, даже если отведенный ему квант времени уже исчерпан. Процесс должен самостоятельно освободить процессор. Это обычно происходит в тот момент, когда процесс приостанавливает свою работу в ожидании необходимого ресурса или какого-то события; когда процесс завершил функционирование в режиме ядра и когда собирается возвращаться в режим задачи. В любом случае, так как процесс освобождает процессор добровольно, он может быть уверен, что ядро системы находится в корректном состоянии. (Современные системы UNIX, работающие в режиме реального времени, поддерживают возможность вытеснения при определенных условиях, см. подробнее раздел 5.6.)

Создание ядра системы не вытесняющим является гибким решением большинства проблем, связанных с синхронизацией. Вернемся к примеру, показанному на рис. 2.6. В данном случае ядро системы может обрабатывать связанный

список без его блокировки, не беспокоясь о возможном вытеснении. Существуют три ситуации, при возникновении которых необходима синхронизация:

- ◆ операции блокировки;
- ◆ прерывания;
- ◆ работа многопроцессорных систем.

2.5.1. Операции блокировки

Операция блокировки – это операция, которая блокирует процесс (то есть переводит процесс в *спящий* режим до тех пор, пока блокировка не будет снята). Так как ядро системы не является вытесняющим, оно может манипулировать большинством объектов (структурами данных и ресурсами) без возможности причинения им какого-либо вреда, так как заранее известно, что никакой другой процесс не может получить к ним в это время доступ. Однако существует некоторое количество объектов, которым необходима защита на время проведения блокировки. Для этого существуют специальные дополнительные механизмы. Например, процесс может производить операцию *read* (чтения) из файла в блочный буфер диска, находящийся в памяти ядра. Так как необходима процедура ввода-вывода с диска, процесс должен ожидать ее завершения, позволив в этот период времени выполнять другому процессу. Однако в этом случае ядру необходимо точно знать, что новый процесс ни в коем случае не получит доступ к буферу, так как тот находится в незавершенном состоянии.

Для защиты таких объектов ядро ассоциирует с ними атрибут защиты *lock*. Он может иметь простейшую реализацию и представлять собою однобитовый флаг, значение которого устанавливается, если объект заблокирован и сбрасывается в противоположном случае. Перед тем как начать пользоваться каким-либо объектом, каждый процесс обязан проверять, не заблокирован ли требуемый объект. Если тот свободен, процесс устанавливает флаг блокировки, после чего начинает использование ресурса. Ядро системы также ассоциирует с объектом еще один флаг – *wanted* (необходимость). Флаг устанавливается на объект процессом, если тот ему необходим, но в данный момент времени заблокирован. Когда другой процесс заканчивает использование объекта, то перед его освобождением проверяет флаг *wanted* и «будит» те процессы, которые ожидают данный объект. Такой механизм позволяет процессу блокировать ресурс на долгий период времени, даже если процесс приостанавливает свою работу, передавая возможность выполнения другими удерживая при этом необходимый ему ресурс.

На рис. 2.7 показан алгоритм блокировки ресурсов. Необходимо учитывать следующие замечания:

- ◆ Процесс блокирует себя в том случае, если не может получить необходимый ресурс, или в случае ожидания события, например завершения ввода-вывода. Для этого процесс вызывает процедуру *sleep()*. Вышеописанный процесс называется *блокировкой* по событию или ресурсу.

- ◆ Процедура `sleep()` помещает процесс в специальную очередь блокированных процессов, изменяет его режим на спящий и вызывает функцию `switch()` для инициализации переключения контекста и дальнейшего разрешения выполнения следующего процесса.
- ◆ Процесс, освобождающий ресурс, вызывает процедуру `wakeup()` для пробуждения *всех* процессов, ожидающих этот ресурс¹. Функция `wakeup()` находит все эти процессы, изменяет их режим на работоспособный и помещает их в очередь планировщика, в которой они ожидают выполнения.
- ◆ Иногда промежуток, прошедший между моментом, когда процесс был разбужен, и временем, когда подошла его очередь выполняться, бывает очень большим. Возможна такая ситуация, что другие текущие процессы могут снова занять необходимый ему ресурс.
- ◆ Следовательно, после пробуждения процессу необходимо снова проверить, доступен ли необходимый ресурс. Если тот оказывается занятым, то процесс снова переходит в спящий режим.

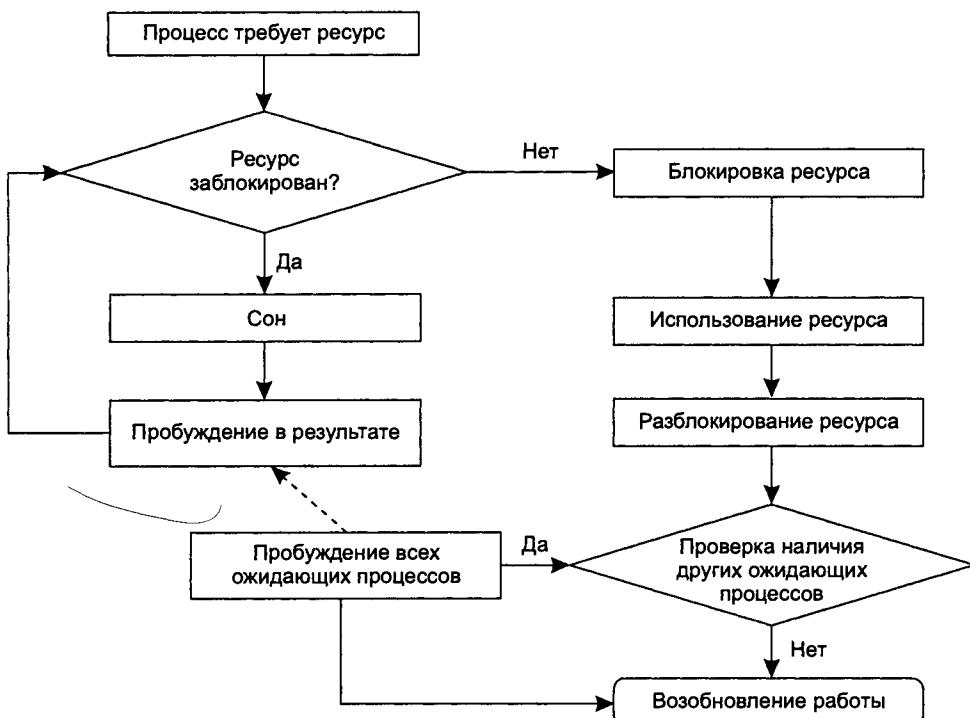


Рис. 2.7. Алгоритм блокировки ресурсов

¹ Более поздние версии системы UNIX поддерживают несколько альтернативных вариантов вызова `wakeup()`, таких как `wake_one()` и `wakeprocs()`.

2.5.2. Прерывания

Несмотря на то что ядро системы защищено от вытеснения другим процессом, процесс, манипулирующий структурами ядра, может быть прерван различными устройствами. Если обработчик прерывания попытается получить доступ к таким структурам, то обнаружит, что они находятся в состоянии нарушения целостности. Возникшую проблему можно решить при помощи блокировки прерываний на период доступа к критически важным структурам данных. Ядро использует триггеры, приведенные в табл. 2.1 для повышения уровня *ipl* и блокировки всех прерываний. Такие области кодов называются *критическими секциями* (см. пример в листинге 2.1).

Листинг 2.1. Блокировка прерываний для критических областей

```
int x = splbio(); /* повышает уровень ipl, возвращает предыдущее
                   значение ipl */
...
/* изменение дискового кэша */
...
splx(x); /* восстанавливает предшествующий уровень ipl */
```

При маскировании прерываний следует учитывать следующие важные соображения:

- ◆ Прерывания обычно требуют незамедлительной обработки, следовательно, они не могут удерживаться слишком долго. Таким образом, критические области кода должны быть по возможности краткими и малочисленными.
- ◆ Необходимо блокировать только те прерывания, обработка которых требует обращения к данным, использующимся в критической области. В приведенном примере целесообразно блокировать только дисковые прерывания.
- ◆ Два различных прерывания могут иметь один и тот же уровень приоритета. Например, на многих системах прерывания терминала и диска происходят на уровне *ipl* 21.
- ◆ Блокирование прерывания приводит к блокированию всех прерываний, имеющих такой же или более низкий уровень приоритета.

ПРИМЕЧАНИЕ

При описании подсистем *UNIX* слово «блокирование» употребляется в нескольких различных значениях. Процесс блокируется на ресурсе или событии, переходя в спящий режим и ожидая освобождения необходимого ресурса или наступления события. Ядро блокирует прерывание или сигнал, задерживая на время его передачу. И наконец, подсистема ввода-вывода передает данные на устройства хранения или от них в блоках фиксированного размера.

2.5.3. Многопроцессорные системы

Появление многопроцессорных систем стало причиной возникновения нового класса проблем синхронизации. Защита ядра, основанная на отсутствии вытеснения, здесь уже не может быть применена. В однопроцессорных системах ядро может производить манипуляции с большинством структур данных, не опасаясь за их целостность, так как работа ядра не может быть вытеснена никем. Необходимо защищать только те данные, которые могут оказаться доступны обработчикам прерываний, а также те, целостность которых зависит от работы вызова `sleep()`.

В многопроцессорных системах два процесса в состоянии одновременно выполняться в режиме ядра на разных процессорах¹, а также выполнять параллельно одну и ту же функцию. Таким образом, каждый раз, когда ядро обращается к глобальным структурам данных, оно должно защищать их от получения доступа с других процессоров. Сами механизмы защиты также должны быть защищены от особенностей выполнения в многопроцессорных системах. Если два процесса, выполняющиеся на различных процессорах, попытаются одновременно заблокировать один и тот же объект, только один должен завершить успешно эту процедуру.

Защита от прерываний является достаточно сложной задачей, так как все процессоры могут обрабатывать прерывания. Нецелесообразно производить блокировку на каждом процессоре, так как это чревато значительным снижением производительности системы. Многопроцессорные системы требуют более сложных механизмов синхронизации, которые будут подробнее описаны в главе 7.

2.6. Планирование процессов

Центральный процессор представляет собой ресурс, который используется всеми процессами системы. Часть ядра, распределяющая процессорное время между процессами, называется *планировщиком* (*scheduler*). В традиционных системах UNIX планировщик использует принцип *вытесняющего циклического планирования*. Процессы, имеющие одинаковые приоритеты, будут выполняться циклически друг за другом, и каждому из них будет отведен для этого определенный период (квант) времени, обычно равный 100 миллисекундам. Если какой-либо процесс, имеющий более высокий приоритет, становится выполняемым, то он вытеснит текущий процесс (конечно, если тот не выполняется в режиме ядра) даже в том случае, если текущий процесс не исчерпал отведенного ему кванта времени.

В традиционных системах UNIX приоритет процесса определяется двумя факторами: фактором «любезности» и фактором *утилизации*. Пользователи

¹ Под словом «многопроцессорных», автор, видимо, имел в виду двухпроцессорную систему. — Прим. ред.

могут повлиять на приоритет процесса при помощи изменения значения его «любезности», используя системный вызов `nice` (но только суперпользователь имеет полномочия увеличивать приоритет процесса). Фактор утилизации определяется степенью последней (то есть во время последнего обслуживания процесса процессором) загруженности CPU процессом. Этот фактор позволяет системе динамически изменять приоритет процесса. Ядро системы периодически повышает приоритет процесса, пока тот не выполняется, а после того, как процесс все-таки получит какое-то количество процессорного времени, его приоритет будет понижен. Такая схема защищает процессы от «зависания»¹, так как периодически наступает такой момент, когда ожидающий процесс получает достаточный уровень приоритета для выполнения.

Процесс, выполняющийся в режиме ядра, может освободить процессор в том случае, если произойдет его блокирование по событию или ресурсу. Когда процесс снова станет работоспособным, ему будет назначен приоритет ядра. Приоритеты ядра обычно выше приоритетов любых прикладных задач. В традиционных системах UNIX приоритеты представляют собой целые числа в диапазоне от 0 до 127, причем чем меньше их значение, тем выше приоритет процесса (так как система UNIX почти полностью написана на языке C, в ней используется стандартный подход к началу отсчета от нуля). Например, в ОС 4.3BSD приоритеты ядра варьируются в диапазоне от 0 до 49, а приоритеты прикладных задач — в диапазоне от 50 до 127. Приоритеты прикладных задач могут изменяться в зависимости от степени загрузки процессора, но приоритеты ядра являются фиксированными величинами и зависят от причины засыпания процесса. Именно по этой причине приоритеты ядра также известны как *приоритеты сна*. В табл. 2.2 приводятся примеры таких приоритетов для операционной системы 4.3BSD UNIX.

Более подробно работа планировщика будет изложена в главе 5.

Таблица 2.2. Приоритеты сна в ОС 4.3BSD UNIX

Приоритет	Значение	Описание
PSWP	0	Свопинг
PSWP + 1	1	Страницный демон
PSWP + 1/2/4	1/2/4	Другие действия по обработке памяти
PINOD	10	Ожидание освобождения inode
PRIBIO	20	Ожидание дискового ввода-вывода
PRIBIO + 1	21	Ожидание освобождения буфера
PZERO	25	Базовый приоритет
TTIPRI	28	Ожидание ввода с терминала
TTOPRI	29	Ожидание вывода с терминала

¹ Из-за отказа операционной системы его обслуживать. — Прим. ред.

Приоритет	Значение	Описание
PWAIT	30	Ожидание завершения процесса-потомка
PLOCK	35	Консультативное ожидание блокированного ресурса
PSLEP	40	Ожидание сигнала

2.7. Сигналы

Для информирования процесса о возникновении асинхронных событий или необходимости обработки исключительных состояний в системе UNIX используются *сигналы*. Например, когда пользователь нажимает на своем терминале комбинацию клавиш Ctrl+C, процессу, с которым пользователь в данный момент интерактивно работает, передается сигнал SIGINT. Когда процесс завершается, он отправляет своему процессу-родителю сигнал SIGCHLD. В ОС UNIX поддерживается определенное количество сигналов (31 в 4.3BSD и SVR3). Большинство из них зарезервированы для специальных целей, однако сигналы SIGUSR1 и SIGUSR2 доступны для использования в приложениях в произвольном назначении.

Сигналы используются для многих операций. Процесс может выслать сигнал одному или нескольким другим процессам, используя системный вызов *kill*. Драйвер терминала вырабатывает сигналы в ответ на нажатия клавиш или происходящие события для процессов, присоединенных к нему. Ядро системы вырабатывает сигналы для уведомления процесса о возникновении аппаратного исключения или в случаях возникновения определенных ситуаций, например превышения квот.

Каждый сигнал имеет определенную по умолчанию реакцию на него, обычно это завершение процесса. Некоторые сигналы по умолчанию игнорируются, а небольшая часть из них приостанавливает процесс. Процесс может указать системе на необходимость иной реакции на сигналы, отличной от заданной по умолчанию. Для этого используется вызов *signal* (System V), *sigvec* (BSD) или *sigaction* (POSIX.1). Действия, отличные от принятых по умолчанию, могут заключаться в запуске обработчика сигнала, определенного разработчиком приложения, его игнорировании, а иногда и в процедурах, противоположных тем, что производятся в обычных случаях. Процесс также имеет возможность временной блокировки сигнала. В таком случае сигнал будет доставлен процессу только после того, как тот будет разблокирован.

Процесс не в состоянии прореагировать на сигнал немедленно. После выработывания сигнала ядро системы уведомляет об этом процесс при помощи установки бита в маске ожидающих сигналов, расположенной в структуре *proc* данного процесса. Процесс должен постоянно быть готовым к получению сигнала и ответу на него. Это возможно только в том случае, если он находится в очереди на выполнение. В начале выполнения процесс обрабатывает

все ожидающие его сигналы и только затем продолжает работать в режиме задачи. (Эта процедура не включает функционирование самих обработчиков сигналов, которые также выполняются в режиме задачи.)

Что же происходит в том случае, если сигнал предназначен для спящего процесса? Будет ли он ожидать того момента, когда процесс снова станет работоспособным, или его «сон» будет прерван? Это зависит от причины приостановки работы процесса. Если процесс ожидает события, которое вскоре должно произойти (например, завершение операции ввода-вывода с диска), то нет необходимости «будить» такой процесс. С другой стороны, если процесс ожидает такое событие, как ввод с терминала, то заранее не известно, через какой промежуток времени оно может произойти. В таких случаях ядро системы будит процесс и прерывает выполнение системного вызова, из-за которого данный процесс был заблокирован. В операционной системе 4.3BSD поддерживается системный вызов `siginterrupt`, который служит для управления реакцией системного вызова¹. Используя `siginterrupt`, разработчик может указать, будет ли обработка системных вызовов, прерываемых сигналами, прекращена или возобновлена. Более подробно сигналы будут описаны в главе 4.

2.8. Новые процессы и программы

Система UNIX является многозадачной средой, и различные процессы действуют в ней все время. Каждый процесс в один момент времени выполняет только одну программу, однако несколько процессов могут выполнять одну и ту же программу в параллельном режиме. Такие процессы умеют разделять между собой единую копию текста (кода) программы, хранящейся в памяти, но при этом содержать собственные области данных и стека. Более того, процесс способен загружать одну или больше программ в течение своего времени жизни. UNIX таким образом разделяет процессы и программы, которые могут выполняться при их помощи.

Для поддержки многозадачной среды в UNIX существует несколько различных системных вызовов, создающих и завершающих процессы, а также запускающих новые программы. Системные вызовы `fork` и `vfork` служат для создания новых процессов. Вызов `exec` загружает новую программу. Следует помнить о том, что работа процесса может быть завершена после принятия определенного сигнала.

2.8.1. Вызовы `fork` и `exec`

Системный вызов `fork` создает новый процесс. При этом вызывающий процесс становится родительским, а новый процесс является его потомком. Связь «родитель-потомок» создает иерархию процессов, графически изображенную

¹ Из-за которого процесс заблокирован. — Прим. ред.

на рис. 2.4. Процесс-потомок является точной копией своего родительского процесса. Его адресное пространство полностью повторяет пространство родительского процесса, а программа, выполняемая им изначально, также не отличается от той, что выполняет процесс-родитель. Фактически процесс-потомок начинает работу в режиме задачи после возврата из вызова `fork`.

Так как оба процесса (родитель и его потомок) выполняют одну и ту же программу, необходимо найти способ отличия их друг от друга и предоставления им возможности нормального функционирования. С другой стороны, различные процессы¹ невозможно заставить выполнять различные действия. Поэтому системный вызов `fork` возвращает различные значения процессу-родителю и его потомку: для потомка возвращается значение 0, а для родителя — идентификатор PID потомка.

Чаще всего после вызова `fork` процесс-потомок сразу же вызывает `exec` и тем самым начинает выполнение новой программы. В библиотеке языка C существует несколько различных форм вызова `exec`, таких как `execve`, `execvpe` и `execvpr`. Все они незначительно отличаются друг от друга набором аргументов и после некоторых предварительных действий используют один и тот же системный вызов. Общее имя `exec` относится к любой неопределенной функции этой группы. В листинге 2.2 приведен фрагмент кода программы, использующий вызовы `fork` и `exec`.

Листинг 2.2. Использование вызовов `fork` и `exec`

```
If ((result==fork))==0 {
    /* код процесса-потомка */

    ...
    if (execve("new_program", ...)<0)
        perror("execve failed");
        exit(1);
} else if (result<0) {
    perror("fork"); /* вызов fork неудачен */
}
/* здесь продолжается выполнение процесса-предка*/
```

После наложения новой программы на существующий процесс при помощи `exec` потомок не возвратит управление предыдущей программы, если не произойдет сбоя вызова. После успешного завершения работы функции `exec` адресное пространство процесса-потомка будет заменено пространством новой программы, а сам процесс будет возвращен в режим задачи с установкой его указателя команд на первую выполняемую инструкцию этой программы.

Так как вызовы `fork` и `exec` часто используются вместе, возникает закономерный вопрос: а может стоит использовать для выполнения задачи единый системный вызов, который создаст новый процесс и выполнит в нем новую программу. Первые системы UNIX [9] теряли много времени на дублирова-

¹ Имеется в виду оригинал и копия. — Прим. ред.

ние адресного пространства процесса-родителя для его потомка (в течение выполнения `fork`) для того, чтобы потом все равно заменить его новой программой.

Однако существует и немало преимуществ в разделении этих системных вызовов. Во многих клиент-серверных приложениях сервер может создавать при помощи `fork` множество процессов, выполняющих одну и ту же программу¹. С другой стороны, иногда процессу необходимо запустить новую программу без создания для ее функционирования нового процесса. И наконец, между системными вызовами `fork` и `exec` процесс-потомок может выполнить некоторое количество заданий, обеспечивающих функционирование новой программы в желаемом состоянии. Это могут быть задания, выполняющие:

- ◆ операции перенаправления ввода, вывода или вывода сообщений об ошибках;
- ◆ закрытие не нужных для новой программы файлов, наследованных от предка;
- ◆ изменение идентификатора UID или GID (группы процесса);
- ◆ сброс обработчиков сигналов.

Если все эти функции попробовать выполнить при помощи единственного системного вызова, то такая процедура окажется громоздкой и неэффективной. Существующая связка `fork-exes` предлагает высокий уровень гибкости, а также является простой и модульной. В разделе 2.8.3 мы расскажем о способах минимизации проблем, связанных с быстродействием этой связи (из-за использования раздельных вызовов).

2.8.2. Создание процесса

Системный вызов `fork` создает новый процесс, который является почти точной копией его родителя. Единственное различие двух процессов заключается только в необходимости отличать их друг от друга. После возврата из `fork` процесс-родитель и его потомок выполняют одну и ту же программу (функционирование которой продолжается сразу же вслед за `fork`) и имеют идентичные области данных и стека. Системный вызов `fork` во время своей работы должен совершить следующие действия:

- ◆ зарезервировать пространство свопинга для данных и стека процесса-потомка;
- ◆ назначить новый идентификатор PID и структуру `proc` потомка;
- ◆ инициализировать структуру `proc` потомка. Некоторые поля этой структуры копируются от процесса-родителя (такие как идентификаторы

¹ С появлением современных многопоточных систем UNIX такая возможность больше не вос требована: теперь сервер может иметь возможность создания необходимого количества нитей выполнения.

пользователя и группы, маски сигналов и группа процесса), часть полей устанавливается в 0 (время нахождения в резидентном состоянии, использование процессора, канал сна и т. д.), а остальным присваиваются специфические для потомка значения (поля идентификаторов PID потомка и его родителя, указатель на структуру `proc` родителя);

- ◆ разместить карты трансляции адресов для процесса-потомка;
- ◆ выделить область и потомка и скопировать в нее содержимое области и процесса-родителя;
- ◆ изменить ссылки области и на новые карты адресации и пространство свопинга;
- ◆ добавить потомка в набор процессов, разделяющих между собой область кода программы, выполняемой процессом-родителем;
- ◆ постранично дублировать области данных и стека родителя и модифицировать карты адресации потомка в соответствии этими новыми страницами;
- ◆ получить ссылки на разделяемые ресурсы, наследуемые потомком, такие как открытые файлы и текущий рабочий каталог;
- ◆ инициализировать аппаратный контекст потомка посредством копирования текущего состояния регистров его родителя;
- ◆ сделать процесс-потомок выполняемым и поместить его в очередь планировщика;
- ◆ установить для процесса-потомка по возврату из вызова `fork` нулевое значение;
- ◆ вернуть идентификатор PID потомка его родителю.

2.8.3. Оптимизация вызова `fork`

Системный вызов `fork` должен предоставить процессу-потомку логически идентичную копию адресного пространства его родителя. В большинстве случаев потомок заменяет предоставленное адресное пространство, так как сразу же после выполнения `fork` вызывает `exec` или `exit`. Таким образом, создание копии адресного пространства (так, как это реализовано в первых системах UNIX) является неоптимальной процедурой.

Вышеописанная проблема была решена двумя различными способами. Сначала был разработан метод *копирования при записи*, впервые нашедший реализацию в ОС System V и в настоящий момент используемый в большинстве систем UNIX. При таком подходе страницы данных и стека родителя временно получают атрибут «только для чтения» и маркируются как «копируемые при записи». Потомок получает собственную копию карт трансляции адресов, но использует одни и те же страницы памяти вместе со своим родительским процессом. Если кто-то из них (родитель или потомок) попытается изменить страницу памяти, произойдет ошибочная исключительная

ситуация (так как страницы доступны только для чтения), после чего ядро системы запустит обработчик произошедшей ошибки. Обработчик увидит, что страница помечена как «копируемая при записи», и создаст новую ее новую копию, которую уже можно изменять. Таким образом, происходит копирование только тех страниц памяти, которые требуется изменять, а не всего адресного пространства целиком. Если потомок вызовет `exec` или `exit`, то защита страниц памяти вновь станет обычной, и флаг «копирования при записи» будет сброшен.

В системе BSD UNIX представлен несколько иной подход к решению проблемы, реализованный в новом системном вызове `vfork`. Программист может воспользоваться им вместо `fork`, если планирует вслед за ним вызвать `exec`. Функция `vfork` не производит копирования. Вместо этого процесс-родитель предоставляет свое адресное пространство потомку и блокируется до тех пор, пока тот не вернет его. Затем происходит выполнение потомка в адресном пространстве родительского процесса до того времени, пока не будет произведен вызов `exec` или `exit`, после чего ядро вернет родителю его адресное пространство и выведет его из состояния сна. Системный вызов `vfork` выполняется очень быстро, так как не копирует даже карты адресации. Адресное пространство передается потомку простым копированием регистров карты адресации. Однако следует отметить, что *вызов `vfork` является достаточно опасным, так как позволяет одному процессу использовать и даже изменять адресное пространство другого процесса*. Это свойство `vfork` используют различные программы, такие как `csh`.

2.8.4. Запуск новой программы

Системный вызов `exec` заменяет адресное пространство вызывающего процесса на адресное пространство новой программы. Если процесс был создан при помощи `vfork`, то вызов `exec` возвращает старое адресное пространство родительскому процессу. В ином случае этот вызов высвобождает старое адресное пространство. Вызов `exec` предоставляет процессу новое адресное пространство и загружает в него содержимое новой программы. По окончании работы `exec` процесс продолжает выполнение с первой инструкции новой программы.

Адресное пространство процесса состоит из нескольких определенных компонентов¹:

- ◆ **Текст (text).** Содержит выполняемый код.
- ◆ **Инициализированные данные (initialized data).** Содержат объекты данных, которые в программе уже имеют начальные значения и соответствуют секции инициализированных данных в выполняемом файле.

¹ Такое разделение представляется весьма функциональным в теории, однако ядро не распознает так много различных компонентов. Например, в системе SVR4 адресное пространство видится просто как набор разделяемых или приватных отображений.

- ◆ **Неинициализированные данные** (uninitialized data). Имеет исторически сложившееся название *блока статического хранения* (block static storage, bss)¹. Содержит переменные, которые были в программе описаны, но значения им присваивались. Объекты в этой области всегда заполнены нулями при первом обращении к ним. Так как хранение нескольких страниц нулей в выполняемом файле представляется нерациональным, в заголовке программы принято просто описывать общий размер этой области и предоставлять операционной системе самой генерировать заполненные нулями страницы.
- ◆ **Разделяемая память** (shared memory). Многие системы UNIX позволяют процессам совместно использовать одни и те же области памяти.
- ◆ **Разделяемые библиотеки** (shared libraries). Если система поддерживает библиотеки динамической связи, процесс может обладать несколькими отдельными областями памяти, содержащими библиотечный код, а также библиотечные данные, которые могут использоваться и другими процессами.
- ◆ **Куча** (heap). Источник динамически выделяемой памяти. Процесс берет память из кучи при помощи системных вызовов brk или sbrk, либо используя функцию malloc() из стандартной библиотеки C. Ядро предоставляет кучу каждому процессу и расширяет ее по мере необходимости.
- ◆ **Стек приложения** (user stack). Ядро выделяет стек каждому процессу. В большинстве традиционных реализаций системы UNIX ядро прозрачно отслеживает возникновение исключительных состояний, связанных с переполнением стека, и расширяет стек до определенного в системе максимума.

Применение разделяемой памяти является стандартной возможностью System V UNIX, но не поддерживается в системе 4BSD (до версии 4.3). Многие коммерческие реализации UNIX, основанные на BSD, поддерживают как разделяемую память, так и некоторые формы разделяемых библиотек в качестве дополнительных возможностей системы. В последующем описании работы `exec` мы рассмотрим программу, которая не использует ни одну из этих возможностей.

Система UNIX поддерживает различные форматы выполняемых файлов. Первым поддерживаемым форматом был a.out, имеющий 32-байтовый заголовок с последующими секциями текста, данных и таблицы символов. Заголовок программы содержит размеры *текста*, областей *инициализированных* и *неинициализированных* *данных*, а также *точку входа*, которая является адресом первой инструкции программы, которая будет выполнена. Заголовок

¹ Морис Бах в широко известной книге «Архитектура операционной системы UNIX» пишет, что сокращение bss имеет происхождение от ассемблерного псевдооператора для машины IBM 7090 и расшифровывается как block started by symbol (блок, начинающийся с символа). — Прим. ред.

вок также содержит **магическое число**, идентифицирующее файл как действительно выполняемый и дающее дополнительную информацию о его формате, такую как: требуется ли ему разбиение на страницы или начинается ли секция данных на краю страницы. Набор поддерживаемых магических чисел определен в каждой реализации UNIX по-своему.

Системный вызов `exec` выполняет следующие действия:

1. Разбирает путь к исполняемому файлу и осуществляет доступ к нему.
2. Проверяет, имеет ли вызывающий процесс полномочия на выполнение файла.
3. Читает заголовок и проверяет, что он действительно исполняемый¹.
4. Если для файла установлены биты SUID или SGID, то эффективные идентификаторы UID и GID вызывающего процесса изменяются на UID и GID, соответствующие владельцу файла.
5. Копирует аргументы, передаваемые в `exec`, а также *переменные среды* в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Выделяет пространство свопинга для областей данных и стека.
7. Высвобождает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи `vfork`, производится возврат старого адресного пространства родительскому процессу.
8. Выделяет карты трансляции адресов для нового текста, данных и стека.
9. Устанавливает новое адресное пространство. Если область текста активна (какой-то другой процесс уже выполняет ту же программу), то она будет совместно использоваться с этим процессом. В других случаях пространство должно инициализироваться из выполняемого файла. Процессы в системе UNIX обычно разбиты на страницы, что означает, что каждая страница считывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек приложения.
11. Сбрасывает все обработчики сигналов в действия, определенные по умолчанию, так как функции обработчиков сигналов не существуют в новой программе. Сигналы, которые были проигнорированы или заблокированы перед вызовом `exec`, остаются в тех же состояниях.
12. Инициализирует аппаратный контекст. При этом большинство регистров сбрасывается в 0, а указатель команд получает значение точки входа программы.

¹ Вызов `exec` также может выполнять сценарии командного интерпретатора, имеющие первую строку типа `#!/shell_name`.

2.8.5. Завершение процесса

Функция ядра `exit()` предназначена для завершения процесса. Она вызывается изнутри, когда процесс завершается по сигналу. С другой стороны, программа может выполнить системный вызов `exit`, который, в свою очередь, вызовет функцию `exit()`. Функция `exit()` производит следующие действия:

1. Отключает все сигналы.
2. Закрывает все открытые файлы.
3. Освобождает файл программы и другие ресурсы, например текущий каталог.
4. Делает запись в журнал данной учетной записи.
5. Сохраняет данные об использованных ресурсах и статус выхода в структуре `proc`.
6. Изменяет состояние процесса на `SZOMB` (зомби) и помещает его структуру `proc` в список процессов-зомби.
7. Устанавливает процесс `init` любому существующему потомку завершающегося процесса в качестве родителя.
8. Освобождает адресное пространство, область `u`, карты трансляции адресов и пространство свопинга.
9. Посыпает родителю завершающегося процесса сигнал `SIGCHLD`. Этот сигнал обычно игнорируется и реально необходим только в тех случаях, если по какой-то причине родительскому процессу необходимо знать о завершении работы его потомка.
10. Будит родительский процесс, если тот был приостановлен.
11. Вызывает `switch()` для перехода к выполнению следующего процесса в расписании.
12. После завершения работы `exit()` процесс находится в состоянии зомби. Вызов `exit` не освобождает структуру `proc` завершенного процесса, так как его родителю, возможно, будет необходимо получить статус выхода и информацию об использовании ресурсов. За освобождение структуры `proc` потомка отвечает его процесс-родитель, как будет описано подробнее позже. По завершении этой процедуры структура `proc` возвращается в список свободных структур, и на этом процесс «чистки следов» завершается.

2.8.6. Ожидание завершения процесса

Часто родительскому процессу необходимо обладать информацией о завершении работы своего потомка. Например, когда командный интерпретатор, исполняя команду, порождает интерактивный процесс (переводящий

ввод и вывод на себя) и, становясь его родителем, должен ждать завершения своего потомка, чтобы после этого вновь быть готовым ко вводу очередной команды. Когда завершается фоновый процесс, командному интерпретатору может потребоваться сообщить об этом пользователю (выводом соответствующего сообщения на терминал). Командный интерпретатор также может запрашивать статус выхода процесса-потомка, так как дальнейшие действия пользователя зависят от того, завершился ли процесс успешно или имела место ошибка. В системах UNIX поддерживаются следующие системные вызовы, дающие возможность отслеживания завершения работы процессов:

```
wait(stat_loc);           /* System V, BSD и POSIX.1 */
wait3(statusp, options, rusagep); /* BSD */
waitpid(pid, stat_loc, options); /* POSIX.1 */
waitid(idtype, id, infop, options); /* SVR4 */1
```

Системный вызов `wait` позволяет процессу ожидать завершения работы его потомка. Так как потомок может уже оказаться завершенным к моменту совершения вызова, то ему необходимо уметь распознавать такую ситуацию. После запуска вызов `wait` первоначально проверяет, имеет ли вызывающий его процесс потомков, функционирование которых завершено или приостановлено. Если он находит такие процессы, то немедленно происходит возврат из этого системного вызова. Если таких процессов нет, то `wait` блокирует вызвавший его процесс до тех пор, пока один из потомков не завершит свою работу, после чего последует возврат из вызова. В обоих случаях вызов `wait` возвратит PID завершившегося процесса, запишет его статус выхода в `stat_loc` и освободит его структуру `proc` (если более одного процесса-потомка завершили свою работу, `wait` обработает только первый из найденных). Если процесс-потомок находится в режиме трассировки, возврат из `wait` также произойдет, когда процесс-потомок получит сигнал. Ошибку `wait` вернет в случае, когда родительский процесс не имеет ни одного потомка (функционирующего или уже завершенного) либо если его работа была прервана поступившим сигналом.

Система 4.3BSD поддерживает вызов `wait3` (названный так потому, что он имеет три аргумента), который также возвращает информацию об использовании ресурсов потомком (время работы в режиме ядра и в режиме задачи процесса-потомка, а также о всех его завершенных потомках). В стандарте POSIX.1 [6] описан системный вызов `waitpid`, в котором используется аргумент `pid` для ожидания потомка, имеющего определенный идентификатор процесса или группы процесса. Системные вызовы `wait3` и `waitpid` поддерживают две опции: `WNOHANG` и `WUNTRACED`. Опция `WNOHANG` заставляет вызов

¹ В операционной системе SVR4 поддержка команд `wait3` и `waitpid` реализована в виде библиотечных функций.

`wait3` немедленно завершить работу, если он не нашел ни одного завершившегося процесса. Опция `WUNTRACED` завершает работу вызова, если потомок приостанавливается, или вновь продолжает функционирование. В ОС SVR4 системный вызов `waitid` поддерживает все вышеописанные возможности. Он позволяет вызывающему его процессу задавать PID или GID процесса, завершения которого должен ждать, а также определять события, по которым также произойдет возврат из вызова. Возвращает более подробную информацию о процессе-потомке.

2.8.7. Процессы-зомби

Когда процесс завершается, он остается в состоянии «зомби» (*zombie*) до тех пор, пока окончательно не будет «вытерт» родительским процессом. В этом режиме единственным занимаемым ресурсом остается структура `proc`, в которой хранится статус выхода, а также информация об использовании ресурсов системы¹. Эта информация может быть важна для родительского процесса, который получает ее посредством вызова `wait`, который также освобождает структуру `proc` потомка. Если родительский процесс завершается раньше, чем его потомок, то тот усыновляется процессом `init`. После завершения работы потомка процесс `init` вызовет `wait` для освобождения его структуры `proc`.

Определенная проблема возникает в том случае, если процесс завершится раньше своего родителя и последний не вызовет `wait`. Тогда структура процесса-потомка `proc` не будет освобождена, а потомок останется в состоянии зомби до тех пор, пока система не будет перезагружена. Такая ситуация возникает очень редко, так как разработчики командных интерпретаторов знают о существовании проблемы и стараются не допустить ее возникновения в своих программах. Однако потенциальная возможность такой ситуации остается, когда недостаточно внимательно написанные программы не следят за всеми своими процессами-потомками. Это может быть достаточно раздражительным, так как процессы-зомби видимы при помощи `ps`, а пользователи никак не могут их завершить, *так как они уже завершены*. Более того, они продолжают занимать структуру `proc`, уменьшая тем самым максимальное количество активных процессов.

В некоторых более поздних реализациях UNIX поддерживается возможность указания на то, что процесс не будет ожидать завершения работы своих потомков. Например, в системе SVR4 процесс может выставить флаг `SA_NOCLDWAIT` в системном вызове `sigaction`, определяя действие на сигнал `SIGCHLD`. Это дает возможность ядру системы не создавать процессы-зомби, если потомок вызывающего процесса завершит функционирование.

¹ В некоторых реализациях UNIX для хранения таких данных используются специальные структуры `zombie`.

2.9. Заключение

В этой главе мы обсудили взаимодействие ядра системы и прикладных процессов в традиционных системах UNIX. Эти вопросы требуют более широкого обзора, поэтому нам необходимо рассмотреть специфические части системы подробнее. Современные варианты UNIX, такие как SVR4 или Solais 2.x, предлагают дополнительные возможности, описание которых можно найти в следующих главах книги.

2.10. Упражнения

1. Какие элементы контекста процесса необходимо сохранять ядру при обработке:
 - ♦ переключения контекста;
 - ♦ прерывания;
 - ♦ системного вызова?
2. В чем преимущества и недостатки динамического размещения таких объектов, как структура proc, и блоков таблицы дескрипторов?
3. Каким образом ядро системы узнает о том, какой из системных вызовов был сделан? Каким образом происходит доступ к аргументам вызова (хранящимся в пользовательском стеке)?
4. Найдите сходства и различия в обработке системных вызовов и исключений. В чем производимые действия сходны и чем они отличаются друг от друга?
5. Многие реализации UNIX совместимы с другими версиями системы при помощи функций пользовательских библиотек, реализующих системные вызовы других версий ОС. Объясните, различается ли с точки зрения разработчика приложений реализация такой функции в виде библиотеки и в виде системного вызова.
6. На что должен обратить внимание разработчик библиотеки, если он решает реализовать свою функцию в библиотеке как альтернативную системному вызову? Что нужно дополнительно учитывать, если библиотеке необходимо использовать несколько системных вызовов для реализации этой функции?
7. Почему необходимо ограничивать объем работы, выполняемой обработчиком прерывания?
8. Данна система с p различными уровнями приоритетов прерываний. Какое максимальное количество прерываний может поддерживаться системой одновременно? Как влияет это количество на размеры различных стеков?

9. Архитектура процессора Intel 80x86 не предусматривает приоритетов прерываний. Для управления прерываниями используются две инструкции: CLI для запрещения всех прерываний и STI для обратного действия. Опишите алгоритм программной поддержки уровней приоритетов для этого процессора.
10. Когда определенный ресурс системы становится доступным, вызывается процедура `wakeup()`, которая будит все процессы, ожидающие его освобождения. Какие есть недостатки у описанного подхода? Какие вы видите альтернативные решения?
11. Представьте, что существует некоторый системный вызов, комбинирующий функции вызовов `fork` и `exec`. Определите его интерфейс и синтаксис. Каким образом этот вызов будет поддерживать такие возможности системы, как перенаправление ввода-вывода, выполнение в интерактивном или фоновом режиме, а также каналы?
12. Какие возникают проблемы при возврате ошибки от системного вызова `exec`? Как эти проблемы решены ядром?
13. Создайте функцию, позволяющую процессу ожидать завершения работы своего родителя (для любой реализации UNIX по вашему выбору).
14. Представьте, что процессу не нужно блокироваться, пока не будет завершено функционирование его потомка. Как можно убедиться в том, что потомок был удален из системы полностью после завершения?
15. Почему завершившийся процесс будит своего родителя?

2.11. Дополнительная литература

1. Allman, E., «UNIX: The Data Forms», Proceedings of the Winter 1987 USENIX Technical Conference, Jan. 1987, pp. 9–15.
2. American Telephone and Telegraph, «The System V Interface Definition (SVID)», Issue 2, 1987.
3. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
4. Kernighan, B. W., and Pike, R., «The UNIX Programming Environment», Prentice-Hall, Englewood Cliffs, NJ, 1984.
5. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
6. Institute for Electrical and Electronic Engineers, Information Technology, «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]», 1003.1–1990, IEEE, Dec. 1990.

7. Institute for Electrical and Electronic Engineers, POSIX P1003.4a, Threads Extension for Portable Operating Systems, 1994.
8. Salemi, C., Shah, S., and Lund, E., «A Privilege Mechanism for UNIX System V Release 4 Operating Systems», Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 235–241.
9. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1931–1946.

Глава 3

Нити и легковесные процессы

3.1. Введение

Технология использования процессов обладает двумя важными ограничениями. Прежде всего, некоторым приложениям необходимо выполнять несколько крупных независимых друг от друга задач, при этом используя одно и то же адресное пространство, а также другие ресурсы. Примерами таких приложений являются серверные части систем обслуживания баз данных, мониторы прохождения транзакций, а также реализации сетевых протоколов среднего и верхнего уровня. Такие процессы должны функционировать параллельно — и, следовательно, для них должна применяться иная программная модель, поддерживающая параллелизм. Традиционные системы UNIX могут предложить таким приложениям либо выполнять отдельные задачи последовательно, либо придумывать неуклюжие и малоэффективные механизмы поддержки выполнения таких операций.

Вторым ограничением применения традиционной модели является невозможность в полной мере задействовать преимущества многопроцессорных систем, так как процесс способен одновременно использовать только один процессор. Приложение может создать несколько отдельных процессов и выполнять их на имеющихся процессорах компьютера. Необходимо найти методики, позволяющие таким процессам совместно использовать память и ресурсы, а также синхронизироваться друг с другом.

Современные системы UNIX предлагают решение приведенных проблем при помощи различных технологий, реализованных в ОС с целью поддержки параллельного выполнения заданий. К сожалению, скудость стандартной терминологии сильно усложняет задачу описания и сравнения большого количества существующих механизмов параллелизации. В каждом варианте UNIX для их обозначения применяются свои термины, например *нити ядра, прикладные нити, прикладные нити, поддерживаемые ядром, C-threads, pthreads* и «легковесные» процессы (lightweight processes). Эта глава классифицирует используемую терминологию, объясняет основные понятия и описы-

вает возможности основных реализаций UNIX. В конце главы вы увидите анализ преимуществ и недостатков описываемых механизмов. А начнем мы с рассмотрения важности и полезности технологии нитей.

3.1.1. Причины появления технологии нитей

Многие программы выполняют отдельные крупные независимые задачи, которые не могут функционировать последовательно. Например, сервер баз данных находится в режиме приема и обработки множества запросов от клиентов. Так как поступающие запросы не требуется обрабатывать в каком-то определенном порядке, то их можно считать отдельными, независимыми друг от друга задачами, имеющими принципиальную возможность функционирования параллельно. Если система позволяет выполнять параллельно отдельные задачи приложения, то его производительность существенно увеличивается.

В традиционных системах UNIX такие программы используют несколько процессов. Большинство серверных приложений запускают *процесс прослушивания*, функция которого заключается в ожидании запросов клиентов. После поступления запроса процесс вызывает *fork* для запуска нового процесса, обрабатывающего запрос клиента. Так как эта задача часто требует проведения операций ввода-вывода, существует потенциальная возможность блокирования работы процесса, что дает некоторую степень одновременности выполнения даже на однопроцессорных системах.

Представим еще одну ситуацию. Имеется некоторое научное приложение, вычисляющее значения различных элементов массива, каждый из которых независим от остальных. Для решения задачи можно создать отдельные процессы для каждого элемента массива и выполнять их параллельно, выделив для каждого процесса отдельную машину или процессор в многопроцессорной системе. Однако даже на однопроцессорной машине имеет смысл разделить поставленную задачу между несколькими процессами. Если один из них заблокируется, ожидая окончания ввода-вывода или обработки ошибки доступа к страницам памяти, то другие в это время смогут продолжить функционирование. В качестве еще одного примера можно привести утилиту *make* системы UNIX, позволяющую пользователям параллельно производить компиляцию нескольких файлов, при этом каждый из них будет обрабатываться в отдельном процессе.

Использование приложением нескольких процессов имеет некоторые недостатки. При их создании происходят значительные перегрузки системы, так как вызов *fork* является весьма затратным (даже в том случае, если применяется совместное использование адресного пространства при помощи техники *копирования-при-записи*). Каждый процесс находится в своем адресном пространстве, поэтому для взаимодействия между ними необходимо применять такие средства, как сообщения, или разделяемую память. Еще

больше затрат требуется на разделение процессов между несколькими процессорами или компьютерами, передачу информации между такими процессами, ожидание завершения и сбор результатов их работы. В заключение необходимо также упомянуть о том, что система UNIX не обладает необходимыми базовыми элементами для совместного использования таких ресурсов, как сетевые соединения и т. д. Применение подобной модели оправдано в тех случаях, когда преимущества одновременной работы процессов перекрывают расходы на их создание и управление.

Приведенные примеры показывают неудобства модели процессов и необходимость применения более производительных средств параллельной обработки. Мы можем теперь определить концепцию независимого вычислительного блока в качестве части общей задачи приложения. Такие блоки относительно мало взаимодействуют друг с другом и, следовательно, требуют малого количества действий по синхронизации. Приложение может содержать один или несколько блоков. Описываемый вычислительный блок называется *нитью*. Процесс в традиционной системе UNIX является однонитевым, то есть все действия в нем происходят последовательно.

Механизмы, обсуждаемые в этой главе, подчеркивают ограничения технологии процессов. Они, конечно, тоже имеют определенные недостатки, описание которых вы увидите в конце главы.

3.1.2. Нити и процессоры

Преимущества многонитевых систем хорошо заметны, если такие системы сочетать с многопроцессорными архитектурами. Если каждая нить будет функционировать на отдельном процессоре, то можно говорить о настоящей параллельности работы приложения. Если количество нитей превышает количество имеющихся процессоров, то такие нити должны быть мультиплексированы на эти процессоры¹. В идеальном случае, если приложение имеет n нитей, выполняющихся на n процессорах, то однонитевая версия той же программы на однопроцессорной системе будет тратить на выполнение задачи в n раз больше времени. На практике часть времени занимает создание, управление и синхронизация нитей, однако при использовании многопроцессорной системы эта величина стремится к идеальному соотношению.

На рис. 3.1 показан набор однонитевых процессов, выполняющихся на однопроцессорной машине. Операционная система создает некоторую иллюзию одновременности при помощи выделения каждому процессу некоторого определенного промежутка времени для работы (*квантование времени*), после чего происходит переключение на следующий процесс. В приведенном

¹ Это означает, что каждый процессор должен использоваться для обработки работающих нитей. Конечно, в один момент времени на одном процессоре может выполняться только одна нить.

примере первые три процесса являются задачами серверной части клиент-серверного приложения. Серверная программа при поступлении каждого нового запроса запускает еще один процесс. Все процессы обладают схожими адресными пространствами и совместно используют данные при помощи механизмов межпроцессного взаимодействия. Нижние два процесса созданы другим серверным приложением.

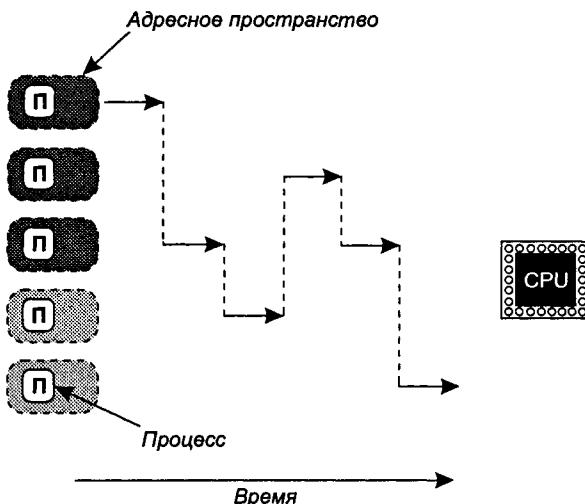


Рис. 3.1. Традиционная система UNIX: однонитевые процессы на однопроцессорной машине

На рис. 3.2 представлены два сервера, реализованные на нитях. Каждый сервер представляет собой единый процесс, имеющий несколько нитей, разделяющих между собой единое адресное пространство. Переключение контекста нитей в рамках одного процесса может обрабатываться либо ядром системы, либо на уровне прикладных библиотек, в зависимости от версии ОС. В обоих случаях в приведенном примере видны преимущества использования нитей. Исключение нескольких схожих адресных пространств приложений, порожденных для параллельной обработки чего-либо, сокращает нагрузку на подсистему памяти (даже при использовании современными системами такого метода разделения памяти, как *копирование при записи*, необходимо оперировать раздельными картами трансляции адресов для каждого процесса). Так как все нити приложения разделяют одно и то же адресное пространство, они могут использовать эффективные легковесные механизмы межнитевого взаимодействия и синхронизации.

Однако существуют и очевидные недостатки применения нитей. Однонитевому процессу не нужно заботиться о защите своих данных от других процессов. Многонитевые процессы должны следить за каждым объектом в собственном адресном пространстве. Если к объекту имеет доступ более

чем одна нить, то с целью предотвращения повреждения данных необходимо применять методы синхронизации.

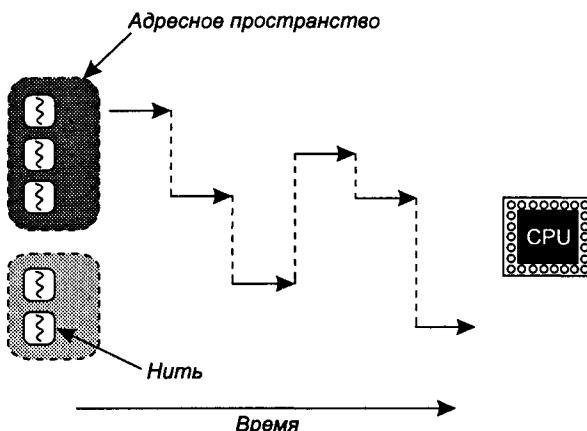


Рис. 3.2. Многонитевые процессы на однопроцессорной машине

На рис. 3.3 показаны два многонитевых процесса, выполняющиеся в многопроцессорной системе. Все нити каждого процесса совместно используют одно и то же адресное пространство, однако каждая из них выполняется на отдельном процессоре. Таким образом, все эти нити функционируют параллельно. Такой подход значительно увеличивает производительность системы, но, с другой стороны, существенно усложняет решение проблем синхронизации.

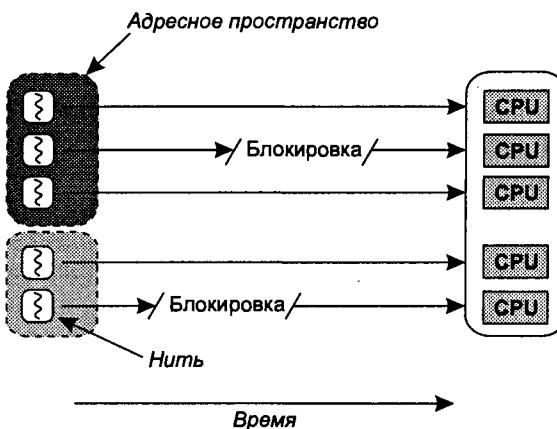


Рис. 3.3. Многонитевые процессы в многопроцессорной системе

Хотя оба средства (многонитевость и многопроцессорность) хорошо сочетаются друг с другом, их вполне можно использовать отдельно друг от друга. Многопроцессорные системы вполне пригодны для однонитевых приложе-

ний, так как в этом случае несколько процессов могут работать параллельно. Точно так же, имеются значительные преимущества в использовании многонитевых приложений на однопроцессорных системах. Если одна нить блокируется в ожидании окончания ввода-вывода или освобождения ресурса, другая нить может стать текущей, а приложение, таким образом, продолжит свое выполнение. Понятие нити более подходит для олицетворения встроенной одновременности выполнения в программе, нежели чем подгонки приложений под возможности многопроцессорных аппаратных архитектур.

3.1.3. Одновременность и параллельность

Для того чтобы понять основные типы элементов нитей, необходимо сначала определить разницу между терминами «одновременность» (concurrency) и «параллельность» (parallelism) [5]. **Параллельность** многопроцессорного приложения — это достигнутая им степень параллельного выполнения, которая в любом случае будет ограничиваться количеством процессоров, физически доступных приложению. Термин **одновременность** описывает максимальную степень параллельности, которую теоретически может достигать приложение с неограниченным количеством доступных процессоров. Эта степень зависит от того, как написано конкретное приложение, а также от количества нитей, которые могут выполняться одновременно, с доступными необходимыми им ресурсами.

Одновременность может поддерживаться на уровне либо системы, либо отдельного приложения. Ядро системы обеспечивает *системную одновременность* при помощи распознавания нескольких нитей внутри процесса (также называемых «горячими» нитями, *hot threads*) и планирования их выполнения независимо. Ядро разделяет такие нити между доступными процессорами. Приложение может воспользоваться преимуществами системной одновременности даже на однопроцессорных системах, поскольку если одна из его нитей будет блокирована в ожидании события или освобождения ресурса, ядро может назначить на выполнение другую нить.

Приложения могут обеспечивать *прикладную одновременность* через использование нитевых библиотек прикладного уровня. Такие нити, или *сопрограммы* (иногда называемые «холодными» нитями, *cold threads*), не распознаются ядром системы и должны управляться и планироваться самими приложениями. Этот подход не дает настоящей одновременности или параллельности, поскольку такие нити не могут в действительности выполняться одновременно, но он является более естественной моделью выполнения программы для приложений, требующих одновременности. Посредством использования неблокирующих системных вызовов приложение может одновременно поддерживать несколько взаимодействий в ходе своего выполнения. Использование прикладных нитей упрощает процесс выполнения программы, поскольку состояние таких взаимодействий считывается и размещается

в локальных переменных каждой нити (в стеке каждой нити), и при этом не используется глобальная таблица переменных.

Каждая модель одновременной обработки имеет определенные ограничения, заложенные в ее основе. Нити используются как средства организации работы и как средства использования многопроцессорных систем. Нити ядра позволяют производить параллельные вычисления на многопроцессорных системах, но они не подходят для структурирования приложений. Например, серверному приложению может потребоваться создание тысяч отдельных нитей, каждая из которых обрабатывает один клиентский запрос. Нити ядра потребляют такие важные ресурсы, как физическую память (так как многие реализации UNIX требуют постоянного хранения структур нитей в памяти); при этом, однако, они бесполезны для такого типа программ. С другой стороны, возможности нитей прикладного уровня полезны только для структурирования приложений, но они не позволяют выполнять код параллельно.

Во многих системах реализована модель *двойной одновременности*, которая совмещает в себе системную и прикладную одновременность. В этой модели нити процесса делятся на те, которые ядро распознает, и те, которые реализованы в библиотеках прикладного уровня и ядру не видимы. Прикладные нити позволяют синхронизировать одновременно исполняемые процедуры программы, не загружая систему дополнительными вызовами, и могут быть востребованы даже в системах, имеющих многонитевые ядра. Более того, уменьшение размера и функций ядра представляется хорошей дальнейшей стратегией развития UNIX, и разделение функциональности по поддержке нитей между ядром и нитевыми библиотеками как раз отвечает этим требованиям.

3.2. Основные типы нитей

Процесс представляет собой составную сущность, которую можно разделить на два основных компонента: набор нитей и набор ресурсов. *Нить* – это динамический объект, в процессе представленный отдельной точкой управления и выполняющий последовательность команд, отсчитываемую от этой точки¹. Ресурсы, включающие адресное пространство, открытые файлы, полномочия, квоты и т. д., используются всеми нитями процесса совместно. Каждая нить, кроме того, обладает собственными объектами, такими как указатель команд, стек или контекст регистров. В традиционных системах UNIX процесс имеет единственную выполняющуюся нить. Многонитевые системы расширяют эту концепцию, позволяя каждому процессу иметь более одной выполняющейся нити.

¹ Проще говоря, выполняемая параллельно в рамках процесса часть программы. – *Прим. ред.*

Централизация ресурсов процесса имеет и некоторые недостатки. Представьте серверное приложение, которое производит различные файловые операции от имени удаленных клиентов. Для проверки прав доступа к файлам серверу при обслуживании клиента необходимо производить его идентификацию. Для этого сервер запускается в системе с полномочиями суперпользователя и периодически вызывает `setuid`, `setgid` и `setgroups` для временного изменения своих полномочий в соответствии с клиентскими. Использование многонитевости для такого сервера, сделанное с целью повышения одновременности его функционирования, может привести к серьезным проблемам безопасности. Так как процесс обладает всего одним набором полномочий, в один момент времени он может работать только от одного клиента. Таким образом, чтобы поддержать необходимый уровень безопасности, серверу приходится выполнять все системные вызовы последовательно, то есть в режиме выполнения одной-единственной нити.

Существуют различные типы нитей, каждая из которых обладает различными свойствами и применением. В этом разделе мы опишем три важнейших их типа: *нити ядра, легковесные процессы и прикладные нити*.

3.2.1. Нити ядра

Нити ядра не требуют связи с каким-либо прикладным процессом. Они создаются и уничтожаются ядром и внутри ядра по мере необходимости и отвечают за выполнение определенных функций. Такие нити используют совместно доступные области кода и глобальные данные ядра, но обладают собственным стеком в ядре. Они могут независимо назначаться на выполнение и используют стандартные механизмы синхронизации ядра, такие как `sleep()` или `wakeup()`.

Нити ядра применяются для выполнения таких операций, как асинхронный ввод-вывод. Вместо поддержки каких-либо специальных механизмов ядро просто создает новую нить для обработки запросов для каждой такой операции. Запрос обрабатывается нитью синхронно, но для ядра представляется асинхронным событием. Нити ядра могут быть также использованы для обработки прерываний, подробнее об этом будет сказано в разделе 3.6.5.

Нити ядра являются малозатратными при создании и дальнейшем использовании. Единственными используемыми ими ресурсами являются стек ядра и область, в которой сохраняется контекст регистров на период приостановки работы нити (необходимо также поддерживать некую структуру данных, хранящую информацию для назначения ее на выполнение и синхронизацию). Переключение контекста между нитями ядра также происходит быстро, так что нет необходимости обновлять отображение памяти.

Применение нитей в ядре не является новым подходом. Такие системные процессы, как `pagedaemon`, в традиционных системах UNIX функционально

похожи на нити ядра. Процессы-демоны, наподобие nfsd (сервер Network File System), запускаются на прикладном уровне, однако после запуска полностью выполняются в ядре. После входа в режим ядра их прикладной контекст становится ненужным. Они также эквивалентны нитям ядра. Так как в традиционных системах UNIX отсутствовало понятие разделения применительно к представлению нитей ядра, такие процессы были вынуждены «таскать» за собою ненужный багаж, присущий традиционным процессам, в виде таких структур, как proc и user. Многонитевые системы позволили реализовать демоны намного проще в качестве нитей ядра.

3.2.2. Легковесные процессы

Легковесный процесс (или LWP, lightweight process) — это прикладная нить, поддерживаемая ядром. LWP является абстракцией высокого уровня, основанной на нитях ядра. Каждый процесс может иметь один или более LWP, любой из которых поддерживается отдельной нитью ядра (рис. 3.4). Легковесные процессы планируются на выполнение независимо от процесса, но совместно разделяют адресное пространство и другие ресурсы процесса. Они обладают возможностью производить системные вызовы и блокироваться в ожидании окончания ввода-вывода или освобождения ресурсов. На много-процессорных системах процесс в состоянии использовать преимущества настоящей параллельной работы, так как каждый LWP может выполняться на отдельном процессоре. Однако даже на однопроцессорных системах применение LWP дает существенные преимущества, поскольку при ожидании ресурсов или окончания ввода-вывода блокируется не весь процесс в целом, а только отдельные LWP.

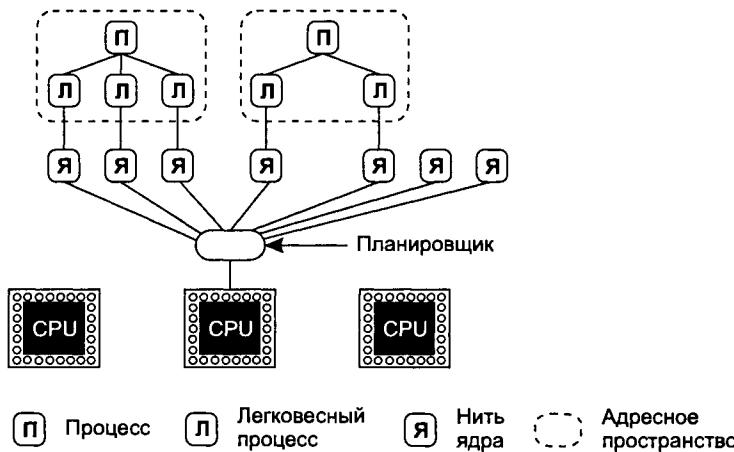


Рис. 3.4. Легковесные процессы

Кроме стека ядра и контекста регистров легковесному процессу необходимо поддерживать некоторое состояние задачи, что в первую очередь включает контекст регистров задачи, который необходимо сохранять перед тем, как обслуживание LWP будет прервано. Несмотря на то что каждый LWP ассоциирован с нитью ядра, некоторые нити ядра могут выполнять системные задачи и не относиться к какому-либо LWP.

Многонитевые процессы применяются прежде всего в тех случаях, когда каждая нить является полностью независимой и редко взаимодействует с другими нитями. Код приложения является полностью вытесняемым, при этом все LWP в процессе совместно используют одно и то же адресное пространство процесса. *Если доступ к каким-либо данным производится одновременно несколькими LWP, необходимо обеспечить некоторую синхронизацию доступа.* Для этого ядро системы предоставляет средства блокировки разделяемых переменных, при попытке прохождения в которые LWP будут блокированы. Такими средствами являются взаимные исключения (*mutual exclusion*, или *mutex*), атрибуты защиты, или защелки (*locks*), семафоры и условные переменные, которые будут более подробно рассмотрены в главе 7.

Рассказывая о легковесных процессах, необходимо также упомянуть и об их ограничениях. Многие операции над такими процессами, например создание, уничтожение и синхронизация, требуют применения системных вызовов. Однако системные вызовы по ряду причин являются весьма затратными операциями. Каждый вызов требует двух *переключений режима*: сначала из режима задачи в режим ядра и обратное переключение после завершения работы функции. При каждом переключении режима LWP пересекает *границу защиты* (*protection boundary*). Ядро должно скопировать все параметры системного вызова из пространства задачи в пространство ядра и привести их при необходимости в корректное состояние для защиты обработчиков этих вызовов от вредоносных или некорректно работающих процессов. При возврате из системного вызова ядро должно скопировать данные обратно в пространство задачи.

Когда легковесным процессам необходимо часто пользоваться разделяемыми данными, затраты на синхронизацию могут свести на нет увеличение производительности от их применения. Многие многопроцессорные системы поддерживают блокировку ресурсов, которая может быть активизирована из прикладного уровня при условии отсутствия удержания данного ресурса другой нитью [18]. Если нити необходим ресурс, который в данный момент недоступен, она может выполнить цикл *активного ожидания* (*busy-wait*) его освобождения без вмешательства ядра. Такие циклы подходят только для доступа к тем ресурсам, которые занимаются на короткие промежутки времени, в иных случаях необходимо блокирование нити. *Блокирование LWP требует вмешательства ядра* и вследствие этого является затратной процедурой.

Каждый легковесный процесс потребляет значительные ресурсы ядра, включая физическую память, отводимую под стек ядра. По этой причине система не может поддерживать большое количество LWP. Более того, так как система имеет единую реализацию LWP, такие процессы должны обладать достаточной универсальностью для поддержки наиболее типичных приложений. Таким образом, они могут содержать в себе такие свойства, которые окажутся ненужными большинству приложений. Применение LWP совершенно не подходит для приложений, использующих большое количество нитей или часто создающих и уничтожающих их. Наконец, легковесные процессы должны планироваться на выполнение ядром. Приложения, которым необходимо часто передавать управление от одной нити к другой, не могут делать это так легко, используя LWP. Применение легковесных процессов также способно повлечь некоторые неожиданные последствия, например, приложение имеет возможность монополизировать использование процессора, создав большое количество LWP.

Подводя итоги, скажем, что хотя ядро системы предоставляет механизмы создания, синхронизации и обработки легковесных процессов, за их правильное применение отвечает программист. Многим приложениям больше подходит реализация на прикладных нитях, описанных в следующем разделе.

ПРИМЕЧАНИЕ

Термин «легковесные процессы» (LWP) позаимствован из терминологии систем SVR4/MP и Solaris 2.x. Возможно, это не совсем точно, так как после появления четвертой версии SunOS термин LWP используется для обозначения прикладных нитей, описание которых вы можете увидеть в следующей главе. Однако в этой книге мы продолжим использование термина LWP для обозначения прикладных нитей, поддерживаемых ядром. В некоторых системах применяется термин «виртуальный процессор», который имеет то же значение, что и LWP¹.

3.2.3. Прикладные нити

Существует возможность поддержки нитей полностью на прикладном уровне, при этом ядру об их существовании ничего известно не будет. Указанная возможность реализована в таких библиотечных пакетах, как *C-threads* системы Mach и *pthreads* стандарта POSIX. Эти библиотеки содержат все необходимые функции для создания, синхронизации, планирования и обработки нитей без какой-либо специальной помощи ядра. Функционирование таких нитей не требует обслуживания ядром и вследствие этого является необычайно быстрым². На рис. 3.5, а представлена схема функционирования прикладных нитей.

¹ Иногда легковесные процессы называют еще тяжеловесными нитями. — Прим. ред.

² Большинство нитевых библиотек требуют вмешательства ядра для поддержки средств асинхронного ввода-вывода.

Рисунок 3.5, б иллюстрирует совместное использование прикладных нитей и легковесных процессов, вместе создающих мощную среду выполнения программы. Ядро распознает, планирует на выполнение и управляет LWP. Библиотеки прикладного уровня мультиплексируют прикладные нити в LWP и предоставляют возможности для межнитевого планирования, переключения контекста и синхронизации без участия ядра. Получается, что библиотека функционирует как миниатюрное ядро для тех нитей, которыми она управляет.

Реализация прикладных нитей возможна по причине того, что прикладной контекст нити может сохраняться и восстанавливаться без вмешательства ядра. Каждая прикладная нить обладает собственным стеком в адресном пространстве процесса, областью для хранения контекста регистров прикладного уровня и другой важной информации, такой как маски сигналов. Библиотека планирует выполнение и переключает контекст между прикладными нитями, сохраняя стек и состояние регистров текущей нити и загружая стек и состояние регистров следующей по расписанию нити.

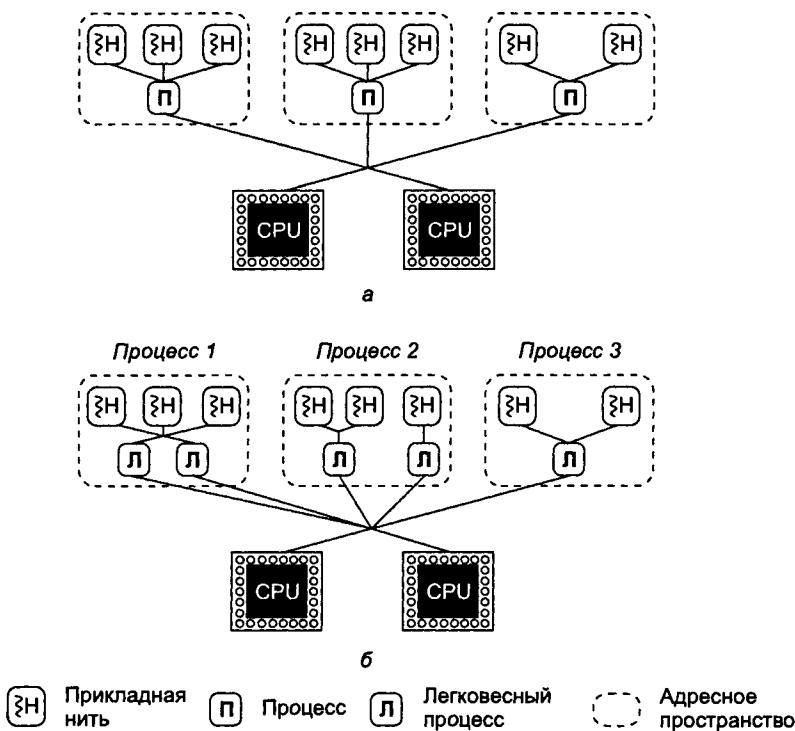


Рис. 3.5. Применение пользовательских нитей: а — прикладные нити обычных процессов; б — мультиплексирование прикладных нитей

Ядро системы отвечает за переключение процессов, так как только оно обладает привилегиями, необходимыми для изменения регистров управления памятью (регистров MMU). Поэтому прикладные нити не являются по-

настоящему планируемыми задачами, и ядро ничего не знает о них. Ядро просто планирует выполнение процесса (или LWP), содержащего в себе прикладные нити, который, в свою очередь, использует библиотечные функции для планирования выполнения своих нитей. Если процесс или LWP вытеснен кем-то, то той же участи подвергаются и все его нити. Точно так же, если нить делает блокирующий системный вызов, то он заблокирует и LWP этой нити. Соответственно, если процесс обладает всего одним LWP (или если прикладные нити реализованы на однонитевой системе), будут заблокированы все его нити.

Нитевая библиотека также включает в себя объекты синхронизации, обеспечивающие защиту совместно используемых структур данных. Такие объекты обычно содержат переменную блокировки (например, семафор) и очередь нитей, заблокированных по ней. Нити должны проверить блокировку перед тем, как начать доступ к защищенной структуре данных. Если объект уже заблокирован, библиотека приостановит выполнение нити, связав ее с очередью заблокированных нитей, и передаст управление следующей нити.

Современные системы UNIX поддерживают механизмы асинхронного ввода-вывода, позволяющие процессам выполнять ввод-вывод без блокирования. Например, в системе SVR4 для любого устройства, работа с которым может быть организована через STREAMS, предлагается ioctl-команда `I_O_SETSIG` (технология STREAMS описана в главе 17). Последующие¹ операции `write` или `read` с потоком просто встают в очередь операций и возвращают значения без блокирования. Когда ввод-вывод завершится, процесс будет проинформирован об этом при помощи сигнала `SIGPOLL`.

Асинхронный ввод-вывод является весьма полезным средством, так как позволяет процессу выполнять другие задачи во время ожидания завершения ввода-вывода. Но, с другой стороны, его реализация ведет к усложнению модели выполнения программы. Одним из удачных решений является организация возможности асинхронной работы на уровне операционной системы, предоставляющая для выполнения программы синхронную среду. Нитевая библиотека обеспечивает такой подход посредством предоставления синхронного интерфейса, использующего асинхронный внутренний механизм. Каждый запрос является синхронным по отношению к делающей его нити, которая блокируется в ожидании окончания операции ввода-вывода. Однако при этом процесс будет продолжать свое выполнение, так как библиотека внутри себя преобразует запрос в виде асинхронной операции и назначит на выполнение следующую по очереди прикладную нить. После завершения ввода-вывода библиотека снова поместит в расписание выполнения приостановленную нить.

Использование прикладных нитей имеет несколько преимуществ. Они предоставляют более естественный способ программирования многих приложе-

¹ То есть сделанные после вышеуказанного вызова `ioctl`. – Прим. ред.

ний, например таких, как оконные системы. Прикладные нити также обеспечивают синхронный подход к выполнению программы, поскольку все сложные асинхронные операции скрыты в недрах нитевых библиотек. Одно только это делает применение прикладных нитей весьма привлекательным, даже в системах, не обладающих поддержкой нитей на уровне ядра. Система может предлагать разработчику несколько различных нитевых библиотек, каждая из которых оптимизирована для различных классов приложений.

Важнейшим преимуществом прикладных нитей является их производительность. Эти нити являются легковесными и не потребляют ресурсов ядра, кроме связанных с LWP. В основе производительности работы прикладных нитей лежит реализация всей их функциональности на прикладном уровне без применения системных вызовов. Такой подход не требует дополнительной обработки системных прерываний и перемещения параметров и данных через границы защиты. Одним из важных параметров нити является *критический размер нити* [4], который показывает тот объем работы, который нить должна выполнить для того, чтобы оправдать свое существование в качестве отдельной сущности. Этот размер зависит от затрат на создание и использование нити. Для прикладных нитей критический размер составляет порядка нескольких сотен инструкций, количество которых может быть сокращено менее чем на сотню, в зависимости от используемого компилятора. Прикладные нити требуют значительно меньшего количества времени для создания, уничтожения и синхронизации. Таблица 3.1 показывает длительность различных операций, производимых процессами, LWP и прикладными нитями на машине SPARCstation 2 [21].

Таблица 3.1. Длительность операций, производимых прикладными нитями, LWP и процессами (на SPARCstation 2)

	Создание, мкс	Синхронизация с использованием семафоров, мкс
Пользовательская нить	52	66
LWP	350	390
Процесс	1700	200

Однако прикладные нити обладают и рядом ограничений, большинство из которых является следствием полного разделения информации между ядром и нитевой библиотекой. Так как ядро системы ничего не знает о прикладных нитях, оно не может использовать свои механизмы для их защиты друг от друга. Каждый процесс имеет свое адресное пространство, защищаемое ядром от несанкционированного доступа других процессов. Прикладные нити лишены такой возможности, так как функционируют в общем адресном пространстве процесса. Нитевая библиотека должна обеспечивать средства синхронизации, необходимые для совместной работы нитей.

Использование модели разделенного планирования может стать причиной возникновения множества других проблем. Нитевая библиотека занимается планированием выполнения прикладных нитей, ядро планирует выполнение процессов или легковесных процессов, в которых эти нити функционируют, и в итоге никто из них не знает о действиях друг друга. Например, ядро может вытеснить LWP, чья прикладная нить находится в области, защищаемой *циклической блокировкой* (*spin lock*), удерживая эту блокировку активной. Если иная прикладная нить иного LWP попытается снять эту блокировку, то она перейдет в цикл активного ожидания до тех пор, пока удерживающая циклическую блокировку нить не получит возможности работать снова. С другой стороны, так как ядро системы не знает относительных приоритетов прикладных нитей, оно вполне может вытеснить один легковесный процесс с выполняемой внутри нитью высокого приоритета, поставив на выполнение другой LWP, нить которого имеет более низкий приоритет.

Механизмы синхронизации, организованные на прикладном уровне, могут в некоторых случаях работать некорректно. Большинство приложений создаются с предположением, что все работающие нити периодически попадают в очередь на выполнение. Это действительно так, если каждая нить находится в отдельном LWP, но предположение может не выполняться в тех случаях, когда некоторое количество прикладных нитей мультиплексируется в меньшем количестве LWP. Так как LWP может блокироваться в ядре, когда его прикладная нить делает блокирующий системный вызов, такой LWP процесса умеет останавливать свою работу даже тогда, когда остаются работающие нити, а в системе — доступные процессоры. Разрешить эту проблему можно при помощи использования механизмов асинхронного ввода-вывода.

И наконец, следует упомянуть о том, что без явной поддержки на уровне ядра прикладные нити в состоянии увеличить одновременность выполнения, но не могут увеличить параллельность. Даже на многопроцессорных системах прикладные нити, разделяющие между собой один легковесный процесс, не могут выполняться параллельно.

В этом разделе были описаны три основных, наиболее используемых типа нитей. Нити ядра являются объектами самого нижнего уровня и невидимы для приложений. Легковесные процессы — это нити, видимые на прикладном уровне, но распознаваемые при этом ядром и опирающиеся на нити ядра. Прикладные нити представляют собой объекты высокого уровня, не видимые ядром. Они могут использовать легковесные процессы (если такие поддерживаются системой) или реализовываться в обычных процессах UNIX без специальной поддержки ядром. И прикладные нити, и LWP имеют некоторые существенные недостатки, ограничивающие их область применения. В разделе 3.5 вы увидите описание новой фундаментальной структуры, основанной на *активациях планировщика*, которая показывает многие из этих проблем. Однако сначала мы затронем некоторые вопросы, касающиеся прикладных нитей и LWP, подробнее.

3.3. Легковесные процессы: основные проблемы

Существует несколько различных факторов, влияющих на устройство легковесных процессов. В первую очередь, необходимо сохранять семантику, принятую в системах UNIX, по крайней мере, для однонитевых вариантов. Это означает, что процесс, содержащий единственный LWP, должен функционировать как традиционный процесс UNIX. (Стоит снова упомянуть, что под LWP мы понимаем прикладные нити, поддерживаемые ядром, а не легковесные процессы системы SunOS 4.0, которые являются объектами полностью прикладного уровня.)

Существуют различные области, где концепцию UNIX нельзя легко перенести на многонитевую систему. Следующие разделы посвящены описанию возникающих проблем и способам их решения.

3.3.1. Семантика вызова fork

Системные вызовы в многонитевых средах принимают несколько необычные значения. Многие вызовы, имеющие отношение к созданию процессов, манипуляциям над адресным пространством или действиям над ресурсами процесса (такими как открытые файлы), должны быть переписаны заново. Есть две важные рекомендации. Во-первых, системный вызов должен придерживаться традиционной семантики системы UNIX в случае применения одной нити. Во-вторых, при использовании многонитевых процессов системный вызов обязан вести себя приемлемым образом, не уходя далеко от его семантики на однонитевых системах. Помня об этих рекомендациях, давайте проведем анализ некоторых важнейших системных вызовов, которые необходимо изменить для многонитевых реализаций.

В традиционных системах UNIX вызов `fork` создает процесс-потомок, являющийся точной копией его родителя. Их единственное различие заключается в необходимости отличия потомка от его родителя. Семантика вызова `fork` полностью отвечает требованиям однонитевого процесса. В случае многонитевых процессов имеется дополнительный параметр, позволяющий дублировать либо все LWP родителя, либо только тот из них, что вызвал функцию `fork`.

Представьте, что вызову `fork` необходимо произвести копирование вызывающего его LWP в новый процесс, и только его. Такой вариант является более эффективным. Он также весьма удобен в тех случаях, если процесс-потомок после своего появления запустит в себе новую программу при помощи `exec`. Однако подобное взаимодействие имеет и некоторые проблемы [20]. LWP часто используются для поддержки нитевых библиотек прикладного уровня. Такие библиотеки представляют каждую прикладную нить в виде структуры данных в пространстве процесса. Если вызов `fork` продублирует только вызывающий его LWP, то новый процесс будет содержать прикладные нити, не

относящиеся ни к одному своему LWP. Более того, процесс-потомок не должен пытаться снять блокировку, удерживаемую нитями, не существующими в нем, так как это может привести к состоянию клинча. В реальности иногда сложно избежать конфликта, так как библиотеки часто создают скрытые нити, о которых ничего не знает разработчик приложения.

Представим иной случай, когда вызов `fork` дублирует все LWP родительского процесса. Такой вариант наиболее предпочтителен в случае, когда необходимо сделать именно копию всего процесса, нежели выполнить новую программу. Однако и в этом случае возникают определенные проблемы. Какой-либо LWP родителя может быть заблокирован системным вызовом, и это состояние не будет определено в потомке. Возможность обойти данную ситуацию заключается в том, чтобы заставить системный вызов вернуть код ошибки `EINTR` (системный вызов прерван), что позволило бы LWP сделать его заново по мере необходимости. LWP также может иметь открытые сетевые соединения. Закрытие соединения в потомке может стать причиной отправки на удаленный узел незапланированных сообщений. Некоторые LWP умеют обрабатывать внешние общие структуры данных, которые могут быть повреждены при клонировании такого LWP вызовом `fork`.

Ни одно из решений не в силах правильно обработать все возможные ситуации. Во многих системах определенный компромисс достигнут при помощи двух различных вариантов `fork`, один из которых применяется для дублирования процесса полностью, а второй — копирует только одну нить. Для последнего варианта в таких системах определен набор безопасных функций, которые могут быть вызваны потомком перед выполнением `exec`. Альтернативным вариантом является разрешение на регистрацию процессом одного или нескольких обработчиков `fork`, которые являются функциями, запускаемыми в родителе или потомке до или после вызова `fork` в зависимости от параметров, указанных при их регистрации.

3.3.2. Другие системные вызовы

Для корректной работы в многонитевых системах необходимо пересматривать не только `fork`, но и многие другие системные вызовы. Все LWP процесса разделяют между собой общий набор файловых дескрипторов. Это может стать причиной возникновения конфликта в случае, если один из LWP закроет файл, который в текущий момент времени считывается или в него ведется запись другим LWP. Файловый *указатель на смещение*¹ также используется совместно всеми нитями через дескриптор, поэтому применение функции `lseek` одним из LWP повлияет на работу с этим файлом всех остальных нитей. Эта проблема проиллюстрирована на рис. 3.6. Легковесному процессу `L1`

¹ Используется при считывании из файла или записи в файл для запоминания той позиции, откуда необходимо начинать чтение или запись при следующем вызове `read` или `write`. — Прим. ред.

необходимо прочесть данные из файла, начиная со смещения `off1`, для чего он вызывает функцию `lseek`, а затем — `read`. Между двумя означенными вызовами другой процесс `L2` применяет `lseek` в отношении того же файла, указывая при этом другое смещение. Такая ситуация приведет к тому, что `L1` считает не те данные. Приложение должно решать подобные проблемы самостоятельно, используя какой-либо протокол блокирования файлов. Альтернативным решением являются механизмы ядра системы, которые позволяют производить произвольный ввод-вывод атомарно (см. подробнее в разделе 3.6.6).

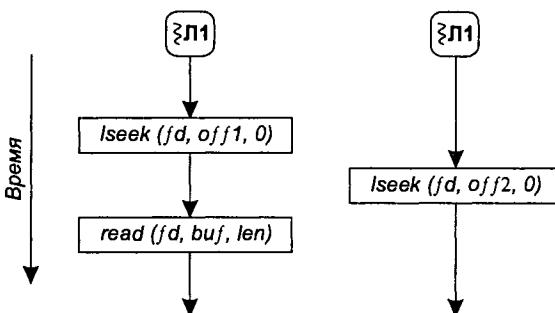


Рис. 3.6. Проблемы, возникающие при одновременном доступе к файлу

Каждый процесс имеет один текущий рабочий каталог и использует одну структуру пользовательских полномочий. Так как полномочия могут изменяться в любой момент времени, ядро должно использовать их атомарно и только однажды перед системным вызовом.

Все LWP процесса используют совместно одно и то же адресное пространство и могут манипулировать им одновременно при помощи различных системных вызовов, таких как `mmap` или `brk`. Такие вызовы должны быть безопасными (в отношении нитей) во избежание повреждения адресного пространства процесса. Программистам следует внимательно относиться к последовательности таких операций, так как в противном случае результат может быть непредсказуем.

3.3.3. Доставка и обработка сигналов

В системах UNIX доставка и обработка сигналов производится на уровне процесса. В многонитевых системах необходимо определять, какой из LWP процесса будет заниматься обработкой сигналов. При использовании прикладных нитей имеется аналогичная проблема: после того, как ядро передаст сигнал в LWP, нитевая библиотека должна определить, в какую нить его направить. Существует несколько вариантов решения данной проблемы:

- ◆ пересылка сигналов каждой нити;
- ◆ объявление одной из нитей процесса «главной», после чего все сигналы передаются только этой нити;

- ◆ отправка сигналов в любую произвольно выбранную нить;
- ◆ использование эвристических методов для определения, какой из нитей необходимо отправить данный сигнал;
- ◆ создание новой нити для обработки каждого сигнала.

Первый метод является очень затратным, и, более того, он несовместим с большинством обычных ситуаций, в которых используются сигналы. Однако в некоторых случаях он весьма удобен. Например, если пользователь нажимает комбинацию **Ctrl+Z** на терминале, он может желать приостановки всех нитей процесса. Второй метод приводит к асимметричной обработке нитей, что несовместимо с современным подходом к нитям и с симметричными многопроцессными системами, часто ассоциируемыми с многонитевыми ядрами. Последнее решение, приведенное в списке, подходит только для определенных ситуаций.

Выбор между двумя оставшимися методами зависит от природы выработанного сигнала. Некоторые сигналы, например **SIGSEGV** (ошибка сегментации) и **SIGILL** (непредусмотренное исключение), создаются вследствие действий нити. Наиболее удобным решением в данном случае представляется доставка такого сигнала той нити, которая и стала причиной его возникновения. Другие сигналы, такие как **SIGSTP** (сигнал остановки, вырабатываемый терминалом) или **SIGINT** (сигнал прерывания), создаются при возникновении внешних событий и не могут быть как-то ассоциированы с конкретной нитью процесса.

Еще одним аспектом, о котором стоит упомянуть, является применяемый метод обработки и маскирования сигналов. Должны ли все нити использовать общий набор обработчиков сигналов или каждая будет определять свой собственный? Хотя последний вариант является более гибким и универсальным, он привносит каждой нити дополнительные затраты, что противоречит главной цели применения многонитевых процессов. Такие же проблемы возникают при маскировании сигналов, поскольку обычно маскирование происходит с целью защиты важных участков кода. Следовательно, лучшим вариантом представляется разрешение каждой нити на указание собственной маски сигналов. Перегрузки, возникающие при применении таких масок, менее значительны и поэтому более приемлемы.

3.3.4. Видимость

Важно определить, в какой степени LWP будет видимым вне процесса. Безспорно, ядро системы знает о существующих LWP и планирует их выполнение независимо. Однако большинство реализаций систем не позволяет процессам обладать информацией о конкретных LWP других процессов, а также взаимодействовать с ними.

Вместе с тем внутри процесса необходимо предоставлять какому-либо LWP возможность получать информацию о существовании остальных LWP в рамках своего процесса. Многие системы предлагают для этой цели специаль-

ные системные вызовы, которые позволяют одному LWP отправлять сигналы другому LWP, принадлежащему тому же самому процессу.

3.3.5. Рост стека

Если какой-нибудь из процессов в системе UNIX переполняет свой стек, в результате этого возникает ошибка нарушения сегментации. Ядро распознает появление таких ситуаций в сегменте стека и автоматически увеличивает размер стека¹, не посыпая никаких сигналов процессу.

Многонитевые процессы обладают несколькими стеками, по одному на каждую прикладную нить. Эти нити размещаются на прикладном уровне при помощи нитевых библиотек. Если ядро системы попытается расширить стек, то возникнет определенная проблема, так как такая операция может привести к конфликту с обработчиком стека в нитевой библиотеке прикладного уровня.

Даже в многонитевой системе ядро не имеет никакого представления о стеках прикладных нитей². Такие стеки не всегда являются специальными областями и могут быть взяты прямо из кучи. Обычно если нить указывает размер необходимого ей стека, то библиотека может защитить стек от переполнения при помощи размещения страницы памяти, защищенной от записи, сразу после конца стека. Такой подход приводит к ошибке защиты при возникновении переполнения стека³, и в этом случае ядро системы посыпает сигнал SIGSERV соответствующей нити. После этого нить может либо увеличить размер стека, либо решить проблему иным путем⁴.

3.4. Нитевые библиотеки прикладного уровня

При разработке пакетов функций для работы с прикладными нитями необходимо найти ответы на следующие два важных вопроса: какого рода программный интерфейс библиотеки будет представлен программисту и как такой пакет

¹ До определенного установленного лимита. В системе SVR4 размер стека ограничивается значением переменной RLIMIT_NOFILE. Эта переменная содержит *жесткие* и *мягкие границы*. Для получения значения границ применяется системный вызов getrlimit. При помощи вызова setrlimit можно уменьшить жесткий лимит, а также уменьшить или увеличить мягкую границу до значения, не превышающего жесткий лимит.

² Некоторые многонитевые системы, например SVR4.2/MP, имеют средства, позволяющие автоматически увеличивать стек прикладной нити.

³ Помещаемые в стек данные попадут в область стоящей за стеком страницы. — Прим. ред.

⁴ Конечно, такой сигнал должен обрабатываться в специальном стеке, раз уже обычный стек не имеет свободного места для функционирования обработчика сигнала. В современных системах UNIX существуют способы задания приложением альтернативного стека для обработки сигналов (см. подробнее в разделе 4.5).

может быть реализован при помощи средств, предлагаемых конкретной операционной системой. Существуют различные варианты нитевых библиотек, например Chorus [2], Topaz [23] и *C-threads* ОС Mach [7]. Группой Р1003.3а IEEE стандартов POSIX было разработано несколько предварительных вариантов пакета функций для работы с нитями под названием *pthreads* [13]. Современные системы UNIX должны поддерживать *pthreads* для совместимости с этими стандартами (см. подробнее в разделе 3.8.3).

3.4.1. Программный интерфейс

Интерфейс, обеспечиваемый пакетами функций для работы с нитями, должен обладать несколькими важными средствами. Он должен поддерживать большой набор различных операций над нитями, таких как:

- ◆ создание и уничтожение нитей;
- ◆ перевод нитей в режим ожидания и их восстановление в работоспособное состояние;
- ◆ назначение приоритетов для отдельных нитей;
- ◆ планирование выполнения нитей и переключение контекста;
- ◆ синхронизацию действий при помощи таких средств, как семафоры или взаимные исключения;
- ◆ обмен сообщениями между нитями.

Пакеты функций для работы с нитями должны по возможности минимизировать участие ядра, так как переключение между режимом задачи и режимом ядра может быть весьма затратной процедурой. Поэтому нитевые библиотеки предоставляют столько возможностей, сколько могут. Обычно ядро системы не обладает информацией о прикладных нитях, однако нитевые библиотеки могут использовать системные вызовы для реализации ряда своих возможностей. Из этого вытекает ряд важных моментов. Например, приоритет нити не имеет никакого отношения к приоритету процесса или LWP этой нити, назначенному в расписании ядра. Приоритет нити имеет смысл только внутри своего процесса и используется *планировщиком нитей* для выбора одной из них на выполнение.

3.4.2. Реализация нитевых библиотек

Реализация конкретной библиотеки зависит от средств многонитевости, предоставляемых ядром системы. Многие существующие пакеты функций для работы с нитями созданы для традиционных вариантов UNIX, вообще не имеющих специальной поддержки нитей. В таких системах нитевые библиотеки функционируют как миниатюрные ядра, обрабатывая самостоятельно всю информацию о состоянии каждой нити и производя все операции над

ними на прикладном уровне. Хотя этот подход обеспечивает довольно эффективную последовательную обработку, он дает и некоторую степень одновременности при использовании средств асинхронного ввода-вывода системы.

Во многих современных системах ядро поддерживает многонитевые процессы через LWP. В таком случае библиотеки прикладных нитей могут быть реализованы различными способами:

- ◆ Каждой нити назначается свой легковесный процесс. Вариант наиболее прост для реализации, но требует большого количества ресурсов ядра и дает лишь малое увеличение производительности. Он также требует участия ядра во всех операциях синхронизации и планирования выполнения.
- ◆ Прикладные нити мультиплексируются в меньший набор легковесных процессов. Этот вариант более эффективен, так как требует меньшего количества ресурсов ядра. Он наиболее подходит в тех случаях, когда все нити процесса примерно эквивалентны друг другу. Однако здесь нет простого способа, гарантирующего предоставление ресурсов определенной нити.
- ◆ Связанные и несвязанные нити смешиваются в одном процессе. Такой вариант дает возможность приложению полностью использовать параллельность и одновременность, предоставляемые системой. Он также позволяет обрабатывать преимущественно связанные нити посредством повышения уровня приоритетов LWP, их содержащего, или вообще давая им LWP эксклюзивное право на пользование процессором. О связанных и несвязанных нитях подробнее в разделе 3.6.3.

Нитевая библиотека содержит алгоритм планирования, по которому выбирается очередная нить для выполнения. Он обрабатывает приоритеты и состояния каждой нити, не зависящие от состояния или приоритета LWP, внутри которого находятся эти нити. На рис. 3.7 показан пример шести прикладных нитей, мультиплексированных в два LWP. Библиотека планирует выполнение по одной нити в каждом LWP. Такие нити (в данном случае H5 и H6) находятся в *выполняющемся* состоянии, даже если сам легковесный процесс, их содержащий, был заблокирован системным вызовом или был вытеснен и находится в ожидании своей очереди на выполнение.

Нить (такая как H1 или H2 на рис. 3.7) изменяет свое состояние на *блокированное* в том случае, если попытается использовать объект синхронизации, заблокированный другой нитью. После освобождения объекта библиотека разблокирует нить и переведет ее в очередь на выполнение. Нити H3 и H4 уже находятся в состоянии *готовности* и ожидают своей очереди на выполнение. Планировщик нитей выбирает на выполнение нить из этой очереди, исходя из приоритета и связанного с ней LWP. *Этот механизм очень схож с алгоритмами планирования и ожидания ресурсов ядра.* Как уже говорилось ранее, нитевая библиотека функционирует как миниатюрное ядро для нитей, которыми управляет.

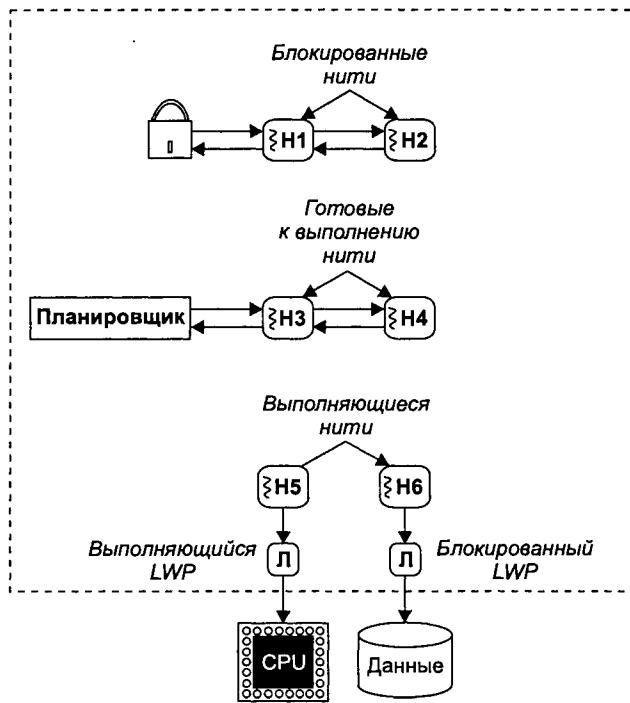


Рис. 3.7. Состояния прикладных нитей

Более подробно о реализациях прикладных нитей можно прочесть в [9], [18] и [20].

3.5. Активации планировщика

В двух предыдущих разделах описывались преимущества и недостатки легковесных процессов и прикладных нитей. Ни одна из приведенных моделей не является полностью удовлетворительной. Разработчики хотят сочетать производительность прикладных нитей и их гибкость. Однако такие нити имеют меньшую функциональность, чем легковесные процессы, поскольку не имеют интеграции с ядром системы. В [1] описывается совершенно новая архитектура нитей, в которой используются преимущества обеих моделей. Эта архитектура получила признание среди специалистов по операционным системам и появилась в коммерческих реализациях механизмов нитей от таких производителей, как SGI¹ [4].

Основной принцип новой модели заключался в тесной интеграции прикладных нитей и ядра. Ядро отвечает за выделение процессора, а нитевая

¹ Silicon Graphics. — Прим. ред.

библиотека — за планирование. Библиотека информирует ядро системы о событиях, которые требуют выделения процессора. Она может запросить либо дополнительные процессоры, либо оставить свой текущий. Ядро полностью контролирует использование процессоров и может периодически «забирать» процессоры, предоставляя их другим процессам.

Библиотека, получив некоторые процессоры, всецело контролирует нити и планирует их выполнение на этих процессорах. Если ядро забирает один из процессоров, то оно проинформирует об этом библиотеку, которая переназначит нити соответствующим образом. Если нить блокируется в ядре, процесс не потеряет занимаемый им процессор. Ядро сообщит об этом факте библиотеке, которая немедленно поставит на выполнение другую прикладную нить на тот же самый процессор.

Для реализации описываемой модели необходимо ввести два новых понятия — *обратного вызова* (*upcall*) и *активации планировщика* (*scheduler activation*). Обратный вызов — это вызов, сделанный ядром системы в нитевую библиотеку. Активация планировщика — контекст выполнения, который может быть использован для выполнения прикладной нити. Он подобен LWP и имеет собственный стек в пространстве процесса и стек ядра. Когда ядро осуществляет обратный вызов, оно передает в библиотеку активацию, которая будет использована для обработки события, выполнения новой нити или для какого-либо системного вызова. Ядро системы не квантует время активаций на процессоре. В любой момент времени процесс обладает одной активацией для каждого процессора из назначенных ему.

Отличительной особенностью модели активаций планировщика является ее обработка блокирующих операций. Когда прикладная нить блокируется в ядре, последнее производит новую активацию и сообщает об этом библиотеке посредством обратного вызова. Библиотека сохраняет состояние нити от предыдущей активации и информирует ядро системы о том, что эту активацию можно использовать заново. Затем библиотека переходит к выполнению следующей нити с новой активацией. Когда блокирующая операция завершается, ядро производит еще один обратный вызов, указывающий библиотеке на произошедшее событие. Такой вызов требует новой активации. Ядро может назначить новый процессор для выполнения этой активации или вытеснить одну из текущих активаций данного процесса. В последнем случае вызов оповестит библиотеку о двух событиях: во-первых, о том, что можно продолжить выполнение изначальной нити (которая была блокирована), и во-вторых, о том, что нить, выполняющаяся ранее на этом процессоре, была вытеснена. Библиотека поместит обе нити в список нитей, готовых к выполнению, и затем решит, какая из них будет выполняться первой.

Технология активаций планировщика обладает рядом преимуществ. Прежде всего, активации происходят очень быстро, так как большинство операций не требует участия ядра. В работе [1] приведены измерения, которые показа-

ли, что пакеты функций работы с нитями, основанные на активациях, функционируют производительнее, чем другие типы нитевых библиотек. Так как ядро системы информирует библиотеку о событиях блокирования или вытеснения, библиотека может самостоятельно принимать решения по планированию и синхронизации, а также избегать взаимных блокировок и некорректной семантики. Например, если ядро заберет процессор, который в текущий момент занят нитью, находящейся в состоянии циклической блокировки, библиотека может переключить выполнение этой нити на другой процессор и выполнять ее там до тех пор, пока ожидаемый нитью ресурс не снимет блокировку.

Оставшаяся часть главы посвящена описанию реализаций нитей в системах Solaris, SVR4, Mach и Digital UNIX.

3.6. Многонитевость в Solaris и SVR4

Корпорация Sun Microsystems начала поддерживать нити на уровне ядра в системе Solais версий 2.x¹. Компания UNIX System Laboratories для своей ОС SVR4.2/MP приняла технологию нитей системы Solaris. Разработанная архитектура предлагает большое количество базовых элементов как для уровня ядра, так и на прикладном уровне, что позволяет создавать приложения с широкими возможностями.

Система Solaris поддерживает нити ядра, легковесные процессы и прикладные нити. Процесс может обладать несколькими сотнями нитей, сохранив параллельность выполнения программы. Нитевая библиотека проведет мультиплексирование этих нитей в меньшее количество легковесных процессов. Существует возможность управления количеством LWP для оптимизации использования ресурсов системы. Также существует возможность группирования некоторых нитей в отдельные легковесные процессы (см. подробнее в разделе 3.6.3).

3.6.1. Нити ядра

Нить ядра в системе Solaris — это основной легковесный объект, который может независимо планироваться и отправляться на выполнение одному из процессоров системы. Такой объект не нуждается в ассоциации с каким-либо процессом, он может быть создан, запущен и уничтожен ядром при помощи специальных функций. В результате ядру системы не нужно переотображать виртуальное адресное пространство при переключении нитей ядра [16]. Следовательно, переключение контекста нити ядра менее затратно, чем переключение контекста процесса.

¹ Нити ядра были представлены в Solaris 2.0, а доступный разработчикам интерфейс в Solaris 2.2.

Нить ядра требует минимального количества ресурсов в виде небольшой структуры данных и стека. Структура данных нити ядра содержит следующую информацию:

- ◆ сохраненная копия регистров ядра;
- ◆ приоритет и информация, связанная с расписанием;
- ◆ указатели на местонахождение нити в очереди планировщика или, если выполнение нити блокировано, в очереди *ожидания ресурсов*;
- ◆ указатель на стек;
- ◆ указатели на связанные с нитью структуры *lwp* и *proc* (равны *NULL*, если нить выполняется не в рамках LWP);
- ◆ указатели на очередь всех нитей процесса и очередь всех нитей в системе;
- ◆ информация об LWP, связанном с нитью, если таковой существует (см. раздел 3.6.2).

Ядро системы Solaris организовано как набор внутренних нитей. Некоторые из них выполняют легковесные процессы, другие отвечают за внутренние функции ядра. Нити ядра являются полностью вытесняемыми. Они могут относиться к любому классу расписания задач системы (см. раздел 5.5), в том числе и к классу реального времени. Нити используют специализированные версии базовых элементов синхронизации (семафоров, условий и т. д.), защищающих от *инверсии приоритетов*, то есть ситуации, при которой нить с более низким приоритетом удерживает ресурс, необходимый нити, имеющей более высокий приоритет, что замедляет ее выполнение. Такие средства системы будут подробно описаны в разделе 5.6.

Нити ядра используются при проведении асинхронных операций, таких как отложенная запись на диск, выполнение обслуживающих процедур STREAMS и *отложенные вызовы* (callouts, см. раздел 5.2.1). Это позволяет ядру системы задавать приоритет каждой из подобных операций (посредством установки приоритета нити) и исходя из этих приоритетов планировать их выполнение. Нити также используются для поддержки легковесных процессов. Для этого каждый процесс LWP «прикреплен» к ядру нитью ядра (однако не все нити ядра имеют свой LWP).

3.6.2. Реализация легковесных процессов

Легковесные процессы обеспечивают многонитевое выполнение внутри одного процесса. Планирование выполнения LWP происходит независимо, и такие процессы могут выполняться параллельно на многопроцессорных системах. Каждый LWP связан со своей собственной нитью ядра, такая связь не прерывается на протяжении всего жизненного цикла процесса.

Традиционные структуры `proc` и `user` недостаточны для представления многонитевых процессов. Данные в этих структурах должны быть разделены на информацию, касающуюся каждого процесса и каждого LWP. В системе Solaris структура `proc` используется для хранения всех данных каждого процесса, в том числе и процессо-зависимой части традиционной области `i`.

В добавок к структурам, описывающим процесс в ядре, появляется новая структура — `lwp`, которая хранит информацию о каждой LWP-составляющей контекста процесса. Структура `lwp` содержит следующую информацию:

- ◆ сохраненные значения регистров прикладного уровня (когда LWP не выполняется);
- ◆ аргументы системного вызова, результаты работы и код ошибки;
- ◆ информацию об обработке сигналов;
- ◆ данные об использовании ресурсов и данные профиля процесса;
- ◆ время подачи сигналов тревоги;
- ◆ значения времени работы в режиме задачи и использования процессора;
- ◆ указатель на нить ядра;
- ◆ указатель на структуру `proc`.

Структура `lwp` может быть выгружена вместе с легковесным процессом, поэтому невыгружаемая по определению информация, например маски некоторых сигналов, хранится в структуре нити, связанной с LWP. В реализации системы под архитектуру Sparc для хранения указателя на текущую нить используется глобальный регистр `%g7`, что дает возможность быстрого доступа к текущему LWP и процессу.

Для легковесных процессов (и нитей ядра) доступны основные средства синхронизации, такие как взаимные исключения, условные переменные, семафоры и защелки чтения-записи. Эти средства будут более подробно описаны в разделе 7. Каждое из приведенных средств может определять различные варианты поведения нитей. Например, если нить попытается получить доступ к объекту `mutex`, удерживаемому другой нитью, то она либо перейдет в цикл активного ожидания освобождения объекта `mutex`, либо блокируется до момента этого события. Когда объект синхронизации инициализируется, источник вызова этого объекта должен указать, какое поведение для него ожидается.

Все LWP используют общий набор обработчиков сигналов. Однако каждый LWP может обладать собственной маской сигналов, решая самостоятельно, какие из полученных сигналов нужно игнорировать или блокировать. Любой LWP также имеет возможность определить свой собственный альтернативный стек для обработки сигналов. Все сигналы делятся на две категории: ловушки и прерывания. *Ловушки* представляют собой сигналы, вырабатываемые в ходе действий самого LWP (например, `SIGSEGV`, `SIGFPE` и `SIGSYS`). Такие сигналы всегда передаются легковесному процессу, действия которого при-

вели к их появлению. Сигналы прерываний (такие как SIGSTOP и SIGINT) могут быть доставлены любому LWP, который не маскирует эти сигналы.

Легковесные процессы не имеют глобального пространства имен и вследствие этого невидимы для других процессов. *Процесс не может направить сигнал напрямую определенному LWP, принадлежащему другому процессу, а также знать, какой LWP послал ему сообщение.*

3.6.3. Прикладные нити

Прикладные нити реализованы в системе при помощи *нитевой библиотеки*. Они могут создаваться, уничтожаться и обрабатываться без участия ядра. Библиотека также предоставляет средства синхронизации и планирования нитей. Это позволяет процессу использовать большое количество нитей, не потребляя при этом ресурсы ядра и не загружая систему лишними вызовами. Хотя в системе Solaris прикладные нити реализованы на основе LWP, нитевая библиотека скрывает эти детали, позволяя разработчикам приложений иметь дело только с прикладными нитями.

По умолчанию библиотека создает для каждого процесса набор LWP и мультиплексирует в него все прикладные нити. Размер такого набора зависит от количества процессоров и прикладных нитей. Разработчик приложения волен изменить начальные установки и самостоятельно указать количество создаваемых легковесных процессов. Он также вправе потребовать от системы назначения LWP какой-то определенной нити. Таким образом, процесс может обладать прикладными нитями двух типов: *свободными* (связанными с единственным LWP) нитями и *несвязанными*, разделяющими общий набор легковесных процессов (рис. 3.8).

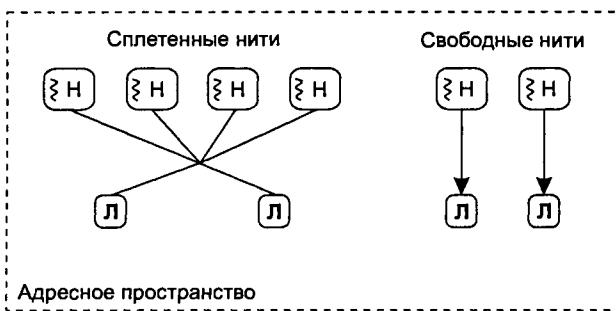


Рис. 3.8. Схема процесса в системе Solaris 2.x

Мультиплексирование большого количества нитей в небольшое число LWP дает возможность одновременной работы при достаточно низких затратах. Например, в оконной системе каждый объект (диалоговое окно, меню, пиктограмма и т. д.) может быть представлен в виде нити. В один момент времени обычно активно лишь некоторое количество окон, следовательно,

только эти нити должны поддерживаться LWP. Количество легковесных процессов определяет максимальную параллельность, которую может достигать приложение (как минимум равную количеству имеющихся процессоров). Оно также ограничивает количество одновременно имеющихся блокированных операций в процессе. В некоторых случаях превышение количества нитей над LWP является недостатком. Например, при вычислении произведения двух двухмерных массивов мы могли бы вычислять каждый элемент результирующего массива в отдельной нити. Если количество процессоров невелико, то этот метод, вероятно, окажется непродуктивным, так как библиотека может терять много времени на переключение контекста между нитями. Более эффективным решением может стать создание одной нити для каждой строки вычисляемого массива и привязка каждой нити к отдельному LWP.

Использование связанных и несвязанных прикладных нитей в одном приложении бывает весьма эффективным в тех случаях, когда требуется какая-либо обработка, время выполнения которой критично. Такая обработка может быть выполнена нитями, связанными с LWP, которым в расписании назначен приоритет реального времени. При этом другие нити отвечают за фоновые операции, имеющие более низкий приоритет. В приведенном ранее примере оконной системы нити реального времени могут использоваться для обработки движений мыши, так как их результаты должны немедленно появляться на экране монитора¹.

3.6.4. Реализация прикладных нитей

Каждая прикладная нить должна поддерживать информацию следующего содержания:

- ◆ **Идентификатор нити** (thread ID). Позволяет нитям взаимодействовать друг с другом в рамках процесса при помощи сигналов и прочих средств.
- ◆ **Сохраненное состояние регистров** (saved register state). Содержит указатель команд и указатель стека.
- ◆ **Стек в приложении** (user stack). Каждая нить обладает своим собственным стеком, размещаемым при помощи библиотеки. Ядро системы не знает о существовании подобных стеков.
- ◆ **Маска сигналов** (signal mask). Каждая нить может обладать собственной маской сигналов. Когда приходит сигнал, библиотека доведет его до соответствующей нити, исходя из информации, содержащейся в масках всех нитей.

¹ Неудачный пример. *Обработка* (не отображение на уровне драйвера) движений мыши в развитых графических оболочках (UNIX, Windows, OS/2 – для Intel 80x86) возможна только после выборки сообщений из общей очереди, что несмотря на все приоритеты не позволяет получить немедленные результаты. Можно наблюдать иллюстрацию сказанного при высокой загрузке процессора. Представлять каждый оконный объект в виде нити также нецелесообразно. – Примеч. ред.

- ◆ **Приоритет** (priority). Прикладная нить имеет приоритет внутри процесса, который используется планировщиком нитей. Ядро системы не обладает информацией о таких приоритетах и производит планирование выполнения только для LWP, содержащих эти нити.
- ◆ **Локальная область хранения нити** (thread kernel storage). Каждой нити позволено иметь некоторую собственную область для хранения данных (управляемую библиотекой) с целью поддержки реентерабельных версий интерфейсов библиотеки C [13]. Например, многие функции библиотеки C возвращают код ошибки в глобальную переменную `errno`. Если одну из таких функций вызовут одновременно несколько нитей, то это может привести к хаосу. Для предупреждения подобных проблем многонитевые библиотеки помещают значения `errno` в локальную область хранения нити [20].

Нити используют средства синхронизации, предоставляемые библиотекой, которые похожи на аналогичные средства ядра (условные переменные, семафоры и т. д.). Система Solaris позволяет нитям разных процессов синхронизироваться друг с другом при помощи переменных синхронизации, помещаемых в совместно используемый участок памяти. Такие переменные могут также быть размещены в файлах, а доступ к ним может быть организован при помощи механизма отображения файла `ттмар`. Такой подход позволяет объектам синхронизации иметь время жизни большее, чем у создавшего их процесса, и, следовательно, задействовать эти переменные для синхронизации нитей разных процессов.

3.6.5. Обработка прерываний

Обработчики прерываний часто манипулируют данными, используемыми также и ядром системы. Это требует синхронизации доступа к разделяемым данным. В традиционных системах UNIX ядро обеспечивает ее, повышая *уровень приоритета прерываний* (`ipl`) для блокирования тех прерываний, которые в состоянии получить доступ к таким данным. Часто объекты защищаются от прерываний, хотя обращение к ним со стороны каких-либо прерываний малореально. Например, очередь спящих процессов должна быть защищена от прерываний, хотя большинству из них нет необходимости обращаться к этой очереди.

Эта модель обладает несколькими существенными недостатками. Во многих системах процедура увеличения или уменьшения уровня `ipl` является весьма затратной и требует выполнения нескольких инструкций. Прерывания представляют собой важные и обычно экстренные события, поэтому их блокирование уменьшает производительность системы в большинстве случаев. В многопроцессорных системах эта проблема еще актуальнее, так как ядру

системы приходится защищать намного большее количество объектов и обычно приходится блокировать прерывания на всех имеющихся процессорах.

В операционной системе Solaris традиционная модель прерываний и синхронизации заменена новой технологией, увеличивающей производительность, в первую очередь, на многопроцессорных системах [11], [17]. Новая модель не использует уровни *ipl* для защиты от прерываний. Вместо этого применяется набор различных объектов ядра для осуществления синхронизации, таких как взаимные исключения или семафоры. Для обработки прерываний используется набор нитей ядра. Такие *нити прерываний* могут быть созданы «на лету»; им будет назначен более высокий приоритет выполнения, чем у любых других существующих нитей. Нити прерываний используют те же основные элементы синхронизации, что и любые другие нити, и, следовательно, могут блокироваться в тех случаях, когда необходимый им ресурс занят другой нитью. Ядро блокирует обработку прерываний только при возникновении малого количества исключительных ситуаций, например при попытке освободить mutex, защищающий очередь спящих процессов.

Хотя создание нитей ядра является относительно несложной процедурой, однако организация новой нити для каждого прерывания слишком накладна. Ядро содержит набор предварительно выделенных и частично инициализированных нитей. По умолчанию такой набор состоит из количества нитей, равного количеству уровней прерываний помноженному на количество процессоров плюс еще одна общая системная нить для таймера. Поскольку каждая нить требует около 8 Кбайт для хранения стека и данных, весь набор занимает значительное количество памяти. На системах, обладающих небольшим объемом памяти, имеет смысл уменьшить количество нитей в наборе, так как ситуация, когда необходимо будет одновременно обрабатывать все прерывания, маловероятна.

На рис. 3.9 показана схема обработки прерываний в системе Solaris. Нить H1 выполняется на процессоре P1 в тот момент времени, когда он получает прерывание. Обработчик прерывания в первую очередь поднимает уровень *ipl* для предотвращения дальнейших прерываний того же или более низкого уровня (предохраняющая семантика системы UNIX). Далее происходит выделение нити прерывания H2 из набора нитей и переключение контекста на нее. Пока нить H2 выполняется, H1 остается *прикрепленной* (pinned). Это означает, что она не может выполняться на другом процессоре. После завершения H2 происходит обратное переключение контекста к нити H1, которая затем продолжает свою работу.

Нить прерываний H2 выполняется без полной инициализации. Это означает, что она не является полноценной нитью и не может быть вытеснена. Инициализация завершается только в том случае, если нить имеет причину для блокирования. Тогда она сохраняет свое состояние и становится независимой нитью, выполняющейся на любом свободном процессоре. Если нить H2 заблокируется, управление возвратится к H1, таким образом откреп-

ляя нить H1. В итоге перегрузка, вызываемая полной инициализацией нити, ограничивается случаями, когда нить прерывания должна заблокироваться.

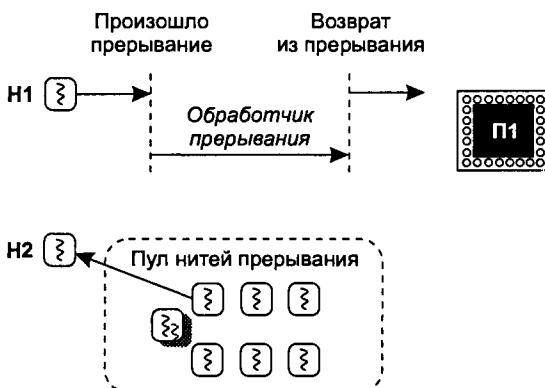


Рис. 3.9. Использование нитей для обработки прерываний

Реализация обработки прерываний через нити несколько увеличивает загруженность системы (на Sparc это порядка 40 инструкций). С другой стороны, такой подход избавляет от необходимости блокирования прерываний для каждого объекта синхронизации, что каждый раз экономит порядка 12 инструкций. Так как операции синхронизации являются более частыми, чем прерывания, общим результатом применения вышеописанного метода является увеличение производительности системы, поскольку прерывания блокируются не так часто.

3.6.6. Обработка системных вызовов

В системе Solaris вызов `fork` дублирует каждый LWP родителя для создаваемого потомка. Любые легковесные процессы, находящиеся в состоянии выполнения какого-либо системного вызова, вернутся с ошибкой `EINTR`. В ОС Solaris предлагается дополнительный системный вызов `fork1`, который похож на `fork`, но клонирует только ту нить, которая его вызвала. Вызов `fork1` удобен в тех случаях, когда процесс-потомок вскоре собирается запустить новую программу.

Решение проблемы одновременных произвольных операций ввода-вывода над файлом сведено в Solaris 2.3 к добавлению системных вызовов `pread` и `pwrite`, которые принимают в качестве аргументов смещение в файле. К сожалению, система не предоставляет аналогичных вызовов, заменяющих `readv` и `writenv`, которые осуществляют ввод-вывод методом сборки-разъединения (`scatter-gather I/O`, подробнее см. в разделе 8.2.5).

В заключение упомянем о том, что система Solaris предлагает богатый набор программных интерфейсов с ранее описанной двухуровневой моделью.

Поддержка прикладных нитей совместно с легковесными процессами дает возможность разделять то, что видит разработчик приложения, и то, как это представлено в системе. *Программист может создавать приложения, изначально используя только нити, и далее оптимизировать их путем манипуляции с содержащими эти нити легковесными процессами, добиваясь необходимой степени одновременности для данного приложения.*

3.7. Нити в системе Mach

Операционная система Mach изначально создавалась многонитевой. Mach поддерживает нити как на уровне ядра, так и при помощи библиотек прикладного уровня. Система предлагает дополнительные механизмы для управления функционированием нитей на разных процессорах многопроцессорных систем. ОС Mach поддерживает семантику 4.3BSD UNIX на уровне программного интерфейса полностью, включая все системные вызовы и библиотеки¹. Этот раздел описывает реализацию нитей в системе Mach. Следующий раздел расскажет о системном интерфейсе Digital UNIX — системы, основанной на Mach. В разделе 3.9 вы увидите описание нового механизма под названием *продолжений* (continuations), впервые представленного в Mach 3.0.

3.7.1. Задачи и нити в системе Mach

Ядро Mach поддерживает два фундаментальных элемента системы: *задачу* и *нить* [22]. *Задача* — это статический объект, занимающий адресное пространство и набор системных ресурсов, называющихся *правами порта* (см. раздел 6.4.1). Сама по себе задача является не выполняемым объектом, а средой, в которой может выполняться одна или большее количество нитей.

Нить является основным выполняемым элементом, функционирующим в контексте задачи. Задача может содержать ноль и более нитей, каждая из которых разделяет ее ресурсы. Нить обладает стеком ядра, используемым для обработки системных вызовов. Она также имеет собственные переменные состояния, такие как указатели команд и стека, регистры общего назначения и т. д. Планирование нитей на выполнение процессором происходит независимо. Нити, относящиеся к прикладным задачам, эквивалентны легковесным процессам. Нити ядра, используемые ядром, относятся к *задачам ядра*.

Система Mach также поддерживает *наборы процессоров*, которые более подробно описаны в разделе 5.7.1. Все процессоры, доступные системе, могут быть поделены на не перекрывающие друг друга наборы. Каждая задача или нить может быть связана с любым набором процессоров (большинство опе-

¹ Система Mach версии 2.5 поддерживает функции 4.3BSD на уровне ядра. Версия 3.0 реализует эти функции в виде серверной программы на прикладном уровне.

раций над наборами процессоров требуют привилегий суперпользователя). Такой подход позволяет назначить несколько процессоров многопроцессорной системы для выполнения одной или более определенных задач. Это гарантирует доступность ресурсов для наиболее приоритетных задач.

Структура `task` описывает *задачу* и хранит следующую информацию:

- ◆ указатель на карту адресации, описывающую виртуальное адресное пространство задачи;
- ◆ заголовок списка нитей, относящихся к задаче;
- ◆ указатель на набор процессоров, с которым связана задача;
- ◆ указатель на структуру `utask` (см. раздел 3.8.1);
- ◆ порты и другую информацию, относящуюся к межпроцессному взаимодействию (IPC, см. подробнее в разделе 6.4).

Ресурсы, удерживаемые задачей, совместно используются всеми ее нитями. Каждая нить описана структурой `thread`, содержащей:

- ◆ связь нити с очередью планировщика или очередью ожидания;
- ◆ указатели на `task` и на набор процессоров, к которому относится нить;
- ◆ связь нити со списком всех нитей задачи и со списком всех нитей набора процессоров;
- ◆ указатель на блок *управления процессом* (PCB), содержащий сохраненный контекст регистров;
- ◆ указатель на стек ядра;
- ◆ состояние выполнения (готовый к работе, спящий, блокированный и т. д.);
- ◆ информацию для планировщика, такую как приоритет, правила планирования и данные по использованию процессора;
- ◆ указатели на связанные с нитью структуры `uthread` и `utask` (см. раздел 3.8.1);
- ◆ информацию об IPC, относящуюся к нити (см. подробнее в разделе 6.4.1).

Задачи и нити играют дополняющие друг друга роли. Задача владеет ресурсами системы, включая адресное пространство. Нить выполняет код. Процесс в традиционной системе UNIX представляет собой задачу, содержащую единственную нить. Многонитевый процесс состоит из одной задачи и некоторого количества нитей.

Система Mach предлагает набор системных вызовов для работы с задачами и нитями. Вызовы `task_create`, `task_terminate`, `task_suspend` и `task_resume` используются для работы с задачами. Вызовы `thread_create`, `thread_terminate`, `thread_suspend` и `thread_resume` выполняют операции над нитями. Название каждого вызова красноречиво самодокументирует выполняемые им действия. В добавок к перечисленным вызовам имеются вызовы `thread_status` и `thread_mutate` для чтения и изменения состояния регистра нити, а вызов `task_threads` возвращает список всех нитей задачи.

3.7.2. Библиотека C-threads

Система Mach содержит библиотеку C-threads, которая обеспечивает простой интерфейс для создания и управления нитями. Например, функция

```
cthread_t cthread_fork (void* (*func)(), void* arg);
```

создает новую нить, в которой будет запущена функция `func()`. Нить может вызывать

```
void* cthread_join (cthread_t T);
```

для приостановки своей работы до тех пор, пока не закончит функционирование нить `T`. В вызывающий код нити вернется значение функции верхнего уровня нити `T` или код выхода, выработанный после выполнения нитью `T` функции `cthread_exit()`.

Библиотека C-threads поддерживает взаимные исключения и условные переменные с целью осуществления синхронизации. В библиотеке имеется функция `cthread_yield()`, которая запрашивает планировщик на разрешение выполнения другой нити вместо себя. Такая функция необходима только в случаях применения сопрограмм, описанных ниже.

Существуют три различных реализаций библиотеки C-threads. Разработчик приложения может использовать вариант, наиболее отвечающий перечисленным ниже требованиям к разрабатываемому продукту.

- ◆ **Приложение основано на сопрограммах** (coroutine-based), где прикладные нити мультиплексируются в однонитевую задачу (процесс UNIX). Такие нити являются не вытесняемыми, библиотека выполнит переключение в другую нить только в случае выполнения процедур синхронизации (когда текущая нить должна быть блокирована на взаимном исключении или семафоре). Помимо этого библиотека полагается на нити, вызывающие функцию `cthread_yield()`, что защитит другие нити от долгого «стояния» в очереди. Эта реализация библиотеки подходит для отладки, так как порядок переключения контекста нитей является повторяемым.
- ◆ **Приложение основано на нитях** (thread-based), каждая нить C-thread использует отдельную нить, поддерживаемую ядром Mach. Такие нити являются вытесняемыми и могут выполняться на многопроцессорных системах параллельно. Эта реализация является стандартной и используется для разработки различных вариантов программ, базирующихся на нитях C-thread.
- ◆ **Приложение основано на задачах** (task-based), где используется одна задача, поддерживаемая ядром Mach (процесс UNIX) на каждую нить C-thread. Для совместного использования памяти между нитями применяются элементы виртуальной памяти, обеспечиваемые ядром Mach. Такой вариант используется только в случаях, когда необходима специализированная семантика разделения памяти.

3.8. Digital UNIX

Операционная система Digital UNIX (ранее известная как DEC OSF/1) основана на ядре Mach 2.5. С точки зрения разработчика приложений эта система предоставляет полный программный интерфейс UNIX. Однако внутренняя реализация многих возможностей традиционного варианта UNIX в этой системе опирается на базовые элементы ОС Mach. Эта реализация организована на уровне совместимости системы Mach с ОС 4.3BSD, расширенного фондом Open Software Foundation до совместимости с SVR3 и SVR4. Это свойство существенно повлияло на устройство системы Digital UNIX.

Система предлагает изящный набор средств, расширяющих понятие процесса [19]. Многонитевые процессы поддерживаются как на уровне ядра, так и на уровне совместимых со стандартами POSIX нитевых библиотек. Процесс UNIX реализован как верхний уровень задач и нитей системы Mach.

3.8.1. Интерфейс UNIX

Хотя задачи и нити достаточно адекватно обеспечивают интерфейс выполнения программ системы Mach, они не в полной мере описывают процесс UNIX. Процесс обеспечивает некоторые свойства, которые не отражены в Mach, такие как полномочия пользователя, дескрипторы открытых файлов, обработчики сигналов и группы процессов. Более того, для предотвращения изменения традиционного интерфейса UNIX был осуществлен перенос кода уровня, обеспечивающего совместимость Mach 2.5 с 4.3BSD, который, в свою очередь, был перенесен из оригинальной реализации 4.3BSD. Точно так же был произведен перенос многих драйверов устройств из системы Digital ULTRIX, также основанной на ОС BSD. Перенесенный код делает множественные ссылки на структуры proc и user также для обеспечения совместимости.

Применение оригинального варианта структур proc и user является причиной возникновения двух проблем. Во-первых, некоторая информация из этих структур уже отражена в структурах task и thread. Во-вторых, они не могут адекватно представлять многонитевые процессы. Например, традиционная область и содержит блок управления процессом, который хранит контекст регистров процесса. В случае многонитевости каждая нить обладает собственным контекстом регистров. Следовательно, обе структуры должны быть существенно изменены.

Область и заменена двумя объектами: единой структурой utask, которая используется задачей целиком, и по одной структуре uthread выделено для каждой нити задачи. Новые структуры не занимают фиксированное адресное пространство процесса и не участвуют в его свопинге.

Структура utask содержит следующую информацию:

- ◆ указатели на объекты vnode текущего и корневого каталогов;
- ◆ указатель на структуру proc;

- ◆ массив обработчиков сигналов и других полей, относящихся к сигналам;
- ◆ таблицу дескрипторов открытых файлов;
- ◆ маску создания файлов, используемую по умолчанию (*cmask*);
- ◆ данные об использовании ресурсов, квотах и информацию профиля.

Если одна из нитей открывает файл, то его дескриптор может быть использован совместно всеми нитями задачи. Также все нити будут иметь один и тот же текущий рабочий каталог. Структура *uthread* описывает ресурсы, относящиеся к каждой нити процесса UNIX, и содержит следующую информацию:

- ◆ указатель на сохраненные регистры прикладного уровня;
- ◆ поля для просматриваемых путей;
- ◆ текущие и ожидающие сигналы;
- ◆ обработчики сигналов, определенные для данной нити.

Для упрощения переноса ссылки на поля старой области и были преобразованы в ссылки на поля структуры *utask* или *uthread*. Такое преобразование можно осуществить при помощи макроса, например:

```
#define u_cmask      utask->uu_cmask
#define u_pcbs     uthread->uu_pcbs
```

Структура *proc* претерпела незначительные изменения, но большинство ее функций теперь возложено на структуры *task* и *thread*. В результате большинство ее полей не используется, хотя они и сохранены из «исторических» соображений. Например, поля, относящиеся к расписанию и приоритету, не являются необходимыми, так как в ОС Digital UNIX каждая нить планируется на выполнение индивидуально. Структура *proc* операционной системы Digital UNIX содержит следующую информацию:

- ◆ связи с очередью размещенных процессов, процессов-зомби или свободных процессов;
- ◆ маски сигналов;
- ◆ указатель на структуру полномочий процесса;
- ◆ информацию об идентификации и иерархии процесса (PID процесса, PID предка, указатели на процесс-предок, процессы-потомки, процессы того же уровня и т. д.);
- ◆ группу процесса и информацию сеанса;
- ◆ поля, относящиеся к расписанию (не используются);
- ◆ поля для хранения состояния и использования ресурсов при выходе;
- ◆ указатели на структуры *task* и *utask*, а также на первую в списке структуру *thread*.

На рис. 3.10 показана связь между структурами данных в системе Mach и традиционной UNIX. Структура task содержит связанный список своих нитей. Структура task указывает на utask, а каждая структура thread указывает на соответствующую структуру uthread. Структура proc содержит указатели на структуры task, utask и ссылается на первую в списке структуру thread. Структура utask содержит обратный указатель на proc, а каждая thread включает в себя обратные указатели на task и utask. Такая организация взаимосвязей дает возможность быстрого доступа ко всем структурам.

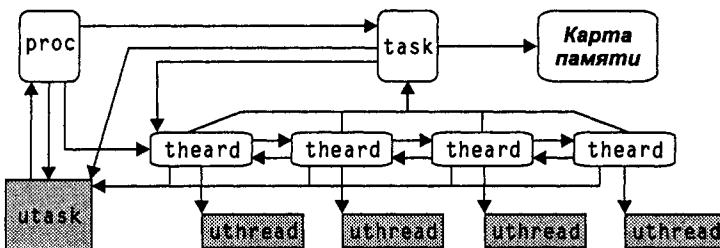


Рис. 3.10. Структуры данных задач и нитей в системе Digital UNIX

Не всем нитям присваивается прикладной контекст. Некоторые могут создаваться непосредственно ядром для выполнения различных системных функций, таких как замена страниц памяти. Такие нити связаны с задачей ядра, которая не имеет прикладного адресного пространства. Задача ядра и нити ядра не имеют связанных с ними структур utask, uthread и proc.

3.8.2. Системные вызовы и сигналы

В операционной системе Digital UNIX системный вызов fork создает новый процесс, обладающий единственной нитью, которая является точной копией нити, вызвавшей fork. Система не поддерживает альтернативного варианта вызова, дублирующего все нити.

Так же как и в ОС Solaris, все сигналы классифицируются на синхронные (системные прерывания, или ловушки, — *traps*) или асинхронные сигналы (прерывания, *interrupts*). Ловушка доставляется той нити, которая стала причиной ее срабатывания. Сигналы прерываний доставляются любой нити, которая примет их. Однако, в отличие от операционной системы Solaris, все нити процесса используют единый набор масок сигналов, который хранится в структуре proc. Каждая нить может создавать собственный набор обработчиков синхронных сигналов, но все нити процесса должны пользоваться единым набором обработчиков асинхронных сигналов.

3.8.3. Библиотека pthreads

Библиотека pthreads предлагает совместимый с POSIX программный интерфейс прикладного уровня для реализации нитей, являющийся более простым, чем

системные вызовы системы Mach. С каждой нитью pthread библиотека связывает одну нить Mach. Функционально pthreads подобны C-threads или иной нитевой библиотеке, но именно в pthreads реализован интерфейс, принятый как стандарт.

Библиотека pthreads реализует функции асинхронного ввода-вывода, определенные стандартом POSIX. Например, если нить вызывает определенную в POSIX функцию `aioread()`, то библиотека создаст новую нить для синхронного чтения. Когда чтение завершится, ядро разбудит заблокированную на этой операции нить, которая, в свою очередь, уведомит вызывающую ее нить при помощи сигнала. Асинхронный ввод-вывод проиллюстрирован на рис. 3.11.

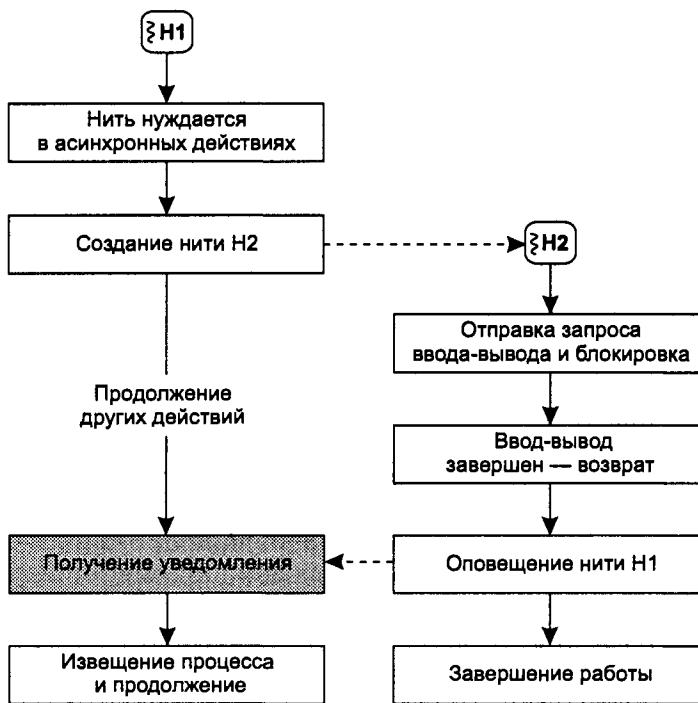


Рис. 3.11. Реализация асинхронного ввода-вывода при помощи создания отдельной нити

Библиотека pthreads предоставляет полный программный интерфейс, включающий функции обработки сигналов и планирования, а также набор элементов синхронизации. Синхронизация между нитями может быть реализована на прикладном уровне, однако если требуется заблокировать LWP, должно вмешаться ядро системы.

Digital предлагает свою собственную библиотеку, `cmu_threads`, поддерживающую некоторые дополнительные возможности [8]. Программы, использующие эту библиотеку, будут работать только на платформах Digital VMS и Windows, на других UNIX-системах они функционировать не будут.

3.9. Продолжения в системе Mach

Хотя нити ядра более легковесны, чем процессы, они все равно занимают больший объем памяти ядра, используя ее преимущественно для стека. Обычно стеки ядра занимают как минимум 4 Кбайт памяти, что составляет почти 90% пространства ядра, используемого нитью. В системах с большим количеством нитей (достигающим нескольких сотен) это может привести к уменьшению производительности. Одним из решений проблемы является мультиплексирование прикладных нитей в меньшее количество нитей Mach или легковесных процессов, что предотвратит потребность в стеке ядра для каждой прикладной нити. Такой подход имеет свои минусы, так как прикладные нити не могут планироваться на выполнение независимо и, следовательно, не дают такого же уровня одновременности, как в случае соответствия каждой прикладной нити одной нити ядра. Более того, раз нити ядра не переходят границы задачи, соответственно каждая задача должна обладать по крайней мере одной нитью ядра, что создает проблемы на системах с большим количеством активных задач. В этом разделе описан подход к решению вышеописанных проблем, предлагаемый системой Mach 3.0, при помощи средства, получившего название *продолжений* (continuations).

3.9.1. Модели выполнения программ

Ядро системы UNIX использует модель выполнения программ, называемую *моделью процессов*. Каждая нить имеет стек ядра, востребуемый нитью, когда она переходит в режим ядра для выполнения системного вызова или обработки исключения. Если нить блокируется в ядре, ее стек содержит состояние ее выполнения, в том числе последовательность вызова¹ и автоматические переменные. Основным достоинством описанного подхода является его простота, поскольку нити ядра могут блокироваться без необходимости явного сохранения каких-либо состояний. Главный недостаток модели процессов — это большой расход памяти.

Некоторые операционные системы, такие как QuickSilver [14] и V [6], используют иную программную модель под названием *модели прерываний*. Ядро обрабатывает системные вызовы и исключения как прерывания, организуя для всех операций ядра единый стек (один для каждого процессора). Следовательно, если нити необходимо заблокироваться, находясь в ядре, ей в первую очередь нужно где-то сохранить свое состояние. Ядро использует сохраненную информацию для восстановления состояния нити при следующем ее запуске.

Основным достоинством модели прерываний является экономия памяти, достигаемая за счет использования единого стека ядра. Главным недостат-

¹ То есть когда одна функция вызывает другую; она, в свою очередь, — третью, и так далее, и на каком-то шаге происходит блокирование. — Примеч. ред.

ком является необходимость сохранения состояния нити при проведении любой операции, потенциально могущей привести к ее блокированию. Это осложняет применение модели, так как сохраняемая информация может пересечь границы модуля. Следовательно, если нить блокируется, находясь в глубоко вложенной процедуре, ей необходимо определить (для сохранения) состояние, необходимое для всех вызовов в последовательности.

Условия, при которых нить должна быть блокирована, определяются той моделью, которая является наиболее пригодной. Наиболее подходящим вариантом является задание блокируемой нитью условий, при которых необходимо использовать ту или иную модель. Если нить блокируется где-то глубоко внутри последовательности вызовов, ей больше подойдет модель процессов. Однако если нити требуется сохранить при блокировании информацию о состоянии небольшого размера, то в этом случае модель прерываний окажется более предпочтительной. К примеру, многие серверные программы периодически блокируются в ядре, ожидая запроса клиента, и затем обрабатывают запросы по мере их получения. Такая программа может легко освобождать свой стек.

Механизм продолжений системы Mach комбинирует преимущества обеих моделей и позволяет ядру выбрать метод блокирования в зависимости от условий. Следующий раздел книги посвящен описанию устройства и реализации этого средства.

3.9.2. Использование продолжений

Для блокирования нити система Mach использует функцию `thread_block()`. В Mach версии 3.0 эта функция была изменена и теперь обладает аргументом:
`thread_block (void (*contfn)());`

где `contfn()` — это *функция продолжения*, которая будет запущена при следующем выполнении нити. Передача функции аргумента `NULL` указывает на необходимость традиционного поведения при блокировании. При таком подходе нить может выбирать, какое продолжение задействовать далее.

Если нить собирается использовать продолжение, то, в первую очередь, она нуждается в сохранении того состояния, которое будет ей необходимо при возобновлении выполнения. Структура нити содержит для этой цели 28-байтовую временную область. Если потребуется больший объем пространства, то нить выделит дополнительную структуру данных. Ядро блокирует нити и забирает ее стек. После возобновления функционирования нити ядро дает ей новый стек и вызывает функцию продолжения. Эта функция восстанавливает состояние из сохраненной области. Такой подход требует, чтобы продолжение и вызываемая функция имели точное представление о том, какое состояние сохранено и где.

Использование продолжений показано на следующем примере. Листинг 3.1 демонстрирует традиционный подход к блокированию нити.

Листинг 3.1. Блокировка нити без использования продолжений

```
syscall_1 (arg1)
{
    ...
    thread_block();
    f2(arg1);
    return;
}
f2(arg1)
{
    ...
    return;
}
```

Листинг 3.2 иллюстрирует применение продолжений.

Листинг 3.2. Блокировка нити с использованием продолжений

```
syscall_1 (arg1)
{
    ...
    сохранение arg1 и другой информации о состоянии
    ...
    thread_block(f2);
    /* сюда выполнение не доходит */
}
f2()
{
    ...
    восстановление arg1 и другой информации о состоянии
    ...
    thread_syscall_return (status);
}
```

Следует упомянуть о том, что в случае вызова функции `thread_block()` с аргументом не произойдет возврата в вызвавший ее код. После восстановления функционирования нити ядро передает управление `f2()`. Функция `thread_syscall_return()` используется для возврата на прикладной уровень из системного вызова. Весь этот процесс является прозрачным для разработчика, так как он видит лишь синхронный выход из системного вызова.

Ядро использует продолжения при условии, что сохраненное перед блокированием состояние будет небольшим. Например, одна из наиболее частых блокирующих операций происходит при ошибке обработки страницы. В традиционных реализациях UNIX код обработчика вызывается вследствие запроса чтения с диска и блокирует работу до тех пор, пока чтение не закончится. Затем ядро системы возвращает нить на прикладной уровень, после чего приложение может дальше продолжать функционирование. Работа, которая должна быть выполнена после окончания операции чтения с диска,

требует небольшого сохраненного состояния (например, указателя на считанную страницу и данные отображения памяти, которые должны быть обновлены). Этот пример показывает ситуацию, когда применение продолжений оправданно.

3.9.3. Оптимизация работы

Главным достоинством продолжений можно назвать сокращение количества стеков в ядре системы. Продолжения также позволяют провести ряд важных оптимизаций. Представьте, что при переключении контекста ядро обнаружило, что предыдущая и последующая нити используют продолжения. Предыдущая нить уже освободила свой стек ядра, а следующая нить еще не имеет такового. В этом случае ядро может передать стек от старой нити новой напрямую, как это продемонстрировано на рис. 3.12. Помимо исключения перегрузок, которые бы происходили при выделении нового стека, такой подход помогает сократить кэш-промахи¹ и буферы ассоциативной трансляции (translation lookaside buffer, TLB, см. подробнее в разделе 13.3.1), ассоциированные с переключением контекста, так как используется та же область памяти.

Преимущества продолжений используются также в реализации IPC (межпроцессное взаимодействие) системы Mach. Передача сообщения включает две стадии. Клиентская нить использует для отправки сообщения и ожидания ответа системный вызов `mach_msg`, а серверная нить использует тот же вызов для отправки ответа клиентам и ожидания новых запросов. Сообщение отправляется в порт, а также принимается из порта, являющегося защищенной очередью сообщений. Отправка и получение сообщений осуществляются независимо друг от друга. Если получатель не готов, ядро поместит сообщение в очередь порта.

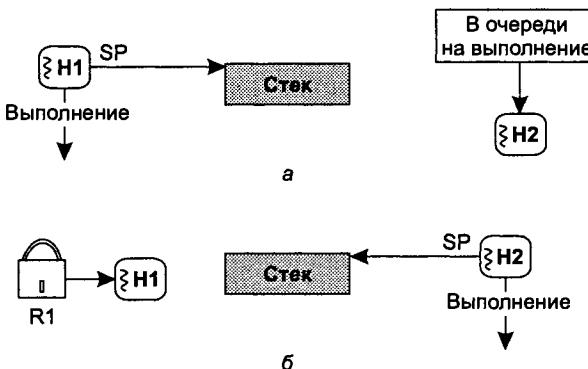


Рис. 3.12. Передача стека при использовании продолжений: а — перед блокировкой нити H1 с функцией продолжения; б — после контекстного переключения

¹ То есть когда требуемые данные в кэше отсутствуют. — Прим. ред.

Если получатель находится в режиме ожидания, то процедура пересылки может быть оптимизирована при помощи продолжений. Когда отправитель обнаружит, что получатель находится в режиме ожидания, он передаст свой стек получателю и заблокируется на функции продолжения `mach_msg_continue()`. Получающая нить восстановит свою работу, используя при этом стек отправителя, который уже содержит всю необходимую информацию о передаваемом сообщении. Такой подход предотвращает перегрузку, возникающую при помещении в очередь и извлечении из нее сообщения, а также ощутимо увеличивает скорость обмена сообщениями. После ответа сервера происходит передача его стека клиентской нити и возобновление работы клиента описанным выше способом.

3.9.4. Анализ производительности

Механизм продолжений системы Mach показал себя очень эффективным. Так как его применение не является обязательным, нет необходимости менять программную модель целиком, и его использование может быть наращиваемым. Механизм продолжений очень сильно сокращает количество запросов, размещаемых в памяти ядра. Измерения производительности показали [10], что в среднем в системе, которой требуется 2002 байт стека в ядре на каждый процессор, пространство ядра для каждой нити сокращается с 4664 до 690 байт.

Операционная система Mach 3.0 очень неплохо подходит для продолжений, так как обладает микроядром, имеющим небольшое количество базовых элементов и предлагающим скромный интерфейс. В частности, та часть кода, которая сохраняла совместимость системы с UNIX, была удалена из ядра, и ее реализация состоялась в виде серверов прикладного уровня [12]. В результате оказалось только 60 потенциальных мест, в которых ядро может блокировать выполнение, а 99% всех случаев блокирования происходит в шести «горячих точках». При концентрировании внимания на них обеспечивается определенное преимущество, заключающееся в уменьшении усилий, затрачиваемых на разработку приложений. В противоположность Mach, традиционные системы UNIX могут производить блокирование в сотнях мест, при этом не имея ни одной так называемой «горячей точки».

3.10. Заключение

В этой главе вы увидели несколько вариантов устройства многонитевых систем. Существует большое количество типов средств, относящихся к нитям, а комбинирование их системой дает возможность создания сложной с элементами одновременности программной среды. Нити могут поддерживаться

как на уровне ядра, так и на уровне прикладных библиотек, допустима и комбинация обоих вариантов.

Разработчикам приложений предложен выбор приемлемого соотношения прикладных средств и возможностей, предоставляемых ядром системы. Единственная проблема, с которой им придется столкнуться, это различия в наборах системных вызовов для создания и управления нитями, предлагаемые в различных операционных системах, что затрудняет написание переносимого многонитевого кода, который бы эффективно использовал системные ресурсы. Стандарт POSIX 1003.4a определяет функции нитевой библиотеки, однако в нем не описываются интерфейсы или реализации для ядра.

3.11. Упражнения

1. Для каждого из перечисленных ниже приложений проанализируйте пригодность использования легковесных процессов, прикладных нитей или иных программных моделей:
 - серверный компонент распределенной службы имен;
 - оконная система, такая как X server;
 - научное приложение, работающее на многопроцессорной системе и производящее большое количество параллельных вычислений;
 - утилита *make*, которая компилирует файлы параллельно по мере возможности.
2. В каких ситуациях применение нескольких процессов в приложении будет более оправдано, нежели использование LWP или прикладных нитей?
3. Почему каждому LWP необходим отдельный стек ядра? Может ли система экономить ресурсы, создавая стек ядра только в тех случаях, когда LWP делает системный вызов?
4. Структура *proc* и область *i* содержат атрибуты и ресурсы процесса. Какие из полей этих структур в многонитевых системах можно разделить между всеми LWP процесса, а какие из них должны существовать для каждого LWP отдельно?
5. Представьте, что LWP вызывает *fork* в тот момент времени, когда другой LWP, принадлежащий тому же процессу, вызывает *exit*. Что произойдет в результате, если система использует *fork* для дублирования всех LWP процесса? Что случится, если вызов *fork* будет дублировать только один (вызывающий *fork*) LWP?
6. Возникнут ли какие-нибудь проблемы с вызовом *fork* в многонитевой системе, поддерживающей его как единый вызов, атомарно совмещающий *fork* и *exec*?

7. Раздел 3.3.2 описывает проблемы, возникающие при использовании единого разделяемого набора ресурсов, таких как дескрипторы файлов или текущий каталог. Почему не следует эти ресурсы предоставлять отдельно для каждого легковесного процесса или прикладной нити? Более подробно поднятая проблема исследуется в [3].
8. Стандартная библиотека определяет для каждого процесса переменную `errno`, содержащую код ошибки, возвращенный последним сделанным системным вызовом. Какие проблемы это может создать в многонитевом процессе? Каким образом их можно преодолеть?
9. Во многих системах библиотечные функции подразделяются на ните-защищенные и нитенезащищенные. К чему может привести использование функции, которая не защищена при использовании в многонитевых приложениях?
10. Перечислите недостатки использования нитей для запуска в них обработчиков прерываний.
11. Какие вы видите недостатки в планировании выполнения LWP посредством ядра?
12. Предложите интерфейс, который позволил бы управлять планированием и выбирать, какой из LWP первым отправится на выполнение. К возникновению каких проблем это может привести?
13. Сравните базовые многонитевые элементы в системах Solaris и Digital UNIX. Какие преимущества имеются у каждого из них?

3.12. Дополнительная литература

1. Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., «Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism», Proceedings of the Thirteenth Symposium on Operating System Principles, Oct. 1991, pp. 95–109.
2. Armand, F., Hermann, F., Lipkis, J., and Rozier, M., «Multi-threaded Processes in Chorus/MIX», Proceedings of the Spring 1990 European UNIX Users Group Conference, Apr. 1990.
3. Barton, J. M., and Wagner, J. C., «Beyond Threads: Resource Sharing in UNIX», Proceedings of the Winter 1988 USENIX Technical Conference, Jan. 1988, pp. 259–266.
4. Bitar, N., «Selected Topics in Multiprocessing», USENIX 1995 Technical Conference Tutorial Notes, Jan. 1995.
5. Black, D. L., «Scheduling Support for Concurrency and Parallelism in the Mach Operating System», IEEE Computer, May 1990, pp. 35–43.

6. Cheriton, D. R., «The V Distributed System», Communications of the ACM, Vol. 31, No. 3, Mar. 1988, pp. 314–333.
7. Cooper, E. C., and Draves, R. P., «C Threads», Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, Sep. 1990.
8. Digital Equipment Corporation, «DEC OSF/1 — Guide to DECthreads», Part No. AA-Q2DPB-TK, July 1994.
9. Doeppner, T. W., Jr., «Threads, A System for the Support of Concurrent Programming», Brown University Technical Report CS-87-11, Jun. 1987.
10. Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W., «Using Continuations to Implement Thread Management and Communication in Operating Systems», Technical Report CMU-CS-9-115R, Department of Computer Science, Carnegie Mellon University, Oct. 1991.
11. Eykholt, J. R., Kleiman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M., and Williams, D., «Beyond Multiprocessing: Multithreading the SunOS Kernel», Proceedings of the Summer 1992 USENIX Technical Conference, Jun. 1992, pp. 11–18.
12. Golub, D., Dean, R., Forin, A., and Rashid, R., «UNIX as an Application Program», Proceedings of the Summer 1990 USENIX Technical Conference, Jun. 1990, pp. 87–95.
13. Institute for Electrical and Electronic Engineers, «POSIX PI003.4a, Threads Extension for Portable Operating Systems», 1994.
14. Haskin, R., Malachi, Y., Sawdon, W., and Chan, G., «Recovery Management in Quicksilver», ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 82–108.
15. Kepecs, J., «Lightweight Processes for UNIX Implementation and Applications», Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 299–308.
16. Keppel, D., «Register Windows and User-Space Threads on the SPARC», Technical Report 91-08-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, Aug. 1991.
17. Kleiman, S. R., and Eykholt, J. R., «Interrupts as Threads», Operating Systems Review, Vol. 29, No. 2, Apr. 1995.
18. Mueller, F., «A Library Implementation of POSIX Threads under UNIX», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 29–41.
19. Open Software Foundation, «Design of the OSF/1 Operating System — Release 1.2», Prentice-Hall, Englewood Cliffs, NJ, 1993.

20. Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M., «SunOS Multi-thread Architecture», Proceedings of the Winter 1991 USENIX Technical Conference, Jan. 1991, pp. 65–80.
21. Sun Microsystems, «SunOS 5.3 System Services», Nov. 1993.
22. Tevanian, A., Jr., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E., and Young, M. W., «Mach Threads and the UNIX Kernel: The Battle for Control», Proceedings of the Summer 1987 USENIX Technical Conference, Jun. 1987, pp. 185–197.
23. Vandevenne, M., and Roberts, E., «WorkCrews: An Abstraction for Controlling Parallelism», International Journal of Parallel Programming, Vol. 17, No. 4, Aug. 1988, pp. 347–366.

Глава 4

Сигналы и управление сеансами

4.1. Введение

Сигналы используются для оповещения процесса о возникновении системных событий. Еще одной функцией сигналов является простой механизм, используемый для коммуникаций и синхронизации между прикладными процессами. Программный интерфейс, поведение, а также внутренняя реализация сигналов сильно отличаются от одной версии UNIX к другой, а иногда и в различных версиях одной и той же операционной системы. Словно для того, чтобы еще более запутать разработчика, операционная система предоставляет дополнительные системные вызовы и библиотечные функции для поддержки ранних интерфейсов сигналов, а также для обеспечения обратной (backward) совместимости¹.

Оригинальная реализация сигналов в ОС System V была изначально неэффективной и ненадежной. Многие ее проблемы были решены после появления системы 4.2BSD UNIX, в которой был предложен новый, надежный механизм сигналов (расширенный в следующей версии, 4.3BSD). Однако механизм системы 4.2BSD явился несовместимым с интерфейсом System V по некоторым аспектам, что послужило основой для возникновения определенных проблем как у разработчиков приложений, желающих создавать переносимые программы, так и у поставщиков операционных систем, стремившихся к совместимости своего продукта одновременно с BSD и System V.

Стандарт POSIX 1003.1 (также известный как POSIX.1, [5]) дал возможность наведения некоторого порядка в хаосе различных реализаций сигналов. Он определил стандартный интерфейс, который должны поддерживать все совместимые с POSIX операционные системы. Однако стандарт не опи-

¹ Такой подход приводит к возникновению и других проблем. Если библиотека, связанная с приложением, использует один набор сигнальных интерфейсов, а приложение — другой, результатом может оказаться некорректное функционирование программы.

сывает, каким именно образом этот интерфейс должен быть реализован. Разработчики операционных систем могут решать сами, на каком уровне они будут поддерживать рекомендации стандарта: в ядре, или через прикладные библиотеки, или через комбинацию обеих составляющих.

Разработчики ОС SVR4 ввели в систему новую, POSIX-совместимую реализацию сигналов, включающую в себя многие возможности механизма сигналов системы BSD. Современные варианты UNIX (такие как Solaris, AIX, HP-UX, 4.4BSD и Digital UNIX) также предлагают совместимые с POSIX решения. Реализация сигналов в 3SVR4 помимо совместимости со стандартом сохранила совместимость с более ранними версиями System V.

Эта глава начинается с объяснения, что такое сигналы, и последующего анализа проблем, имеющихся в оригинальной ОС System V. Затем вы увидите, как эти проблемы были решены в современных операционных системах, располагающих механизмом надежных сигналов. В конце главы мы расскажем об управлении заданиями и сессиями — понятиями, имеющими тесную связь с сигналами.

4.2. Генерирование и обработка сигналов

Сигналы дают возможность вызвать какую-либо процедуру при возникновении события из определенного их набора. События обозначаются целыми числами и представляются символьными константами. Некоторые из событий являются асинхронными уведомлениями (возникающими, например, когда пользователь посыпает сигнал прерывания, нажав комбинацию `Ctrl+C` на терминале), в то время как другие представляют собой синхронные ошибки или исключения (например, при попытке обращения по несуществующему адресу).

Процесс оповещения состоит из двух этапов: генерирования и доставки. Сигнал генерируется после того, как возникает определенное событие, требующее уведомления о нем процесса, который явился виновником его появления. Сигнал считается доставленным (или обработанным), когда получивший его процесс определяет факт доставки сигнала и производит необходимые действия. Между этими двумя событиями сигнал находится в режиме ожидания процесса.

В оригинальном варианте ОС System V определено 15 различных сигналов. Системы 4BSD и SVR4 поддерживают по 31 сигналу. Каждому из них присваивается номер от 1 до 31 (установка номера сигнала в 0 для различных функций имеет специальные значения, например «никаких сигналов»). Адресация сигналов по их номерам отличается в системах System V и BSD UNIX (например, `SIGSTOP` имеет номер 17 в 4.3BSD и номер 23 в SVR4). Более

того, многие коммерческие реализации UNIX (такие как AIX) поддерживают больше чем 31 сигнал. Обычно программисты предпочитают использовать символические имена для идентификации сигналов. Стандарт POSIX 1003.1 определяет символические имена для всех поддерживаемых им сигналов. Эти имена являются переносимыми как минимум для всех реализаций систем, совместимых со стандартом POSIX.

4.2.1. Обработка сигналов

Каждый сигнал обладает некоторым *действием, установленным по умолчанию*, которое производится ядром системы, если процесс не имеет определенного альтернативного обработчика. Всего таких действий пять.

- ◆ **Аварийное завершение (abort).** Завершает процесс после создания *дампа состояния процесса* (*core dump*), представляющего собой содержимое адресного пространства процесса и его контекст регистров, записанные в файл *core*, расположенный в текущем каталоге процесса¹. Создаваемый файл может быть подвергнут дальнейшему анализу при помощи отладчика или других утилит.
- ◆ **Выход (exit).** Завершает процесс без создания дампа состояния процесса.
- ◆ **Игнорирование (ignore).** Игнорирует сигнал.
- ◆ **Остановка (stop).** Приостанавливает процесс.
- ◆ **Продолжение (continue).** Возобновляет работу приостановленного процесса (или игнорируется в ином случае).

Процесс может переопределить действия, производимые по умолчанию для любого сигнала. Таким альтернативным вариантом может быть игнорирование сигнала или запуск определенной в приложении функции, называемой *обработчиком сигнала*. Процесс может в любое время указать новое действие либо, наоборот, сбросить установки на действия по умолчанию. Процесс вправе временно блокировать сигнал (не поддерживается в SVR2 и более ранних версиях этой системы). В таком случае сигнал не будет доставлен до тех пор, пока не будет разблокирован. Сигналы SIGKILL и SIGSTOP являются специальными, и приложения не могут игнорировать, блокировать или определять собственные обработчики для них. Полный список сигналов приведен в табл. 4.1, в которой также имеются ссылки на их действия по умолчанию и существующие ограничения.

¹ В системе 4.4BSD файл дампа имеет название *core.prog*, где *prog* — это первые 16 символов программы, выполнявшейся во время получения сигнала (такой подход более разумен, так как в случае аварийного завершения еще какого-либо процесса, имеющего того же владельца, файл *core* не затрется новым). — Прим. ред.

Таблица 4.1. Сигналы UNIX

Сигнал	Описание	Действие по умолчанию	Доступен в ¹	Примечания ²
SIGABRT	Процесс аварийно завершен	Аварийное завершение	APSБ	
SIGNALRM	Сигнал тревоги реального времени	Выход	OPSБ	
SIGBUS	Ошибка шины	Аварийное завершение	OSБ	
SIGCHLD	Потомок завершил работу или приостановлен	Игнорирование	OJSБ	6
SIGCONT	Возобновить приостановленный процесс	Продолжение/игнорирование	JSБ	4
SIGEMT	Ловушка эмулятора	Аварийное завершение	OSБ	
SIGFPE	Арифметическая ошибка	Аварийное завершение	OAPSБ	2
SIGHUP	Освобождение линии терминала	Выход	OPSБ	
SIGILL	Выполнение недопустимой инструкции	Аварийное завершение	OAPSБ	
SIGINFO	Запрос состояния (Ctrl+T)	Игнорирование	В	
SIGINT	Прерывание терминала (Ctrl+C)	Выход	OAPSБ	
SIGIO	Асинхронное событие ввода-вывода	Выход/игнорирование	SB	3
SIGIOT	Ловушка ввода-вывода	Аварийное завершение	OSБ	
SIGKILL	Завершить процесс	Выход	OPSБ	1
SIGPIPE	Запись в канал при отсутствии считающих процессов	Выход	OPSБ	
SIGPOLL	Событие опрашиваемого устройства	Выход	С	
SIGPROF	Профилирование таймера	Выход	SB	
SIGPWR	Неполадки питания	Игнорирование	OS	
SIGQUIT	Сигнал выхода из терминала (Ctrl+\)	Аварийное завершение	OPSБ	

Сигнал	Описание	Действие по умолчанию	Доступен в ¹	Примечания ²
SIGSEGV	Ошибка сегментации	Аварийное завершение	OAPSB	
SIGSTOP	Остановить процесс	Остановка	JSB	1
SIGSYS	Неверный системный вызов	Выход	OAPSB	
SIGTERM	Завершить процесс	Выход	OAPSB	
SIGTRAP	Аппаратная ошибка	Аварийное завершение	OSB	2
SIGTSTP	Сигнал остановки терминала (Ctrl+Z)	Остановка	JSB	
SIGTTIN	Чтение из терминала фоновым процессом	Остановка	JSB	
SIGTTOU	Запись в терминал фоновым процессом	Остановка	JSB	5
SIGURG	Экстренное событие канала ввода-вывода	Игнорирование	SB	
SIGUSR1	Определяется произвольно	Выход	OPSB	
SIGUSR2	Определяется произвольно	Выход	OPSB	
SIGVTALRM	Сигнал тревоги виртуального времени	Выход	SB	
SIGWINCH	Изменение размера окна	Игнорирование	SB	
SIGXCPU	Превышение лимита процессора	Аварийное завершение	SB	
SIGXFSZ	Превышение лимита размера файла	Аварийное завершение	SB	

¹ Обозначения: О — сигнал системы SVR2; А — ANSI C; В — 4.3BSD; С — SVR4; Р — POSIX.1; І — POSIX.1, при условии поддержки управления заданиями.

² 1 — не может быть перехвачен, блокирован или игнорирован; 2 — не сбрасывается в значение по умолчанию, даже в реализациях System V; 3 — в SVR4 действие по умолчанию — выход, в 4.3BSD — игнорирование; 4 — по умолчанию выполнение процесса, если тот был приостановлен, в ином случае сигнал игнорируется; 5 — не может быть заблокирован; 6 — процесс может решить: позволить запись в терминал фоновым процессам без генерации этого сигнала или нет; 7 — в SVR3 и более ранних версиях системы назывался SIGCLD.

Важно отметить, что любое действие, в том числе и завершение работы, относится только к тому процессу, которому был доставлен сигнал. Такой подход требует, по крайней мере, чтобы процесс был назначен в текущий момент на выполнение. В загруженных системах для процесса, обладающего низким приоритетом, ожидание в очереди планировщика может занять некоторое значительное количество времени. Еще одна задержка может произой-

ти, если процесс окажется выгруженным, приостановленным или блокированным без возможности прерывания.

Ядро системы предупреждает процесс о наличии ожидающих его сигналов при помощи вызова функции `issig()`, который делается от имени процесса для проверки ожидающих сигналов. Вызов функции `issig()` происходит только в следующих случаях:

- ◆ до возвращения в режим задачи после системного вызова или прерывания;
- ◆ сразу перед блокированием на прерываемом событии;
- ◆ немедленно после пробуждения от прерываемого события.

Если функция `issig()` возвращает значение `TRUE`, то ядро вызовет функцию `psig()` для диспетчеризации сигнала. Эта функция завершит процесс, создаст файл `core` (по необходимости) или же вызовет `sendsig()` для запуска обработчика, определенного в приложении. Функция `sendsig()` возвращает процесс в режим задачи, передает управление обработчику сигнала и указывает процессу на необходимость продолжения выполнения прерванного кода после завершения функционирования обработчика. Реализация этих функций сильно зависит от платформы, так как они должны манипулировать стеком приложения, а также сохранять, загружать и изменять контекст процесса.

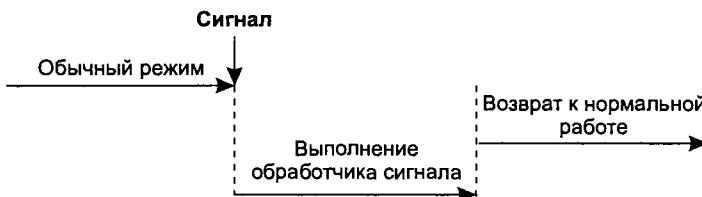


Рис. 4.1. Обработка сигналов

Сигналы вырабатываются вследствие происхождения асинхронных событий, которые могут произойти после любой инструкции в области кода процесса. После завершения обработки сигнала процесс восстанавливает свое функционирование с того места, где его выполнение было прервано сигналом (рис. 4.1). Если сигнал приходит тогда, когда процесс находится в стадии выполнения системного вызова, ядро системы прерывает обработку вызова и возвращает ошибку `EINTR`. В ОС 4.2BSD был введен механизм автоматического рестарта системного вызова после сигнала (см. раздел 4.4.3). Система 4.3BSD предлагает вызов `siginterrupt`, отключающий эту возможность для конкретного сигнала.

4.2.2. Генерирование сигналов

Ядро системы вырабатывает сигналы для процессов в ответ на различные события, причиной возникновения которых может быть сам процесс, получаю-

щий сигнал, другой процесс, а также прерывания или внешние действия. Основными источниками сигналов являются:

- ◆ **исключительные состояния** (exceptions). После возникновения в процессе такого состояния (например, при попытке выполнения недопустимой инструкции) ядро системы уведомляет об этом процесс при помощи сигнала;
- ◆ **другие процессы** (other processes). Процесс может отправлять сигналы другому процессу или набору процессов при помощи системных вызовов `kill` или `sigsend`. Процесс может послать сигнал даже самому себе;
- ◆ **прерывания от терминала** (terminal interrupts). При нажатии определенных комбинаций клавиш, например `Ctrl+C` или `Ctrl+\`, *текущему (интерактивному)* процессу терминала отправляется сигнал. Команда `stty` позволяет пользователю назначать каждому сигналу, создаваемому терминалом, определенные клавиши;
- ◆ **управление заданиями** (job control). Фоновые процессы, желающие произвести операции чтения или записи с терминалом, посыпают сигналы управления заданиями. Командные интерпретаторы, поддерживающие управление заданиями, такие как `csh` или `ksh`, используют сигналы для управления текущими и фоновыми процессами. Когда процесс завершается или приостанавливается, ядро системы уведомляет об этом его родителя посредством сигнала;
- ◆ **квоты** (quotas). Если процесс превысит отведенные ему лимиты использования процессора или допустимого размера файла, ядро пошлет такому процессу сигнал;
- ◆ **уведомления** (notifications). Процессу может потребоваться уведомление о возникновении определенных событий, например готовности устройства на ввод-вывод. Ядро информирует процесс о таких событиях при помощи сигнала;
- ◆ **будильники** (alarms). Процесс может установить будильник на определенное время. Когда отведенное время истечет, ядро предупредит об этом процесс подачей сигнала тревоги.

Существуют три различных типа будильников, которые используют различные типы отсчетов. Так, отсчет `ITIMER_REAL` приводит к измерению реального времени и вырабатыванию будильником сигнала `SIGALRM`. При указании `ITIMER_VIRTUAL` измеряется виртуальное время, то есть тот промежуток, когда процесс находился в режиме задачи, и подается сигнал `SIGVTALRM`. С `ITIMER_PROF` ведется отсчет общего времени, используемого процессом как в режиме задачи, так и в режиме ядра, при этом генерируется сигнал `SIGPROF`.

Реализация будильников и отсчетов различается у разных производителей операционных систем.

4.2.3. Типичные примеры возникновения сигналов

Разберем несколько примеров генерирования и доставки сигналов. Представьте, что пользователь нажимает комбинацию клавиш `Ctrl+C` на своем терминале. Это действие приводит к возникновению прерывания терминала (точно так же, как и ввод любого иного символа с клавиатуры). Драйвер терминала распознает комбинацию как символы, по вводу которых вырабатывается определенный сигнал, после чего отправляет созданный им сигнал `SIGINT` текущему процессу данного терминала (если текущее *задание* состоит из более чем одного процесса, драйвер пошлет сигнал каждому его процессу). Когда этот процесс будет выбран планировщиком на выполнение, то он увидит сигнал, пытающийся вернуться в режим задачи после переключения контекста. Иногда интерактивный процесс уже является *текущим* во время возникновения прерывания. В таком случае обработчик прерываний прервет его выполнение и отправит процессу сигнал¹. После возврата из прерывания процесс осуществит проверку на наличие ожидающих сигналов и обнаружит этот сигнал.

Однако *исключительные состояния* (или *исключения*) обычно приводят к возникновению синхронных сигналов. Причиной их часто являются ошибки в программе (например, попытка деления на ноль, недопустимые инструкции и т. д.), которые будут возникать в том же самом месте программы в случае, если она будет запущена с тождественными условиями (то есть если повторяется та же ветвь кода с теми же данными). Когда в программе происходит исключение, оно является причиной возникновения *ловушки* в режиме ядра. Обработчик ловушки, находящийся внутри ядра, распознает исключительное состояние и отправляет соответствующий сигнал текущему процессу. Перед возвратом в режим задачи обработчик ловушки вызывает функцию `issig()` для получения процессом сигнала.

Возможна ситуация, когда процесс одновременно ожидают несколько сигналов. В таком случае все сигналы будут обрабатываться по одному. Сигнал также может прийти в момент выполнения обработчика другого сигнала, что может стать причиной наложения обработчиков. В большинстве реализаций систем UNIX приложение имеет право запрашивать ядро о выборочной блокировке определенных сигналов перед запуском определенного обработчика (см. подробнее в разделе 4.4.3). Такой подход позволяет исключить или контролировать наложение обработчиков сигналов.

¹ На многопроцессорных системах процесс может оказаться функционирующим на другом процессоре. В таком случае обработчик должен создать специальное межпроцессорное прерывание для доставки сигнала его приемнику.

4.2.4. Спящие процессы и сигналы

Что происходит в случае, если спящий процесс получит сигнал? Должен ли он быть разбужен для обработки этого сигнала или сигнал будет ожидать того момента, когда процесс выйдет из режима сна?

Ответ на эти вопросы зависит от того, что стало причиной перехода процесса в режим сна. Если процесс спит в ожидании возникновения такого события, как завершение ввода-вывода с диска, то есть события, которое может наступить вскоре, имеет смысл немного подождать с доставкой сигналов. С другой стороны, если процесс ожидает ввода символа с клавиатуры, то такое ожидание может продолжаться очень долго. Нам необходимо определиться, как прерывать такие процессы сигналами.

Система UNIX поддерживает два вида сна: прерываемый и непрерываемый. Процесс, находящийся в состоянии сна до события, которое может произойти в ближайшее время (например, завершение дискового ввода-вывода), пребывает в непрерываемом сне и не может быть побеспокоен поступающими сигналами. Процесс, ожидающий такого события, как ввод-вывод терминала, который может не произойти в течение продолжительного времени, находится в прерываемом состоянии сна и будет разбужен посланным ему сигналом.

Если сигнал предназначается для процесса, находящегося в непрерываемом сне, то такой сигнал будет помечен как ожидающий, и больше никаких действий со стороны этого сигнала по отношению к процессу происходить не будет. Процесс не будет уведомлен о сигнале и после выхода из состояния сна до тех пор, пока он не окажется в состоянии возврата в режим задачи или не будет блокирован по прерываемому событию.

Если процесс находится в состоянии, когда он будет заблокирован по прерываемому событию, то перед этой процедурой он проверит наличие сигналов. Если такие будут найдены, процесс обработает их и прервет выполнение системного вызова. Если сигнал будет создан уже после того, как процесс был заблокирован, ядро системы разбудит такой процесс. Пробуждение и начало работы процесса может быть вызвано двумя причинами: либо произошло ожидаемое событие, либо его сон был прерван сигналом, поэтому в первую очередь процесс вызовет функцию `issig()` и проверит, есть ли для него сигналы. В случае когда сигнал ожидает процесс, после вызова `issig()` всегда происходит еще один вызов `psig()`, как это показано ниже:

```
if (issig())
    psig();
```

4.3. Ненадежные сигналы

Изначальная реализация сигналов (в системах SVR2 и более ранних версиях) была ненадежной и малоэффективной [2]. Эта реализация придерживается базовой модели, описанной в предыдущем разделе, и обладает рядом недостатков.

Наиболее важной проблемой является надежность доставки сигналов. Обработчики сигналов не являются постоянно установленными и не маскируют повторения одного и того же сигнала. Представьте, что программист установил обработчик на определенный сигнал. После возникновения этого сигнала ядро системы сбросит действия, определенные для него, на установки по умолчанию перед тем, как вызовет обработчик. Программа, желающая обработать все повторные сигналы, должна каждый раз переустанавливать обработчик, как это показано в листинге 4.1.

Листинг 4.1. Переустановка обработчика сигналов

```
void sigint_handler (sig)
int sig:
{
    signal (SIGINT, sigint_handler); /* переустановка обработчика */
    /* обработка сигнала*/
}
main()
{
    signal (SIGINT, sigint_handler); /* установка обработчика */
}
```

Однако этот подход приводит к состоянию состязательности. Представьте, что пользователь дважды быстро нажимает комбинацию клавиш **Ctrl+C**. Первое нажатие влечет за собой возникновение сигнала **SIGINT**, действие которого сбрасывается на принятое по умолчанию, после чего запустится обработчик. Если повторное нажатие произошло до переустановки обработчика, ядро системы выполнит действие по умолчанию и завершит процесс. Такой подход приводит к существованию определенного промежутка времени между запуском и переустановкой обработчика, в течение которого сигнал не может быть перехвачен. По этой причине говорят, что ранние реализации **UNIX** имеют *ненадежные сигналы*.

Существует также и проблема производительности, относящаяся к спящим процессам. В ранних реализациях вся информация, относящаяся к диспозиции сигналов¹, сохраняется в массиве **u_signal[]**, который содержит одну запись о каждом типе сигналов. Массив располагается в области **u**. Элемент этого массива содержит адрес обработчика, определенного в приложении, а также параметр **SIG_DFL**, в котором регламентируется действие по умолчанию, или используется **SIG_IGN** для указания на необходимость игнорирования сигнала.

Так как ядро может считывать данные только из области и текущего процесса, оно не знает, каким образом другой процесс должен распорядиться сигналом. Более того, если ядру системы необходимо послать сигнал про-

¹ То есть поведению при их получении. — *Прим. ред.*

цессу, находящемуся в состоянии прерываемого сна, оно не может знать, игнорирует процесс такой сигнал или нет. Таким образом оно отправит сигнал и разбудит тем самым процесс, предполагая, что тот обработает сигнал. Если процесс обнаружит, что он разбужен сигналом, который им игнорируется, он просто снова заснет. Такие совершенно излишние пробуждения приводят в результате к совершенно ненужным переключениям контекста и потере времени на обработку сигналов. Очевидно, что лучшим вариантом представляется распознавание сигналов ядром иброс тех из них, которые должны игнорироваться, без необходимости участия в этом процесса.

В завершение скажем, что система SVR2 не обладает средствами временного блокирования сигналов для задержки их доставки до тех пор, пока они не будут разблокированы. Эта система также не имеет поддержки управления заданиями, при которой группы процессов могут быть приостановлены и возобновлены для того, чтобы получить доступ к терминалу.

4.4. Надежные сигналы

Проблемы, озвученные в предыдущем разделе, были впервые решены в системе 4.2BSD, в которой был представлен управляющий механизм надежных и гибких в обработке сигналов. В ОС 4.3BSD были сделаны дополнительные усовершенствования, но базовые средства остались неизменными. Между тем компания AT&T представила собственную версию надежных сигналов в своей системе SVR3 [1]. Эта версия была несовместима с интерфейсом BSD и оказалась не настолько развитой. Разработчики сохранили в реализации SVR3 совместимость с исходным механизмом сигналов, который был в SVR2. В обеих системах, 4.2BSD и SVR3, была предпринята попытка решения одних и те же проблем разными способами. В итоге каждая из этих ОС обладает собственным набором системных вызовов, используемых для доступа к средствам управления сигналами. Эти вызовы имеют как различные имена, так и отличающуюся друг от друга семантику.

Стандарт POSIX.1 явился попыткой навести порядок в имеющемся хаосе, определив стандартный набор функций, которые должны быть реализованы во всех системах, претендующих на совместимость с ним. Функции согласно этому стандарту могут быть реализованы как в виде системных вызовов, так и в виде библиотечных процедур. На основе требований POSIX в системе SVR4 был представлен новый интерфейс, который удовлетворял стандарту POSIX и оказался совместим с BSD и со всеми предыдущими версиями UNIX, созданными ранее компанией AT&T.

Этот раздел начинается с описания основных возможностей механизма надежных сигналов. Затем вы увидите краткое описание интерфейсов систем SVR3 и 4.3BSD, а в конце раздела вы познакомитесь с подробным изложением сигнального интерфейса ОС SVR4.

4.4.1. Основные возможности

Все реализации механизма надежных сигналов обладают некоторыми общими возможностями, перечисленными ниже.

- ◆ **Постоянно установленные обработчики** (persistent handlers). Обработчики сигналов остаются установленными даже после возникновения сигнала и не требуют дополнительных переустановок. Такой подход защищает от существования временного интервала между запуском обработчика сигнала и его переустановкой, во время которого повторно поступивший сигнал может завершить процесс.
- ◆ **Маскирование** (masking). Сигнал может быть временно маскирован (слова «блокирован» и «маскирован» являются синонимами и взаимозаменямы при разговоре о сигналах). Если вырабатывается сигнал, который уже блокирован процессом, ядро системы будет помнить о этом и не станет посыпать такой сигнал процессу незамедлительно. Сигнал будет переправлен и обработан после того, как процесс разблокирует его. Такой подход дает возможность программисту защитить критические области кода от прерывания его выполнения при возникновении определенных сигналов.
- ◆ **Спящие процессы** (sleeping process). Некоторая информация о диспозиции сигналов в процессе является видимой для ядра (посредством хранения данных в структуре `pgos` вместо области `u`), даже если процесс не находится в текущий момент на выполнении. Следовательно, если ядро генерирует сигнал для процесса, находящегося в прерываемом сне, но тот игнорирует или блокирует данный сигнал, то ядро не станет будить такой процесс.
- ◆ **Разблокирование и ожидание** (unblock and wait). Системный вызов `pause` блокирует процесс до тех пор, пока ему не будет доставлен сигнал. Механизм надежных сигналов предлагает еще один вызов, `sigpause`, который атомарно демаскирует сигнал и блокирует процесс до тех пор, пока тот не получит такой сигнал. Если демаскированный сигнал уже находится в ожидании, то произойдет немедленный возврат из этого системного вызова.

4.4.2. Сигналы в системе SVR3

Система SVR3 поддерживает все возможности, описанные в предыдущем разделе. Однако реализация механизма сигналов в этой ОС имеет некоторые недостатки. Проиллюстрируем это на примере с использованием системного вызова `sigpause`.

Представьте, что процессом объявлен обработчик, перехватывающий сигнал `SIGQUIT` и устанавливающий глобальный флаг при его захвате. Процесс

единожды проверяет флаг и, если тот не установлен, ждет его установки. Проверка и последующее ожидание вместе представляют собой *критический участок* кода: если сигнал будет доставлен после проведения проверки, но до начала ожидания, он будет потерян¹ и, следовательно, процесс может ожидать сигнал вечно. Таким образом, процессу необходимо маскировать сигнал SIGQUIT на время проверки флага. Но если процесс войдет в режим ожидания с маскированным сигналом, то в таком случае сигнал никогда не будет доставлен. Следовательно, нам необходим некий атомарный вызов, который бы демаскировал сигнал и блокировал процесс в ожидании. Эту функцию обеспечивает системный вызов `sigpause`. Пример кода, работающего в SVR3, представлен в листинге 4.2.

Листинг 4.2. Использование `sigpause` для ожидания сигнала

```
int sig_received = 0;
void handler(int sig)
{
    sig_received++;
}
main()
{
    sigset(SIGQUIT, handler);
    ...
/* ждем сигнала, если он уже не находится в режиме ожидания*/
    sighold(SIGQUIT);
    while (sig_received==0) /* сигнал еще не доставлен */
        sigpause(SIGINT);2
    /* сигнал прибыл, обработка*/
    ...
}
```

Пример показывает некоторые возможности системы SVR3 по обработке сигналов. Вызовы `sighold` и `sigrelse` позволяют блокировать и разблокировать сигнал. Вызов `sigpause` атомарно деблокирует сигнал и переводит процесс в состояние сна до тех пор, пока тот не получит сигнал, который им не игнорируется и не заблокирован. Системный вызов `sigset` определяет постоянный обработчик, не сбрасываемый в действие по умолчанию после возникновения сигнала. Старый вызов `signal` остался в системе для обратной совместимости. Обработчики, задаваемые при помощи этого вызова, не являются постоянными.

Такой интерфейс обладает некоторыми недостатками [9]. Важно то, что системные вызовы `sighold`, `sigrelse` и `sigpause` могут работать только с одним

¹ Так как и флаг установится уже после его проверки. — Прим. ред.

² В строке, видимо, ошибка, и вместо `SIGINT` должно быть `SIGQUIT`, иначе незачем вызывать `sighold` на `SIGQUIT`, ибо это приведет к блокированию обработчика `SIGQUIT` и, следовательно, флаг никогда не изменит своего значения, что, в свою очередь, не даст выйти из цикла. — Прим. ред.

сигналом в один момент времени. Не существует способа атомарного блокирования или деблокирования нескольких сигналов одновременно. Если обработчик, показанный в листинге 4.2, будет использован несколькими сигналами сразу, не существует приемлемого способа программно выделить критическую область. Мы можем блокировать сигналы по одному в один момент времени, но вызов `sigpause` не сумеет атомарно разблокировать все эти сигналы и далее перевести процесс в состояние ожидания.

В системе SVR3 также отсутствует управление заданиями и такие возможности, как автоматический перезапуск системных вызовов. Такие возможности (наряду с некоторыми другими) представлены в ОС 4BSD.

4.4.3. Механизм сигналов в BSD

Впервые механизм надежных сигналов был представлен в ОС 4.2BSD. Возможности, предлагаемые механизмом сигналов в системе BSD [7], являются более развитыми, чем аналогичные возможности, представленные в ОС SVR3. В большинстве системных вызовов одним из входных аргументов передается 32-битовая маска сигналов, биты которой отображают, с какими из сигналов будет оперировать функция (по одному биту на каждый сигнал). Такой подход позволяет одному системному вызову работать сразу с несколькими сигналами. Вызов `sigsetmask` используется для указания набора блокируемых сигналов. Вызов `sigblock` добавляет в этот набор один или несколько дополнительных сигналов. Реализация вызова `sigpause` в системе BSD атомарно устанавливает новую маску блокируемых сигналов и переводит процесс в состояние сна до прихода сигнала.

Системный вызов `sigvec` заменил собою `signal`. Точно так же, как и `signal`, `sigvec` устанавливает обработчик для одного из сигналов. Дополнительно вызов `sigvec` может задавать маску, ассоциируемую с этим сигналом. После вырабатывания такого сигнала ядро системы перед вызовом обработчика установит новую маску блокируемых сигналов, являющуюся объединением текущей маски, маски, заданной в `sigvec`, и текущего сигнала.

Таким образом, обработчик всегда запускается, когда текущий сигнал блокирован, — следовательно, повторяющийся сигнал не будет доставлен до тех пор, пока обработчик не завершит свою работу. Такое устройство сигнального механизма наиболее соответствует типичным сценариям вызова обработчиков сигналов. Блокирование дополнительных сигналов, вырабатываемых во время функционирования обработчика, является весьма необходимой функцией, так как сами по себе обработчики сигналов обычно являются критическими участками кода. После возврата из обработчика происходит восстановление маски блокированных сигналов в ее предыдущее значение.

Еще одной важной возможностью является обработка сигналов в отдельном стеке. Представьте, что процесс управляет своим собственным стеком. Он может установить обработчик для сигнала `SIGSEGV`, вырабатываемого при

переполнении стека. В обычной ситуации обработчик запустится с тем же, уже переполненным стеком, что приведет к вырабатыванию еще одного сигнала SIGSEGV. Если обработчик будет стартовать, используя при этом отдельный стек, то проблема не возникнет. Отдельный стек для сигнала также полезен и для нитевых библиотек прикладного уровня. Системный вызов `sigstack` задает отдельный стек, который будет использоваться обработчиком сигнала. Ответственность за правильное указание размера такого стека лежит на разработчике, поскольку ядро системы ничего не знает о его границах.

В ОС BSD представлено несколько дополнительных сигналов, в том числе специально выделенных для управления заданиями¹. Задание – это группа связанных между собой процессов, обычно формирующих единый конвейер. Пользователь может выполнять несколько заданий одновременно из одного сеанса терминала, но только один из них будет текущим. Текущему заданию позволено писать в терминал и считывать с него. Фоновым заданиям, пытающимся получить доступ к терминалу, посылаются сигналы, которые обычно приостанавливают процесс. Командные интерпретаторы Korn shell (`ksh`) и C shell (`csh`) [6] используют сигналы управления заданиями для манипулирования заданиями, посылая эти сигналы текущим и фоновым заданиям, приостанавливая и возобновляя их работу. Более подробно об управлении заданиями будет рассказано в разделе 4.9.1.

В заключение отметим, что система 4BSD позволяет автоматически перезапускать медленные системные вызовы, выполнение которых было прервано сигналами. Медленные вызовы включают в себя функции `read` и `write`, осуществляющие чтение и запись на символьные устройства, а также сетевые соединения, каналы, вызовы `wait`, `waitpid`, `ioctl`. Если один из этих вызовов прерывается сигналом, происходит его автоматический перезапуск после возврата из обработчика сигнала вместо прерывания работы с ошибкой `EINTR`. В системе 4.3BSD добавлен вызов `siginterrupt`, позволяющий выборочно разрешить или запретить такую возможность для каждого сигнала отдельно.

Интерфейс сигналов системы BSD является весьма гибким и мощным. Его основным недостатком остается несовместимость с оригинальным интерфейсом систем корпорации AT&T (и даже с вариантом, представленным в SVR3, хотя эта система была создана позже). Это дало возможность сторонним производителям предлагать различные библиотечные интерфейсы, которые пытались удовлетворить приверженцев обеих ветвей генеалогического дерева UNIX. Позже в системе SVR4 был представлен интерфейс, совместимый со стандартом POSIX и при этом обладающий совместимостью с предыдущими реализациями System V и семантикой, принятой в ОС BSD.

¹ Поддержка управления заданиями впервые появилась в 4.1BSD.

4.5. Сигналы в SVR4

Операционная система SVR4 предлагает набор системных вызовов [11], которые обеспечивают универсальные возможности обработки сигналов как ОС SVR3, так и BSD, а также поддерживается устаревший механизм ненадежных сигналов. Ниже представлены основные функции SVR4 для работы с сигналами.

- ◆ `sigprocmask(how, setp, osetp);`
- ◆ Аргумент `setp` используется для изменения маски блокируемых сигналов. Если `how` имеет значение `SIG_BLOCK`, то маска `setp` объединяется операцией ИЛИ с существующей. Если `how` определено как `SIG_UNBLOCK`, то сигналы, заданные в `setp`, деблокируются по существующей маске блокированных сигналов. Если `how` равняется `SIG_SETMASK`, то происходит замена текущей маски на набор, определенный в `setp`. При возврате из функции `osetp` содержит значение маски перед модификацией.
- ◆ `sigaltstack(stack, old_stack);`
- ◆ Задает новый стек `stack` для обработки сигналов. Альтернативный стек (если он требуется) необходимо определять перед установкой обработчика. Остальные обработчики используют стек, заданный по умолчанию. После возврата из функции в переменной `old_stack` содержится указатель на предыдущий альтернативный стек.
- ◆ `sigsuspend(sigmask);`
- ◆ Устанавливает маску блокируемых сигналов в значение `sigmask` и переводит процесс в состояние сна до тех пор, пока этому процессу не будет отправлен сигнал, который не игнорируется и не заблокирован. Если такой сигнал был послан и при изменении маски он будет разблокирован, произойдет немедленный выход из функции.
- ◆ `sigpending(setp);`
- ◆ Возвращает в `setp` набор сигналов, ожидающих процесс. Вызов не производит никаких изменений в состоянии сигналов и используется только для получения информации.
- ◆ `sigsendset(procset, sig);`
- ◆ Расширенная версия `kill`. Посыпает сигнал `sig` набору процессов, заданных в `procset`.
- ◆ `sigaction(signo, act, oact);`
- ◆ Определяет обработчик для сигнала `signo`. Является аналогом вызова `sigvec` в ОС BSD. Аргумент `act` указывает на структуру `sigaction`, содержащую диспозицию сигналов (`SIG_IGN`, `SIG_DFL` или адрес обработчи-

ка), маску, ассоциированную с сигналом (аналогичную маске вызова `sigvec`), а также один или несколько следующих флагов:

<code>SA_NOCLDSTOP</code>	Не генерировать сигнал <code>SIGCHLD</code> , когда процесс-потомок приостановлен
<code>SA_RESTART</code>	Автоматический рестарт системного вызова при прерывании его сигналом
<code>SA_ONSTACK</code>	Обработка сигнала с альтернативным стеком, если такой стек был указан через <code>sigaltstack</code>
<code>SA_NOCLDWAIT</code>	Используется только с <code>SIGCHLD</code> . Просит систему не создавать процессы-зомби, если потомки вызывающего процесса завершают свою работу. Если процесс далее вызовет <code>wait</code> , то он будет находиться в режиме ожидания до тех пор, пока не завершат работу все его потомки
<code>SA_SIGINFO</code>	Обеспечивает дополнительную информацию для обработчика сигнала. Используется для обработки аппаратных исключений и т. д.
<code>SA_NODEFER</code>	Позволяет не блокировать автоматически сигнал в течение выполнения его обработчика
<code>SA_RESETHAND</code>	Сбрасывает действия на заданные по умолчанию перед вызовом обработчика

- ◆ Флаги `SA_NODEFER` и `SA_RESETHAND` используются для совместимости с изначальной реализацией механизма ненадежных сигналов. Во всех случаях переменная `oact` возвращает данные, установленные перед вызовом `sigaction`.
- ◆ *Интерфейс совместимости*
- ◆ Для обеспечения совместимости с предыдущими версиями системы SVR4 также поддерживает вызовы `signal`, `sigset`, `sighold`, `sigrelse`, `sigignore` и `sigpause`. Системы, не требующие совместимости на бинарном уровне, могут реализовывать эти вызовы в виде библиотечных функций.

Все перечисленные системные вызовы (кроме указанных в последнем пункте списка) полностью удовлетворяют стандарту POSIX.1 по имени, передаваемым параметрам и семантике.

4.6. Реализация сигналов

Для эффективной реализации сигналов ядру необходимо содержать некоторое состояние в области `i` и в структуре `proc`. В этом разделе описывается реализация сигналов в системе SVR4, которая отличается от аналогичного набора ОС BSD именами некоторых переменных и функций. Область `i` содержит информацию, требующуюся для правильного запуска обработчиков сигналов, которую составляют нижеперечисленные поля области `i`.

<code>u_signal[]</code>	Вектор обработчиков для каждого сигнала
<code>u_sigmask[]</code>	Маски сигналов, ассоциированные с каждым обработчиком
<code>u_sigaltstack</code>	Указатель на альтернативный стек сигнала
<code>u_sigonstack</code>	Маска сигналов, обрабатываемых с альтернативным стеком
<code>u_oldsig</code>	Набор обработчиков, который должен имитировать устаревший механизм ненадежных сигналов

Структура `proc` содержит определенные поля, относящиеся к созданию и отправке сигналов, в том числе:

<code>p_cursig</code>	Текущий сигнал, в данный момент обрабатываемый
<code>p_sig</code>	Маска ожидающих сигналов
<code>p_hold</code>	Маска блокируемых сигналов
<code>p_ignore</code>	Маска игнорируемых сигналов

Рассмотрим далее реализации на уровне ядра различных функций, относящихся к доставке сигналов.

4.6.1. Генерация сигналов

После вырабатывания сигнала ядро проверяет структуру `proc` процесса, которому этот сигнал предназначен. Если сигнал необходимо проигнорировать, то ядро на этом завершит обработку, не предпринимая никаких действий. В иных случаях он будет добавлен в набор ожидающих сигналов, расположенный в поле `p_cursig`¹. Так как `p_cursig`² является всего лишь битовой маской, где каждому сигналу отводится только один бит, ядро не может записать в нее несколько повторных экземпляров одного сигнала. Следовательно, процесс будет знать только об одном приходе ожидающего сигнала.

Если процесс находится в прерываемом сне и сигнал не заблокирован, то ядро системы разбудит такой процесс для получения сигнала. Более того, сигналы управления заданиями, такие как `SIGSTOP` или `SIGCONT`, напрямую приостанавливают или продолжают функционирование процесса без проведения их доставки.

4.6.2. Доставка и обработка

Процесс ведет проверку на наличие сигналов при помощи функции `issig()`, вызываемой после обработки системного вызова или прерывания, но перед возвращением из режима ядра. Вызов `issig()` производится также и в случаях

¹ Здесь, видимо, ошибка, и имеется в виду все-таки поле `p_sig`. — Прим. ред.

² Опять же, должно быть, не `p_cursig`, а `p_sig`. — Прим. ред.

перехода в режим прерываемого сна или выхода из него (то есть пробуждения). Функция `issig()` считывает установленные биты в поле `p_cursig`¹. Если какой-либо бит установлен, то эта функция проверяет `p_hold` на предмет существования блокировки этого сигнала. Если сигнал не блокируется, то `issig()` сохраняет номер сигнала в `p_sig`² и возвращает значение TRUE.

Когда сигнал является ожидающим, ядро вызывает для его обработки `psig()`. Функция `psig()` проверяет информацию области `u`, относящуюся к этому сигналу. Если не задан ни один обработчик, произойдет действие по умолчанию, обычно это завершение процесса. Если необходимо вызвать обработчик, то происходит изменение маски блокируемых сигналов `p_hold`, в которую добавляется текущий сигнал, а также любой другой сигнал, ассоциируемый с ним в маске `u_sigmask`. Текущий сигнал не добавляется в маску в том случае, если для его обработчика был установлен флаг `SA_NODEFER`. Точно так же, если был установлен флаг `SA_RESETHAND`, действия, задаваемые в массиве `u_signal[]`, сбрасываются на установленные по умолчанию.

Последним этапом действий является вызов `sendsig()`, производимый функцией `psig()`, который заставляет процесс вернуться в режим задачи и передать управление обработчику. Вызов `sendsig()` гарантирует, что после завершения работы обработчика процесс восстановит свое выполнение с того места программы, на котором он был прерван сигналом. Если используется альтернативный стек, вызов `sendsig()` загрузит обработчик с указанным стеком. Реализация функции `sendsig()` является машинно-зависимой, так как ей необходимо знать о деталях работы со стеком и о манипуляциях с контекстом.

4.7. Исключительные состояния

Исключительные³ состояния (исключения) возникают, когда программа оказывается в необычной ситуации, и происходит это, как правило, вследствие ошибки. Например, попытка обращения по несуществующему адресу в памяти или деление на ноль повлекут за собой исключение. При возникновении исключения срабатывает ловушка в ядре, которая вырабатывает сигнал, уведомляющий процесс о возникшем исключении.

В системе UNIX для оповещения процесса об исключениях служат сигналы. Тип вырабатываемого сигнала зависит от природы исключения. Например, попытка обращения по несуществующему адресу может повлечь за собой генерацию сигнала `SIGSEGV`. Если в приложении указан для него обработчик, то ядро системы запустит его. Если нет, то выполняется действие по умолчанию (обычно это завершение работы процесса). Такой подход позволяет каждой

¹ Все-таки, наверное, в маске ожидающих сигналов `p_sig`. — Прим. ред.

² Вот тут, видимо, должно быть поле `p_cursig`. — Прим. ред.

³ Этот раздел описывает только аппаратные исключения. Их не следует путать с программными исключениями, поддерживаемыми различными языками программирования, такими как C++.

программе устанавливать свои собственные обработчики исключений. Некоторые языки программирования (например, Ада) обладают встроенными механизмами обработки исключений. Обработчики могут быть реализованы в библиотеках языка.

Исключения часто используются отладчиками. При отладке (или трассировке) программы вырабатывают исключения в точках останова, а также по завершении выполнения системного вызова `exec`. Отладчик должен перехватывать эти исключения для контроля над программой. Отладчику также может понадобиться перехватывать и другие выбранные им исключения и сигналы, вырабатываемые отлаживаемой программой. Системный вызов `ptrace` системы UNIX разрешает такой перехват (более подробно о его работе можно прочесть в разделе 6.2.4).

Существует ряд недостатков в том способе, которым UNIX обрабатывает исключения. Во-первых, *обработчик исключения запускается в том же контексте, в котором это исключение произошло*. Это означает, что обработчик не имеет доступа к полному контексту регистров, существовавшему во время возникновения исключения. Когда происходит исключение, ядро системы передает лишь некоторый объем его контекста обработчику. Этот объем зависит от конкретной реализации UNIX, а также от аппаратной платформы, на которой работает система. Вообще говоря, одна нить должна работать с двумя контекстами — с контекстом обработчика и тем контекстом, в котором произошло исключение.

Во-вторых, *механизм сигналов изначально разрабатывался для однонитевых процессов*. Системы UNIX, поддерживающие многонитевые процессы, столкнулись с определенными сложностями в адаптации подобной схемы сигналов. И последний недостаток, о котором необходимо упомянуть, это ограничение системного вызова `ptrace`, который *позволяет отладчику, основанному на нем, контролировать только своих непосредственных потомков*.

4.8. Обработка исключительных состояний в Mach

Ограничения механизмов обработки исключений в UNIX стали причиной разработки нового унифицированного средства, представленного в системе Mach [4]. Целью разработчиков ОС стало создание такого механизма, который, с одной стороны, оставался бы совместимым с традиционной UNIX на бинарном уровне, а с другой — поддерживал многонитевые приложения. Это средство стало не только частью описываемой системы, но и частью ОС OSF/1, основанной на Mach.

Разработчики системы Mach отказались от идеи запуска обработчика в том же контексте, в котором произошло исключение. В традиционной UNIX вариант одного контекста применялся только потому, что обработчику требовался доступ и функционирование в том же адресном пространстве, где про-

изошло исключение. Так как система Mach является многонитевой, обработчик может стартовать как отдельная нить той же задачи. (Нити и задачи ОС Mach подробнее описаны в разделе 6.4. Вкратце *задача* объединяет набор ресурсов, в том числе и адресное пространство. *Нить* выполняется внутри задачи и представлена контекстом выполнения и контрольной точкой. Традиционный процесс UNIX в Mach можно интерпретировать как задачу, содержащую единственную нить.)

В системе Mach выделяются два понятия: нить-«жертва» (нить, в которой произошло исключение) и ее обработчик. На рис. 4.2 показано взаимодействие между ними. Нить-жертва *устанавливает* исключение, уведомляя ядро о его возникновении. Затем она *ожидает* окончания функционирования обработчика этого исключения. Обработчик *перехватывает* возникшее исключение, получая уведомление от ядра. Это уведомление содержит информацию о нити-жертве и о типе исключения. Затем происходит обработка исключения и его *очистка*, что дает возможность нити-жертве продолжить свое функционирование. Если обработка исключения не смогла завершиться успешно, то нить, в которой оно возникло, будет завершена.

Описанные взаимодействия в чем-то похожи на поток управления исключениями UNIX, отличаясь от него тем, что обработчик функционирует как отдельная нить. В результате операции *установки*, *ожидания*, *перехвата* и *очистки* составляют вместе *удаленный вызов процедуры*, реализованный в системе Mach через средства взаимодействия процессов (IPC, см. о них подробнее в разделе 6.4).

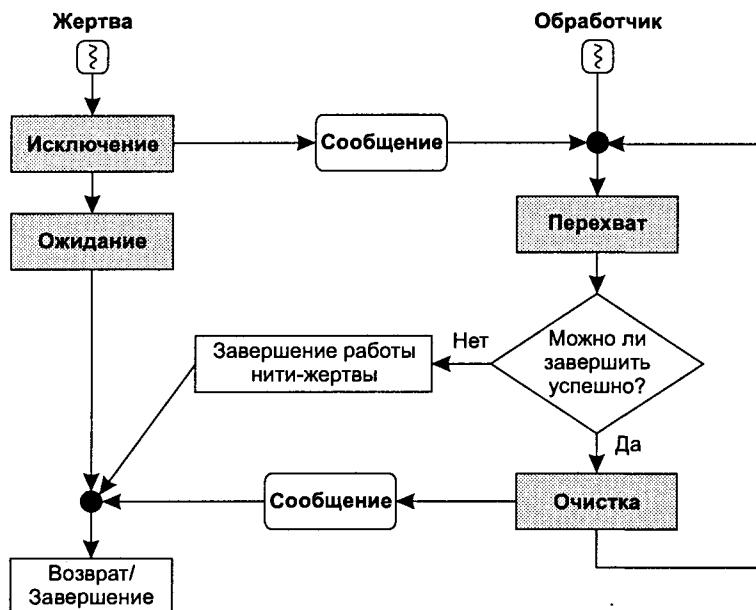


Рис. 4.2. Обработка исключительных состояний в системе Mach

При обработке каждого исключения создаются два сообщения. Когда нить-жертва устанавливает исключение, она посыпает сообщение обработчику и затем ожидает ответа. Обработчик перехватывает это исключение сразу после получения сообщения и очищает его при помощи отправки ответного сообщения нити-жертве. После получения ответного сообщения нить-жертва может продолжить свою работу.

4.8.1. Порты исключительных состояний

В операционной системе Mach сообщения отправляются в определенный *порт*, который представляет собой защищенную очередь сообщений. Несколько задач могут обладать *правом на отправку* сообщений в конкретный порт, но только одна задача имеет право получать сообщения из порта. В системе Mach каждой задаче, а также каждой нити задачи назначается по одному порту исключений. Это обеспечивает два способа обработки исключений, которые соответствуют двум вариантам применения исключений: обработке ошибок и отладке.

Обработчики ошибок ассоциируются с нитями, так как ошибки обычно влияют только на ту нить, в которой они породили исключение. Таким образом, каждая нить может иметь отличный от других нитей обработчик ошибки. Порт обработчика регистрируется как порт исключений нити. При создании новой нити ее порт исключительных состояний инициализируется в NULL, что означает, что нить изначально не имеет обработчика исключений.

Отладчик прикрепляется к задаче посредством регистрации одного из своих портов в качестве порта исключительных состояний задачи. Отладчик запускается в виде отдельной задачи и обладает *правами на получение* сообщений, отправленных в этот порт. Каждая задача наследует порт исключительных состояний от своего родителя. Это позволяет отладчикам контролировать всех потомков отлаживаемой задачи.

Так как исключение может использовать как порт исключений нити, так и порт исключений задачи, необходимо найти способ разрешения конфликта. Для этого необходимо заметить, что порт исключений нити используется обработчиками ошибок и будет прозрачным для отладчиков. Например, обработчик может ответить на ошибку потери значимости числа с плавающей точкой нулем как результатом операции. Такие исключения, как правило, не интересуют отладчик, который в обычных случаях применяется для перехвата только неисправимых ошибок. Таким образом, если в приложении установлен обработчик ошибок, то Mach при возникновении ошибки в приложении отдаст предпочтение в ее обслуживании обработчику, нежели отладчику.

Когда происходит исключение, оно отправляется в порт исключений нити, если таковой существует. То есть те исключения, для которых определен обработчик, являются невидимыми для отладчика. Если установленный обработчик не сможет успешно очистить исключение, то он перенаправит

его в порт исключений задачи. (Так как обработчик является одной из нитей той же задачи, что и нить-жертва, он имеет право доступа к порту исключений задания.) Если ни один из обработчиков не исправит ошибку, то ядро системы завершит работу нити-жертвы.

4.8.2. Обработка ошибок

Когда нить-жертва устанавливает исключение, в порт исключений самой нити либо задачи посыпается сообщение, содержащее адрес обратного порта, идентифицирующего нить и задачу, в которой произошло исключение, и тип исключения. После обработки исключения обработчик отправляет ответное сообщение обратно в указанный порт. Задача, в которой произошло исключение, обладает правами на получение сообщений из этого порта, и нить-жертва ожидает считывания из порта ответного сообщения. Когда сообщение приходит, нить-жертва получает его и восстанавливает свое выполнение в обычном режиме.

Так как обработчик и жертва являются нитями одной задачи, обработчик разделяет адресное пространство нити-жертвы. Он также может иметь доступ и к контексту регистров нити-жертвы, используя для этого вызовы `thread_get_state` и `thread_set_state`.

Система Mach поддерживает совместимость с UNIX, поэтому обработчики сигналов должны в ней запускаться в том же контексте, что и нить, в которой возникло исключение. Такой подход противоположен философии Mach, согласно которой для обработки ошибок запускается отдельная нить. В Mach эта разница была сглажена за счет использования инициируемого системой обработчика. Когда происходит исключение, для которого установлен обработчик UNIX-сигнала, в особый инициируемый системой обработчик посыпается сообщение. Этот обработчик изменяет нить-жертву так, что обработчик сигнала исполняется, когда нить-жертва восстанавливает свое выполнение. Он очищает исключение, приведшее к возникновению сигнала. За коррекцию стека после завершения работы обработчика сигнала отвечает приложение.

4.8.3. Взаимодействие с отладчиком

Отладчик управляет задачей посредством регистрации порта, в котором он имеет такие же права на получение, что и порт исключений задачи. Когда в нити этой задачи возникает исключение, которое не может быть очищено ее обработчиком ошибок, ядро отправляет сообщение в этот порт, и отладчик получает это сообщение. Исключение приводит только к останову нити-жертвы, все остальные нити задачи продолжают работу. Однако отладчик может при необходимости приостановить функционирование всей задачи, используя вызов `task_suspend`.

ОС Mach располагает некоторыми средствами, которые отладчик вправе использовать для управления задачей. Он может обращаться к адресному пространству нити-жертвы при помощи функции `vm_read` или `vm_write`, а также к ее контексту регистров, используя для этого функцию `thread_get_state` или `thread_set_state`. Отладчик также может приостанавливать или возобновлять работу приложения, а также завершить его функционирование при помощи функции `task_terminate`.

Механизм IPC (межпроцессное взаимодействие) в системе Mach является узло-независимым, то есть сообщения могут отправляться в порт как на той же самой машине, так и на удаленные хосты. Специальная прикладная задача-сервер под названием `netmsgserver` расширяет возможности IPC, делая этот механизм прозрачным при использовании через сеть. Сервер выделяет специальные прокси-порты для всех удаленных портов, принимает на них все сообщения, адресованные соответствующим удаленным портам, и затем пересыпает эти сообщения по сети. Таким образом, весь механизм передачи сообщений делается для отправителя этих сообщений прозрачным¹. Это дает возможность отладчику управлять задачей на любом узле сети точно так же, как и на локальном узле.

4.8.4. Анализ

Технология обработки исключений, реализованная в операционной системе Mach, разрешила многие проблемы, имеющиеся в традиционных вариантах UNIX. Она оказалась более гибкой и предоставила некоторые возможности, не поддерживаемые в других реализациях ОС. Перечислим некоторые важные преимущества описываемой технологии:

- ◆ преодоление ограничения на отладчик, по которому он управлял только своими непосредственными потомками. Теперь он может контролировать любое задание, если имеет соответствующие полномочия;
- ◆ отладчик может присоединяться к работающему заданию². Для этого он регистрирует один из своих портов как порт исключений отлаживаемой задачи. Он также может отсоединиться от задания, устанавливая порт исключений задачи в его первоначальное значение. Порт исключений является всего лишь средством связи между отладчиком и задачей, ядро системы не содержит в себе каких-либо средств поддержки отладки;
- ◆ расширение действия средств IPC системы Mach на сеть позволяет создавать распределенные отладчики;

¹ Ведь сообщения отправляются в порт на локальной машине. — *Прим. ред.*

² На сегодняшний день большинство отладчиков используют файловую систему /proc, которая позволяет получать доступ к адресному пространству несвязанным процессам. Таким образом, отладчики имеют возможность легкого присоединения и отсоединения от процесса. Во времена разработки системы Mach такие средства были редкостью.

- ◆ наличие отдельной нити для обработчика обеспечивает чистое разделение контекста обработчика и нити-жертвы, при этом позволяя обработчику иметь полный доступ к контексту нити-жертвы;
- ◆ происходит корректная обработка многонитевых процессов. При возникновении исключения приостанавливается работа только одной нити, вызвавшей это исключение, в то время как остальные продолжают функционировать в обычном режиме. Если исключения произойдут сразу в нескольких нитях, то каждая из них создаст отдельное сообщение, и исключение каждой будет обрабатываться независимо от остальных.

4.9. Группы процессов и управление терминалом

В системе UNIX существует понятие *групп процессов*. Оно используется для управления доступом с терминала и поддержки сеансов входа в систему. Устройство и реализация таких средств в различных вариантах UNIX сильно отличаются друг от друга. Этот раздел начинается с описания общих концепций, после чего приводится анализ основных реализаций для конкретных ОС.

4.9.1. Общие положения

Группы процессов. Каждый процесс относится к определенной группе процессов, которая идентифицируется через *идентификатор группы процессов* (process group ID). Этот механизм используется ядром для проведения некоторых действий сразу над всеми процессами группы. Каждая группа может иметь лидера. *Лидер группы* — это процесс, имеющий PID, совпадающий с идентификатором группы процессов. Обычно процесс наследует идентификатор группы от своего предка, а все остальные процессы в группе являются потомками лидера.

Управляющий терминал. Любой процесс может обладать управляющим терминалом. Чаще всего это терминал входа в систему, от которого процесс был создан. Все процессы одной группы разделяют между собой один и тот же управляющий терминал.

Файл /dev/tty. С управляющим терминалом каждого процесса связан специальный файл /dev/tty. Драйвер устройства, связанный с этим файлом, перенаправляет все запросы на соответствующий терминал. Например, в системе 4.3BSD номер устройства управляющего терминала хранится в поле u_ttyd области u. Чтение из терминала реализовано следующим образом:

```
(*cdevsw[major(u.u_ttyd)].d_read) (u.u_ttyd, flags);
```

Следовательно, если два процесса относятся к различным сессиям входа в систему и они оба откроют файл `/dev/tty`, то результатом станет доступ к разным терминалам.

Управляющая группа. Каждый терминал ассоциируется с группой процессов. Такая группа, называемая управляющей группой терминала, идентифицируется при помощи поля `t_pgrp` структуры `tty` этого терминала¹. *Сигналы, вырабатываемые при вводе с клавиатуры, такие как SIGINT или SIGQUIT, посылаются всем процессам управляющей группы терминала*, то есть всем процессам, чье поле `r_pgrp` в структуре `proc` равно значению `t_pgrp` структуры `tty` этого терминала.

Управление заданиями. Этот механизм (реализованный в системах 4BSD и SVR4) позволяет приостановить и возобновить работу группы процессов, а также управлять ее доступом к терминалу. Командные интерпретаторы, поддерживающие управление заданиями, такие как `csh` или `ksh`, распознают специальные управляющие символы (обычно `Ctrl+Z`) и команды, такие как `fg` и `bg`, для доступа к описываемым возможностям. Драйвер терминала обеспечивает дополнительное управление, посредством которого процессы, не входящие в управляющую группу терминала, защищены от чтения из терминала или записи в него.

Оригинальная реализация System V формирует группы процессов главным образом как представления сессий входа в систему и не имеет средств управления заданиями. В системе 4BSD с каждой введенной командной строкой командного интерпретатора ассоциируется новая группа процессов (следовательно, все процессы, связанные между собой конвейером командного интерпретатора, относятся к одной группе). Таким образом, в этой операционной системе появилось понятие задания. В SVR4 произведена унификация этих различающихся друг от друга подходов при помощи понятия сессии. В следующих разделах вы увидите описание всех трех технологий, перечисленных выше, а также анализ их преимуществ и недостатков.

4.9.2. Модель SVR3

В операционной системе SVR3 (и более ранних версиях ОС компании AT&T) группы процессов показывают характеристики сессии входа в систему терминала. Технология доступа с терминала в SVR3 проиллюстрирована на рис. 4.3. Ниже перечислены ее основные характеристики.

Группы процессов. Каждый процесс во время создания вызовом `fork` наследует идентификатор группы от своего родителя. Единственный способ изменения группы процесса заключается в вызове `setgrp`, который переустанавливает

¹ Драйвер содержит отдельную структуру `tty` для каждого терминала.

группу делающего этот вызов процесса на значение, равное его PID. Таким образом, процесс становится лидером новой группы. Все потомки этого процесса, созданные при помощи `fork`, присоединяются к его группе.

Управляющий терминал. Терминал принадлежит своей управляющей группе. То есть если процесс создает новую группу, то он теряет свой управляющий терминал. После этого первый терминал, им открываемый (который еще не является управляющим), становится его управляющим терминалом. Значение поля `t_pgrp` структуры `tty` для этого терминала установится в значение поля `r_pgrp` структуры `proc` процесса. Все потомки процесса наследуют управляющий терминал от своего лидера. Не может существовать две группы процессов с одним управляющим терминалом.

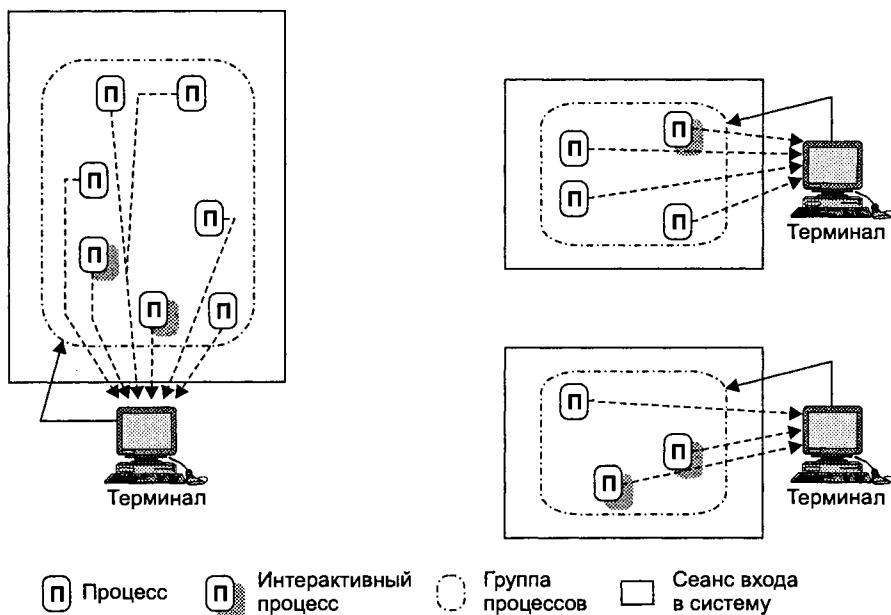


Рис. 4.3. Группы процессов в системе SVR3 UNIX

Типичный сценарий работы. Процесс `init` порождает процесс-потомок для каждого терминала, описанного в файле `/etc/inittab`. Потомок вызывает `setgrp`, становясь в результате этого лидером группы, после чего при помощи `exec` запускает программу `getty`, которая выводит приглашение на вход в систему и ждет ввода. Когда пользователь вводит свое регистрационное имя, программа `getty` посредством `exec` запускает программу `login`, которая запрашивает и верифицирует пароль, и затем стартует командный интерпретатор, назначенный¹ при входе в систему. Таким образом, командный интерпретатор,

¹ Конкретному пользователю. — Прим. ред.

запускаемый при входе в систему, является прямым потомком процесса `init` и лидером группы процессов. Обычно остальные процессы не создают собственных групп (исключение составляют системные демоны, запускаемые из сеанса входа в систему), — следовательно, все процессы, относящиеся к сеансу входа в систему, относятся к одной группе.

Доступ к терминалу. Не поддерживает управление заданиями. Все процессы, имеющие открытый терминал, обладают одинаковыми правами доступа к нему независимо от того, являются ли они фоновыми или интерактивными. Вывод таких процессов может производиться на терминал вперемежку. Если несколько процессов попытаются прочесть данные с терминала одновременно, то может оказаться неочевидным, какой из них «перехватит» ту или иную строку ввода.

Сигналы терминала. Такие сигналы, как `SIGQUIT` или `SIGINT`, создаваемые при нажатии определенных клавиш клавиатуры, посылаются всем процессам управляющей группы терминала, то есть обычно всем процессам сеанса входа в систему. Однако такие сигналы в большинстве случаев необходимы только интерактивным процессам. Поэтому командный интерпретатор при создании фонового процесса устанавливает для него игнорирование таких сигналов. Он также перенаправляет стандартный ввод таких процессов в `/dev/null`, что не позволяет им читать данные с терминала через его дескриптор (чтобы начать чтение с терминала, процесс может открыть другие дескрипторы).

Отсоединение от терминала. Терминал отсоединяется от его управляющей группы при установке поля `t_pgrp` своей структуры `tty` в ноль. Такое происходит, если ни один из процессов не связан более с данным терминалом или когда лидер группы завершает работу (обычно процесс, порожденный при входе в систему с данного терминала).

Завершение работы лидера группы. Лидер группы становится управляющим процессом своего терминала и отвечает за управление терминалом для всей группы. Когда лидер завершает работу, его управляющий терминал отсоединяется от группы (поле `t_pgrp` сбрасывается в ноль). Более того, всем остальным процессам группы оправляется сигнал `SIGHUP`, и значения их полей `r_pgrp` также сбрасываются в ноль, после чего эти процессы¹ больше не относятся ни к одной группе (становясь тем самым осиротевшими).

Реализация. Поле `r_pgrp` структуры `proc` содержит идентификатор группы процессов. Область `i` имеет два поля, относящиеся к терминалу: `u_ttyp` (указатель на структуру `tty` управляющего терминала) и `u_ttyd` (номер устройства управляющего терминала). Поле `t_pgrp` структуры `tty` содержит идентификатор группы процессов, управляющей терминалом.

¹ Оставшиеся «в живых», так как сигнал `SIGHUP` по умолчанию приведет к завершению получившего его процесса. — Прим. ред.

4.9.3. Ограничения

Реализация механизма групп процессов в системе SVR3 обладает нескольки-ми ограничениями [8]:

- ◆ не существует способов, которыми группа процессов могла бы закрыть свой управляющий терминал и выделить себе другой;
- ◆ хотя группы процессов формируются уже после открытия сеансов входа в систему, не существует возможности сохранять такие сеансы после отсоединения их управляющего терминала. В идеальном варианте в системе должен присутствовать механизм, позволяющий сохранять сеансы так, чтобы к ним позже можно было присоединить другой терминал и восстановить сеанс в том состоянии, в котором он был сохранен;
- ◆ нет какого-либо приемлемого способа обработки «потери несущей» управляющим терминалом. Такой терминал выделен группе и в случае «потери несущей» может быть перекоммутирован на другую группу с иной реализацией;
- ◆ ядро не синхронизирует доступ к терминалу между различными процессами группы. Фоновые и интерактивные процессы могут читать или вести запись на терминал в произвольном порядке;
- ◆ по завершении работы лидера группы ядро системы рассыпает сигнал SIGHUP всем процессам группы. При этом процессы, игнорирующие этот сигнал, могут продолжить осуществлять доступ к терминалу даже в том случае, если он переназначен уже другой группе. Это может привести к тому, что пользователь системы будет наблюдать неожиданный для него вывод таких процессов или, что хуже, процесс станет считывать данные, введенные уже новым пользователем. Последнее может стать причиной возникновения «дыр» в защите системы;
- ◆ если какой-то процесс, отличный от процесса, осуществляющего вход в систему, вызовет setgrp, то он будет отсоединен от управляющего терминала. При этом процесс вправе продолжать осуществлять доступ к терминалу через какие-либо существующие дескрипторы файлов. Однако такой процесс уже не управляем терминалом и он не может получить сигнал SIGHUP;
- ◆ в рассматриваемой системе отсутствуют средства управления заданиями, такие как возможность переключения процессов с интерактивного на фоновый и обратно;
- ◆ такие программы, как эмуляторы терминала, открывающие устройства, отличные от своего управляющего терминала, не имеют возможности получать уведомление о потере несущей от этих устройств.

В операционной системе 4BSD были решены некоторые из перечисленных проблем. Модель, реализованная в этой ОС, будет представлена в следующем разделе.

4.9.4. Группы и терминалы в системе 4.3BSD

В системе 4.3BSD группа процессов представляет собой *задание* (иногда называемое *задачей*, task) в рамках сеанса входа в систему. Задание – это набор связанных процессов, которые управляются как единый блок относительно доступа к терминалу. Основные принципы доступа к терминалу системы 4.3BSD продемонстрированы на рис. 4.4.

Группы процессов. Процесс наследует идентификатор группы от своего родителя. Процесс может изменить свой идентификатор группы или идентификатор группы любого другого процесса при помощи вызова `setgrp` (последнее зависит от полномочий: у всех этих процессов должен быть либо общий владелец, либо вызывающий процесс должен иметь права суперпользователя). Системный вызов 4.3BSD `setgrp` имеет два параметра: идентификатор процесса и новый присваиваемый ему идентификатор группы. Таким образом, в системе 4.3BSD процесс может освободить место лидера или присоединиться к любой другой группе. Более того, группа процессов вообще может не иметь лидера.

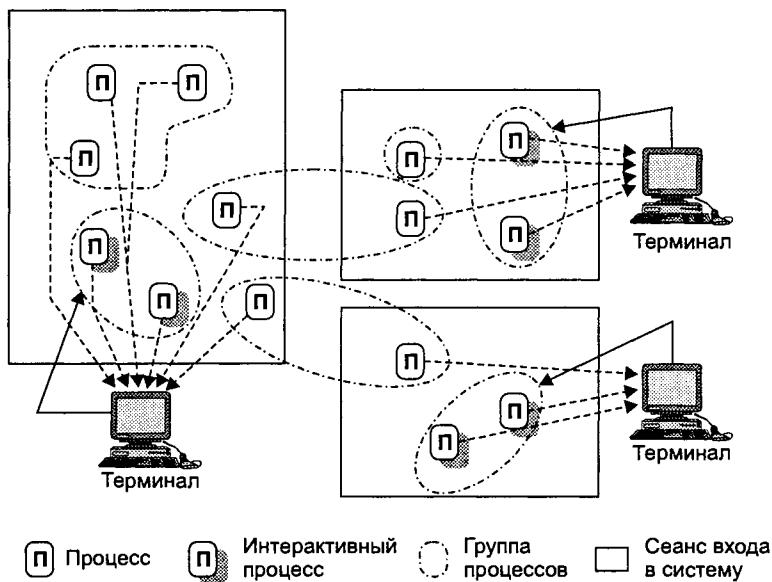


Рис. 4.4. Группы процессов в системе 4.3BSD UNIX

Задания. Командные интерпретаторы, поддерживающие управления заданиями, такие как `csh`, обычно создают новую группу процессов для каждой введенной командной строки, независимо от того, будут ли они выполняться в фоновом или текущем режиме. Таким образом, задание обычно состоит из одного процесса или набора процессов, соединенных между

себой конвейерами¹. Потомки этих процессов также будут являться членами группы.

Сеансы входа в систему. В системе 4.3BSD каждый сеанс входа в систему может создавать несколько групп процессов (или заданий), которые функционируют одновременно, разделяя между собой один и тот же терминал. Поле `t_pgrp` структуры `tty` терминала всегда содержит группу текущего выполняющегося задания.

Управляющие терминалы. Если процесс с идентификатором, группы равным нулю, открывает терминал, то такой терминал становится управляющим для данного процесса, а сам процесс присоединяется к текущей управляющей группе терминала (поле `p_pgrp` структуры `proc` процесса устанавливается равным полю `t_pgrp` структуры `tty` терминала). Если терминал в данный момент не является управляющим терминалом какой-либо группы, то процесс становится лидером (то есть поля `p_pgrp` структуры `proc` процесса и `t_pgrp` структуры `tty` терминала устанавливаются равными значению PID процесса). Прямые наследники `init` (то есть все командные интерпретаторы, установленные на вход в систему) первоначально обладают идентификатором группы, равным нулю. Установить идентификатор группы процесса в ноль может только суперпользователь.

Доступ к терминалу. Интерактивные процессы (то есть процессы, принадлежащие текущей управляющей группе терминала, полученной из поля `t_pgrp` структуры `tty` этого терминала) всегда обладают беспрепятственным доступом к терминалу. Если фоновый процесс попытается прочесть с терминала, драйвер пошлет сигнал `SIGTTIN` всем процессам, принадлежащим его группе. По умолчанию сигнал `SIGTTIN` приводит к приостановке работы получивших его процессов. Операция записи в терминал по умолчанию разрешена всем фоновым процессам. Система 4.3BSD предлагает настройку терминала (бит `LTOSTOP`, изменяемый при помощи вызова `TIOCLSET ioctl`), установка которой приводит к отправке сигнала `SIGTTOU` фоновому процессу, пытающемуся произвести запись в терминал. Задания, приостановленные сигналом `SIGTTIN` или `SIGTTOU`, могут продолжить работу после получения сигнала `SIGCONT`.

Управляющая группа. Процесс, имеющий доступ к терминалу на чтение, может осуществить вызов `TIOSPGRP ioctl` для изменения значения идентификатора управляющей группы терминала (поле `t_pgrp` структуры `tty`) на любое другое. Командный интерпретатор использует эту возможность системы для вызова процесса из фонового выполнения в интерактивный режим и наоборот. Например, пользователь может возобновить работу приостановленной

¹ В системе также существует возможность комбинирования двух или более не соединенных между собой процессов в одну группу при помощи ввода сразу нескольких команд в одной и той же строке. Такие команды заключаются в круглые скобки и отделяются друг от друга точкой с запятой: `% (cc tanman.c; cp file1 file2; echo done >newfile)`.

группы процессов и назначить ее активной, сделав эту группу управляющей и отправив ей сигнал `SIGCONT`. Для этой цели в командных интерпретаторах `csh` и `ksh` предусмотрены команды `fg` и `bg`.

Закрытие терминала. Когда нет ни одного процесса, для которого данный терминал открыт, то такой терминал не связан с группой и его поле `t_pgrp` обнуляется. Это действие производится при помощи процедуры драйвера `close`, вызываемой в момент, когда последний дескриптор терминала закрывается.

Повторная инициализация линии терминала. ОС 4.3BSD обеспечивает системный вызов `vhangup`, который обычно используется процессом `init` для завершения текущего сеанса входа в систему и старта нового. Вызов просматривает таблицу открытых файлов, находит каждый элемент, относящийся к этому терминалу, и делает его неиспользуемым. Это достигается посредством удаления состояния «открыт» в элементах таблицы открытых файлов либо в тех реализациях, в которых поддерживается интерфейс `vnode` (см. раздел 8.6), изменением вектора `vnodeops` на такой набор функций, которые просто возвращают ошибку. Затем `vhangup` вызывает процедуру терминала `close()` и посыпает сигнал `SIGHUP` управляющей группе этого терминала. Такой подход в ОС 4.3BSD является решением проблемы управления процессами, которые продолжают функционирование уже после завершения сеанса входа в систему.

4.9.5. Недостатки модели 4.3BSD

Несмотря на то что модель управления заданиями в 4.3 BSD является развитой и универсальной, она обладает рядом существенных недостатков, перечисленных ниже:

- ◆ не существует четкого представления сеанса входа в систему. Изначальный процесс входа в систему не является особенным и может даже не являться лидером группы. Обычно по завершении такого процесса сигнал `SIGHUP` не рассыпается;
- ◆ не существует какого-то единственного процесса, ответственного за управление терминалом. Таким образом, состояние потери несущей передается всем процессам его текущей управляющей группы (при помощи сигнала `SIGHUP`), процессы которой могут даже игнорировать данный сигнал. Например, удаленный пользователь, работающий в системе через модемное соединение, останется в системе, даже если произойдет физическое отключение от линии связи;
- ◆ процесс может изменить управляющую группу терминала на любую другую, даже на несуществующую. Если позже будет создана группа с таким идентификатором, то она унаследует терминал и будет получать от него «незаслуженные» сигналы;
- ◆ программный интерфейс является несовместимым с интерфейсом System V.

Ясно, что нам необходим подход, при котором сохранялась бы концепция сеансов входа в систему и задач, выполняемых в таких сеансах. Последующий раздел посвящен описанию архитектуры сеансов операционной системы SVR4 и тому, как она решает эту проблему.

4.10. Архитектура сеансов в системе SVR4

Все ограничения моделей групп и терминалов в системах SVR3 и 4.3BSD могут быть отнесены к одной фундаментальной проблеме. Единое понятие группы процессов не в состоянии адекватно представлять сеансы входа в систему и задания, выполняемые в таких сеансах. В системе SVR3 неплохо реализовано управление поведением сеанса входа в систему, но она не поддерживает координацию заданий. Система 4.3BSD обладает развитыми средствами управления заданиями, но не умеет корректно изолировать друг от друга сеансы входа в систему.

В современных операционных системах, таких как SVR4 или 4.4BSD, эти проблемы были преодолены посредством представления сеансов и заданий раздельными, но взаимосвязанными между собой механизмами. Группа процессов определена единым заданием. Новый объект *сеанс* представляет собой сеанс входа в систему. В следующих разделах вы увидите описание архитектуры сеансов, представленной в ОС SVR4. Раздел 4.10.5 расскажет о реализации сеансов в 4.4BSD, которая схожа по функциональности с архитектурой, используемой в SVR4, но дополнительно обладает совместимостью со стандартами POSIX.

4.10.1. Задачи, поставленные перед разработчиками

Архитектура сеансов восполняет некоторые недостатки, имеющиеся в более ранних моделях. Основными целями новой архитектуры явились:

- ◆ поддержка на должном уровне как сеансов входа в систему, так и заданий;
- ◆ обеспечение управления заданиями, в стиле BSD;
- ◆ сохранение обратной совместимости с более ранними версиями System V;
- ◆ предоставление сеансам входа в систему возможности подключения и отключения нескольких управляющих терминалов в течение времени существования сеансов (конечно, в один момент времени они могут иметь только один управляющий терминал). Все эти изменения должны быть прозрачно распространены на все процессы сеанса;
- ◆ реализация лидера сеанса (процесса, создающего сеанс входа в систему), ответственного за обеспечение его целостности и безопасности;

- ◆ предоставление доступа к терминалу, основываясь исключительно на правах доступа к файлу. В частности, если процесс благополучно открыл терминал, то он может иметь к нему доступ на все то время, пока тот открыт;
- ◆ устранение несовместимости с предыдущими реализациями. Например, в ОС SVR3 существует определенная нестыковка в случае, если лидер группы порождает потомка при помощи `fork` до того, как назначит управляющий терминал. Такой процесс-потомок будет получать сигналы (`SIGINT` и т. д.) от этого терминала, но не будет иметь к нему доступа через `/dev/tty`.

4.10.2. Сеансы и группы процессов

Архитектура сеансов операционной системы SVR4 продемонстрирована на рис. 4.5 [12]. Каждый процесс относится как к определенному сеансу, так и к группе. Точно так же каждый управляющий терминал связан с сеансом и активной группой процессов (то есть управляющей группой терминала). Сеанс играет роль группы процессов системы SVR3, а лидер сесанса является ответственным за управление сеансом входа в систему и его изоляцию от других сеансов. Правом назначения или освобождения контролируемого терминала обладает только лидер сесанса.

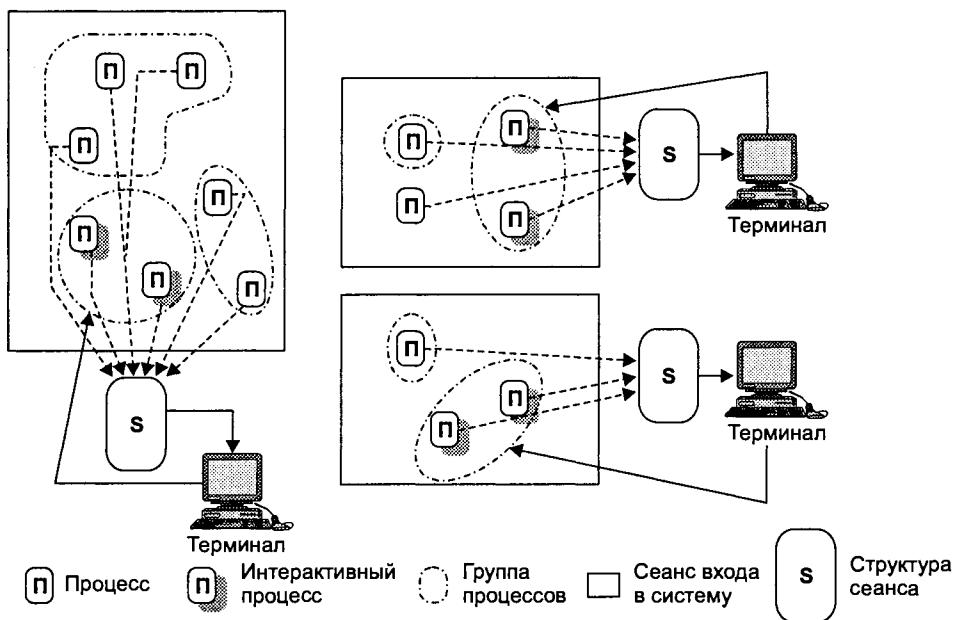


Рис. 4.5. Архитектура сеансов системы SVR4

Процесс создает новый сеанс при помощи вызова `setsid`, который устанавливает идентификаторы сеанса и группы в значение собственного PID. Таким образом, вызов `setsid` делает вызвавший его процесс одновременно лидером группы и сеанса. Если процесс уже является лидером группы, то он не может стать лидером сеанса, и выполнение вызова завершится с ошибкой.

Группы процессов системы SVR4 обладают основными чертами групп ОС 4.3BSD и обычно представляют собой задание внутри сеанса входа в систему. Таким образом, в одном сеансе входа в систему одновременно могут функционировать несколько групп процессов. Одна из таких групп является текущей и обладает неограниченным доступом к терминалу (то есть является управляющей группой терминала). Так же как и в системе 4.3BSD, фоновые процессы, пытающиеся получить доступ к управляющему терминалу, получат сигналы `SIGTTIN` и `SIGTTOU` (последний для этого должен быть разрешен, как это было описано в разделе 4.9.4).

Процесс наследует свою группу от родителя и может изменить ее при помощи вызовов `setpgid` или `setgrp`. Вызов `setgrp` идентичен варианту, представленному в SVR3, и устанавливает значение группы в PID сделавшего этот вызов процесса, делая его таким образом лидером группы. Вызов `setpgid` схож с `setgrp` системы 4.3BSD, но добавляет в действие некоторые существенные ограничения. Синтаксис `setpgid` таков:

```
setpgid (pid, pgid);
```

Задача этого вызова состоит в изменении группы процесса, указанного в `pid`, на значение, определенное в `pgid`. Если `pgid` равняется нулю, то идентификатор группы процесса устанавливается в значение, равное `pid`, что делает процесс лидером группы. Однако, как уже говорилось, в реализации `setpgid` существует несколько важных ограничений. Так, процесс, над которым производится действие, должен сам вызывать эту функцию, либо это должен делать один из его потомков, еще не сделавший `exec`. Вызывающий процесс и процесс, на который направлено действие, также должны относиться к одному и тому же сеансу. Если значение `pgid` не совпадает с PID целевого процесса (или равно нулю, что дает одинаковый эффект), то оно должно быть равным одному из существующих идентификаторов группы внутри того же сеанса.

По указанным выше причинам процессы имеют возможность перемещаться из одной группы в другую в течение продолжительности сеанса. Единственным способом покинуть сеанс для процесса является вызов `setsid`, открывающий новый сеанс с этим процессом как единственным ее членом. Процесс, являющийся лидером группы, может оставить свое лидерство, переместив себя в другую группу. Однако этот процесс не может создать новый сеанс до тех пор, пока его PID является идентификатором группы других процессов (то есть если группа, в которой он оставил лидерство, не пуста). Такой подход защищает от возникновения ситуации, когда группа процессов обладает тем же идентификатором, что и сеанс, в который эта группа не входит.

Точно так же текущая (управляющая) группа может быть изменена только процессом сеанса, который управляет терминалом, и только на группу, которая реально существует в сеансе. Эта возможность используется командными процессорами, поддерживающими управление заданиями для перевода задач в интерактивный и фоновый режимы работы.

4.10.3. Структуры данных

На рис. 4.6 показаны структуры данных, используемые для управления сессиями и группами процессов. Вызов `setsid` выделяет новую структуру сеанса и сбрасывает поля `p_sessp` и `p_pgidp` структуры `proc`. Изначально сеанс не имеет управляющего терминала.

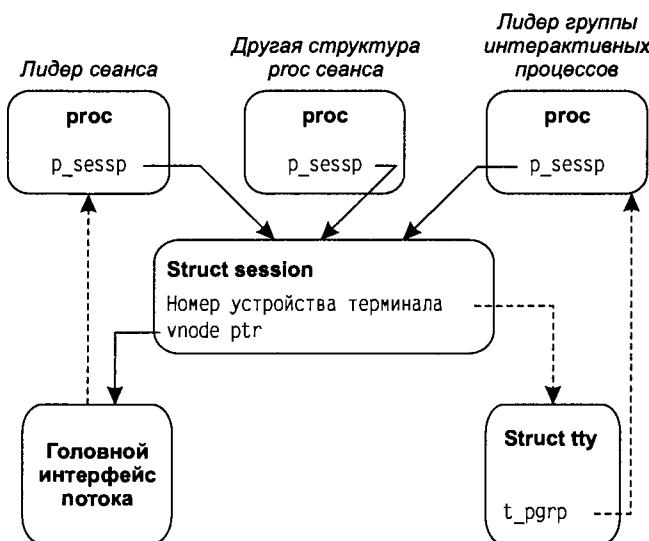


Рис. 4.6. Структуры данных управления сеансом в SVR4 UNIX

Когда лидер сеанса впервые открывает терминал (после того как он станет лидером), этот терминал станет управляющим терминалом сеанса, если только вызывающий процесс не передаст флаг `O_NOCTTY` в соответствующий вызов при открытии этого терминала. Структура сеанса инициализируется указателем на `vnode` этого терминала, а `vnode`, в свою очередь, указывает на головной интерфейс потока (`stream head`) для устройства.

Процессы-потомки лидера сеанса наследуют значение указателя в поле `p_sessp` структуры `proc`, благодаря чему могут постоянно следить за изменениями объекта сеанса. Поэтому процессы-потомки наследуют управляющий терминал, даже если тот был открыт уже после того, как эти процессы были созданы.

4.10.4. Управляющие терминалы

Файл `/dev/tty` снова выступает в качестве псевдонима управляющего терминала. Драйвер разрешает любые вызовы в `/dev/tty` посредством просмотра указателя на сеанс в структуре `proc` и, используя его, получает дескриптор `vnode` управляющего терминала. Если управляющий терминал освобождается, то ядро системы устанавливает указатель на `vnode` объекта сеанса в `NULL`, после чего все попытки доступа к `/dev/tty` закончатся неудачно. Если процесс открыл напрямую определенный терминал (в противоположность открытию `/dev/tty`), то он сможет продолжить доступ к нему даже после того, как терминал перестанет быть связанным с текущим сеансом входа в систему.

Когда пользователь входит в систему, процесс, обеспечивающий вход, производит следующие действия:

1. Вызывает `setsid` для того, чтобы стать лидером сеанса.
2. Закрывает `stdin`, `stdout` и `stderr`¹.
3. Вызывает `open` для открытия выбранного терминала. Так как этот терминал является первым из открытых лидером сеанса, то он становится управляющим терминалом для него. Вызов `open` возвращает дескриптор реального файла устройства терминала.
4. Дублирует и сохраняет полученный дескриптор, с тем чтобы не использовать `stdin`, `stdout` и `stderr` для ссылки на реальный файл устройства. После закрывает оригинальный дескриптор. Управляющий терминал остается открытм через дублированный дескриптор.
5. Открывает `/dev/tty` как `stdin` и дублирует его в `stdout` и `stderr`. Это действие эффективно открывает заново управляющий терминал через псевдоним устройства. Таким образом, лидер и все остальные процессы сеанса (которые наследуют эти дескрипторы) имеют доступ к терминалу только посредством `/dev/tty` (если только другой процесс прямо не откроет файл устройства терминала).
6. В конце происходит закрытие сохраненного дескриптора и удаление любых прямых контактов с управляющим терминалом.

Если драйвер терминала обнаружит разорванное соединение (например, потерю несущей при модемном подключении), то он пошлет сигнал `SIGHUP` только лидеру сеанса. Такой подход явно отличается от отправки сигнала текущей группе в системе 4.3BSD и оправки сигнала всем процессам управляющей группы (сеанса) в SVR3. При таком подходе лидер сеанса является доверенным процессом, и ожидается, что он произведет корректные действия при потере управляющего терминала.

Драйвер также посыпает сигнал `SIGSTP` текущей группе процессов, если она не является группой лидера сеанса. Это защищает интерактивные процессы от получения неожиданных ошибок при попытке доступа к терминалу.

¹ Стандартные устройства ввода, вывода и вывода ошибок. — Прим. ред.

Управляющий терминал остается закрепленным за сеансом. Это дает возможность лидеру сеанса попытаться заново соединиться с терминалом после восстановления соединения.

Лидер сеанса может завершить соединение с текущим управляющим терминалом и открыть новый. Ядро системы установит указатель объекта vnode сеанса на указатель vnode нового терминала. В результате все процессы этого сеанса входа в систему будут прозрачно для них переключены на новый управляющий терминал. Такая косвенная связь, обеспеченная /dev/tty, облегчает решение задачи распространения такого изменения.

Когда лидер сеанса заканчивает свою работу, он завершает и сеанс входа в систему. Управляющий терминал освобождается при помощи установки указателя vnode сеанса в NULL. В результате ни один процесс этого сеанса не сможет иметь доступа к терминалу посредством /dev/tty (но они смогут продолжать осуществлять доступ при непосредственном открытии файла устройства терминала). Процессы текущей группы получают сигнал SIGHUP. Все прямые потомки существующего процесса будут унаследованы процессом init.

4.10.5. Реализация сеансов в 4.4BSD

В архитектуре сеансов системы SVR4 адекватно представлены как сеанс входа в систему, так и задания, выполняющиеся в этом сеансе. В то же время она совместима со стандартом POSIX 1003.1 и ранними версиями System V. Реализация сеансов в операционных системах 4.4BSD и OSF/1 очень похожа на архитектуру SVR4 и обладает сравнимыми с ней возможностями. Различия между реализациями проявляются только в отдельных деталях.

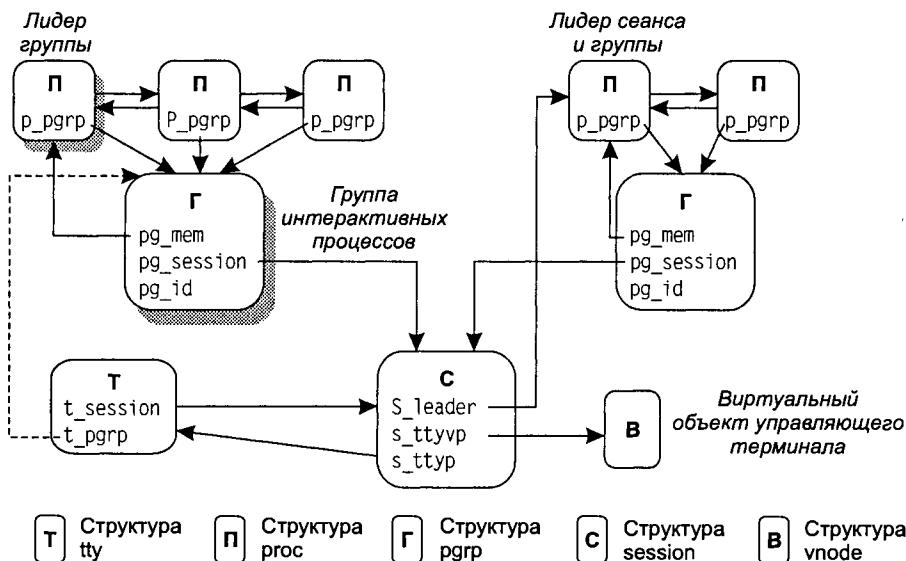


Рис. 4.7. Структуры данных управления сеансами в 4.4BSD UNIX

На рис. 4.7 для сравнения проиллюстрированы структуры данных, используемые в 4.4BSD [10]. Одним из важнейших отличий является то, что структура `proc` не имеет прямой ссылки на объект `session`. Вместо этого она ссылается на объект группы процессов (структура `pggrp`), который, в свою очередь, содержит указатель на структуру `session`.

4.11. Заключение

Появление стандарта POSIX 1003.1 помогло объединить различающиеся между собой и несовместимые реализации поддержки сигналов и управления терминалами. В результате интерфейсы оказались весьма удачными и в большой степени отвечающими ожиданиям типичных приложений и разработчиков.

4.12. Упражнения

Ответы на некоторые из перечисленных ниже вопросов могут быть различными в зависимости от используемого варианта UNIX. Отвечающий может выбрать для ответа на вопросы одну из наиболее близких для него реализаций системы.

1. Почему обработчики сигналов не сохраняются после выполнения системного вызова `exec`?
2. Почему сигнал `SIGCHLD` по умолчанию игнорируется?
3. Что происходит в случае возникновения сигнала для процесса, находящегося в стадии выполнения вызовов `fork`, `exec` или `exit`?
4. При каких условиях сигнал, отправленный `kill`, не завершит немедленно выполнение процесса?
5. В традиционных системах UNIX приоритет сна используется для двух целей: решения, будет ли процесс разбужен для приема этого сигнала, и определения приоритета процесса в расписании после выхода процесса из режима сна. Какие недостатки имеются в классической модели и как эти проблемы были решены в современных системах?
6. Какие существуют недостатки использования постоянно установленных обработчиков? Существуют ли какие-либо определенные сигналы, для обработки которых нецелесообразно применять постоянно установленные обработчики?
7. Чем отличаются между собой реализации вызова `sigpause` в системах 4.3BSD и SVR3? Опишите возможную ситуацию, при которой применение первого варианта представляется более удобным.
8. Чем предпочтительнее поддержка повторного старта прерванного системного вызова на уровне ядра, нежели перезапуск его приложением?

9. Что произойдет, если процесс получит несколько экземпляров одного и того же сигнала перед тем, как он обработает первый экземпляр этого сигнала? Будет ли в такой ситуации целесообразным применение иных сигнальных моделей?
10. Представьте, что процесс ожидает получения двух сигналов и что в нем были описаны обработчики для каждого из этих сигналов. Каким образом ядро системы может убедиться в том, что процесс обработает второй сигнал сразу же после обработки первого?
11. Что произойдет, если процесс получит сигнал во время обработки другого? Как в этом случае процесс может контролировать свои действия?
12. В каких случаях процессу целесообразно использовать флаг `SA_NOCLDWAIT`, определенный в системе SVR4? В каких случаях этого делать не следует?
13. Зачем обработчику исключения необходим полный контекст процесса, в котором исключение произошло?
14. Какой процесс может создать новую группу процессов: а) в системе 4.3BSD, б) в системе SVR4?
15. Какими преимуществами обладает архитектура сеансов SVR4 по сравнению со средствами работы с терминалами и управления заданиями системы 4.3BSD?
16. В [3] описывается менеджер сеансов прикладного уровня, использующийся для поддержки сеансов входа в систему. Насколько сходно это средство с архитектурой сеансов системы SVR4?
17. Что должно сделать ядро системы SVR4 в том случае, когда лидер сеанса освободит управляющий терминал?
18. Каким образом в системе SVR4 реализована поддержка переподключения сеанса к управляющему терминалу? В каких ситуациях применима такая возможность системы?

4.13. Дополнительная литература

1. American Telephone & Telegraph, «UNIX System V Release 3: Programmer's Reference Manual», 1986.
2. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
3. Bellovin, S. M., «The Session Tty Manager», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988.
4. Black, D. L., Golub, D. B., Hauth, K., Tevanian, A., and Sanzi, R., «The Mach Exception Handling Facility», CMU-CS-88-129, Computer Science Department, Carnegie Mellon University, Apr. 1988.

5. Institute for Electrical and Electronic Engineers, Information Technology – «Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) (C Language}», 1003.1–1990, Dec. 1990.
6. Joy, W., «An Introduction to the C Shell», Computer Science Division, University of California at Berkeley, Nov. 1980.
7. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.
8. Leaner, D. C., «A System V Compatible Implementation of 4.2 BSD Job Control», Proceedings of the Summer 1986 USENIX Technical Conference, Jun. 1986, pp. 459–474.
9. Stevens, W. R., «UNIX Network Programming», Prentice-Hall, Englewood Cliffs, NJ, 1990.
10. Stevens, W. R., «Advanced Programming in the UNIX Environment», Addison-Wesley, Reading, MA, 1992.
11. UNIX Systems Laboratories, «Operating System API Reference: UNIX SVR4.2», UNIX Press, 1992.
12. Williams, T., «Session Management in System V Release 4», Proceedings of the Winter 1989 USENIX Technical Conference, Jan. 1989, pp. 365–375.

Глава 5

Планирование процессов

5.1. Введение

Центральный процессор наряду с памятью и терминалами является общим ресурсом, разделяемым между всеми процессами системы. Принятие решения о том, как должны использоваться ресурсы различными процессами, возложено на операционную систему. Планировщик является компонентом ОС, определяющим, какой из процессов должен выполняться в данный момент времени и как долго он может занимать процессор. Система UNIX изначально создавалась как ОС разделения времени, что означает возможность одновременного выполнения в ней нескольких процессов. Конечно, в определенной степени одновременное выполнение является лишь иллюзией (по крайней мере, в однопроцессорных системах), так как в один момент времени на одном процессоре может выполняться только что-то одно. В системах UNIX эмулируется одновременность работы при помощи чередования процессов на основе принципа разделения времени. Планировщик предоставляет процессор каждому процессу системы на небольшой период времени, после чего производит переключение на следующий процесс. Такой период называется *квантом времени* (quantum или slice).

В описании планировщика UNIX необходимо учитывать два важных аспекта. Первый из них касается *политики* — правил, используемых при принятии решения о том, какой из процессов следует назначить на выполнение и когда произвести переключение на выполнение другого процесса. Второй рассматриваемый аспект относится к *реализации*, представляющей собой набор структур данных и алгоритмов, используемых для проведения этих политик. Политика планирования должна по мере возможности удовлетворять различным требованиям: определенному времени реакции для интерактивных приложений, высокой производительности для фоновых заданий, недопущению полного необслуживания процессов и т. д. Попытка удовлетворения одновременно всех поставленных целей часто приводит к конфликтам, поэтому планировщик должен обеспечить наилучшее соотношение между ними. Также от него требуется эффективная реализация политики планирования при минимальных перегрузках системы.

На самом низком уровне планировщик заставляет процессор производить переключения от одного процесса к другому. Это действие называется *переключением контекста*. Ядро системы сохраняет аппаратный контекст текущего выполняющегося процесса в *блоке управления процессом* (process control block, PCB), который обычно является частью его области памяти. Контекст представляет собой «моментальный снимок» текущих значений регистров общего назначения, регистров управления памятью и других специальных регистров процесса. После завершения процедуры сохранения ядро системы загружает аппаратные регистры с контекстом следующего процесса, готовящегося к выполнению (контекст загружается из его блока PCB). Это действие приводит к тому, что CPU начинает выполнение нового процесса. Основной задачей планировщика является принятие решения о том, когда произвести переключение контекста и какой из процессов назначить на выполнение.

Переключения контекста являются весьма затратными операциями. Кроме сохранения копии регистров процесса, ядро системы должно выполнить еще множество архитектурно зависимых действий. На некоторых системах оно должно очистить данные, инструкции или буфер трансляции адресов для предупреждения некорректного доступа к памяти новым процессом (см. разделы 15.9–15.13). В результате в начале своей работы процесс тратит в виде накладных расходов несколько операций обращения к памяти. Это уменьшает его производительность, так как доступ к памяти является значительно более медленной процедурой, чем доступ к кэшу. Также стоит упомянуть о конвейерных архитектурах, таких как *процессоры с сокращенным набором команд* (Reduced Instruction Set Computers, RISC), в которых ядро перед переключением контекста должно очистить конвейер команд. Упомянутые факторы могут влиять не только на реализацию планировщика, но и на его политику.

Эта глава начинается описанием обработки прерываний таймера и заданий, основанных на его работе. Таймер является важным компонентом функционирования планировщика, так как последнему часто необходимо вытеснить выполняющийся процесс по окончанию выделенного ему кванта времени. Остальная часть главы рассказывает об устройстве различных планировщиков и о том, как их функционирование влияет на работу системы.

5.2. Обработка прерываний таймера

На каждой UNIX-машине существует аппаратный таймер, который вырабатывает прерывание в системе через фиксированные промежутки времени. В некоторых машинах операционная система должна увеличивать значение времени, отсчитываемого каждым прерыванием таймера, в других таймер это делает самостоятельно. Период времени между двумя последующими преры-

ваниями таймера называется *тиком процессора*, *тиком таймера* или просто *тиком*. Большинство компьютеров поддерживают переменные тиковые интервалы. В системах UNIX продолжительность тика составляет обычно 10 миллисекунд¹. Во многих реализациях UNIX частота таймера (количество тиков в секунду) хранится в специальной константе `HZ`, которая обычно определена в файле `param.h`. Для тика продолжительностью 10 миллисекунд значение `HZ` будет равно 100. Функции ядра чаще всего измеряют время в количестве тиков, редко используя для этого секунды или миллисекунды.

Обработка прерываний сильно зависит от используемой системы. Этот раздел рассказывает о стандартной реализации, которую можно встретить во многих традиционных версиях UNIX. Обработчик прерываний таймера запускается в ответ на возникновение аппаратного прерывания таймера, являющегося вторым по приоритету событием в системе (после прерывания по сбою питания). Следовательно, обработчик должен запускаться как можно быстрее, а его время работы желательно сводить к минимуму. Обработчик прерываний таймера выполняет следующие задачи:

- ◆ ведет счет тиков аппаратного таймера по необходимости;
- ◆ обновляет статистику использования процессора текущим процессом;
- ◆ выполняет функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- ◆ посылает текущему процессу сигнал `SIGXCPU`, если тот превысил выделенную ему квоту использования процессора;
- ◆ обновляет часы и другие таймеры системы. Например, в SVR4 имеется переменная `lbolt`, хранящая количество тиков, отсчитанных с момента загрузки системы;
- ◆ обрабатывает отложенные вызовы (см. раздел 5.2.1);
- ◆ пробуждает в нужные моменты системные процессы, такие как `swapper` и `pagedaemon`;
- ◆ обрабатывает сигналы тревоги (см. раздел 5.2.2).

Некоторые из перечисленных задач не требуют выполнения на каждом тике. В большинстве систем UNIX определено понятие *основного тика*, который равен n тикам таймера (число n зависит от конкретного варианта системы). Планировщик выполняет некоторые из своих задач только с приходом основного тика. Например, в 4.3BSD пересчет приоритетов происходит на каждый четвертый тик, в то время как SVR4 обрабатывает сигналы тревоги и возобновляет по необходимости работу системных процессов с частотой один раз в секунду.

¹ Это число не является универсальным и зависит от конкретного варианта UNIX. Продолжительность тика также зависит от разрешения аппаратного таймера.

5.2.1. Отложенные вызовы

Отложенный вызов (callout) представляет собой запись функции, которую ядро системы должно будет вызвать через определенный промежуток времени. Например, в системе SVR4 любая подсистема ядра может зарегистрировать отложенный вызов следующим образом:

```
int to_ID = timeout (void (*fn)(), caddr_t arg, long delta);
```

где fn() — функция ядра, которую необходимо запустить, arg — аргумент, который следует передать fn(). delta — временной интервал, через который эта функция должна быть вызвана, выраженный в тиках процессора. Ядро выполняет функцию, определенную в отложенном вызове, в системном контексте. Следовательно, такая функция не должна переходить в режим ожидания или осуществлять доступ к контексту процесса. Возвращаемое значение to_ID может быть использовано для отмены выполнения отложенного вызова:

```
void untimeout (int to_ID);
```

Отложенные вызовы могут быть использованы для выполнения различных повторяющихся задач, таких как:

- ◆ повторная пересылка сетевых пакетов;
- ◆ некоторые функции планировщика и управления памятью;
- ◆ мониторинг устройств для предотвращения потери прерываний;
- ◆ периодический опрос устройств, не поддерживающих прерывания.

Отложенные вызовы считаются обычными процедурами ядра и не должны выполняться с приоритетами прерываний. Поэтому обработчик прерываний таймера не выполняет эти вызовы напрямую. На каждом тике обработчик прерываний таймера проверяет, не нужно ли начать выполнение отложенного вызова. Если он находит ожидающий вызов, то выставляет флаг, указывающий на необходимость запуска *обработчика отложенного вызова*. Система проверяет этот флаг при возврате в основной приоритет прерываний и, если тот установлен, запускает обработчик. Обработчик начнет выполнение каждого ожидаемого отложенного вызова. Следовательно, ожидаемый отложенный вызов будет выполнен с максимально возможной быстротой, но только после обработки всех ожидающих прерываний¹.

Ядро системы поддерживает список ожидающих отложенных вызовов. Организация такого списка влияет на производительность системы, так как он может содержать несколько вызовов. Поскольку список проверяется на каждом тике с высоким приоритетом прерывания, проверяющий алгоритм должен оптимизировать время проверки. Время, затрачиваемое на вставку

¹ Во многих реализациях UNIX обеспечена некоторая оптимизация механизма обработки прерываний для случая, когда никакие иные прерывания не находятся в ожидании завершения работы наиболее приоритетного обработчика. В такой ситуации обработчик таймера напрямую уменьшает приоритет прерывания и запускает обработчик отложенного вызова.

нового отложенного вызова в список, является менее критичным, так как вставка обычно происходит при более низких приоритетах и с меньшей частотой, чем одна операция на каждый тик.

Существует несколько способов реализации списка отложенных вызовов. Метод, используемый в 4.3BSD [12], сортирует список в порядке *возрастания времени запуска вызовов* (time to fire). Каждый элемент списка содержит разницу между временем своего запуска и запуска предыдущего отложенного вызова. Ядро системы уменьшает время первого элемента списка на каждом тике и запускает вызов после того, как это значение станет равным нулю. Если другие вызовы должны выполниться в это же время, то и они запускаются ядром. Описанная технология графически проиллюстрирована на рис. 5.1.

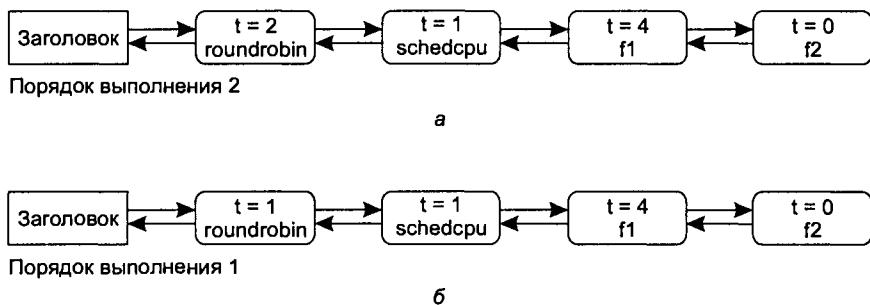


Рис. 5.1. Реализация отложенных вызовов в BSD UNIX: а — начальное состояние списка отложенных вызовов; б — состояние через один тик таймера

Еще одним применяемым методом является использование сходного сортированного списка, но хранится в нем абсолютное время запуска для каждого элемента. В таком случае на каждом тике ядро системы сравнивает текущее абсолютное время со временем первого элемента списка и запускает отложенный вызов, если значения совпадут.

Оба приведенных метода требуют организации сортированного списка, который может оказаться довольно большим и, следовательно, довольно затратным для системы. Альтернативное решение предполагает использование *карусели* (timing wheel), представляющей собой циклически замкнутый массив очередей отложенных вызовов. На каждом тике обработчик прерываний таймера смещает указатель к следующему элементу массива, циклически возвращаясь по достижении конца массива. Если в текущей очереди находятся отложенные вызовы, то происходит проверка их времени запуска. Новые вызовы добавляются в очередь, отстоящую на N элементов от текущей, где N — время до запуска отложенного вызова, выраженное в тиках¹.

¹ Скорее всего, N — это не просто смещение от текущего элемента, а остаток от деления времени запуска на длину массива; например, при длине массива в 100 элементов и времени запуска через 535 тиков N будет равняться 35. — Прим. ред.

В результате циклического планирования хэшируются отложенные вызовы на основе времени истечения их ожидания, то есть времени, через которое они должны отработать. Внутри каждой очереди отложенные вызовы могут храниться как в упорядоченном, так и в не сортированном виде. Сортировка сокращает время на обработку непустых очередей, но увеличивает затраты на внесение нового элемента в список. В работе [18] описываются способы улучшения продуктивности этого метода путем использования нескольких иерархически связанных временных колес с целью оптимизации производительности таймера.

5.2.2. Будильники

Процесс может запросить ядро системы послать ему сигнал через определенный промежуток времени, что похоже на работу обычного будильника. Существуют три типа *будильников*: реального времени, виртуального времени и профиля процесса. *Будильник реального времени* работает в действительном времени, по истечении заданного промежутка которого ядро посыпает процессу сигнал SIGALRM. *Будильник профиля процесса* измеряет время работы процесса и использует сигнал SIGPROF. *Будильник виртуального времени* следит только за количеством времени работы процесса в режиме задачи и посыпает сигнал SIGVTALRM.

В BSD UNIX существует системный вызов setitimer, который позволяет процессу запросить будильник любого типа, указывая интервал времени в микросекундах. Внутри системы этот интервал преобразуется в соответствующее ему количество тиков, так как именно тик является минимальной временной единицей, различаемой ядром. В System V для обращения к будильнику реального времени применяется системный вызов alarm. Задаваемый ему интервал времени должен выражаться в секундах. В систему SVR4 был добавлен системный вызов hrtsys, который обеспечивает таймерный интерфейс с высоким разрешением, позволяющим задавать временной интервал в микросекундах. Этот вызов позволяет реализовать совместимость с BSD посредством setitimer (а также getitimer, gettimeofday и settimeofday) в виде библиотечных функций.

Высокое разрешение будильников реального времени не предполагает высокой точности их работы. Предположим, что приложение использует будильник реального времени для воспроизведения звука по истечении 60 миллисекунд с момента запроса. После того как пройдет указанный промежуток времени, ядро пошлет сигнал SIGALRM вызывающему процессу. Однако процесс не сможет увидеть (а следовательно, и среагировать) на сигнал до тех пор, пока в очередной раз не будет назначен планировщиком на выполнение. Таким образом, перед началом обработки сигнала неизбежно возникает задержка, величина которой зависит от приоритета процесса и степени загруженности системы. Таймеры высокого разрешения полезны при использовании их

высокоприоритетными процессами, обладающими минимальными задержками при планировании. Но даже и эти процессы могут быть задержаны, если текущий процесс, выполняясь в режиме ядра, еще не достиг точки, с которой он может быть вытеснен. Более подробно описание таймеров реального времени см. в разделе 5.5.4.

Виртуальные будильники и будильники профиля не имеют вышеизложенной проблемы, так как они не привязаны к реальному времени. Точность этих будильников определяется другими факторами. Обработчик прерываний таймера измеряет число полных тиков работы текущего процесса, даже в том случае, если было использовано не целое количество тиков. Таким образом, измеряемый промежуток времени отражает количество прерываний таймера, произошедших в течение выполнения процесса. При большой продолжительности работы эта величина является достаточно точным индикатором. Однако для любого единичного будильника результат будет весьма неточен.

5.3. Цели, стоящие перед планировщиком

Планировщик должен по возможности наиболее справедливо распределять процессорное время между всеми процессами системы. Естественно также и то, что при увеличении общей загруженности системы каждый процесс получает меньшее количество процессорного времени и, следовательно, работает медленнее, чем на незагруженной системе. Планировщик должен следить за тем, чтобы система предоставляла приемлемую производительность каждому приложению при общей занятости системы в рамках нормы.

Обычно операционная система предоставляет возможность одновременного выполнения нескольких приложений. Эти приложения можно весьма фривольно категоризировать по следующим классам, в зависимости от их требований к планированию и производительности работы:

- ◆ **Интерактивные приложения.** Приложения типа командных интерпретаторов, редакторов и программ с графическим пользовательским интерфейсом, постоянно взаимодействующих с пользователями. Большую часть времени такие приложения находятся в ожидании действий пользователя, таких как ввод с клавиатуры, либо манипуляций мышью. После получения ввода приложение должно быстро его обработать, иначе пользователь будет наблюдать «торможение» системы. Следовательно, системе необходимо уменьшать среднее время между действием пользователя и реакцией приложения на него, так, чтобы пользователь не заметил задержки. Приемлемая величина задержек реакции на ввод с клавиатуры или движения мыши составляет 50–150 миллисекунд.
- ◆ **Пакетные приложения.** К ним относятся такие действия, как сборка программ или научные вычисления, то есть программы, не требующие взаимодействия с пользователем и часто выполняющиеся в фоновом

режиме. Для таких задач мерой эффективности планирования является время завершения их работы при функционировании среди других процессов, сравниваемое со временем их выполнения как единственной задачи системы.

- ◆ **Приложения реального времени.** Это достаточно широкий класс приложений, для которых время часто является весьма критичным. Хотя на сегодняшний день существует множество различных видов приложений реального времени, обладающих отличными друг от друга наборами требований, между ними существует множество сходств. Обычно все они требуют упреждающего поведения планировщика, с гарантированными границами времени реакции. Например, приложение работы с видео может выводить фиксированное количество *кадров в секунду* (frames per second, fps). Такое приложение может больше заботиться об уменьшении времени реакции системы, нежели просто запрашивать дополнительное процессорное время. Пользователи могут настраивать частоту кадров в диапазоне 10–30 fps со средней частотой в 20 fps.

На рабочей станции одновременно способны выполняться сразу нескольких типов приложений. Планировщик должен пытаться сбалансировать требования каждого из них. Также он обязан следить за тем, чтобы функции ядра, такие как поддержка страничной памяти, обработка прерываний и управление процессами, могли работать именно тогда, когда это требуется.

В хорошо сбалансированной системе все приложения должны продолжать выполнение. Ни одно приложение не должно влиять на продолжение выполнения остальных, кроме случая, когда пользователь принудительно укажет на это. Более того, система обязана постоянно находиться в состоянии получения и обработки интерактивного пользовательского ввода, в противном случае пользователь не будет иметь возможности управлять системой.

Выбор политики планирования оказывает важное влияние на способность системы удовлетворять требованиям различных типов приложений. Следующий раздел описывает традиционный вариант планировщика (SVR3/4.3BSD), который поддерживает только интерактивные и пакетные задания. Остальная часть пятой главы посвящена планировщикам современных систем UNIX, поддерживающих приложения реального времени.

5.4. Планирование в традиционных системах UNIX

В этом разделе обсуждается традиционный алгоритм планирования, применяемый как в SVR3, так и в 4.3BSD. Эти ОС создавались как системы разделения времени, обладающие интерактивными средствами для нескольких пользователей, которые могли одновременно запускать несколько пакетных

и интерактивных процессов. Цель политики планирования заключается в увеличении скорости реакции при интерактивном взаимодействии пользователя с системой, одновременно отслеживая протекание фоновых задач, защищая их от зависания из-за недостатка процессорного времени.

Механизм планирования в традиционных системах базируется на приоритетах. Каждый процесс обладает приоритетом планирования, изменяющимся с течением времени. Планировщик всегда выбирает процессы, обладающие более высоким приоритетом. Для диспетчеризации процессов с равным приоритетом применяется вытесняющее квантование времени. Изменение приоритетов процессов происходит динамически на основе количества используемого ими процессорного времени. Если какой-либо из высокоприоритетных процессов будет готов к выполнению, планировщик вытеснит ради него текущий процесс даже в том случае, если тот не израсходовал свой *квант времени*.

Традиционное ядро UNIX является строго невытесняющим. Если процесс выполняется в режиме ядра (например, в течение исполнения системного вызова или прерывания), то ядро не заставит такой процесс уступить процессор какому-либо высокоприоритетному процессу. Выполняющийся процесс может только добровольно освободить процессор в случае своего блокирования в ожидании ресурса, иначе он может быть вытеснен при переходе в режим задачи. Реализация ядра невытесняющим решает множество проблем синхронизации, связанных с доступом нескольких процессов к одним и тем же структурам данных ядра (см. раздел 2.5).

Следующие подразделы посвящены описанию устройства и реализации планировщика в системе 4.3BSD. Вариант, применяемый в SVR3, имеет лишь незначительные отличия во второстепенных деталях, таких как имена некоторых функций и переменных.

5.4.1. Приоритеты процессов

Приоритет процесса задается любым целым числом, лежащим в диапазоне от 0 до 127. Чем меньше такое число, тем выше приоритет. Приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетом в диапазоне 50–127. Структура `proc` содержит следующие поля, относящиеся к приоритетам:

<code>p_prī</code>	Текущий приоритет планирования
<code>p_usrprī</code>	Приоритет режима задачи
<code>p_cpri</code>	Результат последнего измерения использования процессора
<code>p_nice</code>	Фактор «любезности», устанавливаемый пользователем

Поля `p_prī` и `p_usrprī` применяются для различных целей. Планировщик использует `p_prī` для принятия решения о том, какой процесс направить на вы-

полнение. Когда процесс находится в режиме задачи, значение его `p_pr1` идентично `p_usrpr1`. Когда процесс просыпается после блокирования в системном вызове, его приоритет будет временно повышен для того, чтобы дать ему предпочтение для выполнения в режиме ядра. Следовательно, планировщик использует `p_usrpr1` для хранения приоритета, который будет назначен процессу при возврате в режим задачи, а `p_pr1` — для хранения временного приоритета для выполнения в режиме ядра.

Ядро системы связывает *приоритет сна* с событием или ожидаемым ресурсом, из-за которого процесс может заблокироваться. Приоритет сна является величиной, определяемой для ядра, и потому лежит в диапазоне 0–49. Например, значение приоритета сна для терминального ввода равно 28, в то время как для операций дискового ввода-вывода оно имеет значение 20. Когда замороженный процесс просыпается, ядро устанавливает значение его `p_pr1`, равное приоритету сна события или ресурса¹. Поскольку приоритеты ядра выше, чем приоритеты режима задачи, такие процессы будут назначены на выполнение раньше, чем другие, функционирующие в режиме задачи. Такой подход позволяет системным вызовам быстро завершать свою работу, что является желательным, так как процессы во время выполнения вызова могут занимать некоторые ключевые ресурсы системы, не позволяя пользоваться ими другим.

Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в значение текущего приоритета в режиме задачи. Измененный таким образом приоритет может оказаться ниже, чем приоритет какого-либо иного запущенного процесса; в этом случае ядро системы произведет переключение контекста.

Приоритет в режиме задачи зависит от двух факторов: «любезности» (*nice*) и последней измеренной величины использования процессора. *Степень любезности* (*nice value*) является числом в диапазоне от 0 до 39 со значением 20 по умолчанию. Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения степени благоприятствия. Уменьшить эту величину для какого-либо процесса может только суперпользователь, поскольку при этом поднимется его приоритет. Степень любезности называется так потому, что одни пользователи могут быть поставлены в более выгодные условия другими пользователями посредством увеличения кем-либо из последних значения уровня любезности для своих менее важных процессов².

Системы разделения времени пытаются выделить процессорное время таким образом, чтобы конкурирующие процессы получили его примерно в рав-

¹ На котором он был заблокирован. — Прим. ред.

² Команда `nice(1)` принимает любые значения в диапазоне от -20 до 19 (отрицательные величины может задавать только суперпользователь). Это число используется как увеличение текущего значения фактора любезности.

ных количествах. Такой подход требует слежения за использованием процессора каждым из процессов. Поле `p_cri` структуры `proc` содержит величину результата последнего сделанного измерения использования процессора процессом. При создании процесса значение этого поля инициализируется нулем. На каждом тике обработчик таймера увеличивает `p_cri` на единицу для текущего процесса до максимального значения, равного 127. Более того, каждую секунду ядро системы вызывает процедуру `schedcpu()` (запускаемую через отложенный вызов), которая уменьшает значение `p_cri` каждого процесса исходя из *фактора «полураспада»* (decay factor). В системе SVR3 используется фиксированное значение этого фактора, равное $\frac{1}{2}$. В 4.3BSD для расчета фактора полураспада применяется следующая формула:

```
decay = (2*load_average)/(2*load_average + 1);
```

где `load_average` — это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду. Процедура `schedcpu()` также пересчитывает приоритеты для режима задачи всех процессов по формуле

```
p_usrpri = PUSER + (p_cpu/4) + (2*p_nice);
```

где `PUSER` — базовый приоритет в режиме задачи, равный 50.

В результате, если процесс в последний раз¹ использовал большое количество процессорного времени, его `p_cri` будет увеличен. Это приведет к росту значения `p_usrpri` и, следовательно, к понижению приоритета. Чем дольше процесс пристаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его `p_cri`, что приводит к повышению его приоритета. Такая схема предотвращает зависание низкоприоритетных процессов по вине операционной системы. Ее применение предпочтительнее процессам, осуществляющим много операций ввода-вывода, в противоположность процессам, производящим много вычислений. Если процесс большинство времени выполнения тратит на ожидание ввода-вывода (например, командный интерпретатор или текстовый редактор), то он остается с высоким приоритетом и, таким образом, быстро получает процессор при необходимости. Вычислительные приложения, такие как компиляторы или редакторы связей, обычно обладают более высокими значениями `p_cri` и работают на значительно более низких приоритетах.

Фактор использования процессора обеспечивает справедливость и равенство при планировании процессов в режиме разделения времени. Основная идея заключается в хранении приоритетов всех таких процессов примерно в том же диапазоне в течение некоторого периода времени. Приоритеты могут повышаться или понижаться в рамках этого диапазона в зависимости от того, сколько процессорного времени эти процессы получали в последний

¹ До вытеснения другим процессом. — Прим. ред.

раз. Если приоритеты будут меняться слишком медленно, процессы, начавшие работу с низким приоритетами, сохранят их в течение долгого периода времени, что приведет к их фактическому зависанию.

Фактор полураспада обеспечивает экспоненциально взвешенное среднее значение использования процессора в течение всего периода функционирования процесса. Формула, применяемая в системе SVR3, подсчитывает простое экспоненциальное среднее, что имеет побочный эффект, заключающийся в росте приоритетов при увеличении загрузки системы [2]. Рост происходит по причине того, что на сильно загруженной системе каждый процесс получает меньшее процессорное время. При этом величина использования процессора процессом остается низкой, поэтому фактор полураспада со временем еще дополнительно ее сокращает. В результате использование процессора не сильно влияет на приоритеты, и процессы, начавшие работу с более низкими приоритетами, простояивают в ожидании процессора непропорционально.

В системе 4.3BSD фактор полураспада зависит от загруженности системы. Если загрузка велика, он влияет несущественно. Следовательно, для процессов, получающих процессорное время, снижение приоритетов будет происходить быстро.

5.4.2. Реализация планировщика

Планировщик содержит массив `qs`, состоящий из 32 очередей выполнения (рис. 5.2). Каждая очередь соответствует четырем соседствующим приоритетам. Таким образом, очередь 0 используется для приоритетов 0–3, очередь 1 для приоритетов 4–7 и т. д. Каждая очередь содержит начало двунаправленного связанного списка структур `proc`. Глобальная переменная `whichqs` хранит битовую маску, в которой для каждой очереди зарезервирован один бит. Бит устанавливается, если в очереди имеется хотя бы один процесс. В очередях планировщика находятся только готовые к выполнению процессы.

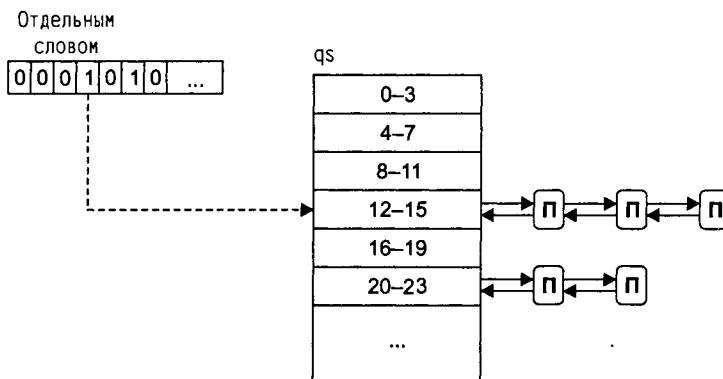


Рис. 5.2. Структуры данных планировщика в системе BSD

Использование массива упрощает задачу выбора процесса для выполнения. Процедура `swtch()`, производящая переключение контекста, проверяет `whichqs` и ищет в ней индекс первого установленного бита. Этот индекс используется для идентификации очереди планировщика, содержащей наиболее приоритетные готовые к выполнению процессы. Процедура `swtch()` удаляет процесс из начала очереди и переключает на него контекст выполнения. По выходу из процедуры `swtch()` вновь назначенный процесс продолжает выполнение.

При переключении контекста происходит сохранение состояния регистров текущего процесса (регистров общего назначения, счетчика команд, указателя стека и т. д.) в его блоке управления, являющемся частью области `u`, с последующей загрузкой регистров из сохраненного контекста очередного процесса. Поле `p_addr` структуры `proc` указывает на элементы таблицы страниц области `u`. Это поле использует процедура `swtch()` для нахождения блока PCB очередного процесса.

Так как машина VAX-11 стала основной платформой для ранних реализаций систем 4BSD и System V, ее архитектура оказала сильное влияние на реализацию планировщика. VAX имела две специальные инструкции — FFS (Find First Set, найти первый установленный) и FFC (Find First Clear, найти первый очищенный), применяемые для работы с 32-битовыми полями. Это послужило причиной размещения 128 приоритетов в 32 очередях. В VAX также имелись специальные инструкции (INSQHI и REMQHI) для атомарных операций вставки и удаления элементов из двунаправленных связанных списков, а также другие команды (LDPCTX и SVPCTX), загружающие и сохраняющие контекст процесса. Все это позволило VAX отрабатывать весь алгоритм планирования, используя при этом небольшой набор машинных инструкций.

5.4.3. Операции с очередью выполнения

Процесс, обладающий наивысшим приоритетом, запускается всегда, если только текущий процесс не выполняется в режиме ядра. Каждому процессу назначается квант времени фиксированного размера (в системе 4.3BSD это 100 миллисекунд). Это обстоятельство действует только при планировании тех процессов, которые находятся в одной очереди выполнения. Через каждые 100 миллисекунд ядро (через отложенный вызов) вызывает процедуру `roundrobin()` для постановки на выполнение очередного процесса из одной и той же очереди. Если в состоянии готовности к выполнению окажутся процессы с более высоким приоритетом, то они будут назначены на выполнение без ожидания вызова `roundrobin()`. *Если все остальные процессы, готовые к выполнению, находятся в очередях с более низким приоритетом, то текущий процесс продолжит выполнение даже после выработывания отведенного ему кванта времени.*

Процедура `schedcpri()` пересчитывает приоритет каждого процесса каждую секунду. Так как приоритет процесса, находящегося в очереди на выполнение, не может быть изменен, процедура `schedcpri()` извлекает процесс из очереди, меняет его приоритет и помещает его обратно, однако не обязательно в ту же очередь. Обработчик прерываний таймера пересчитывает приоритет текущего процесса через каждые четыре тика.

Существуют три ситуации, при которых возникает переключение контекста.

- ◆ Если текущий процесс блокируется в ожидании ресурса или завершает работу. При этом происходит свободное переключение контекста.
- ◆ Если в результате, полученном процедурой пересчета приоритетов, оказалось, что другой процесс обладает более высоким приоритетом по сравнению с текущим.
- ◆ Если текущий процесс или обработчик прерываний разбудил более приоритетный процесс.

Свободное переключение контекста является непосредственным, так как ядро системы напрямую вызывает процедуру `swtch()` из процедуры `sleep()` или `exit()`. События, препятствующие свободному переключению контекста, обычно происходят во время функционирования системы в режиме ядра, следовательно, текущий процесс не может быть вытеснен немедленно. Ядро устанавливает флаг `runrun`, указывающий на ожидание в очереди процесса с более высоким приоритетом. Перед возвратом текущего процесса в режим задачи ядро проверяет этот флаг. Если он окажется установленным, ядро передаст управление процедуре `swtch()`, которая инициирует переключение контекста.

5.4.4. Анализ

Классический алгоритм планирования является простым и эффективным. Он вполне приемлем для систем разделения времени как класса, со смешанным выполнением интерактивных и пакетных заданий. Динамический пересчет приоритетов защищает процессы от зависания из-за недостатка процессорного времени. Традиционный метод наиболее подходит для задач, производящих большой объем ввода-вывода и требующих небольших и нечастых «захватов» процессора.

Однако традиционный планировщик имеет ряд ограничений, которые делают его непригодным для использования с широким спектром коммерческих приложений:

- ◆ он не слишком хорошо умеет масштабировать: если общее количество процессов велико, он неэффективно пересчитывает приоритеты каждую секунду;
- ◆ не существует способов гарантированного предоставления части процессорного времени как ресурса определенной группе процессов или конкретному процессу;

- ◆ не существует никакого гарантированного времени реакции приложений, имеющих свойства приложений реального времени;
- ◆ приложения имеют скудные возможности для управления собственным приоритетом. Механизм, работающий через значение любезности, является слишком примитивным и не отвечающим поставленным требованиям;
- ◆ поскольку ядро системы является невытесняющим, высокоприоритетные готовые к выполнению процессы могут находиться в ожидании в течение довольно значительного интервала времени. Такое свойство системы называется *инверсией приоритетов*.

Современные системы UNIX применяются в самых различных областях деятельности. В частности, существует острая необходимость в планировщике, поддерживающем приложения реального времени, которым необходимо более предсказуемое поведение и ограниченное время реакции. Решение этой проблемы потребовало провести полную переработку планировщика. Все последующие разделы главы посвящены новым средствам планирования, реализованным в системах SVR4, Solaris 2.x, OSF/1 и в некоторых менее распространенных вариантах UNIX.

5.5. Планировщик в системе SVR4

В системе SVR4 был представлен полностью переработанный планировщик [1], в котором разработчики попытались улучшить традиционный метод планирования. Он оказался применимым для широкого спектра приложений за счет большей гибкости и управляемости. Ниже представлены основные качества новой архитектуры.

- ◆ Поддержка более широкого диапазона приложений, в том числе и требующих работы в режиме реального времени.
- ◆ Отделение политики планирования от механизма ее реализации.
- ◆ Предоставление приложениям больших возможностей по управлению своими приоритетами и планированию.
- ◆ Определение механизма планирования с хорошо описанным интерфейсом взаимодействия с ядром системы.
- ◆ Возможность добавления новых политик планирования как отдельных модулей, включая динамически загружаемые реализации планировщика.
- ◆ Ограничение на допустимую задержку реакции для критичных ко времени приложений.

Несмотря на то что некоторые усилия были направлены на поддержку приложений реального времени, данная архитектура оказалась, в общем, дос-

таточной для удовлетворения многих других требований планирования. Фундаментальным понятием архитектуры SVR4 явился *класс планирования*, определяющий политику планирования для всех процессов, относящихся к нему. По умолчанию в системе SVR4 поддерживаются два класса: класс разделения времени и класс реального времени.

Планировщик обеспечивает набор процедур, независимых от используемого класса, в которых реализованы общие службы, такие как переключение контекста, управление очередью выполнения и вытеснение. Он также определяет процедурный интерфейс для классо-зависимых функций, таких как расчет приоритета и наследование. Для каждого класса эти функции реализованы по-разному. Например, класс реального времени использует фиксированные приоритеты, в то время как в классе разделения времени приоритет процесса изменяется динамически, как реакция на определенные события.

Этот объектно-ориентированный подход похож на тот, который применяется в архитектуре vnode/vfs (см. раздел 8.6) и подсистеме памяти (см. раздел 14.3). В разделе 8.6.2 показаны основные концепции объектно-ориентированного подхода, используемого в современных системах UNIX. Согласно этому подходу, планировщик является *абстрактным базовым классом*, а каждый класс планирования становится его подклассом (или порожденным им классом).

5.5.1. Независимый от класса уровень

Независимый от класса уровень отвечает за переключение контекста, управление очередью выполнения и вытеснение. Высокоприоритетный процесс всегда получает процессор (кроме случаев невытесняемой обработки в ядре). Количество приоритетов увеличено до 160, и для каждого из них теперь используется отдельная очередь. В отличие от традиционной реализации больший номер приоритета соответствует более высокому приоритету. Однако назначение и пересчет приоритетов процессов совершается на классо-зависимом уровне.

На рис. 5.3 показаны структуры данных, используемые для управления очередью выполнения. Переменная `dqactmap` является битовой маской, показывающей, какая из очередей выполнения содержит по крайней мере один готовый к выполнению процесс. Для помещения процесса в очередь применяются вызовы `setfrontdq()` и `setbackdq()`, для удаления — `dispdeq()`. Эти функции могут быть вызваны как из основного кода ядра, так и из классо-зависимых процедур планировщика. Обычно очередной готовый к выполнению процесс помещается в конец своей очереди, в то время как процесс, вытесненный до окончания отведенного ему кванта времени, возвращается в начало очереди.

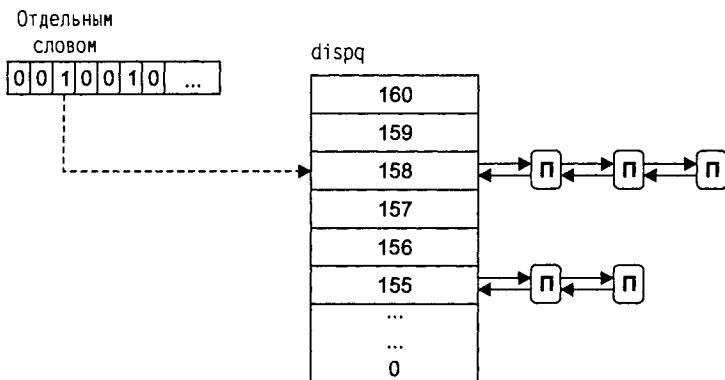


Рис. 5.3. Очереди отправки на выполнение в системе SVR4

Главное ограничение системы UNIX, относящееся к работе приложений реального времени, заключается в невытесняющей природе ядра. Для таких процессов необходимо, чтобы *задержки обслуживания планировщиком* (*dispatch latency*) (временные промежутки между моментом перехода процесса в состояние готовности к выполнению и началом его выполнения) были минимальны. Если процесс реального времени становится готовым к выполнению в то время, когда текущий процесс выполняет системный вызов, задержка перед переключением контекста может оказаться значительной.

Для решения этой проблемы в ядре системы SVR4 были определены несколько *точек вытеснения*. Эти точки являются определенными местами в коде ядра, в которых все структуры данных находятся в устойчивом состоянии, а ядро системы готово начать производство большого объема операций. Когда достигается одна из точек вытеснения, ядро проверяет флаг *kprunrun*, который указывает на готовность к выполнению процесса реального времени. Если флаг установлен, ядро системы вытеснит текущий процесс. Такой подход ограничивает время ожидания процесса реального времени перед тем, как он будет назначен на выполнение¹. Макроопределение *PREEMPT()* проверяет флаг *kprunrun* и вызывает для вытеснения процесса процедуру *preempt()*. Ниже перечислены некоторые примеры точек вытеснения.

- ◆ В процедуре разбора имен путей *lookuppri()*: перед началом анализа каждого индивидуального компонента имени пути.
- ◆ В системном вызове *open*: перед созданием файла, если тот не существует.
- ◆ В подсистеме памяти: перед освобождением страниц памяти, занимаемых процессом.

¹ Этот код не является классо-зависимым, несмотря на явное упоминание процессов реального времени. Ядро только проверяет *kprunrun* для определения, должно ли оно вытеснить текущий процесс. На текущий момент только класс реального времени устанавливает этот флаг, однако в будущем может появиться и новый класс, также требующий вытеснения процессов, работающих в режиме ядра.

Флаг `runrun` используется точно так же, как в традиционных системах, и приводит к вытеснению процессов только при возврате их в режим задачи. Функция `preempt()` вызывает операцию `CL_PREEMPT` для выполнения классо-зависимых действий. Затем она вызывает `swtch()` для инициализации переключения контекста.

Функция `swtch()` вызывает `pswtch()` для проведения машинно-независимой части переключения контекста, после чего вызывает подпрограммы низкого уровня¹ для манипуляций с контекстом регистров, очистки буферов трансляции и т. д. Функция `pswtch()` сбрасывает флаги `runrun` и `krunrun`, выбирает самый высокоприоритетный готовый к выполнению процесс и удаляет его из очереди отправки на выполнение. Эта функция также обновляет `dqactmap` и устанавливает состояние процесса в значение `SONPROC` (выполняемый на процессоре). В завершение она производит обновление регистров управления памяти, отраженных в области `u`, и карт трансляции адресов этого процесса.

5.5.2. Интерфейс с классами планирования

Вся классо-зависимая функциональность обеспечивается через общий интерфейс, реализация виртуальных функций (см. раздел 8.6.2) которого индивидуальна для каждого класса. Интерфейс определяет как семантику этих функций, так и связи, используемые в вызове определенной реализации для класса.

На рис. 5.4 показана реализация классо-зависимого интерфейса. Структура `classfuncs` является вектором указателей на функции, реализующие классо-зависимый интерфейс какого-либо класса. В глобальной таблице классов для каждого класса отводится по одному элементу. Такой элемент содержит имя класса, указатель на функцию инициализации и указатель на вектор `classfuncs` для этого класса.

Когда процесс создается, он наследует приоритет класса от своего родителя. Впоследствии процесс может быть переведен и в другой класс при помощи системного вызова `procctl`, описанного в разделе 5.5.5. Структура `proc` имеет три поля, используемых классами планирования:

<code>p_cid</code>	Идентификатор класса, являющийся индексом в глобальной таблице классов
<code>p_clfuncs</code>	Указатель на вектор <code>classfuncs</code> для класса, к которому относится процесс. Указатель копируется из соответствующего элемента глобальной таблицы классов
<code>p_clproc</code>	Указатель на приватную классо- зависимую структуру данных

Набор макроопределений преобразует вызовы, сделанные через общий интерфейс функций, в соответствующие классо-зависимые функции. Например:

```
#define CS_SLEEP(proc, clproc, ...) \
(*(proc)->p_clfuncs->cl_sleep)(clproc, ...)
```

¹ Автор, видимо, имел в виду машинно-зависимые коды. — Прим. ред.

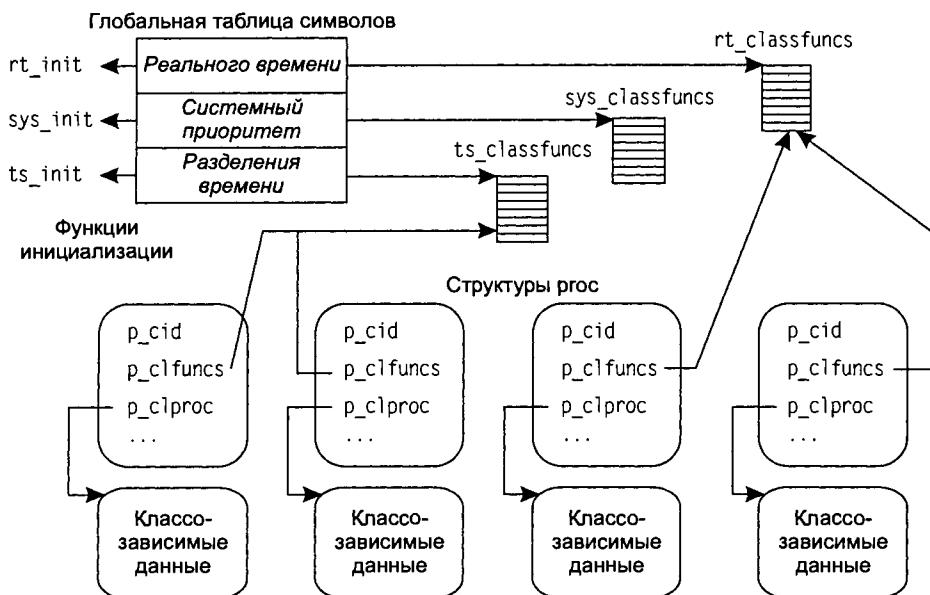


Рис. 5.4. Классо-зависимый интерфейс системы SVR4

Таким образом, классо-зависимые функции могут быть вызваны из классо-независимого кода, а также через системный вызов `prioctl`.

Класс планирования сам выбирает политику для расчета приоритетов и планирования процессов, относящихся к этому классу. Он определяет диапазон приоритетов для принадлежащих ему процессов, а также условия, при которых приоритет процесса может быть изменен. Класс также задает размер кванта времени работы процесса. Квант может быть как одинаковым для всех процессов, так и зависимым от приоритета процесса. Величина кванта может лежать в диапазоне от одного тика до бесконечности. Установка «бесконечного» кванта наиболее подходит для некоторых задач реального времени, которые должны находиться в режиме выполнения постоянно вплоть до завершения.

Точки входа классо-зависимого интерфейса включают в себя:

<code>CL_TICK</code>	Вызывается из обработчика прерываний таймера. Применяется для отслеживания квантов времени, пересчета приоритетов, обработки исчерпания кванта и т. д.
<code>CL_FORK, CL_FORKRET</code>	Вызываются из <code>fork</code> . <code>CL_FORK</code> инициализирует определенную для класса структуру данных потомка. <code>CL_FORKRET</code> может устанавливать флаг <code>runrun</code> , позволяя процессу-потомку стартовать до своего предка
<code>CL_ENTERCLASS, CL_EXITCLASS</code>	Вызываются при перемещении процесса в класс планирования и из него. Отвечают за размещение и освобождение определенной для класса структуры данных соответственно

CL_SLEEP	Вызывается из <code>sleep()</code> . Может пересчитывать приоритет процесса
CL_WAKEUP	Вызывается из <code>wakeprocs()</code> . Помещает процесс в подходящую для него очередь выполнения, может устанавливать флаги <code>runrun</code> и <code>kprunrun</code>

Класс планирования сам решает, какие действия будет производить каждая функция, и в каждом классе эти функции могут быть определены по-своему. Такой подход придает общему планированию большую гибкость. Например, обработчик прерываний таймера традиционного планировщика производит подсчет каждого типа текущего процесса и пересчитывает его приоритет на каждом четвертом тике. В новой архитектуре обработчик вызывает процедуру `CL_TICK`, определенную для класса, которому принадлежит процесс. Процедура самостоятельно решает, как обрабатывать тики времени. Например, класс реального времени использует постоянные приоритеты и не требует их пересчета. Классо-зависимый код определяет момент, когда происходит исчерпание выделенного кванта времени, и устанавливает флаг `runrun` для инициализации процедуры переключения контекста.

По умолчанию все 160 поддерживаемых приоритетов разделяются на следующие три категории:

0–59	класс разделения времени
60–99	системные приоритеты
100–159	класс реального времени

Дальнейшие подразделы посвящены описанию реализации классов разделения времени и реального времени.

5.5.3. Класс разделения времени

Класс *разделения времени* является классом по умолчанию для процесса. Он изменяет приоритеты процесса динамически и реализует *карусельное планирование* для процессов одного приоритета. Класс использует для управления приоритетами процессов и квантами времени статическую *таблицу параметров планировщика*. Величина кванта времени, выделяемого конкретному процессу, зависит от его приоритета. В таблице параметров определяются кванты времени для каждого приоритета. По умолчанию чем меньше приоритет процесса, тем больше времени ему предоставляется для выполнения. Такой подход может показаться нелогичным, однако причиной его применения является тот факт, что низкоприоритетные процессы выполняются нечасто и должны получать больше времени, когда наступает их очередь выполнения.

Класс разделения времени использует планирование, управляемое по событиям [17]. Вместо пересчета приоритетов всех процессов каждую секунду

в системе SVR4 изменение приоритета процесса происходит в ответ на определенные события, относящиеся к этому процессу. Планировщик уменьшает приоритет процесса каждый раз, как он вырабатывает отведенный ему квант времени. С другой стороны, система увеличивает приоритет процесса, если тот блокируется по событию или находится в ожидании ресурса или если он долгое время не может израсходовать свой квант времени. Так как каждое событие обычно относится только к одному процессу, пересчет приоритета происходит очень быстро. В таблице параметров диспетчера определено, как различные события влияют на приоритет процесса.

Класс разделения времени использует для хранения классо-зависимых данных структуру `tsproc`. Эта структура включает следующие поля:

<code>ts_timeleft</code>	оставшееся время кванта
<code>ts_cpupri</code>	системную часть приоритета
<code>ts_upri</code>	прикладную часть приоритета (фактор «любезности»)
<code>ts_utmdpri</code>	приоритет режима задачи (<code>ts_cpupri + ts_upri</code> , но не более чем 59)
<code>ts_dispwait</code>	число секунд таймера после начала кванта

Когда процесс возобновляет свою работу после сна, находясь в режиме ядра, его приоритет соответствует приоритету ядра и определяется условиями сна. Когда позже процесс возвращается в режим задачи, его приоритет восстанавливается из `ts_utmdpri`. Приоритет режима задачи является суммой `ts_cpupri` и `ts_upri`, но ограничивается диапазоном значений от 0 до 59. Значение `ts_upri` лежит в диапазоне от -20 до +19 и по умолчанию равно 0. Значение этого поля может быть изменено вызовом `priocntl`, однако права на его увеличение даны только суперпользователю системы. Значение `ts_cpupri` выбирается из таблицы параметров диспетчера, как это будет описано ниже.

Таблица параметров содержит по одному элементу для каждого приоритета класса. Хотя в системе SVR4 каждый класс обладает такой таблицей (плюс еще одна для системных приоритетов), каждая из этих таблиц имеет различный формат. Она не является обязательной структурой для каждого класса, существует также возможность создания нового класса без таблицы. Для класса разделения времени каждый элемент таблицы состоит из следующих полей:

<code>ts_globpri</code>	Глобальный приоритет элемента (для класса разделения времени это то же самое, что индекс таблицы)
<code>ts_quantum</code>	Квант времени для этого приоритета
<code>ts_tqexp</code>	Новое значение <code>ts_cpupri</code> , устанавливаемое при исчерпании кванта времени
<code>ts_slpret</code>	Новое значение <code>ts_cpupri</code> , устанавливаемое при возврате в режим задачи после сна

<code>ts_maxwait</code>	Количество секунд ожидания до исчерпания кванта времени перед применением <code>ts_lwait</code>
<code>ts_lwait</code>	Используется вместо <code>ts_tqexp</code> , если процесс при исчерпании своего кванта затратил времени больше, чем значение <code>ts_maxwait</code>

Таблица может применяться двумя способами. Она может быть проиндексирована по текущим значениям поля `ts_cprpri` для получения доступа к полям `ts_tqexp`, `ts_slpret` и `ts_lwait`, так как эти поля предоставляют новое значение `ts_cprpri`, основанное на его прежнем значении. Таблица может быть проиндексирована по полю `ts_umdpri` для получения доступа к `ts_globpri`, `ts_quantum` и `ts_maxwait`, поскольку именно эти поля отвечают за общие приоритеты планирования.

Таблица 5.1 является примером типичной таблицы параметров диспетчера для класса разделения времени. Для того чтобы понять, как она может быть использована, представим процесс, обладающий `ts_upri=14` и `ts_cprpri=1`. Значения его глобального приоритета (`ts_globpri`) и `ts_umdpri` одинаковы и равны 15¹. Если процесс исчерпает выделенный ему квант времени, то его `ts_cprpri` будет установлено в значение 0 (следовательно, его `ts_umdpri` будет равно 14). Однако если процессу необходимо более 5 секунд для того, чтобы использовать отведенный ему квант времени, его `ts_cprpri` будет установлено в значение 11 (следовательно, `ts_umdpri` будет равным 25).

Предположим, что перед тем, как исчерпать отведенный ему квант времени, процесс произведет системный вызов, и это приведет к необходимости его блокирования в ожидании ресурса. Когда процесс возобновит работу и, в конечном счете, перейдет в режим задачи, его `ts_cprpri` установится в 11 (из колонки `ts_slpret`²), а поле `ts_umdpri` станет равным 25, в зависимости от того, сколько времени понадобится процессу на исчерпание отведенного ему кванта времени.

Таблица 5.1. Таблица параметров диспетчера для класса разделения времени

Индекс	<code>globpri</code>	<code>Quantum</code>	<code>tqexp</code>	<code>slpret</code>	<code>maxwait</code>	<code>Lwait</code>
0	0	100	0	10	5	10
1	1	100	0	11	5	11
...
15	15	80	7	25	5	25
...
40	40	20	30	50	5	50
...
59	59	10	49	59	5	59

¹ По всей видимости, здесь должно быть `ts_globpri=1`. — Прим. ред.

² Точнее, из Slpret. — Прим. ред.

5.5.4. Класс реального времени

Класс реального времени использует приоритеты в диапазоне 100–159. Эти приоритеты являются более высокими, чем у процессов, принадлежащих классу разделения времени, даже в случае выполнения их в режиме ядра. Это означает, что процессы реального времени будут выбраны для выполнения перед любым процессом, выполняющимся в режиме ядра. Предположим, что текущий процесс работает в режиме ядра в то время, когда процесс из класса реального времени становится готовым к выполнению. Ядро не может сразу же вытеснить текущий процесс, так как это способно повлечь нестабильное состояние системы. Процесс реального времени должен ожидать момента, когда текущий процесс начнет переключаться в режим задачи или достигнет одной из точек вытеснения ядра. В класс реального времени могут войти только процессы суперпользователя, для этого они используют вызов `priocntl`, в параметрах которого указываются необходимый приоритет и квант времени.

Процессы реального времени характеризуются постоянным приоритетом и постоянным выделяемым квантом. Единственным способом изменения этих параметров является применение вызова `priocntl`. Таблица параметров диспетчера для класса реального времени является очень простой, так как в ней хранятся только значения квантов времени для каждого приоритета, принятые по умолчанию. Они используются в том случае, если при входении в класс реального времени процесс не указывает квант самостоятельно. Для более низких приоритетов по умолчанию назначаются более продолжительные кванты времени. Классо-зависимые данные процесса реального времени хранятся в структуре `rtproc`, куда входят текущий квант, время, оставшееся до исчерпания кванта, и текущий приоритет.

Процессы реального времени требуют ограничения времени задержек обслуживания планировщиком и времени реакции. Оба этих понятия продемонстрированы на рис. 5.5. Задержка обслуживания планировщиком — это интервал времени между тем моментом, когда процесс становится готовым к выполнению, и тем, когда он начнет свою работу. Время реакции — промежуток времени между возникновением события, требующего реакции от процесса, и моментом начала обработки процессом этого события. Оба этих параметра должны иметь определенную верхнюю границу, лежащую в разумных пределах.

Время реакции является суммой времени, требуемого для обработки события обработчиком, времени задержки обслуживания планировщиком и времени, необходимого для проведения обработки события самим процессом реального времени. Величина задержки обслуживания планировщиком во многом зависит от ядра. Ядра традиционных систем не в состоянии обеспечить приемлемых границ этой задержки, так как они являются невытесняющими, вследствие чего готовый к выполнению процесс класса реального времени может ждать довольно долго, если текущий процесс «застрянет» в некоторой

замысловатой процедуре ядра. Измерения показали, что время выполнения некоторых участков кода ядра может доходить до нескольких миллисекунд, что совершенно неприемлемо для приложений реального времени.

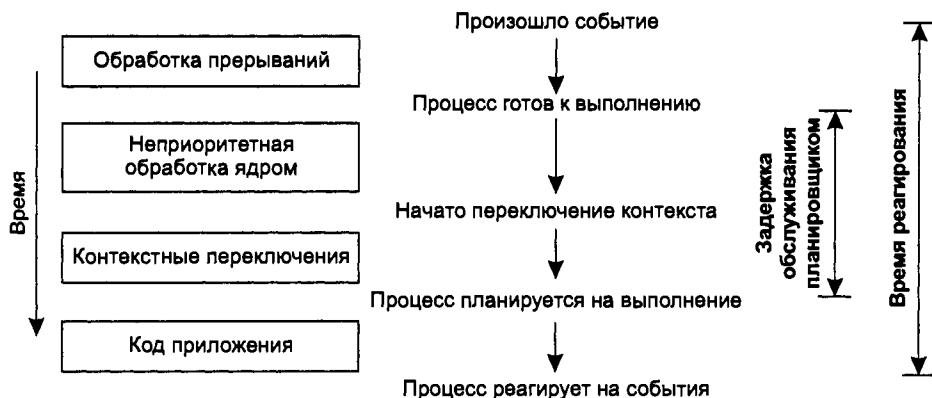


Рис. 5.5. Время реакции и задержки обслуживания планировщиком

В системе SVR4 для разделения длинных алгоритмов ядра на более короткие и ограниченные блоки выполнения применяются точки вытеснения. Когда процесс реального времени становится готовым к выполнению, процедура `rt_wakeup()`, которая обрабатывает классо-зависимую часть процедуры пробуждения, устанавливает определенный в ядре флаг `kprunrun`. Когда текущий процесс (предположительно выполняющийся в режиме ядра) достигает точки вытеснения, ядро проверяет флаг и инициирует переключение контекста для процесса реального времени. Таким образом, время ожидания ограничено временем выполнения максимально длинного участка кода между двумя точками вытеснения, что является удовлетворительным решением для многих задач.

В завершение необходимо упомянуть о том, что соблюдение границ задержки может быть гарантировано только в том случае, если процесс реального времени является самым высокоприоритетным процессом системы. Если в течение любого периода своего ожидания высокоприоритетный процесс становится готовым к выполнению, то он будет назначен на выполнение в первую очередь. После того как процесс получит процессор в свое распоряжение, пересчет задержки начнется с нуля.

5.5.5. Системный вызов `priocntl`

Системный вызов `priocntl` предлагает ряд возможностей для управления приоритетами и планированием процесса. Для него определен набор подкоманд, которые могут быть использованы для различных операций, таких как:

- ◆ изменение приоритета класса процесса;
- ◆ установка значения `ts_upr1` для процессов класса разделения времени;

- ◆ сброс в значение по умолчанию приоритета и кванта времени для процесса реального времени;
- ◆ получение текущего значения ряда параметров планирования.

Большинство из этих операций может производить только суперпользователь, следовательно, они недоступны большинству приложений. В системе SVR4 также имеется системный вызов `priocntlset`, производящий те же операции, но над набором связанных чем-либо процессов, например:

- ◆ всех процессов системы;
- ◆ всех процессов группы или сеанса;
- ◆ всех процессов в определенном классе планирования;
- ◆ всех процессов, принадлежащих определенному пользователю;
- ◆ всех процессов, имеющих одного и того же родителя.

5.5.6. Анализ

В системе SVR4 традиционный планировщик был заменен новым планировщиком, отличным по устройству и по принципам работы. Новый планировщик обеспечивает гибкий подход, позволяющий добавлять новые классы планирования в систему. Производители программного обеспечения получили возможность «скроить» свой планировщик, который бы удовлетворял их приложениям. Применение таблиц параметров диспетчера расширило возможности системного администратора, поскольку появилась способность контролировать поведение системы путем изменения табличных установок и последующей пересборки ядра.

В традиционных системах UNIX пересчет приоритетов процессов ведется каждую секунду. Это может отнимать довольно много времени, если в системе имеется большое количество процессов. Следовательно, такой алгоритм не обладает достаточными возможностями по масштабированию в системах, обслуживающих тысячи процессов. В системе SVR4 класс разделения времени изменяет приоритеты процессов, основываясь на событиях, относящихся к этим процессам. Так как обычно событие относится лишь к одному процессу, применяемый алгоритм обладает высоким быстродействием и масштабируемостью.

Событийно управляемое планирование наиболее подходит для заданий, производящих в основном операции ввода-вывода, и интерактивных заданий, нежели для заданий, ограниченных только обработкой на процессоре. Такой подход имеет несколько недостатков. Пользователи, чьи интерактивные задания также требуют больших объемов вычислений, не сумеют на подобной системе эффективно с ними работать, поскольку такие процессы не смогут вырабатывать достаточное количество увеличивающих приоритет событий для более активного использования процессора. В дополнение к это-

му связанные с событиями операции по оптимальному повышению или понижению приоритетов зависят от общей загруженности системы и характеристик задач, работающих в текущий момент времени. Таким образом, для повышения эффективности системы и скорости реакции эти величины могут часто перенастраиваться.

Добавление какого-либо класса планирования не требует доступа к исходным кодам ядра. Для этой цели разработчик должен произвести следующие последовательные действия.

1. Разработать реализацию каждой классо-зависимой функции нового класса планирования.
2. Инициализировать вектор `classfuncs` указателями на эти функции.
3. Разработать функцию инициализации, выполняющую различные действия (например, выделить память для внутренних структур данных и т. д.) при установке нового класса.
4. Добавить запись для этого класса в таблицу классов в основном файле конфигурации, находящемся обычно в подкаталоге `master.d` каталога сборки ядра. Эта запись должна содержать указатели на функцию инициализации и на вектор `classfuncs`.
5. Произвести пересборку ядра.

Существенным ограничением является то, что в системе SVR4 не обеспечено сколько-нибудь приемлемого способа для перевода процессов класса разделения времени в иной класс. Вызов `priocntl` может быть в распоряжении только суперпользователя. Был бы очень полезен механизм, который умел бы отображать определенные идентификаторы пользователей или программы в класс, отличный от принятого по умолчанию.

Несмотря на то что средства поддержки класса реального времени явились важным шагом, они оказались далеки до соответствия желаемым возможностям. Например, система не поддерживает планирование по крайнему сроку (см. раздел 5.9.2). Код между двумя точками вытеснения все равно остается слишком длинным для многих приложений, критичных ко времени. Более того, настоящие системы реального времени должны обладать полностью вытесняющим ядром. Некоторые из этих аспектов были реализованы в системе Solaris 2.x, в которой были расширены возможности планировщика SVR4. Мы расскажем о них в следующем разделе главы.

Основной проблемой планировщика SVR4 является чрезвычайная сложность более-менее качественной настройки системы при работе со смешанными наборами приложений. В книге [13] описывается эксперимент, при котором в системе одновременно запускались три различных приложения: интерактивный сеанс с клавиатурным вводом данных, пакетное задание и программа для просмотра видео. Это оказалось весьма сложным предприятием, поскольку программа, принимающая клавиатурный ввод, и программа для просмотра видео требовали взаимодействия с X-server.

Для решения этой задачи авторы книги изменяли приоритеты и классы планирования всех четырех процессов (трех приложений и X-server). Выяснилось, что найти комбинацию параметров, обеспечивающих приемлемое протекание работы всех приложений, очень трудно. Например, вполне логичным решением представляется размещение в классе реального времени только программы воспроизведения видео. На деле же оказалось, что даже программа работы с видео не может нормально функционировать. Вся проблема заключалась в X-server, который не получал достаточного количества процессорного времени. Тогда авторы разместили в классе реального времени еще и X-server, в результате чего они добились нормальной скорости воспроизведения видео (после корректного подбора относительных приоритетов), но при этом функционирование интерактивного приложения и пакетного задания застопорилось, а система перестала отвечать на нажатия клавиш и движения мыши.

Возможно, что после длительных экспериментов есть шансы подобрать правильное сочетание приоритетов для данного набора приложений. Однако эти установки будут работать корректно только при использовании именно этого набора программ. В реальности же загрузка системы меняется постоянно, а подстраивать ее вручную при старте каждого нового приложения просто неразумно.

5.6. Расширенные возможности планирования системы Solaris 2.x

В системе Solaris 2.x были расширены возможности архитектуры планирования SVR4 сразу в нескольких направлениях [9]. Solaris является многонитевой, симметрично многопроцессорной операционной системой, следовательно, ее планировщик должен поддерживать все эти особенности. Кроме этого разработчики произвели оптимизацию системы в целях уменьшения задержек обслуживания планировщиком для высокоприоритетных, критичных ко времени процессов. В результате получился планировщик, более пригодный для приложений реального времени.

5.6.1. Вытесняющее ядро

Точки вытеснения в ядре SVR4 являются наилучшим компромиссным решением, позволяющим ограничить задержки, что требуется для процессов реального времени. Ядро системы Solaris 2.x является полностью вытесняющим, что позволяет гарантировать быстроту его реакции. Это явилось радикальным изменением ядра UNIX и имело далеко идущие последствия. Большинство глобальных структур ядра необходимо защищать при помощи соответствующих объектов синхронизации, таких как семафоры или *взаимные исключения* (mutual exclusion locks, mutex). Сделать ядро вытесняющим — очень сложная

задача, однако решение этой проблемы является необходимым требованием для многопроцессорных операционных систем.

Другим подобным изменением в системе Solaris 2.x является реализация прерываний при помощи специальных нитей ядра, использующих стандартные средства синхронизации ядра и блокирующихся в ожидании ресурсов, если это необходимо (см. раздел 3.6.5). В результате в системе нечасто требуется повышать уровень прерываний для защиты критических участков кода и она имеет лишь несколько невытесняемых сегментов кода. Таким образом, высокоприоритетные процессы могут быть назначены на выполнение сразу после того, как они станут готовыми к выполнению.

Нити прерываний всегда выполняются в системе с наивысшими приоритетами. ОС Solaris позволяет динамически загружать классы планирования. При этом приоритеты нитей прерываний пересчитываются для гарантии того, что они останутся с самыми высокими значениями. Если нити прерывания необходимо блокироваться в ожидании ресурса, то возобновить функционирование она сможет только на том же самом процессоре.

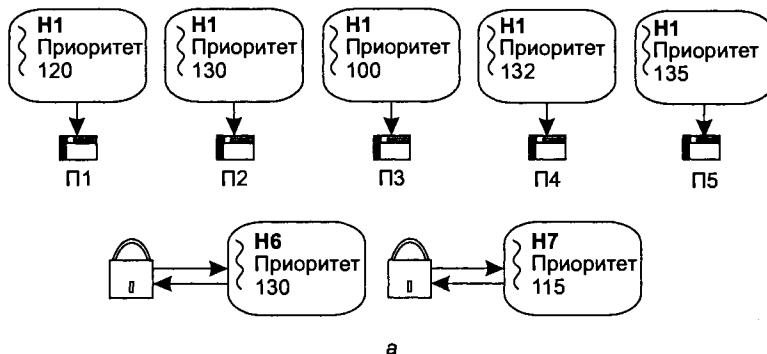
5.6.2. Многопроцессорная поддержка

Система поддерживает единую очередь диспетчеризации для всех процессоров. Тем не менее некоторые нити (например, нити прерываний) могут быть ограничены выполнением только на одном, определенном процессоре. Процессоры могут взаимодействовать друг с другом при помощи отправки *межпроцессорных прерываний*. Каждый процессор обладает следующим набором переменных планирования:

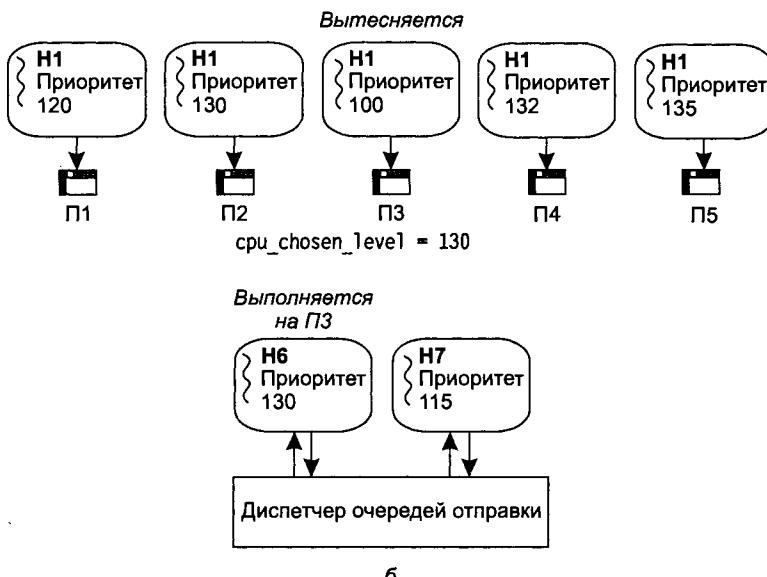
<code>cpu_thread</code>	текущая нить, выполняемая на этом процессоре
<code>cpu_dispthread</code>	нить, выбранная последней для запуска
<code>cpu_idle</code>	нить холостой работы
<code>cpu_runrun</code>	флаг вытеснения, используемый нитями класса разделения времени
<code>cpu_kprunrun</code>	флаг вытеснения, устанавливаемый нитями класса реального времени
<code>cpu_chosen_level</code>	Приоритет нити, собирающейся вытеснить текущую нить

Планирование в многопроцессорных средах показано на рис. 5.6. Событие на процессоре П1 делает нить Н6 (с приоритетом 130) готовой к выполнению. Ядро помещает Н6 в очередь отправки на выполнение и вызывает процедуру `cpri_choose()` для нахождения процессора, на котором выполняется нить, имеющая наименьший приоритет (в данном случае это – П3). Так как приоритет найденной нити окажется меньше, чем у Н6, процедура `cpri_choose()` пометит процессор, на котором эта нить выполняется, для вытеснения, установит его значение переменной `cpri_chosen_level` равным приоритету Н6 (130) и пошлет ему межпроцессорное прерывание. Предположим, что в это время, то есть до того как процессор П3 обработает прерывание и вытеснит нить Н3, другой

процессор, например П2, обрабатывает событие, которое сделает нить Н7 с приоритетом 115 готовой к выполнению. Теперь процедура `cpu_choose()` проверит значение `cpu_chosen_level` процессора П3 и обнаружит его равным 130. Это приведет к тому, что процедура выявит, что на данном процессоре выполняется нить с более высоким приоритетом. Следовательно, процедура `cpu_choose()` оставит Н7 в очереди отправки на выполнение, предупреждая конфликт.



а



б

Рис. 5.6. Многопроцессорное планирование в системе Solaris 2.x: а — начальное состояние; б — после того как Н6 и Н7 станут готовы к выполнению

Существуют определенные ситуации, когда низкоприоритетная нить может заблокировать более высокоприоритетную нить на длительный период времени. Причиной возникновения подобных ситуаций является либо скры-

тое планирование, либо *инверсия приоритетов*. В системе Solaris устраниены подобные проблемы. Как это сделано, описано в следующих разделах главы.

5.6.3. Скрытое планирование

Ядро системы часто совершает некоторые действия асинхронно от имени нити. Ядро планирует эту работу без учета приоритета нити, для которой ее выполняет. Это называется *скрытым планированием*. Примерами таких действий ядра являются процедуры обслуживания STREAMS (см. раздел 17.4) и отложенные вызовы.

В операционной системе SVR4, например, перед каждым возвратом процессы в режим задачи ядро вызывает процедуру `runqueues()` для проверки существования ожидающих запросов на обслуживание. Если таковые существуют, то ядро обрабатывает эти запросы при помощи вызова процедуры `service` соответствующего модуля STREAMS. Таким образом, подобные запросы обслуживаются *текущим процессом* (тем самым, который собирается возвратиться в режим задачи), хотя они относятся к совсем другим процессам. Если приоритет процесса, от чьего имени был сделан запрос, ниже, чем у текущего, запрос будет обработан с неверным приоритетом. В результате нормальная работа текущего процесса прерывается выполнением низкоприоритетной задачи.

В Solaris эта проблема решается посредством перевода функционирования STREAMS на уровень нитей ядра, которые всегда обладают приоритетом меньшим, чем у нитей режима реального времени. Однако такое решение порождает новую проблему: некоторые запросы STREAMS могут быть инициированы нитями реального времени. Поскольку эти запросы также обслуживаются нитями ядра, они обрабатываются с более низким приоритетом, чем должны. Эта проблема не решается без кардинальных изменений в семантике обработки нитей и остается возможным препятствием для функционирования режима реального времени на требуемом уровне.

Также существует проблема, связанная с обработкой отложенных вызовов (см. раздел 5.2.1). В системе UNIX все отложенные вызовы обслуживаются с самым низким уровнем приоритета прерываний, являющимся, однако, выше любого приоритета реального времени. Если такой вызов будет произведен низкоприоритетной нитью, то его обслуживание может задержать постановку на выполнение высокоприоритетной нити. Измерения производительности, проводимые на ранних версиях SunOS, показали, что для обработки очереди отложенных вызовов система способна потратить до 5 миллисекунд.

Для решения означенной проблемы разработчики Solaris переложили обработку таких вызовов на *нить отложенных вызовов*, которая выполняется с максимальным системным приоритетом, который, вместе с тем, ниже любого приоритета реального времени. Вызовы, произведенные процессами реального времени, обрабатываются отдельно и обладают низшими приоритетами прерываний. Это гарантирует своевременное выполнение отложенных вызовов, критичных ко времени.

5.6.4. Инверсия приоритетов

Проблема *инверсии приоритетов* была впервые описана в работе [10] и относится к ситуации, когда низкоприоритетный процесс удерживает ресурс, необходимый процессу с более высоким приоритетом. Таким образом, он блокирует работу более высокоприоритетного процесса. Существует несколько разновидностей сформулированной проблемы. Рассмотрим их на примерах (рис. 5.7).

В простейшем случае нить H1 удерживает ресурс P, который необходим более высокоприоритетной нити H2. Последняя будет ожидать до тех пор, пока H1 не освободит ресурс. Усложним сценарий, добавив нить H3, приоритет которой меньше, чем у H2, но больше, чем у H1 (рис. 5.7, а). Предположим также, что H2 и H3 являются нитями реального времени. Поскольку нить H2 заблокирована, то H3 на данный момент является наиболее высокоприоритетной и готовой к выполнению нитью, и поэтому именно она вытеснит нить H1 (рис. 5.7, б). В результате нить H2 останется блокированной до тех пор, пока нить H3 либо завершит работу, либо заблокируется, и только после этого нить H1 начнет функционировать и освободит ресурс.

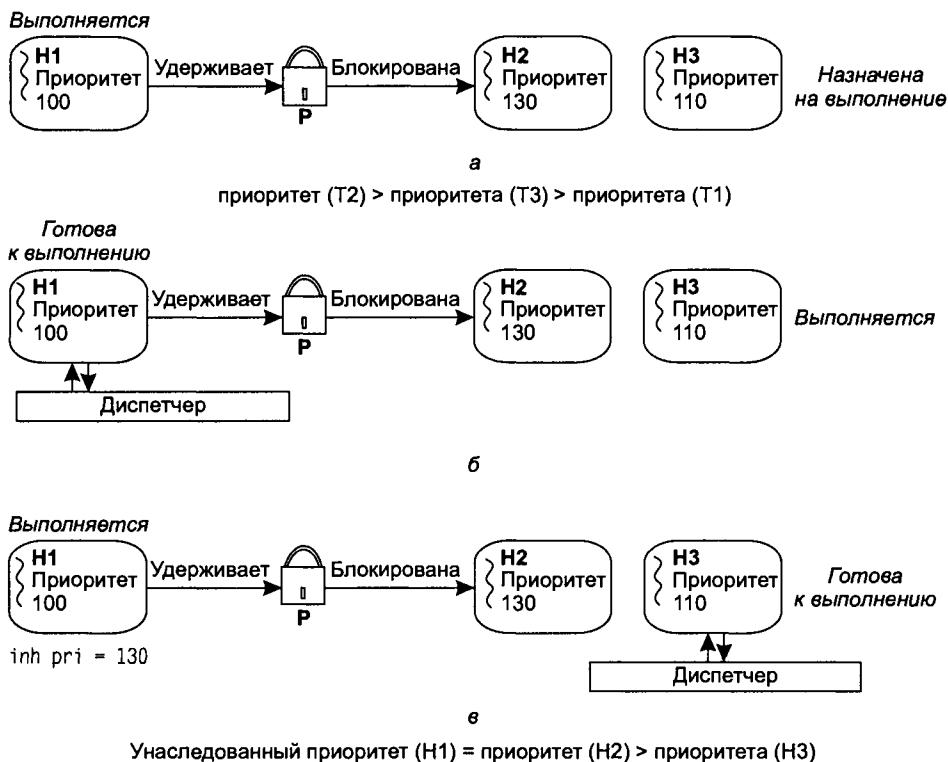
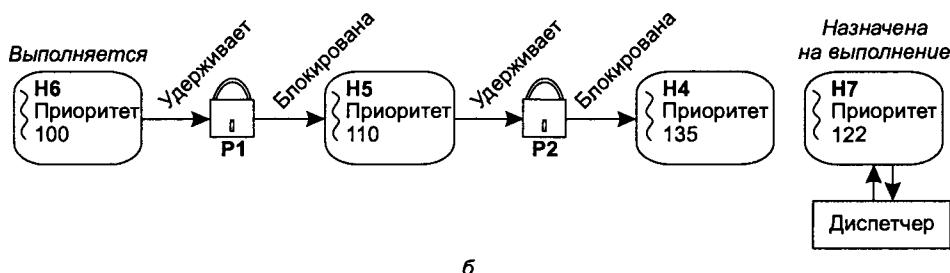
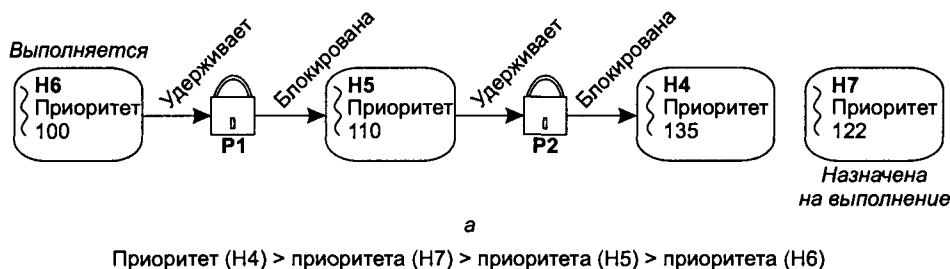


Рис. 5.7. Простейший случай инверсии приоритетов: а — начальная ситуация, б — без наследования приоритетов, в — с наследованием приоритетов

Описываемая проблема может быть решена при использовании метода *наследования приоритетов* или *временной передачи приоритетов*. Когда высокоприоритетная нить блокируется в ожидании ресурса, она временно передает свой приоритет менее приоритетной нити, которая в данный момент удерживает этот ресурс. Таким образом, в приведенном выше примере нить H1 унаследует приоритет нити H2 и теперь не сможет быть вытеснена нитью H3 (рис. 5.7, *в*). Когда нить H1 освободит ресурс, ее приоритет снова вернется в оригинальное значение, что позволит H2 вытеснить ее. Нить H3 будет назначена на выполнение только после того, как H1 освободит ресурс, а H2 закончит работу и отдаст процессор.

Наследование приоритетов должно быть переходным. На рис. 5.8 нить H4 блокируется в ожидании ресурса, удерживаемого нитью H5, которая, в свою очередь, блокирована на ресурсе, удерживаемом нитью H6. Если приоритет нити H4 выше, чем у H5 и H6, то нить H6 должна унаследовать приоритет нити H4 через нить H5. В противном случае нить H7, обладающая приоритетом большим, чем у H5 и H6, но меньшим, чем у H4, вытеснит нить H6 и послужит причиной инверсии приоритетов в отношении H4. Таким образом, унаследованный приоритет нити должен равняться приоритету высокоприоритетной нити, непосредственно или косвенно ожидающей данный ресурс.



(Унаследованный приоритет (H6) = унаследованному приоритету (H5) = приоритету (H4) > (приоритета (H7))

Рис. 5.8. Переходная инверсия приоритетов: *а* — начальная ситуация, *б* — при переходящем наследовании

Для реализации наследования приоритетов ядро системы Solaris должно содержать дополнительную информацию о занятых объектах. Необходимо

идентифицировать, какая нить в данный момент назначена владельцем каждого занятого объекта, а также те объекты, в ожидании которых находится каждая блокированная нить. Так как наследование является переходным, ядро должно иметь возможность просматривать все объекты и блокированные нити в *цепочке синхронизации*, начинающейся от каждого данного объекта. О том, как реализовано наследование приоритетов в системе Solaris, будет рассказано в следующем подразделе.

5.6.5. Реализация наследования приоритетов

Каждая нить обладает двумя приоритетами: *глобальным приоритетом*, определяемым классом планирования, и *унаследованным приоритетом*, зависящим от взаимодействия нити с объектами синхронизации. Унаследованный приоритет обычно равен нулю до тех пор, пока нити не будет передан чей-либо приоритет. Приоритет планирования нити выше, чем ее глобальный и унаследованный приоритеты.

Когда нить должна блокироваться в ожидании ресурса, она вызывает функцию `ri_willto()` для передачи своего приоритета всем нитям, которые прямо или косвенно блокируют ресурс. Так как наследование является переходным, функция `ri_willto()` передает унаследованный приоритет вызывающей нити¹. Функция `ri_willto()` просматривает цепочку синхронизации этой нити, начиная с объекта, на котором нить заблокирована напрямую. Этот объект содержит указатель на свою *нить-владельца*, удерживающую в текущий момент защелку. Если приоритет планирования нити-владельца ниже, чем наследуемый приоритет вызывающей нити, то нить-владелец ресурса унаследует более высокое значение приоритета. Если нить-владелец объекта заблокируется в ожидании другого ресурса, ее структура нити будет содержать указатель на соответствующий объект синхронизации. Функция `ri_willto()`, следуя по этому указателю, передаст приоритет нити-владельцу объекта и т. д. Цепочка синхронизации закончится тогда, когда достигнет незаблокированной нити или объекта, приоритет которого не был инвертирован².

Разберем пример, приведенный на рис. 5.9. Нить Н6 является текущей и обладает глобальным приоритетом, равным 110. Она желает заполучить ресурс Р4, удерживаемый нитью Н5. Ядро вызывает функцию `ri_willto()`, которая просматривает цепочку синхронизации, начинающуюся от Р4, производя при этом описанные ниже действия.

¹ В том случае, видимо, когда унаследованный приоритет выше глобального, то есть функция вызывается нитью, находящейся не в начале цепочки. — Прим. ред.

² Приведенный алгоритм известен под названием *вычисления транзитивного замыкания* (computation of transitive closure), а просматриваемая цепочка представляет собой *прямой ациклический граф*. (Матрица транзитивного замыкания составляется при помощи последовательного удаления узлов, встречающихся дважды.) — Прим. ред.

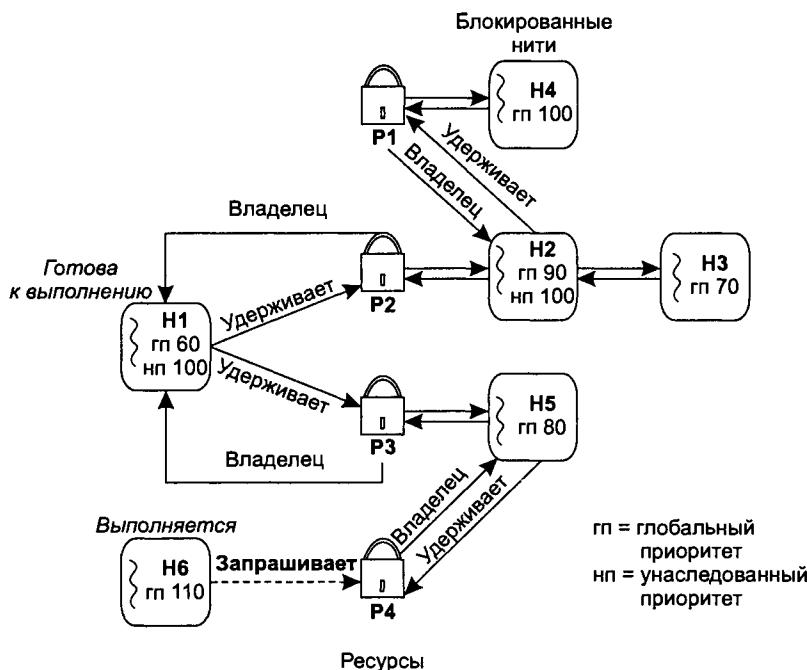


Рис. 5.9. Просмотр цепочки синхронизации

- Нить H5, имеющая глобальный приоритет 80, но без унаследованного приоритета, является владельцем ресурса P4. Так как значение ее приоритета ниже 110, то ей устанавливается унаследованный приоритет, равный 110.
- Нить H5 блокирована в ожидании ресурса P3, владельцем которого является H1. Нить H1 имеет глобальный приоритет 60, но ее унаследованный приоритет равен 100 (через ресурс P2). Так как это число меньше 110, функция увеличит наследуемый приоритет нити H1 до 110.
- Поскольку нить H1 не блокирована на каком-либо ресурсе, просмотр цепочки завершается, функция возвращает управление.

После возврата из `pi_willto()` ядро блокирует H6 и выбирает другую нить для выполнения. Так как приоритет H1 был только что поднят до значения 110, скорее всего именно она будет немедленно назначена на выполнение. На рис. 5.10 показана ситуация после переключения контекста.

Когда нить освобождает объект, она сбрасывает унаследованный приоритет при помощи вызова `pi_waive()`. В некоторых случаях (например, в предыдущем примере) нить может удерживать несколько объектов. Тогда ее унаследованный приоритет будет равняться максимальному значению из всех приоритетов, наследуемых от этих объектов. Когда нить освобождает какой-либо объ-

ект, ее приоритет пересчитывается на основе оставшихся удерживаемых ею объектов. Такие сокращения наследованного приоритета могут привести к тому, что приоритет этой нити станет меньше, чем приоритет другой, готовой к выполнению нити, которая в конечном счете вытеснит первую.

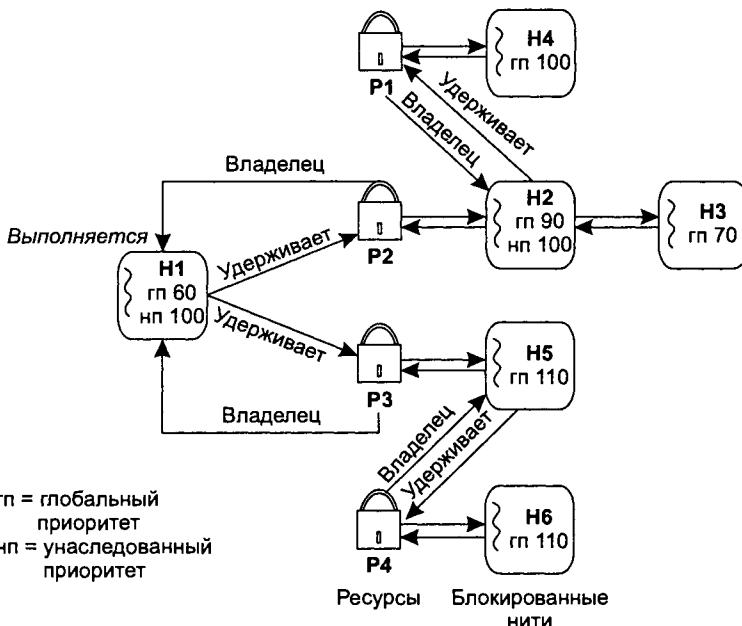


Рис. 5.10. Наследование приоритетов

5.6.6. Ограничения наследования приоритетов

Наследование приоритетов следует использовать только в случаях, когда мы знаем, какая из нитей собирается освободить ресурс. Это возможно, когда ресурс удерживается единственной известной нам нитью. В системе Solaris 2.x поддерживается четыре типа объектов синхронизации: взаимные исключения (mutex), семафоры, условные переменные и защелки чтения/записи (более подробно эти объекты будут рассмотрены в главе 7). При применении взаимных исключений владелец ресурса всегда известен¹. Однако при использовании семафоров или условных переменных владелец обычно не определяется, следовательно, наследование приоритетов не используется. Это является неприятным моментом, так как условные переменные часто применяются в сочетании со взаимными исключениями, реализуя тем самым высоконивневые конструкции синхронизации, а некоторые из последних имеют определяемых владельцем.

¹ Владельца можно определить через функцию `mutex_owned`. — Прим. ред.

Если защелка чтения/записи закрывается для записи, то владелец один, и он известен¹. Однако ресурс может удерживаться сразу несколькими читающими нитями. Пищащая нить должна блокироваться до тех пор, пока все текущие читающие нити не освободят объект. В этом случае объект не имеет единственного владельца, а хранить указатели на всех владельцев непрактично. В системе Solaris для решения подобной проблемы предусмотрено определение *обладателя записи* (owner-of-record), которым является первая читающая нить, получившая защелку и переключившая тем самым ее в режим разрешения доступа «только на чтение». Если высокоприоритетная пищащая нить блокируется в ожидании этого объекта, нить владельца записи унаследует ее приоритет. Когда нить владельца записи освободит объект, возможна ситуация, что какие-либо другие неопределенные читающие нити ещедерживают защелку в режиме чтения. Эти нити не могут унаследовать приоритет пищащей нити. Таким образом, предлагаемое решение является ограниченным, но остается удобным, так как во многих ситуациях удержание защелки производится всего лишь одной читающей нитью.

Наследование приоритетов уменьшает время блокирования высокоприоритетного процесса, находящегося в ожидании ресурсов, удерживаемых низкоприоритетными процессами. Однако в самом худшем случае величина задержки по-прежнему остается достаточно большой, не удовлетворяя требований многих приложений реального времени. Одной из причин этого является тот факт, что цепочка блокирования способна достигать весьма больших размеров. Другой причиной может послужить наличие в высокоприоритетном процессе нескольких критических участков кода, на каждом из которых может происходить блокирование, что в сумме может привести к значительным задержкам. Этой проблеме уделяется большое внимание среди разработчиков. Появились ее альтернативные решения, например *ceiling-протокол* (*ceiling protocol*, см. [16]). Протокол контролирует захват ресурсов процессами для гарантии того, что высокоприоритетный процесс блокируется в ожидании ресурса, удерживаемого низкоприоритетным процессом, не более чем один раз при каждой активации. Хотя такой подход ограничивает задержки блокировки для высокоприоритетных процессов, он заставляет низкоприоритетные процессы чаще блокироваться. Он также требует априорного знания обо всех процессах системы и их требованиях относительно ресурсов. Описанные недостатки сужают область применимости протокола, оправдывая его только для небольшого круга приложений.

5.6.7. Турникеты

Ядро содержит сотни объектов синхронизации, по одному для каждой структуры данных, которую необходимо защищать отдельно. Такие объекты должны хранить огромные объемы информации, например очереди нитей, на них

¹ Тот, кто пишет. – *Прим. ред.*

блокированных. Содержание большой структуры данных для каждого объекта является расточительным, поскольку хотя в ядре находятся сотни таких объектов, но только несколько из них используются в один конкретный момент. В системе Solaris применяется эффективное решение этой проблемы при помощи *турникетов*. Объект синхронизации содержит указатель на турникет, в котором находятся все данные, необходимые для манипулирования объектом, например очереди блокированных нитей и указатель на нить, владеющую ресурсом в текущий момент (рис. 5.11). Турникеты выделяются динамически из пула, который растет в размере по мере увеличения количества нитей в системе. Турникет предоставляетяя первой нитью, которой необходимо заблокировать объект. Когда более не остается блокированных на объекте нитей, турникет освобождается и возвращается обратно в пул.

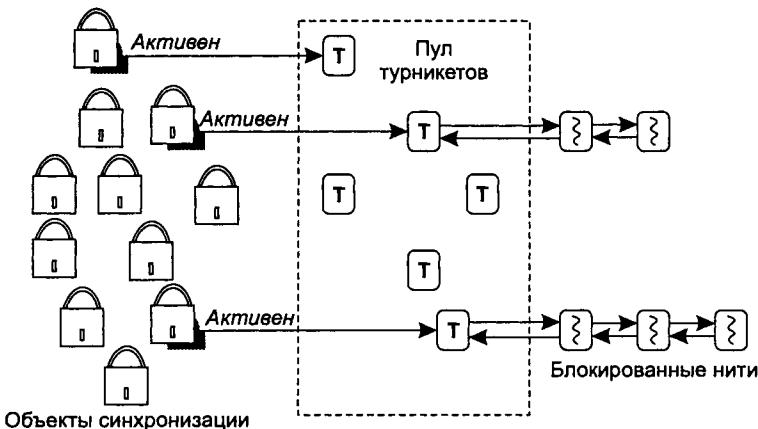


Рис. 5.11. Турникеты

В традиционных системах UNIX ядро связывает с каждым ресурсом или событием, на котором процесс может заблокироваться, особый *канал сна* (см. раздел 7.2.3). Канал обычно представляет собой адрес, связанный с ресурсом или событием. Ядро хэширует процесс в очередь сна, основанную на этом канале ожидания. Поскольку разные каналы ожидания могут отображаться в одну и ту же очередь сна, время, затрачиваемое на ее просмотр, ограничивается только общим количеством нитей системы. В ОС Solaris 2.x этот механизм заменен турникетами, которые ограничивают очередь сна количеством нитей, блокированных на конкретном ресурсе, что дает более разумные границы времени, затрачиваемого на обработку очереди.

Нити внутри турникета выстроены в порядке их приоритетов. Объекты синхронизации поддерживают два способа разблокировки: *сигнал*¹, который

¹ Функциональность этих сигналов не имеет никакого отношения к традиционным сигналам системы UNIX. Так как в UNIX принято использовать терминологию из нескольких источников, некоторые термины системы обладают несколькими значениями.

пробуждает определенную спящую нить, или *широковещательное сообщение* (broadcast), которое будит все нити, блокированные в ожидании ресурса. В системе Solaris сигнал будит наиболее высокоприоритетную нить очереди.

5.6.8. Анализ

В системе Solaris 2.x представлена сложная среда поддержки многонитевой обработки и обработки в режиме реального времени — как для однопроцессорных, так и для многопроцессорных систем. В Solaris были устраниены некоторые недостатки, присущие механизму планирования в SVR4. Измерения, проведенные на Sparcstation 1, показали, что задержки обслуживания планировщиком в большинстве случаев не превышали 2 миллисекунд. Такое значение является следствием полностью вытесняющего ядра и наследования приоритетов.

Несмотря на то что в Solaris имеются средства, подходящие для большинства приложений реального времени, она является главным образом операционной системой общего назначения. Система, создаваемая специально для приложений реального времени, должна поддерживать такие дополнительные возможности, как групповое планирование процессоров, планирование по крайним срокам или планирование, основанное на приоритетах операций устройств ввода-вывода. Эти возможности будут описаны в разделе 5.9.

Рассмотрим несколько алгоритмов планирования, применяемых в коммерческих и экспериментальных вариантах системы UNIX.

5.7. Планирование в системе Mach

Mach является многонитевой операционной системой, поддерживающей многопроцессорные системы. Она разработана для применения в любых типах компьютеров — от однопроцессорных машин до массивно-параллельных систем, содержащих сотни процессоров и разделяющих между собой единое адресное пространство. Из этого следует, что планировщик ОС Mach должен подходить для всех возможных целей применения [2].

Основные понятия системы — задачи и нити — были описаны в разделе 3.7. Нить является главной единицей планирования и система планирует выполнение нитей относительно задачи, к которой они относятся. Такой подход снижает производительность, поскольку переключение контекста между нитями одной задачи происходит намного быстрее по отношению к переключению нитей, относящихся к различным задачам (из-за того, что в первом случае не нужно производить изменение карт управления памятью). Однако политика, которая поощряет переключение нитей внутри процесса, может противоречить целям баланса загрузки и использования системы. Более того, различия производительности двух типов контекстного переключения могут оказаться малозначительными в зависимости от применяемого аппаратного обеспечения и запущенных приложений.

Каждая нить наследует базовый приоритет планирования от задачи, к которой она относится. Этот приоритет сочетается с фактором использования процессора, который хранится и ведется отдельно для каждой нити. Система Mach уменьшает степень использования процессора для каждой нити посредством умножения ее на $5/8$ каждой секунды простояния нити. Алгоритм снижения является распределенным. Каждая нить отслеживает уровень собственного использования процессора и пересчитывает его при пробуждении после блокирования. Обработчик прерываний таймера регулирует фактор использования текущей нити. Для предупреждения зависания (из-за недостатка процессорного времени) низкоприоритетных нитей, продолжающих оставаться в очереди на выполнение без возможности пересчета своих приоритетов, каждые 2 секунды запускается внутренняя нить ядра, которая пересчитывает приоритеты всех готовых к выполнению нитей.

Назначенная на выполнение нить выполняется в течение определенного кванта времени. По исчерпании этого промежутка она может быть вытеснена другой нитью, обладающей равным или большим приоритетом. Перед тем как израсходуется начальный квант у текущей нити, ее приоритет может быть понижен по отношению к другим готовым к выполнению нитям. Однако в системе Mach такие изменения не приведут к переключению контекста раньше, чем нить исчерпает свой квант. Благодаря этой особенности сокращается общее число переключений контекста, что прямо оказывается на равновесии в использовании системы. Текущая нить может быть вытеснена, если более высокоприоритетная нить станет готовой к выполнению, даже если квант текущей нити не был до конца выработан.

В системе Mach поддерживается *автоматическое планирование* (handoff scheduling), посредством которого нить может напрямую передавать процессор другой нити без поиска очередей выполнения. Подсистема межпроцессного взаимодействия (IPC) использует эту технологию для передачи сообщений: если нить уже находится в режиме ожидания сообщения, посылающая нить передает ей процессор непосредственно. Такой подход увеличивает производительность работы вызовов IPC.

5.7.1. Поддержка нескольких процессоров

Как уже говорилось, система Mach функционирует на самых различных аппаратных архитектурах, от небольших персональных компьютеров до машин, имеющих сотни процессоров. Планировщик системы обладает несколькими средствами, позволяющими эффективно управлять процессорами системы.

Для вытеснения в Mach не применяются межпроцессорные прерывания. Предположим, что некоторое событие на одном из процессоров привело к появлению готовой к выполнению нити, имеющей приоритет больший, чем у другой нити, выполняющейся на другом процессоре. Нить с меньшим приоритетом не будет вытеснена до тех пор, пока другой процессор обрабатывает

ет прерывание таймера или иное событие, относящееся к планированию. Отсутствие межпроцессорного вытеснения не оказывает отрицательного влияния на режим разделения времени, но оно, тем не менее, может сказываться на эффективности реакции системы по отношению к приложениям реального времени.

Система Mach позволяет пользователям управлять выделением процессоров различным задачам путем создания *наборов процессоров*, каждый из которых может содержать 0 и более процессоров. Каждый процессор относится к одному из таких наборов, и он также может быть перемещен из одного набора в другой. Каждой задаче или нити назначается определенный набор процессоров, который может быть изменен в любой момент времени. Однако такой возможностью обладают только привилегированные задачи, которым разрешается назначать процессоры, задачи и нити наборам процессоров.

Нить может выполняться на одном из процессоров из выделенного ей набора. Назначение задачи набору процессоров делает его назначаемым по умолчанию всем новым нитям этой задачи. Задача наследует набор от своего предка, а *первоначальной задаче* (initial task) предоставляется *набор процессоров, принятый по умолчанию*. Исходно такой набор состоит из всех процессоров системы, и на этом наборе выполняются внутренние нити ядра и демоны.

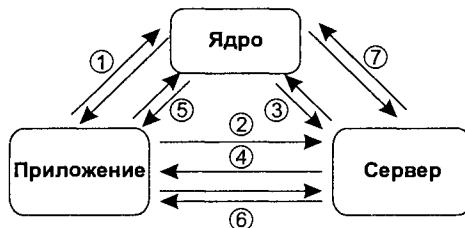


Рис. 5.12. Выделение процессоров в Mach

Выделение процессоров может быть выполнено при помощи программы-сервера прикладного уровня (выполняющейся как привилегированное задание), которая и определяет политику присвоения. На рис. 5.12 показана типичная схема взаимодействий между приложением, сервером и ядром. Приложение выделяет набор процессоров всем своим нитям. Сервер назначает процессоры для этого набора. Последовательность действий при этом описана ниже.

1. Приложение запрашивает ядро о выделении для себя набора процессоров.
2. Приложение запрашивает сервер о процессорах для этого набора.
3. Сервер запрашивает ядро о назначении процессоров набору.
4. Сервер отвечает приложению, подтверждая, что процессоры были выделены.

5. Приложение запрашивает ядро о назначении своих нитей этому набору.
6. Приложение использует процессоры и уведомляет сервер, когда завершает свою работу.
7. Сервер переназначает процессоры.

Такой подход позволяет добиться потрясающей гибкости в управлении процессорами, особенно для массивно-параллельных систем с большим количеством процессоров. Например, существует возможность назначить их некоторое количество одной задаче или группе задач, гарантуя тем самым предоставление им части доступных ресурсов независимо от общей загруженности системы. В крайних случаях приложение может добиваться выделения по одному процессору для каждой своей нити. Такой подход называется *групповым планированием* (*gang scheduling*).

Применение группового планирования удобно для приложений, требующих *барьерной синхронизации*. Такие приложения создают несколько нитей, которые функционируют независимо друг от друга до достижения некой точки синхронизации, называемой барьером. Каждая нить, достигнувшая барьера, должна ожидать, пока к нему приблизятся остальные. После того как все нити сведут свою работу к этой точке, приложение может запустить некоторый однонитевой код и затем создать другой набор нитей, повторяющих действия предыдущего набора.

Для того чтобы подобные приложения функционировали оптимально, задержки при достижении барьера должны быть минимальными. Это требует, чтобы все нити достигали барьера примерно в одно и то же время. Групповое планирование позволяет приложению запустить нити одновременно, представив каждой из них отдельный процессор. Описанный подход сводит к минимуму задержки при достижении барьера.

Групповое планирование также подходит для мелкомодульных приложений, нити которых часто взаимодействуют между собой. В таких приложениях вытеснение одной из нитей способно привести к блокированию других нитей, требующих взаимодействия с ней. Недостатком предоставления каждой нити отдельного процессора является то, что при блокировании нити процессор не может быть использован.

Процессоры системы могут быть неодинаковыми. Некоторые из них могут оказаться более быстрыми, чем другие, некоторые могут содержать блок вычислений с плавающей точкой и т. д. Возможность составления процессоров в наборы позволяет использовать определенные процессоры для выполнения свойственной им задачи. Например, процессоры с блоком вычислений с плавающей точкой логично назначать только нитям, которым необходима максимальная производительность при работе с вещественными числами.

Кроме всего перечисленного, нить может быть временно ограничена определенным процессором. Такая возможность поддерживается, в первую очередь, для совместимости части кода системы Mach с UNIX в области непа-

ралльной (небезопасной в многопроцессорной системе) обработки. Эта часть кода выполняется на одном процессоре, называемом *главным процессором* (*master processor*). Каждый процессор имеет локальную очередь выполнения, а каждый набор — глобальную, используемую совместно всеми процессорами набора. Сначала процессоры проверяют свои локальные очереди, отдавая таким образом предпочтение ограниченным данным процессором нитям (или неограниченным использованием данного процессора нитям, обладающим высокими приоритетами). Такое решение привнесло максимальную производительность для непараллельного кода UNIX, защищая от эффекта «бутылочного горлышка».

5.8. Планировщик реального времени Digital UNIX

Планировщик операционной системы Digital UNIX поддерживает приложения, выполняющиеся как в режиме разделения времени, так и в режиме реального времени [5]. Он совместим с интерфейсом POSIX 1003.1b [8], определяющим программное расширение реального времени. Несмотря на факт, что Digital UNIX произошла от ОС Mach, ее планировщик был полностью переработан. Он поддерживает следующие классы планирования:

- ◆ SHED_OTHER, или класс разделения времени.
- ◆ SHED_FIFO, или класс FIFO («первым вошел, первым вышел»).
- ◆ SHED_RR, или класс карусельного обслуживания.

Для установки класса планирования и приоритета процесса приложение может вызвать `sched_setscheduler`. По умолчанию используется класс разделения времени, в котором приоритеты изменяются динамически на основе величины любезности и уровня использования процессора. Оставшиеся два класса используют постоянные приоритеты. Процессы, действующие по политику класса SHED_FIFO, не имеют определенного кванта времени и выполняются до тех пор, пока сами не освободят процессор или не будут вытеснены более высокоприоритетным процессом. Классам разделения времени и карусельного обслуживания определен квант времени, который влияет на диспетчеризацию процессов, обладающих одинаковым приоритетом. Когда процессы, принадлежащие какому-либо из этих двух классов, исчерпывают выделенный им квант, они помещаются в конец списка процессов того же приоритета. В случае отсутствия готовых к выполнению процессов, обладающих одинаковым или более высоким приоритетом по сравнению с текущим, процесс будет продолжать свою работу.

Планировщик всегда выбирает для выполнения наиболее высокоприоритетный процесс. Каждый процесс имеет приоритет в диапазоне 0–63, где меньшие числа соответствуют меньшим приоритетам. Планировщик для каждого

приоритета содержит упорядоченную очередь и выбирает процесс из начала наивысшей непустой очереди. Когда блокированный процесс становится готовым к выполнению или текущий процесс освобождает процессор, то такие процессы обычно помещаются в конец очереди своего приоритета. Исключительным случаем является вытеснение процесса до того, как он исчерпает выделенный ему квант времени. В этом случае процесс возвращается в начало своей очереди, что позволяет ему завершить использование кванта времени до того, как начнут свое выполнение другие процессы, обладающие тем же приоритетом.

Приоритеты подразделяются на три перекрывающихся класса, которые позволяют увеличить гибкость. Назначение приоритетов процессам регулируется следующими правилами:

- ◆ Процессы класса разделения времени имеют приоритеты в диапазоне от 0 до 29. Для увеличения приоритетов выше 19 требуются привилегии суперпользователя.
- ◆ Пользователи управляют приоритетами процессов класса разделения времени посредством изменения величины их «любезности». Значение величины «любезности» находится в диапазоне от -20 до +20, где меньшие числа указывают на более высокий приоритет (для обратной совместимости). Отрицательные величины, соответствующие приоритетам диапазона 20–29, может задавать только суперпользователь системы.
- ◆ Фактор использования процессора уменьшает приоритеты процессов класса разделения времени в зависимости от полученного ими количества процессорного времени.
- ◆ Системные процессы обладают фиксированными приоритетами в диапазоне 20–31.
- ◆ Процессам с фиксированными приоритетами может быть назначен любой приоритет в диапазоне 0–63. Однако для назначения приоритетов выше 19 требуются привилегии суперпользователя. Процессы реального времени имеют приоритеты в диапазоне от 32 до 63, поскольку системные процессы не должны их вытеснять.

Вызов `sched_setparam` применяется для изменения приоритетов процессов классов FIFO и карусельного обслуживания. Вызов `sched_yield` перемещает вызвавший его процесс в конец очереди его приоритета, что эквивалентно освобождению процессора для любого готового к выполнению процесса, обладающего тем же приоритетом. Если таких процессов не обнаружено, процесс, вызвавший `sched_yield`, продолжит выполнение.

5.8.1. Поддержка нескольких процессоров

Система Digital UNIX позволяет эффективно использовать такое свойство систем, как многопроцессорность, посредством настройки своего планировщика для оптимизации переключений контекста и использования кэша [6].

В идеальном случае планировщик стремится выполнять высокоприоритетные готовые к выполнению нити на всех доступных процессорах. Такая политика требует от планировщика поддержания глобального набора очередей выполнения, разделяемого всеми процессорами. Это может стать причиной эффекта «бутылочного горлышка», когда все процессоры будут стремиться получить монопольный доступ (то есть блокировать доступ другим) к этим очередям. Более того, когда нить начинает свое выполнение, кэши процессора заполняются ее данными и инструкциями. И если на короткий промежуток времени эта нить будет вытеснена и затем снова назначена на выполнение, то по возможности она должна продолжаться на том же процессоре, так как при этом она получит определенный выигрыш из-за наличия в кэшах процессора ранее размещенных данных и инструкций.

Для согласования вышеуказанных моментов в системе Digital UNIX для нитей режима разделения времени используется политика *мягкого сродства* (*soft affinity*). Такие нити хранятся в локальных очередях выполнения процессора, следовательно, они вновь назначаются на выполнение на том же самом процессоре, что и раньше. Это также сокращает борьбу за очереди выполнения. Планировщик следит за очередями для каждого процессора и предотвращает возможный дисбаланс путем перемещения нитей из очереди более загруженного процессора в очередь менее занятого.

Нити с фиксированными приоритетами назначаются на выполнение из глобальной очереди, потому что они должны начать свое выполнение как можно быстрее. По возможности ядро отправляет их на тот же самый процессор, который использовался ими в предыдущий раз. В завершение следует также отметить, что система Digital UNIX обеспечивает вызов `bind_to_scpu`, который принуждает нить к выполнению только на определенном процессоре. Эта возможность полезна для кода, функционирование которого на более чем одном процессоре небезопасно.

Планировщик системы Digital UNIX предоставляет совместимый с POSIX интерфейс планирования реального времени. Однако он не обладает многими возможностями, имеющимися в Mach и SVR4. Он также не обеспечивает интерфейс для выделения наборов процессоров или механизм «ручного» планирования. Ядро Digital UNIX является невытесняющим и не имеет никаких средств для контроля за инверсией приоритетов.

5.9. Другие реализации планирования

Реализация алгоритмов планирования системы зависит, главным образом, от требований приложений, которые будут работать в данной ОС. В некоторых системах работают критичные ко времени программы и приложения реального времени, в других — крупные, работающие в режиме разделения времени приложения, в третьих — и те и другие. В иных системах выполняется

огромное количество процессов, для которых существующие методики планирования не соответствуют требованиям по масштабированию. Такое положение дел побудило к созданию ряда планировщиков, которые частично были реализованы в различных вариантах UNIX. Их описанию и посвящен этот раздел книги.

5.9.1. Планирование справедливого разделения

Планировщик справедливого разделения (fair-share) предоставляет фиксированный объем разделяемых ресурсов процессора каждой *разделяющей группе* процессов. Такие группы могут состоять из единичного процесса, всех процессов одного пользователя, всех процессов сеанса входа в систему и т. д. Выбор распределения процессорного времени между *разделяющими группами* доступен только суперпользователю системы. Ядро отслеживает использование процессора для выбора схемы его распределения. Если одна из групп не использовала выделенную ей часть ресурсов, то их оставшаяся часть обычно делится между остальными группами пропорционально изначально предоставленным им ресурсам.

Такой подход дает каждой разделяющей группе прогнозируемый объем процессорного времени, который не зависит от общей загруженности системы. Это особенно полезно в средах, где время вычисления является строго расписанным ресурсом, поскольку ресурсы могут выделяться пользователям фиксированными объемами. Это также пригодно для гарантии предоставления необходимых ресурсов критичным к ним приложениям в системах разделения времени. Одна из реализаций планировщика равного разделения описана в [7].

5.9.2. Планирование по крайнему сроку

Многие приложения реального времени должны отвечать на события в течение определенного ограниченного периода. Например, мультимедийный сервер может передавать видеокадры клиенту каждые 33 миллисекунд. Если сервер читает данные с диска, то он может установить крайний срок завершения операции чтения. Если в течение этого времени операция не успеет завершиться, кадр будет задержан. Механизм назначения крайних сроков также применим для запросов на ввод-вывод или вычислений. В последнем случае нить может запросить выделения определенного объема процессорного времени, которое должно быть ей предоставлено до окончания установленного крайнего срока.

Производительность описываемых видов приложений повысится при использовании технологии *планирования по крайнему сроку*. Основным принципом этого метода является динамическое изменение приоритетов, увеличение которых происходит при приближении к конечному сроку. Примером реализации подобного алгоритма является планировщик Ferranti-Originated

Real-Time Extensions to UNIX (FORTUNIX), описанный в [3]. Его алгоритм определяет четыре уровня приоритетов:

- ◆ Жесткие приоритеты реального времени для крайних сроков, которые всегда должны быть соблюдены при удовлетворении чего-либо.
- ◆ Мягкие приоритеты реального времени, при которых требования удовлетворить что-либо до наступления крайнего срока должны реализовываться с наибольшей вероятностью, однако при этом случаи неукладывания в граничные сроки возможны.
- ◆ Приоритеты разделения времени без определенных конечных сроков, но с требованием, чтобы были установлены разумные пределы времени реакции.
- ◆ Приоритеты пакетных заданий, для которых крайние сроки выражаются в часах (а не в миллисекундах).

Система осуществляет планирование процессов в соответствии с их уровнями приоритетов. Например, процесс реального времени с мягким приоритетом будет выполнен только в случае отсутствия готовых к выполнению процессов с жесткими приоритетами. Процессы классов 1, 2 и 4 выполняются в соответствии с их крайними сроками: процесс, обладающий наименьшим оставшимся временем, будет назначен на выполнение первым. Процессы перечисленных выше классов будут выполняться до тех пор, пока они не завершатся или не будут блокированы, если только не появится готовый к выполнению процесс из более старшего класса или того же класса, но с меньшим временем, оставшимся до достижения крайнего срока. Процессы режима разделения времени планируются на выполнение в традиционной манере UNIX на основе их приоритетов, зависящих от фактора любезности и использования процессора.

Планирование по крайнему сроку подходит для систем, в которых работают процессы с известными заранее требованиями по времени реакции. Подобная схема приоритетов применима для планирования запросов ввода-вывода к диску и т. д.

5.9.3. Трехуровневый планировщик

Планировщики UNIX не способны гарантированно обеспечить соблюдение требований приложений реального времени, но в то же время позволяют осуществлять произвольную смешанную рабочую нагрузку. Основной причиной недостатков является отсутствие *управления доступом* (admission control). Не существует ограничений на количество или типы приложений реального времени и задач общего назначения, запускаемых в системе. Система позволяет всем процессам состязаться за ресурсы, не контролируя этот процесс. Забота о том, чтобы в системе не возникало перегрузок, лежит фактически на ее пользователях.

В работе [14] описывается трехуровневый планировщик, применяющийся в системах реального времени для мультипротокольных файлов и медиа-серверов. Планировщик поддерживает три класса служб: изохронный, реального времени и общего применения. Изохронный класс подразумевает периодические действия, такие как передача видеокадров, через определенные интервалы с минимальным *дрожанием изображения*, или колебаниями. Класс реального времени требуется для апериодических задач, нуждающихся в ограниченной задержке обслуживания планировщиком. Последний класс служб общего назначения используется для низкоприоритетных фоновых задач. Планировщик гарантирует, что выполнение таких задач не будет оказывать влияния на работу изохронных нитей.

Несколько слов о том, как планировщик проводит политику управления доступом. Перед приемом нового видеопотока резервируются все необходимые для этого потока ресурсы, которые включают в себя часть процессорного времени, пропускную способность при операциях с диском и сетевым контроллером. Если серверу не удается зарезервировать ресурсы, то он не разрешит обработку запроса. Каждая служба реального времени за каждую свою активацию может обрабатывать ограниченное число *рабочих блоков*. Например, сетевой драйвер перед тем, как освободить процессор, вправе обработать лишь определенное количество входящих сообщений. Планировщик также по отдельности устанавливает фиксированный объем всех ресурсов для заданий общего характера, по отношению к которым не используется управление доступом. Такой подход защищает задания общего применения от загружения на сильно загруженных системах.

Для реализации описанной политики система планирует не только процессорное время, но и дисковые и сетевые операции. Запросам на ввод-вывод назначаются приоритеты на основе инициирующей их задачи. Обрабатываются в первую очередь высокоприоритетные запросы. Резервирование пропускной способности всего тракта из всех имеющихся ресурсов дает возможность гарантировать, что система будет удовлетворять требованиям всех допущенных видеопотоков. Такой подход также гарантирует продолжение выполнения низкоприоритетных задач даже в случае максимальной загруженности системы.

В трехуровневом алгоритме планирования общечелевые задания являются полностью вытесняемыми. Задания реального времени могут вытесняться изохронными заданиями только в четко определенных точках вытеснения, которые обычно находятся в конце выполнения каждого блока работы. Изохронные задачи используют алгоритм планирования *постоянного отношения* (*rate-monotonic*, описан в [11]). Каждая такая задача обладает определенным фиксированным приоритетом планирования, зависящим от ее периода. Чем меньше период, тем выше будет приоритет. Высокоприоритетная изохронная задача может вытеснить задачу с более низким приоритетом только в точке вытеснения. В работе [15] показано, что алгоритм постоянного отношения оптimalен для планирования периодических задач с фиксированными приоритетами.

Еще одной проблемой серверов, работающих на традиционных системах UNIX, является их неспособность справляться с перенасыщением, вызванным большой нагрузкой. Системы UNIX производят основную часть обработки входящих сетевых запросов на уровне прерываний. Если входящий трафик слишком высок, система тратит больше времени на обработку прерываний, оставляя слишком малую его долю на обслуживание запросов. Если входная загрузка превышает определенный критический уровень, резко снижается пропускная способность сервера. Эта проблема известна под названием *петли приема* (*receive livelock*). Применение трехуровневого планировщика решает проблему путем перемещения всей обработки сетевых запросов на уровень задач реального времени, для которых ограничен объем обрабатываемого за один прием трафика. Если входящий трафик превысит критическую отметку, сервер отбросит избыточные запросы, что даст возможность продолжения обработки запросов, которые были приняты. Следовательно, после снижения трафика пропускная способность вновь приблизится к постоянной величине, вместо того чтобы снизиться.

5.10. Заключение

В этой главе были рассмотрены несколько различных архитектур планирования и показано, как они влияют на действия системы по отношению к различным типам приложений. Поскольку вычислительные системы применяются в самых различных областях деятельности, каждая из которых обладает определенным набором требований, то ни один планировщик не может идеально устраивать все системы. Планировщик ОС Solaris 2.x подходит для многих типов приложений и предоставляет механизм, позволяющий динамически добавлять новые классы планирования, удовлетворяющие требованиям каких-либо особых приложений. Он лишен некоторых возможностей, таких как поддержка ввода-вывода нитей реального времени или управляемое пользователем планирование работы с диском, однако он является более совершенным, чем традиционный планировщик UNIX. Другие методы планирования, которые были рассмотрены в этой главе, применимы для особых областей приложений, например для параллельной обработки или мультимедиа.

5.11. Упражнения

1. Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерываний таймера?
2. В каких ситуациях для обработки отложенных вызовов использование временных колес будет более эффективным, чем алгоритм, применяемый в системе 4.3BSD?

3. В чем преимущества и недостатки использования в отложенных вызовах относительных интервалов времени по сравнению с абсолютными их значениями?
4. Почему в системе UNIX более предпочтительны процессы ввода-вывода, нежели вычислительные процессы?
5. В чем преимущества объектно-ориентированного интерфейса планировщика системы SVR4? В чем недостатки этой методики?
6. Почему значения `slpreat` и `lwait` всегда больше, чем значения `tgexp` в каждой строке таблицы параметров диспетчера (см. табл. 5.1)?
7. Почему процессам реального времени даются приоритеты выше, чем процессам, работающим в режиме ядра? В чем недостатки такого распределения приоритетов?
8. Почему планирование по событиям наиболее удобно для приложений ввода-вывода и интерактивных задач?
9. В разделе 5.5.6 был упомянут эксперимент, описанный в [13]. Какой будет эффект в том случае, если X-server'у, программе для просмотра видео и интерактивному заданию будут назначены приоритеты класса реального времени, а пакетное задание получит приоритет класса разделения времени?
10. Предположим, что процесс освобождает ресурс, который ожидают сразу несколько других процессов. Что рациональнее: разбудить все ожидающие процессы или же только один из них? Если будить один процесс, то какой из них выбрать?
11. Групповое планирование подразумевает, что каждая нить выполняется на отдельном процессоре. Какие действия предпримет приложение, требующее барьерной синхронизации, в том случае, если доступных процессоров окажется меньше, чем готовых к выполнению нитей? Могут ли в такой ситуации нити, достигшие барьера, находиться в режиме активного ожидания прихода остальных?
12. Какие существуют методы поддержки приложений реального времени в системе Solaris 2.x? В каких случаях их применение неадекватно?
13. Почему планирование по крайнему сроку не подходит для традиционной операционной системы UNIX?
14. Перечислите характеристики процессов реального времени. Приведите несколько примеров периодических и непериодических приложений реального времени.
15. Для уменьшения времени реакции и задержек обслуживания планировщиком можно просто использовать более мощный процессор. Какие различия существуют между системами реального времени и высокопроизводительными системами? Может ли быть такое, что система

с меньшей общей производительностью окажется наиболее подходящей для приложений реального времени?

16. Какие существуют различия между жесткими и мягкими требованиями для классов реального времени?
17. Чем важна для систем реального времени поддержка управления доступом?

5.12. Дополнительная литература

1. American Telephone and Telegraph, «UNIX System V Release 4 Internals Students Guide», 1990.
2. Black, D. L., «Scheduling Support for Concurrency and Parallelism in the Mach Operating System», IEEE Computer, May 1990, pp. 35–43.
3. Bond, P. O., «Priority and Deadline Scheduling on Real-Time UNIX», Proceedings of the Autumn 1988 European UNIX Users' Group Conference, Oct. 1988, pp. 201–207.
4. Digital Equipment Corporation, «VAX Architecture Handbook», Digital Press, 1986.
5. Digital Equipment Corporation, «DEC OSF/I Guide to Realtime Programming», Part No. AA-PS33C-TE, Aug. 1994.
6. Denham, J. M., Long, P., and Woodward, J. A., «DEC OSF/I Version 3.0 Symmetric Multiprocessing Implementation», Digital Technical Journal, Vol. 6, No. 3, Summer 1994, pp. 29–54.
7. Henry, G. J., «The Fair Share Scheduler», AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Oct. 1984, pp. 1845–1857.
8. Institute for Electrical and Electronic Engineers, «POSIX P1003.4b, Real-Time Extensions for Portable Operating Systems», 1993.
9. Khanna, S., Sebree, M., and Zolnowsky, J., «Realtime Scheduling in Sun-OS 5.0», Proceedings of the Winter 1992 USENIX Technical Conference, Jan. 1992.
10. Lampson, B. W. and Redell, D. D., «Experiences with Processes and Monitors in Mesa», Communications of the ACM, Vol. 23, No. 2, Feb 1980, pp. 105–117.
11. Liu, C. L., and Layland, J. W., «Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment», Journal of the ACM, Vol. 20, No. 1, Jan. 1973, pp. 46–61.
12. Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., «The Design and Implementation of the 4.3 BSD UNIX Operating System», Addison-Wesley, Reading, MA, 1989.

13. Nieh, J., «SVR4 UNIX Scheduler Unacceptable for Multimedia Applications», Proceedings of the Fourth International Workshop on Network and Operating Support for Digital Audio and Video, 1993.
14. Ramakrishnan, K. K., Vaitzblit, L., Gray, C. G., Vahalia, U., Ting, D., Tzelnic, P., Glaser, S., and Duso, W. W., «Operating System Support for a Video-on-Demand File Server», *Multimedia Systems*, Vol. 3, No. 2, May 1995, pp. 53–65.
15. Sha, L., and Lehoczky, J. P., «Performance of Real-Time Bus Scheduling Algorithms», *ACM Performance Evaluation Review*, Special Issue, Vol. 14., No. 1, May 1986.
16. Sha, L., Rajkumar, R., and Lehoczky, J. P., «Priority Inheritance Protocols: An Approach to Real-Time Synchronization», *IEEE Transactions on Computers*, Vol. 39, No. 9, Sep. 1990, pp. 1175–1185.
17. Straathof, J. H., Thareja, A. K., and Agrawala, A. K., «UNIX Scheduling for Large Systems», Proceedings of the Winter 1986 USENIX Technical Conference, Jan. 1986.
18. Varghese, G., and Lauck, T., «Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility», Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 25–38.

Глава 6

Межпроцессное взаимодействие

6.1. Введение

В сложных программных средах часто применяется несколько взаимодействующих процессов, выполняющих взаимосвязанные действия. Такие процессы должны иметь возможность общаться и разделять между собой общие ресурсы и данные. Для того чтобы сделать это возможным, ядро системы должно поддерживать определенные механизмы. Такие механизмы получили название *межпроцессного взаимодействия* (interprocess communications, или IPC). В этой главе будут описаны средства IPC, представленные в основных вариантах системы UNIX.

Взаимодействие процессов производится для нескольких определенных целей:

- ◆ **Передача данных.** Одному процессу иногда необходимо передать данные другому процессу. Количество передаваемых данных может варьироваться от одного байта до нескольких мегабайтов.
- ◆ **Совместное использование информации.** Нескольким процессам необходимо обрабатывать разделяемые данные таким образом, чтобы их изменение одним из них сразу же становилось видимым остальным процессам, участвующим в их совместном использовании.
- ◆ **Уведомление о событиях.** Одному процессу иногда требуется уведомлять другой процесс (или набор процессов) о возникновении какого-либо события. Например, при завершении работы процесса об этом необходимо проинформировать всех его потомков. Получатель может предупреждаться асинхронно, в таком случае нужно прерывать нормальную работу. Альтернативным вариантом является ожидание получателем уведомления.
- ◆ **Совместное использование ресурсов.** Хотя ядро системы обычно предоставляет определенную семантику для выделения ресурсов, она может оказаться неподходящей для некоторых приложений. Набор взаимодействующих процессов иногда нуждается в определении собственного

протокола доступа к определенным ресурсам. Подобные правила обычно реализуются на основе схемы блокировки и синхронизации, которая строится поверх основного набора средств, предоставляемых ядром.

- ◆ **Управление процессами.** Некоторым процессам (например, отладчикам) необходимо полное управление выполнением других процессов. Контролирующий процесс может перехватывать все исключительные состояния и аппаратные прерывания целевого процесса и уведомляться обо всех изменениях его состояния.

В системах UNIX предлагаются различные механизмы IPC. Эта глава начинается с описания общего набора средств, которые имеются во всех реализациях UNIX. Это — сигналы, программные каналы и трассировка процессов. Затем мы расскажем о средствах, имеющихся в System V IPC. В конце главы будет рассмотрено взаимодействие между процессами на основе сообщений, поддерживаемое в системе Mach, в которой предлагается богатый набор средств, объединенных единой унифицированной структурой.

6.2. Универсальные средства IPC

Первая внешняя реализация системы UNIX поддерживала три различных средства, которые можно использовать для взаимодействия процессов: сигналы, каналы и трассировку процессов [14]¹. Все перечисленные механизмы являются общими для различных вариантов ОС UNIX. Сигналы и каналы уже были описаны ранее, в этой главе мы остановимся только на том, как их можно использовать для межпроцессного взаимодействия.

6.2.1. Сигналы

Обычно сигналы применяются для уведомления процессов о возникновении асинхронных событий. Изначально они были изобретены для обработки ошибок, но могут быть использованы и в качестве примитивных механизмов IPC. Современные системы UNIX распознают 31 и более различных сигналов. Большинство из них имеют определенные назначения, но по крайней мере два — SIGUSR1 и SIGUSR2 — вправе использоваться приложениями по их собственному усмотрению. Процесс может послать сигнал другому процессу (или процессам) при помощи системного вызова `kill` или `killpg`. Кроме этого, сигнал может быть сгенерирован ядром в ответ на различные события, происходящие в системе. Например, при нажатии комбинации клавиш `Ctrl+C` на терминале ядро посыпает сигнал `SIGINT` приоритетному процессу.

¹ Первые системы UNIX от Bell Telephone Laboratories не поддерживали таких средств. Например, программные каналы были разработаны Д. Мак-Илроем и К. Томпсоном и впервые появились в Version 3 UNIX (см. [12]).

Каждый сигнал приводит к определенным действиям, по умолчанию это завершение процесса. Процесс может указать альтернативное действие, происходящее при получении любого сигнала, предоставив системе функцию обработки сигнала. При вырабатывании сигнала ядро прерывает выполнение процесса, который должен ответить на него запуском обработчика. После завершения обработки сигнала процесс может продолжить работу в обычном режиме.

При помощи сигналов процессы уведомляются о произошедших асинхронных событиях и реагируют на них. Однако сигналы могут также использоваться для синхронизации. Процесс может вызвать `sigpause` для ожидания прибытия сигнала. В первых реализациях системы UNIX совместное использование ресурсов и протоколы блокировки многих приложений базировались на сигналах.

Изначально сигналы разрабатывались для обработки ошибок, например ядро транслировало аппаратные ошибки, такие как *деление на ноль* или *неверные инструкции* в сигналы. Если процесс не имел собственного обработчика для таких ошибок, ядро завершало его работу.

При применении сигналов как механизма взаимодействия процессов существуют несколько ограничений, связанных с тем, что обработка сигналов является затратным действием. Сначала отправитель вызывает системную функцию, после чего ядро прерывает работу получателя и производит интенсивные действия над его стеком, так как ему нужно загрузить обработчик и позже продолжить выполнение прерванного процесса. Более того, сигналы обладают малой пропускной способностью. Во-первых, существует всего 31 сигнал (в системах SVR4 и 4.3BSD, некоторые реализации типа AIX поддерживают большее число сигналов). Во-вторых, сигналы могут нести только ограниченный объем информации. Не существует способа отправки дополнительных данных или входных аргументов при посылке сигналов, создаваемых пользователем¹. Сигналы применимы для уведомления о событиях, но остаются недостаточными для более сложных взаимодействий.

Подробнее сигналы обсуждались в главе 4.

6.2.2. Каналы

В традиционных реализациях UNIX программные каналы (`pipe`)² представляют собой односторонний неструктурированный поток данных фиксированного максимального размера, работающий по принципу FIFO («первым вошел,

¹ Сигналы, генерируемые ядром в ответ на аппаратные сбои, возвращают дополнительную информацию через структуру `siginfo`, передаваемую обработчику.

² На неименованных программных каналах, иначе трубах, реализуется технология конвейеризации, почему их еще называют конвейерами. — Прим. ред.

первым вышел»)¹. Отправители добавляют данные в конец канала, получатели извлекают их из его начала. После того как данные будут прочитаны, они сразу же удаляются из канала и больше недоступны другим получателям. Программные каналы представляют собой простейший механизм управления нитями. Процесс, осуществляющий попытку чтения из пустого канала, будет приостановлен до тех пор, пока в канале не появятся какие-либо данные. Точно так же, если процесс попытается записать в заполненный программный канал, то он будет приостановлен до того момента, пока иной процесс не прочтет данные из канала, тем самым освободив его.

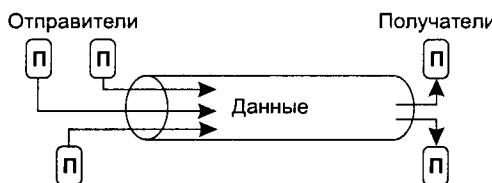


Рис. 6.1. Движение потока данных по программному каналу

Для создания программного канала применяется системный вызов `pipe`, который возвращает два дескриптора файла: один для чтения и один для записи. Эти дескрипторы наследуются процессами-потомками, таким образом разделяя между собой доступ к файлу. Следовательно, в каждый программный канал могут записывать данные и читать из него несколько процессов (рис. 6.1). Каждый процесс вправе как читать, так и записывать информацию, а также производить оба действия сразу. Однако в большинстве случаев программный канал используется двумя процессами, на двух его концах. Операции ввода-вывода над каналами очень похожи на аналогичные операции над файлами. Они производятся через системные вызовы `read` и `write` с помощью дескриптора канала. Процесс иногда может и не знать о том, что работает с каналом, а не с обычным файлом.

Различные приложения, такие как командные интерпретаторы, управляют программными каналами, считая, что у тех имеется только один отправитель (записывающий данные) и один получатель (считывающий данные), тем самым образуется односторонний поток данных. Наиболее общим применением программного канала является перенаправление вывода одной программы на вход другой. Для этого пользователи объединяют две программы в конвейер при помощи оператора `|`.

¹ В традиционных системах, например в SVR2, программные каналы реализованы в файловой системе. Они используют поля блоков прямой адресации в индексных дескрипторах файлов (`inode`, см. раздел 9.2.2) для размещения блоков данных в канале. Это ограничивает размер конвейера до десяти блоков. В современных системах UNIX остается такое ограничение несмотря на то, что реализация каналов в них иная.

С точки зрения межпроцессного взаимодействия программные каналы являются эффективным способом передачи данных от одного процесса другому. Однако они обладают некоторыми ограничениями:

- ◆ так как чтение данных из канала приводит к их удалению, он не может быть использован для широковещательной передачи информации нескольким адресатам;
- ◆ данные канала интерпретируются как поток байтов, границы сообщения заранее не известны. Если отправитель посыпает через канал несколько объектов различной длины, получатель не имеет возможности определять, как много объектов ему было передано, он также не знает, где заканчивается один объект и начинается другой¹;
- ◆ когда данные из каналачитываются несколькими процессами, отправитель не в состоянии передать данные кому-то определенному получателю. Точно так же при наличии нескольких отправителей не существует способа определения, какой из них отоспал данные².

Реализация программных каналов может быть осуществлена различными способами. В традиционных вариантах (например, в SVR2) для этого применяются механизмы файловой системы и ассоциация *индексного дескриптора файла* (*inode*) и *элемента таблицы файлов* для каждого канала. Во многих реализациях системы, основанных на BSD, применяются сокеты. В ОС SVR4 используются двунаправленные каналы STREAMS, которые будут описаны подробнее в следующем разделе.

В ОС System V UNIX и других коммерческих вариантах UNIX представлены *файлы FIFO*, называемые также *именованными каналами* (named pipe). Они отличаются от обычных неименованных каналов способами создания и доступа. Пользователь создает файл FIFO при помощи системного вызова `mknod`, передавая ему имя файла и режим его создания. Поле режима описывает тип файла `S_IFIFO` и обычные правила доступа к нему. После этого процесс, обладающий соответствующими полномочиями, может открывать файл FIFO и производить с ним операции чтения и записи. Семантика чтения-записи для файлов FIFO похожа на аналогичные операции над программными неименованными каналами и будет более подробно описана в разделе 8.4.2. Файл FIFO существует до тех пор, пока не будет отсоединен принудительно, даже если у него не останется ни одного активного отправителя или получателя.

Файлы FIFO обладают некоторыми преимуществами по сравнению с программными каналами. Такие файлы могут быть доступны любым процессам.

¹ Взаимодействующие приложения могут договориться о протоколе, описывающем границы пакета информации в каждом объекте.

² Еще раз повторим, что приложения могут определить соглашение по маркировке источника данных каждого объекта.

Они являются постоянными и, следовательно, могут быть использованы для хранения данных пользователей, уже не активных в системе. Они обладают определенным именем в пространстве имен файловой системы. Конечно, файлы FIFO имеют и некоторые недостатки. По завершении использования их необходимо удалять принудительно. Они менее защищены по сравнению с программными каналами, так как к ним может обращаться любой процесс, обладающий достаточными привилегиями. Программные каналы проще в создании и требуют меньшего количества ресурсов.

6.2.3. Каналы в системе SVR4

В системе SVR4 как базовое средство работы с сетями и реализации каналов и файлов FIFO применяются STREAMS (см. главу 17). Это дает возможность предоставления системой новых полезных средств конвейеризации¹. В этом разделе будут описаны только новые возможности каналов. Детали их организации обсуждаются в разделе 17.9.

В системе SVR4 каналы двунаправлены. Вызов `rpipe` возвращает, как и ранее, два дескриптора, однако теперь они оба открыты как для чтения, так и для записи. Синтаксис вызова (совпадающий с традиционным) приведен ниже:

```
status = pipe(int fildes[2]);
```

В SVR4 вызов `rpipe` создает два независимых канала ввода-вывода, работающих по принципу «первым вошел, первым вышел» и представленных двумя дескрипторами. Данные, записанные в `fildes[1]`, могут быть прочитаны из `fildes[0]`, а данные, записанные в `fildes[0]`, могут быть прочитаны из `fildes[1]`. Такой подход является очень удобным, поскольку многие приложения требуют от системы возможности двухсторонних коммуникаций, для чего в предыдущих версиях SVR4 необходимо было открывать два отдельных канала.

Операционная система SVR4 может позволить процессу присоединить к объекту файловой системы любой дескриптор файла STREAMS [10]. Приложение создает канал посредством вызова `rpipe` и затем связывает его дескрипторы с именем файла при помощи конструкции

```
status = fattach(int fildes, char *path);
```

где `path` — путь к объекту файловой системы, обладателем которого является вызывающий процесс². Такой объект может быть обычным файлом, каталогом или специальным файлом. Он не может быть точкой монтирования (то есть на него не должна монтироваться файловая система) или объектом удаленной файловой системы. Он также не присоединяется к другому дескрип-

¹ Новые средства представлены только для программных каналов. Реализация файлов FIFO в ОС SVR4 почти совпадает с традиционной.

² В ином случае вызывающий процесс должен являться привилегированным.

тору файла STREAMS. Существует возможность связать дескриптор с несколькими путями, ассоциируя с ними несколько имен.

После присоединения все последующие операции над *path* будут на самом деле производиться над файлом STREAMS до тех пор, пока дескриптор не будет отключен от *path* через вызов *fdetach*. Используя описанную возможность, процесс может создавать канал и затем дать доступ к нему другим процессам системы.

Пользователь может поместить модули STREAMS в канал или файл FIFO. Такие модули перехватывают проходящие через канал данные и производят над ними различные действия. Так как эти модули функционируют внутри ядра системы, они умеют выполнять действия, недоступные пользовательским приложениям. Только привилегированные пользователи обладают правом добавлять в систему модули, но всем пользователям доступна возможность помещения модулей в открытые ими потоки.

6.2.4. Трассировка процессов

Системный вызов *ptrace* предлагает базовый набор средств трассировки процессов. Обычно он применяется различными отладчиками, например *sdb* или *dbx*. При помощи *ptrace* процесс может управлять выполнением своего потомка. Он может контролировать и несколько потомков сразу, но такое редко применяется на практике. Синтаксис *ptrace* следующий:

```
ptrace (cmd, pid, addr, data);
```

где *pid* — идентификатор трассируемого процесса, *addr* — ссылка на его адресное пространство. Интерпретация аргумента *data* зависит от *cmd*. Аргумент *cmd* позволяет процессу-предку осуществлять следующие действия:

- ◆ записывать или считывать слово из адресного пространства потомка;
- ◆ записывать или считывать слово из области и потомка;
- ◆ считывать или писать в общеселевые регистры потомка;
- ◆ перехватывать определенные сигналы. Если перехватываемый сигнал генерируется для потомка, ядро приостановит его работу и уведомит его предка о возникновении события;
- ◆ устанавливать или удалять точки наблюдения в адресном пространстве потомка;
- ◆ возобновлять выполнение приостановленного потомка;
- ◆ осуществлять пошаговое выполнение потомка: прервать его работу, затем позволить ему выполнить одну инструкцию, после чего снова приостановить его функционирование;
- ◆ завершить работу потомка.

Одна из команд (`cmd==0`) зарезервирована для потомка. Она применяется для информирования ядра системы о том, что потомок будет подвергнут трассировке своим предком. После этого ядро устанавливает для него флаг `traced` (в структуре `proc`), который влияет на обработку потомком сигналов. Если сигнал генерируется для трассируемого процесса, ядро приостанавливает его выполнение и уведомляет об этом его процесс-предок при помощи сигнала `SIGCHLD` (вместо загрузки обработчика сигнала). Это позволяет родительскому процессу перехватить сигнал и произвести соответствующие действия. Флаг `traced` также изменяет выполнение системного вызова `exec`. Если потомок загружает новую программу, вызов `exec` генерирует для потомка сигнал `SIGTRAP` до того, как он возвратится в пользовательский режим. Это дает возможность родительскому процессу получить контроль над потомком перед началом выполнения потомка.

Обычно родительский процесс создает процесс-потомок, который в дальнейшем вызывает `ptrace` для того, чтобы предок имел возможность управлять им. Затем родительский процесс применяет вызов `wait` для ожидания события, изменяющего режим выполнения потомка. После возникновения такого события ядро системы будет родительский процесс. Величина, возвращаемая `wait`, указывает на то, что потомок был приостановлен быстрее, чем завершен, и передает информацию о событии, заставившем потомка приостановить работу. Затем родительский процесс может контролировать дочерний при помощи одной или нескольких команд `ptrace`.

Хотя появление системного вызова `ptrace` помогло создать многие отладчики, он, к сожалению, обладает некоторыми недостатками и ограничениями:

- ◆ процесс умеет управлять выполнением только своих прямых потомков. Если трассируемый процесс производит вызов `fork`, то отладчик не сможет контролировать новые процессы и их потомков;
- ◆ вызов `ptrace` является весьма неэффективным, так как требует нескольких переключений контекста только для того, чтобы передать одно слово от потомка его предку. Контекстные переключения являются обязательными, поскольку отладчик не обладает прямым доступом к адресному пространству потомка;
- ◆ отладчик не в состоянии следить за процессом, который уже находится в стадии выполнения, так как потомку необходимо сначала вызвать `ptrace` для уведомления ядра системы о том, что им можно управлять;
- ◆ трассировка программы `setuid` приводит к возникновению проблем защиты, если эта программа затем произведет вызов `exec`. Пользователь может использовать отладчик для изменения адресного пространства процесса, тогда вызов `exec` приведет к загрузке оболочки вместо программы, которую он хотел выполнить. В результате пользователь загрузит командный интерпретатор с привилегиями суперпользователя. Для предотвращения возникновения такой ситуации в системах UNIX либо отключают трассировку программ `setuid`, либо запрещают действия `setuid` и `setgid` с последующим вызовом `exec`.

В течение длительного периода времени вызов `ptrace` оставался единственным инструментом программ-отладчиков. В современных системах UNIX, таких как SVR4 и Solaris, имеются более эффективные средства отладки, использующие файловую систему `/proc` [9], описанную в разделе 9.11.2. Эти средства не обладают ограничениями, присущими `ptrace`, и предоставляют различные дополнительные возможности, такие как отладка независимых друг от друга процессов или подключение отладчика к выполняющемуся процессу. После появления технологии `/proc` многие отладчики были переписаны заново и теперь не используют для своей работы вызов `ptrace`.

6.3. System V IPC

Средства, описанные в предыдущих разделах, не удовлетворяют требованиям многих приложений по взаимодействию процессов. Появление ОС System V ознаменовалось значительными расширениями возможностей IPC. Разработчики системы представили три механизма: семафоры, очереди сообщений и разделяемую память. Все они известны под общим названием System V IPC [1]. Изначально они создавались для поддержки приложений обработки транзакций. Позже эти средства были взяты на вооружение большинством поставщиков систем UNIX, в том числе и теми, кто создавал ОС на основе BSD. В данном разделе будут описаны возможности этих механизмов и показано, как они были практически реализованы в системе UNIX.

6.3.1. Общие элементы

Все три механизма очень схожи друг с другом в отношении программного интерфейса и своей реализации. При описании их общих возможностей мы будем использовать термин «ресурс IPC» (или просто «ресурс») вместо указания на набор семафоров, очередь сообщений или область разделяемой памяти. Каждый ресурс IPC обладает набором атрибутов.

- ◆ **Ключ (key).** Поддерживаемое пользователем целое число, идентифицирующее конкретный экземпляр ресурса.
- ◆ **Создатель (creator).** Пользовательские и групповые идентификаторы процесса, создавшего ресурс.
- ◆ **Владелец (owner).** Пользовательские и групповые идентификаторы владельца ресурса. При создании ресурса его владелец и создатель одинаковы. Однако процесс, обладающим правами изменения владельца, может указать позже нового владельца ресурса. Такими правами обладают процессы создателя, текущего владельца и суперпользователя.
- ◆ **Права (permissions).** Права файловой системы на чтение/запись/выполнение для владельца, группы и других пользователей.

Процесс получает ресурс при помощи системных вызовов `shmget`, `semget` и `msgget`, передавая им ключ, необходимые флаги и другие аргументы, зависящие от используемого механизма. Разрешенными флагами являются `IPC_CREAT` и `IPC_EXCL`. Первый из них запрашивает ядро о создании ресурса, если таковой не существует. Флаг `IPC_EXCL` применяется совместно с `IPC_CREAT` и запрашивает ядро о возвращении ошибки, если требуемый ресурс уже существует. Если не указывается ни один из флагов, ядро ищет существующий ресурс с тем же ключом¹. Если оно находит такой ресурс и если вызывающий процесс обладает правами доступа к нему, то ядро возвращает *идентификатор ресурса* (resource ID), который может быть в дальнейшем использован для быстрого обнаружения ресурса.

Каждый механизм обладает управляющим системным вызовом (`shmctl`, `semctl` и `msgctl`), предоставляющим несколько различных команд. Эти команды включают в себя `IPC_STAT` и `IPC_SET`, которые применяются для получения и установки статусной информации (специфичной для каждого механизма), а также `IPC_RMID` для освобождения ресурса. Для управления семафорами поддерживаются дополнительные команды управления, которые используются в целях получения и установки переменных отдельных семафоров набора.

Каждый ресурс IPC должен освобождаться принудительно при помощи команды `IPC_RMID`. В противном случае ядро системы будет считать ресурс активным, даже если все использовавшие его процессы были завершены. Такое свойство ресурсов может оказаться весьма удобным. Например, процесс может записать данные в разделяемую область памяти или в очередь сообщений и затем завершить свою работу. Эти данные может запросить позже другой процесс. Ресурс IPC является постоянным, то есть годится для использования уже после завершения работы процесса, обращавшегося к нему.

Однако такой подход имеет и определенные недостатки, так как ядро системы не может определить, был ли ресурс оставлен активным для новых процессов или он был потерян случайно, например, если работа процесса была завершена принудительно и тот не успел освободить ресурс. В результате ядру системы необходимо продолжать поддерживать ресурс, имеющий неизвестное состояние. Если это случается слишком часто, система может такой ресурс выгрузить, так как он, по крайней мере, занимает определенный объем памяти, которому можно найти другое, более полезное применение.

Правами на выполнение команды `IPC_RMID` обладают только процессы, являющиеся создателем, текущим владельцем ресурса или обладающие привилегиями суперпользователя. Удаление ресурса влияет на все процессы, обращающиеся к нему в текущий момент, следовательно, ядро системы должно

¹ Если ключ является специальной переменной `IPC_PRIVATE`, ядро создаст новый ресурс. Этот ресурс недоступен при помощи других вызовов `get` (так как ядро будет каждый раз создавать еще один ресурс), и, следовательно, вызывающий процесс будет обладать исключительными правами на его владение. Владелец может использовать ресурс совместно со своими потомками, которые наследуют его через вызов `fork`.

удостовериться в том, что процессы постепенно и согласованно обрабатывают такое событие. Специфика предпринимаемых действий зависит от конкретного применяемого механизма и будет подробнее описана в следующих разделах.

Для реализации интерфейса каждый тип ресурса обладает собственной таблицей ресурса фиксированного размера. Размер таблицы является настраиваемым и ограничен общим количеством каждого типа ресурсов, одновременно поддерживаемых системой. Каждый элемент таблицы содержит общую структуру `ipc_perm`, а также данные, специфичные для определенного типа ресурса. Структура `ipc_perm` хранит общие атрибуты ресурса (ключ, идентификаторы создателя и владельца, привилегии), а также последовательность чисел, являющуюся счетчиком, который увеличивается при каждом новом использовании элемента таблицы.

При создании ресурса IPC пользователем ядро возвращает идентификатор ресурса, который вычисляется по формуле:

```
id = seq * table_size + index;
```

где `seq` — последовательность чисел для этого ресурса, `table_size` — размер таблицы ресурса и `index` — индекс ресурса в таблице. Эта формула гарантирует создание нового `id`, если элемент таблицы используется повторно, так как значение `seq` инкрементируется. Это защищает процессы от доступа к ресурсам, использующим устаревший идентификатор.

ПРИМЕЧАНИЕ

Термин «инкрементирование переменной» означает, что ее значение увеличивается на единицу. «Декремент» означает уменьшение значения на единицу. Эти термины взяты из языка С, где применяются операторы инкремента (`++`) и декремента (`--`)¹.

Пользователь передает `id` в качестве аргумента последующим системным вызовам, производящим действия над этим ресурсом. Ядро транслирует `id` для обнаружения ресурса в таблице по формуле

```
index = id % table_size;
```

6.3.2. Семафоры

Семафоры [6] — это объекты, находящиеся в диапазоне целых чисел, которые поддерживают две операции, `P()` и `V()`². Примитив `P()` применяется для декремента значения семафора, если новое значение оказывается меньше нуля, то он блокируется. Операция `V()` используется для инкремента значения, при

¹ Впервые операторы автоувеличения и автоуменьшения появились в компиляторе языка B — предшественника С, они были введены К. Томпсоном для собственных нужд при работе с PDP-7. Позже популярность С и UNIX на PDP-10 обусловилась во многом этими режимами, как пишет Д. Ритчи в *The Development of the C Language*. — Прим. ред.

² Имена `P()` и `V()` происходят от голландских слов, означающих соответствующие операции.

этом если результат оказывается равным нулю или больше, то `V()` пробуждает нить или процесс. Эти операции являются неделимыми.

Семафоры можно применять для реализации различных протоколов синхронизации. Например, представьте, какие проблемы возникают при управлении исчисляемого ресурса (обладающего определенным числом элементов). Процесс пытается получить одну из составляющих ресурса и освободить ее после завершения использования. Ресурс может быть представлен семафором, который применяется для управления доступа к нему. Примитив `P()` применяется при каждой попытке запроса ресурса и уменьшает значение семафора при ее удачном завершении. Если переменная станет равна нулю (то есть свободных ресурсов больше нет), все последующие операции приведут к блокировке. При освобождении ресурса стартует операция `V()`, которая увеличивает значение семафора, что приводит к пробуждению заблокированного процесса.

Во многих системах UNIX семафоры применяются ядром для синхронизации внутренних операций. Они также используются в этом качестве и для пользовательских приложений. В ОС System V поддерживается наиболее общая версия семафоров. Системный вызов `semget` создает или запрашивает массив семафоров (верхнюю границу которого можно назначить). Синтаксис вызова приведен ниже:

```
semid = semget (key, count, flag);
```

где `key` — 32-разрядная переменная, передаваемаязывающим процессом. Функция `semget` возвращает массив `count` семафоров, ассоциированных с ключом (`key`). Если с ключом не связано ни одного набора семафоров, то вызов будет возвращать ошибку до тех пор, пока не будет задан флаг `IPC_CREAT`, создающий новый набор семафоров. Если функции передается флаг `IPC_EXCL`, то `semget` возвращает ошибку в том случае, если набор семафоров для указанного ключа уже существует. Переменная `semid` применяется в последующих операциях над семафорами, она идентифицирует массив семафоров.

Системный вызов `semop` применяется для проведения операций над отдельными семафорами массива. Ее синтаксис таков:

```
status = semop (semid, sops, nsops);
```

где `sops` — указатель на элемент `nsops` массива структур `sembuf`. Каждая структура `sembuf`, как это будет показано ниже, представляет одну из операций над отдельным семафором набора.

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

Здесь `sem_num` идентифицирует один из семафоров массива, а `sem_op` указывает на действия, которые следует с ним произвести. Значения переменной `sem_op` интерпретируются по следующим правилам:

<code>sem_op > 0</code>	Добавляет <code>sem_op</code> к текущему значению семафора. Результатом этой операции может стать пробуждение процесса, ожидавшего увеличения переменной
<code>sem_op = 0</code>	Блокировка до тех пор, пока значение семафора не станет равным нулю
<code>sem_op < 0</code>	Блокировка до момента, когда значение семафора станет равным или выше абсолютного значения <code>sem_op</code> , после чего производится вычитание <code>sem_op</code> от значения семафора. Если оно уже выше, чем абсолютная величина <code>sem_op</code> , вызывающий процесс не блокируется

Таким образом, используя всего лишь один системный вызов `semop`, можно указать несколько различных отдельных операций над семафорами. Ядро системы гарантирует выполнение либо всех этих операций, либо ни одной из них. Более того, ядро отслеживает работу `semop`, не позволяя начать обработку повторного вызова над тем же массивом семафоров до тех пор, пока первый вызов не завершит работу либо не будет заблокирован. Если вызову `semop` необходимо приостановить свое выполнение после завершения некоторых действий, то ядро системы после возобновления его работы повторит операцию сначала (отменит все изменения), гарантируя тем самым неделимость функции.

Значениями аргумента `sem_flg` могут быть два различных флага. Флаг `IPC_NOWAIT` указывает ядру на необходимость возврата ошибки функцией вместо приостановки ее выполнения. Если процесс, удерживающий семафор, завершит работу, не освободив его, появляется вероятность возникновения взаимоблокировки. Тогда другой процесс, пытающийся воспользоваться семафором, может оказаться заблокированным навсегда на стадии выполнения операции `P()`. Для защиты от этого вызову `semop` можно передавать флаг `SEM_UNDO`. В этом случае ядро запоминает произведенные им операции и автоматически прокручивает их назад по завершении работы процесса.

Семафоры необходимо удалять из системы принудительно. Для этого применяется команда `IPC_RMID` вызова `semctl`. В противном случае ядро системы будет поддерживать семафоры, даже если они не используются ни одним из процессов. Такой подход позволяет применять семафоры в течение длительного времени, не зависящего от жизненного цикла процессов. Но, с другой стороны, если приложение по завершении работы не освобождает семафоры, то они продолжают занимать ресурсы системы.

После выполнения процессом команды `IPC_RMID` ядро освобождает семафор в таблице ресурсов. Ядро также пробуждает все процессы, заблокированные во время проведения операций над тем же семафором. Результатом

работы `semop` для таких процессов является возврат статуса `EIDRM`. После удаления семафора процесс больше не имеет доступа к нему (с помощью ключа или идентификатора семафора).

Детали реализации семафоров

Ядро преобразует `semid` для получения элемента таблицы ресурсов семафоров, каждый из которых описывается следующей структурой данных:

```
struct semid_ds {
    struct ipc_rerm sem_perm; /* см. раздел 6.3.1 */
    struct sem* sem_base; /* указатель на массив семафоров в наборе */
    ushort sem_nsems; /* количество семафоров в наборе */
    time_t sem_otime; /* время последней операции */
    time_t sem_ctime; /* время последнего изменения */
    ...
};
```

Ядро поддерживает значение и информацию о синхронизации для каждого семафора в наборе в структуре, показанной ниже.

```
struct sem {
    ushort semval; /* текущее значение */
    pid_t sempid; /* идентификатор процесса, вызвавшего последнюю
                    операцию */
    ushort semncnt; /* количество процессов, ожидающих увеличения значения
                     семафора */
    ushort semzcnt; /* количество процессов, ожидающих установления значения
                     семафора, равного нулю */
};
```

Ядро также поддерживает список отмены для каждого процесса, производящего операции над семафорами с флагом `SEM_UNDO`. Этот список содержит запись операций, каждую из которых можно отменить. Если процесс завершает выполнение, ядро проверяет наличие списка отмены. Если оно находит такой список, то «прокручивает» в обратном направлении все действия, совершенные ранее.

Применение семафоров

Технология семафоров позволяет разрабатывать сложные средства синхронизации, используемые взаимодействующими процессами. Первые системы UNIX не обладали поддержкой семафоров, что заставляло для синхронизации приложений искать и применять другие атомарные операции. Например, системный вызов `link` возвращает ошибку, если новое соединение уже существует. Если два процесса пытаются произвести одну и ту же операцию `link` в один момент времени, только для одного из них результат будет успешным. Однако применение операций файловой системы, таких как `link`, в целях синхронизации процессов является неудобным и громоздким.

Появление семафоров смогло удовлетворить большинство требований создателей программных приложений.

Основными проблемами, связанными с применением семафоров, являются условия состязательности и предупреждение взаимоисключений. Использование одиночных семафоров (вместо их массивов) может привести к взаимной блокировке в том случае, если процессу необходимо запросить несколько семафоров. Например, на рис. 6.2 процесс А, удерживающий семафор C1, пытается получить семафор C2 в то время, как процесс Б уже владеет семафором C2 и стремится к тому же в отношении C1. В таком случае ни один из процессов не сможет продолжить функционирование. Хотя этот простейший случай легко обнаружить (и, следовательно, защититься от его возникновения), клинч может произойти и в более сложных вариациях, затрагивающих не один процесс и семафор.

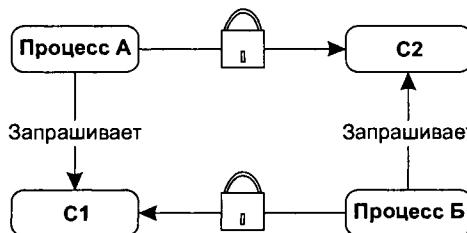


Рис. 6.2. Семафоры могут стать причиной возникновения взаимоблокировок

Коды обнаружения взаимоблокировок и защиты от них реализовывать внутри ядра системы непрактично. Более того, не существует общих и универсальных алгоритмов, защищающих от всевозможных ситуаций. Из этого можно сделать вывод, что ядро оставляет решение проблемы взаимоблокировок приложениям. Предлагая механизм наборов семафоров и неделимые операции над ними, ядро системы дает разработчику интеллектуальные механизмы обработки семафоров. Программисты могут выбрать несколько известных методов защиты от тупиковых ситуаций, некоторые из них будут показаны в разделе 7.10.1.

Одна из основных проблем реализации семафоров в System V связана с тем, что операции по их инициализации и размещению не являются неделимыми. Для размещения набора семафоров пользователь вызывает `semget` вслед за функцией `semctl`, которая инициализирует набор. Такой подход может привести к условию состязательности, которую необходимо пресекать на пользовательском уровне [13].

В завершение скажем, что необходимость принудительного удаления ресурса при помощи команды `IPC_RMID` является общей проблемой любых механизмов IPC. Хотя это свойство и позволяет создателю пережить созданный им ресурс, оно приводит к появлению «мусора» в системе в случае завершения работы процесса без высвобождения всех его ресурсов.

6.3.3. Очереди сообщений

Очередь сообщений — это заголовок, указывающий на связанный список сообщений. Каждое сообщение содержит 32-разрядную переменную *type*, следующую за областью *данных*. Процесс создает или получает очередь сообщений при помощи системного вызова *msgget*, синтаксис которого показан ниже:

```
msgid = msgget (key, flag).
```

Семантика вызова *msgget* совпадает с *semget*. *Key* — это целое число, задаваемое пользователем. Для создания новой очереди сообщений необходимо указать флаг *IPC_CREAT*. Задание флага *IPC_EXCL* ведет к ошибочному завершению работы вызова в том случае, если очередь с указываемым ключом уже существует. Переменная *msgid* используется в дальнейших вызовах для доступа к очереди.

Для того чтобы поместить сообщение в очередь, необходимо произвести следующий вызов:

```
msgsnd (msgqid, msgp, count, flag);
```

где *msgp* указывает на буфер сообщения (содержащий поле типа *type*, следующий за областью данных), *count* — общее количество байтов в сообщении (включает поле *type*). Флаг *IPC_NOWAIT* используется для возврата ошибки, если сообщение невозможно отправить без блокировки (например, когда очередь переполнена, так как очередь обычно обладает настраиваемым ограничением на количество хранящихся в ней данных).

На рис. 6.3 показаны операции над очередями сообщений. Каждая очередь описывается в виде строки в таблице ресурсов очередей сообщений. Эта структура показана ниже:

```
struct msqid_ds {
    struct msg ipc_perm msg_perm; /* см. раздел 6.3.1 */
    struct msg* msg_first; /* первое сообщение в очереди */
    struct msg* msg_last; /* последнее сообщение в очереди */
    ushort msg_cbytes; /* текущая величина очереди в байтах */
    ushort msg_qbytes; /* максимально допустимый размер очереди в байтах */
    ushort msg_qnum; /* текущее количество сообщений в очереди */
    ...
}.
```

Сообщения располагаются внутри очереди в порядке их поступления. Они удаляются из очереди по принципу «первым вошел, первым вышел» при чтении их процессом при помощи вызова

```
count = msgrcv(msgqid, msgp, maxcnt, msctype, flag);
```

в котором *msgp* указывает на буфер, в который помещается входящее сообщение, *maxcnt* ограничивает максимально прочитываемое количество байтов. Если входящее сообщение длиннее, чем *maxcnt*, то оно будет обрезано. Поль-

зователь должен быть уверен в том, что буфер, указанный при помощи `msgp`, имеет достаточный объем для хранения `maxcnt` байтов данных. Возвращаемая функцией величина указывает на успешно прочитанное количество байтов.

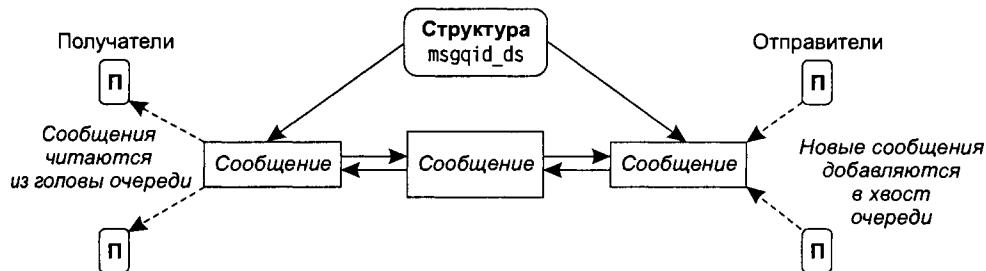


Рис. 6.3. Применение очередей сообщений

Если `msgtype` равно нулю, вызов `msgrcv` вернет первое по счету сообщение очереди. Если значение `msgtype` больше нуля, вызов возвратит первое сообщение типа `msgtype`. Если `msgtype` меньше нуля, функция вернет первое сообщение наименьшего типа, то есть типа, меньшего или равного абсолютной величине `msgtype`. Точно так же, как и в предыдущей функции, флаг `IPC_NOWAIT` заставляет `msgrcv` возвратиться немедленно, если необходимое сообщение отсутствует в очереди.

После прочтения сообщение удаляется из очереди и, следовательно, не может быть прочитано другими процессами. Если часть сообщения отрезана вследствие недостаточности объема буфера, то эта часть теряется навсегда. Никакой индикации о факте обрезки сообщения не производится.

Процесс должен удалять очередь сообщений принудительно при помощи вызова `msgctl` с указанием команды `IPC_RMID`. При этом ядро освобождает очередь и удаляет все сообщения, находившиеся в ней. Если какие-либо процессы были заблокированы в ожидании чтения или записи в очередь, ядро разбудит их, а вызванная им системная функция возвратит статус `EIDRM` (удалено).

Применение очередей сообщений

Каналы и очереди сообщений предлагают схожие услуги, однако последние являются более универсальным механизмом, в котором были преодолены некоторые ограничения, присущие каналам. Очереди сообщений передают данные в виде дискретных сообщений, в то время как каналы работают с неформатированным потоком байтов. Это дает возможность более интеллектуальной обработки передаваемой информации. Поле сообщения `type` (тип) можно использовать различными способами. Например, это поле может указывать на приоритет сообщений, давая возможность получателю проверять более важные сообщения раньше, чем остальные. Если очередь сообщений обрабатывается одновременно несколькими процессами, поле типа может быть использовано для определения адресата.

Очереди сообщений являются эффективным средством передачи небольших объемов данных, но становятся слишком неудобными для больших объемов информации. Когда процесс отправляет сообщение в очередь, ядро системы копирует его во внутренний буфер. Если другой процесс запросит это сообщение, ядро скопирует данные в адресное пространство получателя. Таким образом, передача сообщения требует проведения двух операций копирования, что приводит к низкой производительности. Позднее в этой главе будет рассказано о средствах межпроцессного взаимодействия Mach, позволяющих эффективно передавать большие объемы данных.

Еще одним ограничением очередей сообщений является невозможность указания получателя. Любой процесс, обладающий соответствующими полномочиями, имеет право запрашивать сообщение из очереди. Хотя, как это упоминалось ранее, взаимодействующие процессы могут договориться о протоколе указания адресатов, ядро системы никак не участвует в этом. Также механизм очередей сообщений не поддерживает *широковещательную передачу*, при применении которой процесс может отправлять одно и то же сообщение нескольким получателям.

Средства STREAMS поддерживаются большинством современных систем UNIX. Они обладают богатыми возможностями передачи сообщений. STREAMS являются более функциональными, чем очереди сообщений. Одной из возможностей, которой обладают очереди сообщений, но не поддерживающейся STREAMS, являются селективные запросы сообщений в зависимости от их типов. Однако разработчики большинства приложений считают STREAMS более удобным интерфейсом, поэтому очереди сообщений в современных системах UNIX оставлены больше из соображений обратной совместимости. Более подробно работа STREAMS будет рассмотрена в главе 17.

6.3.4. Разделяемая память

Область разделяемой памяти — это некоторый объем физической памяти, который используется совместно сразу же несколькими процессами. Процесс может присоединить эту область в качестве диапазона виртуальной памяти в адресном пространстве процесса. Диапазон может быть различным для каждого процесса (рис. 6.4). После присоединения процесс обладает доступом к этой области, не отличающимся от доступа к любому другому участку памяти, то есть без необходимости применения системных вызовов для записи или чтения данных из нее. Следовательно, *механизм разделяемой памяти предоставляет процессу максимально быстрый способ доступа к данным*. Если процесс записывает данные в ячейки разделяемой памяти, их содержимое незамедлительно становится видимым остальным процессам, разделяющим между собой эту область¹.

¹ На многопроцессорных системах для гарантии целостности кэша необходимы дополнительные операции. Некоторые из них будут описаны в разделе 15.13.

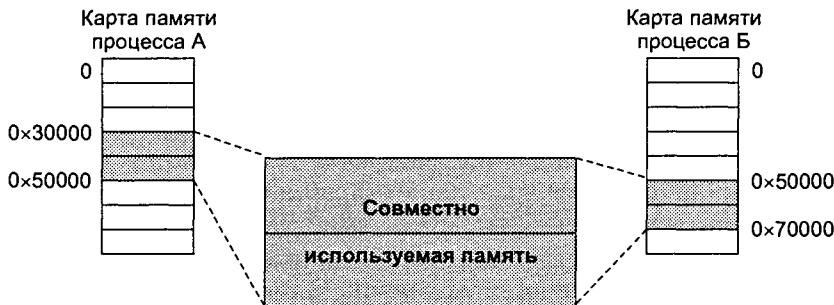


Рис. 6.4. Присвоение разделяемой области памяти

Для первоначального получения или создания области разделяемой памяти процесс использует следующий вызов:

```
shmid = shmget (key, size, flag);
```

где `size` — размер области; остальные параметры и флаги идентичны входным аргументам вызовов `semget` и `msgget`. Затем процесс производит присоединение области виртуальной адресации:

```
addr = shmat (shmid, shmaddr, shmflg);
```

Аргумент `shmaddr` указывает на адрес, к которому может быть присоединена область разделяемой памяти. В качестве аргумента `shmflag` можно указать флаг `SHM_RND`, который запросит у ядра присвоить `shmaddr` приблизительно, исходя из указанного диапазона. Если `shmaddr` равен нулю, ядро вправе выбрать для области любой адрес. Флаг `SHM_RDONLY` указывает на необходимость присвоения области только для чтения. Вызов `shmat` возвращает адрес, присвоенный области.

Процесс может исключить область разделяемой памяти из своего адресного пространства при помощи вызова

```
shmdt (shmaddr);
```

Для полного удаления области процессу необходимо использовать команду `IPC_RMID` системного вызова `shmctl`. Эта функция помечает область как удаленную, которая подвергнется удалению только после того, как все процессы произведут ее отключение от своего адресного пространства. Ядро поддерживает данные о числе процессов, подключенных к каждой области. Если область помечена как удаленная, новые процессы не могут подключаться к ней. Если область не была удалена принудительно, ядро системы будет продолжать поддерживать ее даже в случае отсутствия подключенных к ней процессов. Такой подход удобен для многих приложений, так как процесс имеет возможность перед завершением работы оставить какие-либо данные, которые могут быть получены позже. Для этого взаимодействующий процесс подключается к области памяти, используя тот же ключ `key`.

Реализация разделяемой памяти сильно зависит от архитектуры виртуальной памяти конкретной операционной системы. Некоторые варианты ОС используют для назначения области разделяемой памяти единую таблицу страниц, после чего предоставляют доступ к этой таблице всем процессам, подключенным к области. В других реализациях ОС применяются отдельные для каждого процесса карты трансляции адресов областей. При использовании этой модели, если процесс выполняет действие, изменяющее назначение памяти для разделяемой страницы, то это должно приводить к изменению всех назначений для этой страницы памяти. В системе SVR4 (о реализации обработки памяти которой будет рассказано в главе 14) для размещения страниц области разделяемой памяти используется структура `anon_map`. Таблица ресурсов разделяемой памяти содержит строки, представленные следующей структурой:

```
struct shmid_ds {  
    struct ipc_perm shm_perm; /* см раздел 6.3.1 */  
    int shm_segsz; /* размер сегмента в байтах */  
    struct anon_map *shm_amp; /* указатель на информацию обработки памяти */  
    ushort shm_nattch; /* текущее количество подключений */  
};
```

Механизм разделяемой памяти является быстрым и универсальным средством, позволяющим совместно использовать большие объемы данных без применения копирования или системных вызовов. Основным ограничением механизма является отсутствие средств синхронизации. Если два процесса пытаются изменить одну и ту же разделяемую область памяти, ядро системы не может обеспечить последовательность этих операций, что приведет к смешению записанных данных. Процессы, разделяющие между собой область памяти, должны самостоятельно поддерживать собственный протокол синхронизации. Обычно для этой цели используются простые конструкции, такие как семафоры. Применение таких конструкций приводит к необходимости вызова одной или нескольких системных функций, что уменьшает производительность работы с разделяемой памятью.

Многие современные системы UNIX (в том числе и SVR4) предлагают системный вызов `mmap`, который назначает файл (или его часть) как адресное пространство вызывающего процесса. Процессы могут применять вызов `mmap` для осуществления взаимодействия между собой путем назначения одного и того же файла в адресном пространстве каждого процесса (в режиме `MAP_SHARED`). В результате этих действий появляется область разделяемой памяти, которой является содержимое файла. Если процесс изменяет назначенный файл, то произведенные им изменения становятся сразу видны остальным процессам, подключенным к этому файлу. Обновление файла на диске производится ядром системы. Преимуществом вызова `mmap` является использование имен, принятых в файловой системе, вместо ключей. В отличие от

разделяемой памяти, чьи страницы резервируются в области свопинга (см. раздел 14.7.6), страницы, созданные `mmap`, резервируются при помощи файла, к которому они были присоединены. Более подробно работа `mmap` будет показана в разделе 14.2.

6.3.5. Применение механизмов IPC

Существуют определенные сходства между механизмами IPC и файловой системой. Идентификатор ресурса похож на дескриптор файла. Вызов `get` повторяет `open`, команда `IPC_RMID` схожа с вызовом `unlink`, а вызовы `send` и `receive` аналогичны `read` и `write`. Системный вызов `shmctl` предлагает возможности для разделяемой памяти, сходные с `close`. Однако для очередей сообщений и семафоров не существует эквивалента вызова `close`, так как представляется более предпочтительным удалять эти ресурсы сразу же. В результате процесс, использующий очередь сообщений или семафоры, может обнаружить, что необходимый ему ресурс больше не существует.

Ключи, ассоциируемые с ресурсами и формирующие пространство их имен, значительно отличаются от имен, применяемых в файловых системах¹. Каждый из этих механизмов обладает собственным адресным пространством, ключ используется для его однозначной идентификации. Так как ключ представляет собой простое целое число, выбиралось пользователем произвольно, он может быть использован только на одной машине и не подходит для распределенных сред вычислений. Также задача использования уникальных ключей независимыми между собой процессами представляет определенную трудность, поскольку простота ключей может привести к конфликтам при использовании их другими приложениями. В системе UNIX поддерживается библиотечная процедура `ftok` (описанная в руководстве `stdipc(C3)`), которая применяется для создания ключей, основанных на именах файлов и целых числах. Синтаксис `ftok` показан ниже:

```
key = ftok (char *pathname, int ndx);
```

Процедура `ftok` создает значение ключа на основе `ndx` — номера индексного дескриптора файла. Намного проще решить задачу уникальности имени ключа при применении имен файлов (приложение, к примеру, может использовать в качестве такого имени путь к собственному выполняемому файлу) и тем самым уменьшить вероятность возникновения конфликтов. Параметр `ndx` увеличивает гибкость библиотечной процедуры и может быть использован для указания идентификатора проекта, известного всем взаимодействующим приложениям, или другой полезной информации.

Основной проблемой механизмов IPC является их незащищенность, так как идентификатор ресурса представляет собой ссылку на общую таблицу

¹ В некоторых ОС, таких как Windows NT или OS/2, для имен разделяемых областей используются пути файловой системы (но не в обязательном порядке, разделяемые области могут быть и неименованными). — Прим. ред.

ресурсов. Доступ к ресурсу может получить процесс, не обладающий соответствующими полномочиями. Для этого нужно просто угадать идентификатор ресурса. Таким образом, процесс имеет потенциальную возможность считывать или записывать сообщения в разделяемую память или искажать семафоры других процессов. Назначение определенных полномочий ресурсу может частично решить проблему защиты, однако часто процессам необходимо использовать ресурсы совместно с другими процессами, относящимися к разным пользователям системы. Следовательно, в таких случаях невозможно применять строгие разграничения прав доступа. Использование последовательности чисел в качестве идентификатора ресурса дает очень ограниченную защищенность из-за того, что их не так уж трудно угадать. Их применение ставит определенные проблемы перед приложениями, обладающими требованиями по безопасности.

Большинство средств, представленных в System V IPC, может быть реализовано при помощи других возможностей системы, например блокировки или каналов. Однако инструменты IPC являются более гибкими, эффективными и обладающими большей производительностью, чем компоненты файловой системы ОС.

6.4. Mach IPC

Оставшаяся часть главы будет посвящена рассмотрению средств IPC системы Mach, базирующихся на сообщениях. В Mach средства взаимодействия процессов являются центральным и наиболее важным компонентом ядра. Вместо реализации поддержки IPC системой разработчики Mach создали средства IPC, которые являются основой операционной системы. При разработке Mach IPC были поставлены следующие цели:

- ◆ передача сообщений должна быть фундаментальным механизмом коммуникаций;
- ◆ поддерживаемый объем данных одного сообщения должен варьироваться от нескольких байтов до размера всего адресного пространства системы (то есть до 4 Гбайт). Ядро должно осуществлять передачу больших объемов информации без копирования информации;
- ◆ ядро обязано поддерживать защищенные соединения и позволять отправку и получение сообщений только авторизованным нитям;
- ◆ средства коммуникаций и управления памятью должны быть тесно взаимосвязаны между собой. Подсистема IPC использует механизмы копирования при записи подсистемы памяти для повышения эффективности передачи больших объемов данных. С другой стороны, подсистема памяти прибегает к механизмам IPC для осуществления коммуникаций с пользовательскими программами управления памятью;

- ◆ средства Mach IPC должны поддерживать коммуникации между пользовательскими заданиями, а также между пользователем и ядром. В системе Mach нить производит системный вызов при помощи отправки сообщения ядру, а ядро, в свою очередь, возвращает результат вызова в ответном сообщении;
- ◆ механизм IPC должен подходить для клиент-серверных приложений. В системе Mach для поддержки многих служб реализованы серверные программы пользовательского уровня (например, поддержка файловой системы или управление памятью), которые ранее обычно обрабатывались ядром операционной системы. Такие серверные программы используют Mach IPC для обработки запросов к службам;
- ◆ интерфейс должен быть расширяемым до поддержки распределенных вычислений. Пользователь не должен знать, куда он отправляет сообщения: на локальный сервер или удаленный узел.

Средства IPC долгое время оставались неизменными в различных версиях системы Mach. Механизмы взаимодействия процессов Mach 2.5 будут представлены в разделах 6.4–6.9. Эта версия системы является наиболее популярной, и именно ее взяли за основу разработчики OSF/1 и Digital UNIX. В Mach 3.0 возможности IPC были расширены. О них пойдет речь в разделе 6.10.

В этой главе вы снова увидите термины «задание» или «нить» системы Mach. Более подробно они описывались в разделе 3.7.1. Вкратце, задание является набором ресурсов, в том числе пространства адресов, в котором выполняется одна или несколько нитей. Нить — это динамическая сущность, имеющая независимые счетчик команд и стек (логическая управляющая последовательность) программы. Традиционный процесс UNIX эквивалентен заданию, содержащему одну нить. Все нити задания разделяют между собой его ресурсы.

6.4.1. Основные концепции

Основными понятиями IPC в системе Mach являются сообщения и порты. *Сообщение* — это набор данных определенного типа. *Порт* — это защищенная очередь сообщений. Любое сообщение можно отослать только в порт, но не заданию или нити. В ОС Mach каждому порту присваиваются *права на получение* и *права на отправку*. Эти права поддерживаются задачами. Право на передачу разрешает задаче отправлять в порт сообщения. Право на получение позволяет получать сообщения, которые были посланы в порт. Правом на отправку сообщений в один и тот же порт могут обладать сразу несколько заданий, но только одно из них, называемое владельцем порта, имеет право на по-

лучение¹. Таким образом, порты разрешают производить соединения типа «многие к одному», как это показано на рис. 6.5.

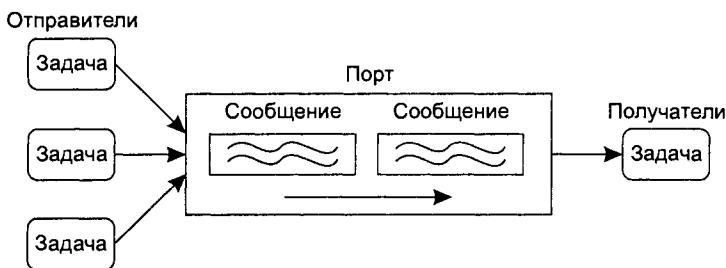


Рис. 6.5. Осуществление коммуникаций через порты системы Mach

Сообщение может быть простым или сложным. Простое сообщение включает в себя обычные данные, не интерпретируемые ядром. Сложное сообщение может содержать обычные данные, внешней памяти (данные, передаваемые при помощи метода «копирования при записи»), а также права на отправку и получение для различных портов. Ядро интерпретирует эту информацию и преобразует ее в форму, необходимую получателю.

Каждый порт имеет счетчик ссылок, отслеживающий количество его прав. Каждое такое право (также называемое совместимостью) представляет одно из имен порта. Имена представляют собой целые числа, диапазон имен является локальным для каждого задания. Следовательно, для одного и того же порта двумя заданиями могут применяться разные имена (рис. 6.6). Точно так же, одно и то же имя порта в разных заданиях может ссылаться на различные порты.

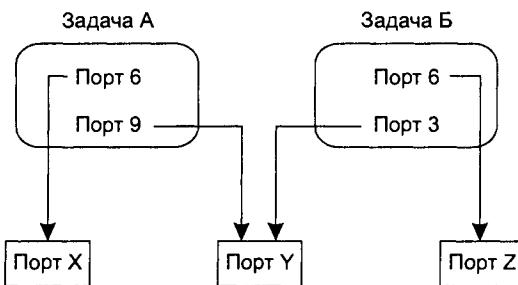


Рис. 6.6. Локальные имена портов

Порты также используются для представления объектов ядра. Следовательно, каждый объект, такой как нить, задание или процессор, может быть

¹ В ранних версиях системы Mach разделялись понятия владельца и обладателя права на получение. В версии 2.5 (и более поздних) права владельца заменены средством восстановления порта, которое будет описано в разделе 6.8.2.

представлен как порт. Права таких портов представляют собой ссылки на объекты и позволяют производить различные операции над объектами. Обслуживание прав таких портов закреплено за ядром.

Каждый порт имеет очередь сообщений конечного размера. Размер очереди является простым механизмом управления нитями. Отправители блокируются при заполнении очереди, а получатели — при пустой очереди.

Каждое задание или нить обладает набором портов, заданных по умолчанию. Например, каждое задание имеет право на отправку в порт `task_self`, представляющий само задание (правами на получение из этого порта обладает ядро) и право на получение из порта `task_notify` (отправлять сообщения в который может ядро). Задание владеет правами на получение из порта `bootstrap`, который предоставляет доступ к серверу имен. Каждая нить имеет право на отправку в порт `thread_self` и право на получение из порта `reply`, используемый для приема ответов от системных вызовов и удаленных вызовов процедур к другим нитям. Существует также порт `exception`, который ассоциируется с каждым заданием или нитью. Владельцем прав на порты нити является задание, в котором выполняется эта нить. Следовательно, такие порты являются доступными каждой нити этого задания.

Задания наследуют права на порты от своих предков. Каждое задание обладает списком *зарегистрированных портов*. Такой подход позволяет заданию иметь доступ к различным системным службам. Порты наследуются новыми заданиями на стадии их создания.

6.5. Сообщения

Ядро системы Mach основано на сообщениях, которые используются для доступа к большинству системных служб. Средства Mach IPC предоставляют возможность коммуникации между пользовательскими задачами, между пользователями и ядром, а также между различными подсистемами внутри ядра системы. Расширение средств IPC на сеть реализовано при помощи программы пользовательского уровня под названием `netmsgserver`, позволяющей приложениям обмениваться сообщениями по сети точно так же, как и внутри одной машины. Фундаментальными компонентами Mach IPC являются сообщения и порты. В этой главе вы увидите описание структур данных и функций, при помощи которых реализованы эти компоненты системы.

6.5.1. Структуры данных сообщения

Сообщение представляет собой набор данных определенного типа. Существует три основных типа данных, располагаемых в сообщениях:

- ◆ обычные данные, не интерпретируемые ядром и передающиеся получателю путем их физического копирования;

- ◆ внешняя память (out-of-line), используемая для передачи больших объемов данных при помощи технологии «копирования при записи» (см. раздел 6.7.2);
- ◆ передача на порты или прием прав.

Каждое сообщение имеет заголовок фиксированной величины, после которого сразу же располагается набор компонентов данных переменного размера (рис. 6.7). Заголовок сообщения содержит следующую информацию:

- ◆ **тип** (type). Простой (только обычные данные) или сложный (в сообщение включена информация внешней памяти или прав портов);
- ◆ **размер** (size). Указывается размер всего сообщения (тело + заголовок);
- ◆ **порт назначения** (destination port);
- ◆ **порт ответа** (reply port). Отправка прав на порт, на который можно послать ответ. Поле устанавливается только в случае необходимости пересылки ответа отправителю;
- ◆ **идентификатор сообщения** (message ID). Используется приложениями по своему усмотрению.

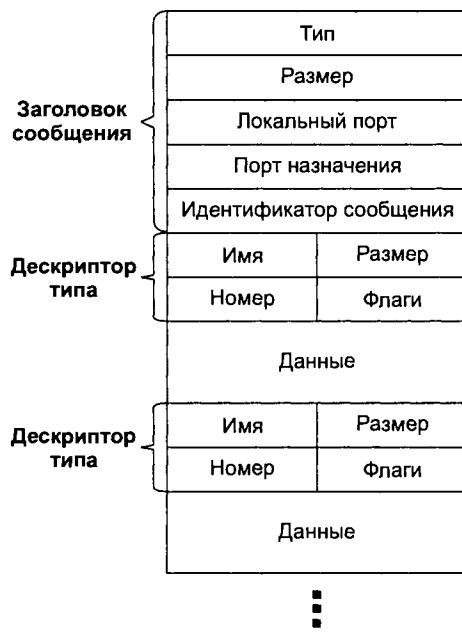


Рис. 6.7. Сообщение системы Mach

Перед отправкой сообщение создается в адресном пространстве задания-отправителя. На этом этапе для указания портов назначения и ответа применяются локальные имена задания. Перед пересылкой сообщения ядро системы преобразует локальные имена в значения, используемые получателем.

Каждый компонент содержит дескриптор типа, после которого следуют данные. Дескриптор представляет следующую информацию:

- ◆ **имя** (name). Указывает тип данных. Система Mach 2.5 распознает 16 различных значений этой переменной, в том числе внутреннюю память, права порта на отправку и получение или диапазон данных (byte, integer (16 и 32 разряда), string или real);
- ◆ **размер** (size). Размер наборов данных, присутствующих в компоненте;
- ◆ **количество** (number). Количество наборов данных компонента;
- ◆ **флаги** (flags). Указывает, находятся ли данные внутри сообщения или снаружи (out-of-line), а также необходимость освобождения памяти или прав на порты для задания-отправителя.

6.5.2. Интерфейс передачи сообщений

Приложения могут пользоваться службой передачи сообщений различными способами:

- ◆ отправлять сообщение, не ожидая ответа;
- ◆ ожидать незапрашиваемые сообщения и обрабатывать их по мере получения;
- ◆ отправлять сообщение и требовать ответа, но не ожидать его. В этом случае приложение получает ответы асинхронно и обрабатывает его через достаточно большой промежуток времени;
- ◆ отправлять сообщение и ожидать ответ.

Программный интерфейс средств передачи сообщений поддерживает три функции, которые при совместном применении обеспечивают вышеперечисленные способы коммуникаций [2].

```
msg_send (msg_header_t* hdr,  
          msg_option_t option,  
          msg_timeout_t timeout);
```

```
msg_rcv (msg_header_t* hdr,  
          msg_option_t option,  
          msg_timeout_t timeout);
```

```
msg_rpc (msg_header_t* hdr,  
          msg_size_t rcv_size,  
          msg_option_t option,  
          msg_timeout_t send_timeout,  
          msg_timeout_t receive_timeout);
```

Вызов `msg_send` используется для отправки сообщения, не требующего ответа. Этот вызов может быть заблокирован в случае отсутствия свободного

места в очереди сообщений принимающего порта. Функция `msg_rcv` также блокируется до тех пор, пока сообщение не будет получено. Каждый из этих вызовов поддерживает флаг `SEND_TIMEOUT` или `RCV_TIMEOUT`. При их задании функция может быть заблокирована на период, не превышающий значения `timeout`, задаваемого в миллисекундах. После исчерпания этого периода времени происходит возврат функции со статусом `timed out` (превышение лимита времени). Флаг `RCV_NO_SENDERS` заставляет функцию `msg_rcv` закончить работу, если больше ни одно задание не имеет прав на отправку сообщений в этот порт.

Вызов `msg_rpc` применяется для отправки исходящего сообщения в случае необходимости получения ответа на него. Эта функция представляет собой оптимизированный вариант использования `msg_send` и последующего `msg_rcv`. Ответ использует тот же буфер, где до этого содержалось исходящее сообщение. Функция `msg_rpc` поддерживает все опции вызовов `msg_send` и `msg_rcv`.

Заголовок сообщения включает в себя данные о его размере. При вызове `msg_rcv` заголовок будет показывать максимальный размер исходящего сообщения, поддерживаемый заданием, вызвавшим функцию. При возврате заголовок будет содержать данные о действительном размере полученного сообщения. При применении `msg_rpc` необходимо задавать максимальный размер отдельно в переменной `rcv_size`, так как заголовок сообщения в этом случае уже содержит размер исходящего сообщения.

6.6. Порты

Порты представляют собой защищенные очереди сообщений. Права на отправку или получение сообщений для портов могут задаваться заданиями. Доступ к портам разрешен только заданиям, обладающим соответствующими правами. Многие задания могут обладать правами на отправку сообщений в тот или иной порт, но только одно задание имеет право получать из порта. Обладатель прав на получение автоматически получает и право на отправку в этот порт.

В ОС Mach порты применяются для представления объектов системы, таких как задания, нити и процессоры. Правами на отправку и получение для таких портов обладает ядро. Порты имеют счетчики ссылок. Каждое право на отправку является ссылкой на объект, представляемый портом. Такие ссылки дают возможность обладателю производить действия над объектом. Например, порт задания `task_self` представляет само задание. Задание может отправлять сообщения на этот порт для запроса служб ядра. Если иное задание (чаще всего отладчик) также обладает правами на отправку в тот же порт, то оно вправе выполнять над этим заданием различные операции, такие как приостановка выполнения, при помощи отправки в порт сообщений.

В следующих разделах описываются диапазон имен и структуры данных, используемые для представления портов.

6.6.1. Диапазон имен портов

Каждое право порта представлено при помощи его имени. Имена портов – это всего лишь целые числа, диапазон возможных значений которых локален для каждого задания. Таким образом, один и тот же порт в разных заданиях может иметь различные имена. С этой точки зрения имена портов схожи с дескрипторами файлов UNIX.

Для каждого порта задания должно быть определено по крайней мере одно имя. Права портов могут передаваться при помощи сообщений, следовательно, задание может запросить право на один и тот же порт несколько раз. Ядро системы следит за тем, чтобы каждый раз использовалось одно и то же имя. В результате задание может сравнивать два имени порта. Если они не совпадают, то они не могут указывать на один и тот же порт.

Каждый порт также представляется при помощи глобальной структуры данных ядра. Ядро производит преобразование локального имени порта в глобальное, то есть на адрес его глобальной структуры данных этого порта, а также обратное преобразование. Если говорить о дескрипторах файлов, то в системах UNIX каждый процесс поддерживает таблицу дескрипторов в своей области и, в которой хранятся указатели на объекты открытых файлов. Однако в системе Mach принят другой метод преобразования, описание которого вы можете увидеть в разделе 6.6.3.

6.6.2. Структура данных порта

Ядро поддерживает структуру данных `kern_port_t` для каждого порта, содержащую следующую информацию:

- ◆ счетчик ссылок на все имена (права) порта;
- ◆ указатель на задание, являющееся обладателем прав на получение;
- ◆ локальное имя порта для задания-получателя;
- ◆ указатель на резервный порт. Если оригиналный порт будет освобожден, все сообщения будут передаваться на порт, указанный в качестве резервного;
- ◆ двунаправленный связанный список сообщений;
- ◆ очередь блокированных отправителей;
- ◆ очередь блокированных получателей. Хотя задание может обладать правами на получение, отдельные его нити могут оставаться в режиме ожидания сообщений;
- ◆ связанный список всех преобразований объекта;
- ◆ указатель на набор портов; указатели на следующий и предыдущий порты набора (если данный порт является частью набора, см. раздел 6.8.3);
- ◆ текущее число сообщений в очереди;
- ◆ максимальное поддерживаемое количество сообщений («журнал заказов»).

6.6.3. Преобразования портов

В Mach поддерживается одно вхождение преобразования для каждого права порта. Все вхождения должны включать в себя набор записей `<task, port, local_name, type>`, где `task` — задание, обладающее правом, `port` — указатель на структуру данных ядра, `local_name` — локальное имя порта, а переменная `type` соответствует получению или отправке. Система Mach использует эти вхождения для различных целей:

- ◆ вызов `msg_send` должен конвертировать `<task, local_name>` в `port`;
- ◆ вызов `msg_rcv` должен конвертировать `<task, port>` в `local_name`;
- ◆ если задание освобождает порт, ядро отыскивает все права порта;
- ◆ при удалении задания ядро отыскивает преобразования для этого задания и освобождает соответствующие ссылки;
- ◆ при удалении порта ядро отыскивает преобразования этого порта и предупреждает об этом все задания, обладающие правами на этот порт.

Для реализации этих механизмов необходима технология, которая бы эффективно поддерживала все перечисленные операции. На рис. 6.8 показаны структуры данных преобразования порта системы Mach 2.5. В ОС Mach применяются две глобальные таблицы хэширования для быстрого поиска вхождений: таблица `TP_table` хэширует вхождения, основанные на `<task, port>`, таблица `TL_table` хэширует их по кортежу `<task, local_name>`. Структуры данных `kernel_port_t` и `task` содержат заголовки связанных списков преобразований для порта и задания соответственно.

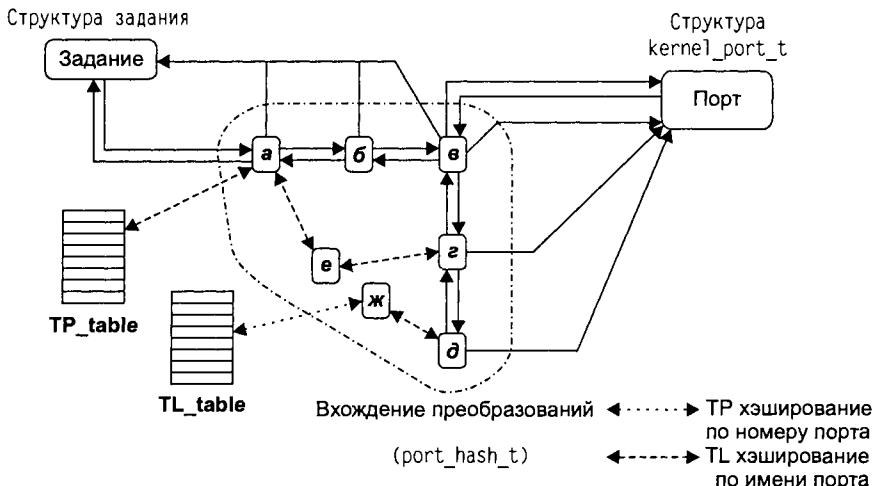


Рис. 6.8. Преобразование портов в ОС Mach

На рис. 6.8 вхождения *a*, *b* и *e* описывают преобразования различных портов одного задания, а вхождения *b*, *g* и *d* — преобразования одного и того

же порта для различных заданий. Вхождения *б*, *г* и *е* транслируются в одинаковый индекс в таблице *TP_table*¹, в то время как вхождения *д* и *ж* хэшируются одинаковым индексом в таблице *TL_table*. Каждое вхождение преобразования описывается структурой *port_hash_t*, содержащей следующую информацию:

- ◆ *task* (задание). Задание-владелец права;
- ◆ *local_name* (локальное имя). Имя права задания;
- ◆ *type* (тип). Отправка или получение;
- ◆ *obj* (объект). Указатель на объект порта в ядре системы.

Кроме этого, каждое вхождение преобразования находится в каждом из нижеперечисленных двунаправленных связанных списков:

- ◆ *TP_chain*. Цепочка хэширования, основанная на кортеже *<task, port>*;
- ◆ *TL_chain*. Цепочка хэширования, основанная на кортеже *<task, local_name>*;
- ◆ *task_chain*. Список всех преобразований, владельцем которых является задание;
- ◆ *obj_chain*. Список всех преобразований порта.

6.7. Передача сообщений

При передаче сообщения производятся следующие действия:

1. Сначала отправитель создает сообщение в своем адресном пространстве.
2. Для его отправки отправитель вызывает системную функцию *msg_send*. Порт назначения указывается в заголовке сообщения.
3. Ядро копирует сообщение во внутреннюю структуру данных (*kern_msg_t*) при помощи процедуры *msg_copyin()*. На этой стадии происходит преобразование прав порта в указатели на объекты порта ядра и копирование внешней памяти в карту хранения.
4. В дальнейшем действия могут отличаться в зависимости от условий:
 - ◆ если нить ожидает сообщение (находится в списке блокированных в ожидании приема из порта), она будет разбужена и сообщение будет передано ей напрямую;
 - ◆ в ином случае, если очередь сообщений окажется заполненной полностью, отправитель будет заблокирован до тех пор, пока из очереди не будет удалено сообщение;

¹ Согласно рисунку не «б», а «а». — Прим. ред.

- в ином случае сообщение будет поставлено в очередь порта, где оно будет находиться до тех пор, пока нить принимающего задания не загрузит `msg_rcv`.
5. Выход ядра из выполнения `msg_send` происходит после того, как сообщение будет поставлено в очередь или передано получателю напрямую.
 6. Когда получатель вызывает системную функцию `msg_rcv`, ядро загружает процедуру `msg_dequeue()` для удаления сообщения из очереди. Если очередь окажется пустой, получатель будет заблокирован до тех пор, пока в ней не появится сообщение.
 7. Затем ядро производит копирование сообщения в адресное пространство получателя при помощи функции `msg_copyout()`, которая производит дальнейшие преобразования внешней памяти и прав портов.
 8. Во многих случаях отправитель сообщения требует ответа на него. Для того чтобы обратная передача была возможна, получатель должен обладать правами отправки в порт, владельцем которого является отправитель. Отправитель передает это право в поле *порта ответа* заголовка сообщения. При этом отправитель может воспользоваться вызовом `mach_rpc` для оптимизации обмена. Вызов `mach_rpc` эквивалентен вызову `msg_send`, за которым следует `msg_rcv`.

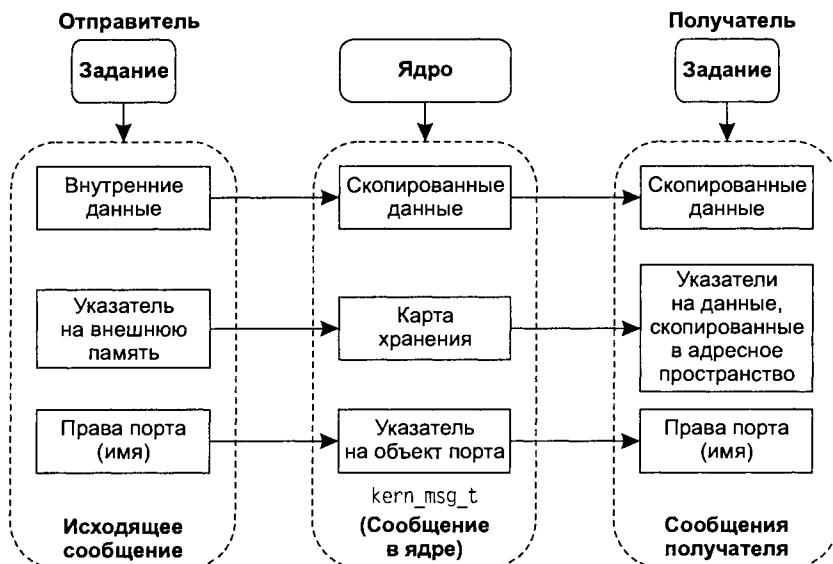


Рис. 6.9. Две стадии передачи сообщений

На рис. 6.9 показаны изменения различных компонентов сообщения, происходящие в процессе его передачи. Давайте рассмотрим некоторые аспекты передачи сообщения более подробно.

6.7.1. Передача прав порта

Существует несколько причин осуществления передачи прав портов через сообщения. Наиболее частым случаем является передача прав на порт ответа (рис. 6.10). Нить задания H1 отправляет сообщение в порт P2, владельцем которого является задание H2. В этом сообщении H1 передает право на отправку для порта P1, для которого у нити H1 имеется право на получение. В результате нить H2 может отправить на порт P1 ответное сообщение, которое будет ожидать нить-отправитель. Описанная ситуация возникает настолько часто, что разработчики внесли в заголовок сообщения отдельное поле для хранения прав на порт ответа.

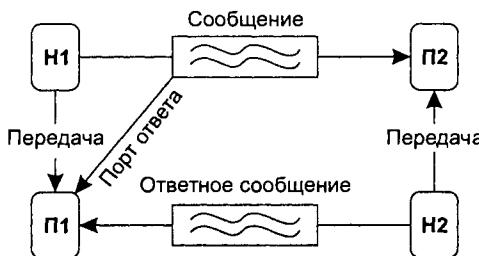


Рис. 6.10. Сообщение может содержать права отправки на порт ответа

Еще одной часто встречающейся ситуацией является взаимодействие между серверной программой, клиентом и сервером имен (рис. 6.11). Сервер имен хранит права отправки для некоторых серверных программ системы. Обычно серверные программы регистрируются в сервере имен самостоятельно сразу после начала работы (1). Все задания наследуют права отправки на сервер имен во время процедуры их создания (значение порта хранится в поле порта инициализации структуры задания).

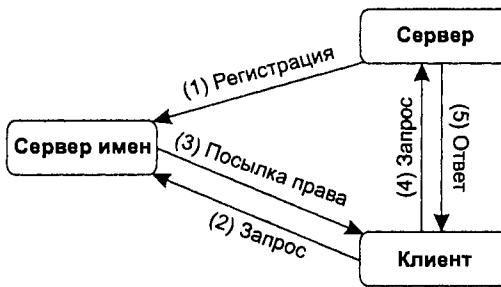


Рис. 6.11. Использование сервера имен для инициализации взаимодействия между клиентом и сервером

Если клиенту необходим доступ к серверной программе, он должен сначала получить право отправки в порт, владельцем которого является сервер. Для этого клиент делает запрос к серверу имен (1), который в ответ возвра-

тит право на отправку сообщений в его порт (3). Этим правом клиент воспользуется для передачи запроса серверу (4). Запрос содержит порт получения, который может быть использован сервером для отправки своего ответа клиенту (5). Все последующие взаимодействия между клиентами и серверами не требуют проведения повторных запросов к серверу имен.

Отправитель передает права порта, используя для него локальное имя. Дескриптор типа для этого компонента сообщения информирует ядро о том, что данное поле является правами порта. Локальное имя ничего не говорит получателю сообщения, следовательно, ядро системы должно перед отправкой преобразовать его. Для этого оно производит поиск вхождений преобразований в `<task, local_name>`, идентифицируя объект ядра (или глобальное имя) для этого порта.

При запросе сообщения ядру необходимо преобразовать глобальное имя в локальное имя, принятое в задании-получателе. Для этого ядро сначала проверяет, не имеет ли получатель прав на этот порт (хэшируя при этом `<task, port>`). Если это так, ядро преобразует его в указанное имя. В противоположном случае ядро выберет новое имя порта получателя и создаст вхождение преобразования, указывающее на этот порт. Имена портов обычно представляют собой целые небольшие числа, поэтому ядро будет при выборе имени использовать наименьшее целое свободное число.

Так как ядро системы создает для этого порта дополнительную ссылку, ему необходимо также и инкрементировать счетчик объектов порта. Ядро производит эту операцию при копировании сообщения в системное адресное пространство, так как новая ссылка создается именно в этот момент. Отправитель также может указать в дескрипторе типа флаг освобождения. Тогда ядро системы освободит право на порт в задании-отправителю и не будет увеличивать счетчик ссылок на этот порт.

6.7.2. Внешняя память

Сообщение может содержать небольшой объем данных. В этом случае оно может быть передано путем физического копирования информации, сначала в буфер ядра, а затем (при запросе сообщения) — в адресное пространство задания-получателя. Однако такой подход является непрактичным при передаче больших объемов данных. Система Mach позволяет передавать в одном сообщении содержание всего адресного пространства (то есть до 4 Гбайт информации в 32-разрядных системах), поэтому необходимо средство, позволяющее вести передачу таких значительных объемов информации более эффективно.

Обычно при передаче данных большого объема большинство из них не изменяется ни отправителем, ни их получателем. В таких случаях нет нужды создавать отдельные копии данных. Страница памяти может дублироваться

только в том случае, если одно или оба задания попытаются внести в нее изменения. До этого момента оба задания будут разделять между собой одну и ту же физическую копию страницы. В системе Mach это реализовано при помощи метода «копирования при записи» подсистемы виртуальной памяти. В главе 15 управление памятью в ОС Mach будет описано более подробно. В этом разделе мы расскажем только об аспектах управления, относящихся к средствам IPC.

На рис. 6.12 показана передача внешней памяти (out-of-line). Отправитель указывает на использование этого типа памяти при помощи флага в *дескрипторе типа*. Процедура `msg_copyin()`, вызываемая из `msg_send`, изменяет карты памяти отправителя, делая передаваемые страницы доступными только для чтения и копируемыми при записи. Затем для этих страниц создается временная карта хранения в ядре, и они также получают атрибуты «только для чтения» и «копирования при записи» (рис. 6.12, *а*). Если получатель вызывает `msg_rcv`, то функция `msg_copyout()` получает диапазон адресов в его пространстве и копирует туда вхождения из карты хранения страниц. Она также маркирует эти новые вхождения как доступные только в режиме чтения и копируемые при записи. Затем происходит освобождение временной карты хранения (рис. 6.12, *б*).

С этого момента отправитель и получатель используют страницы совместно в режиме копирования при записи. Если какое-либо из этих заданий попытается изменить страницу с таким атрибутом, результатом будет возникновение ошибки. Обработчик этой ошибки распознает установки и решает проблему при помощи создания копии страницы и изменения карты адресации задания на эту копию. Обработчик также изменяет атрибуты, разрешая отправителю и получателю создавать собственные копии страницы (рис. 6.12, *в*).

Следует запомнить, что передача внешней памяти минует две стадии. На первой стадии сообщение помещается в очередь, а страницы находятся в стадии пересылки. Позже, при запросе сообщения, страницы становятся разделяемыми между отправителем и получателем. Карты хранения используются на стадии передачи, что гарантирует создание ядром системы новой копии страницы для отправителя, если он попытается внести в нее изменения до того, как страница будет запрошена получателем. В этом случае отправитель будет иметь доступ к ее оригинальной копии.

Описанное средство работает с максимальной производительностью, если ни отправитель, ни получатель не вносят изменения в совместно используемые страницы. Так функционируют многие приложения. *Внутренняя память копируется дважды: сначала от отправителя в ядро и затем из ядра получателю. Внешняя память копируется первый раз только тогда, когда одно из заданий попытается внести в нее изменения.*

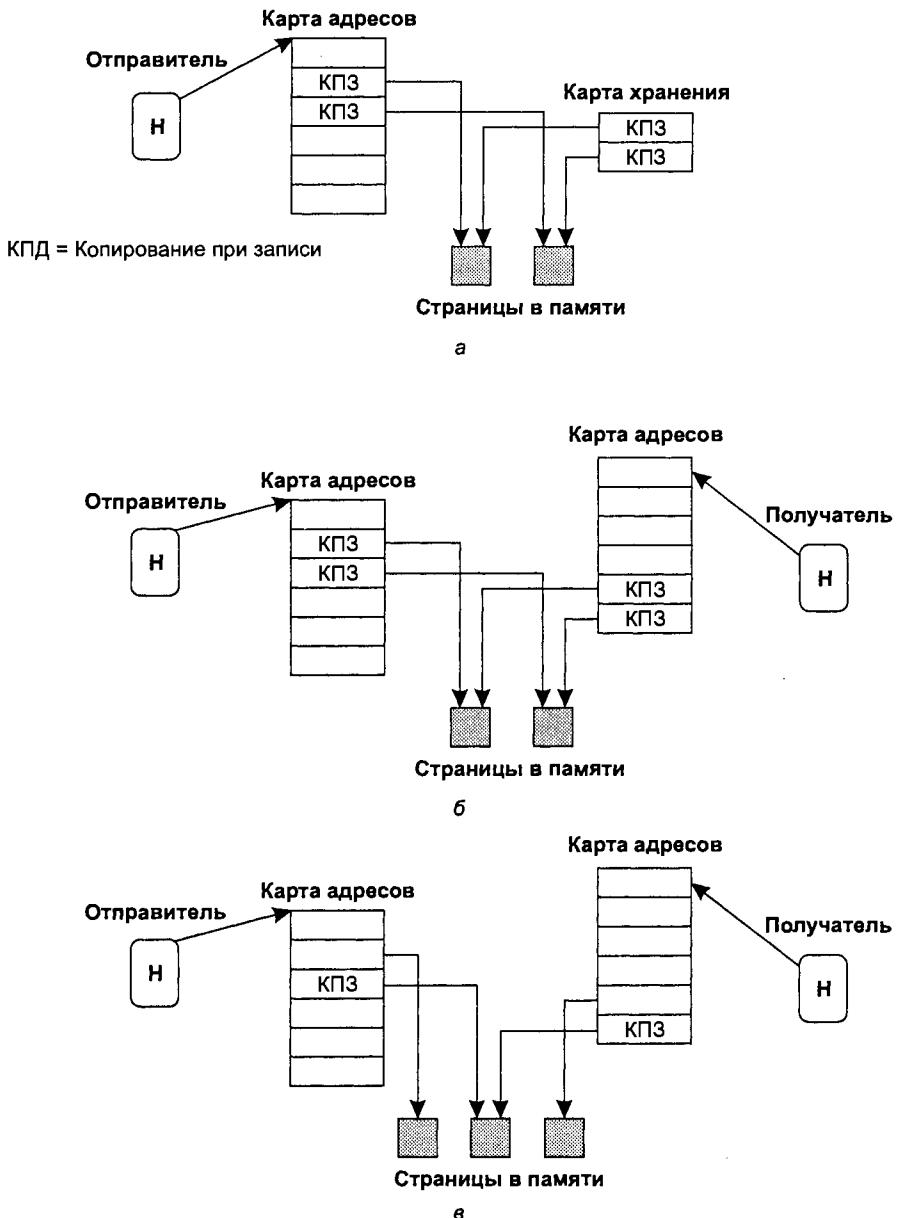


Рис. 6.12. Передача внешней памяти: а — сообщение копируется в карту хранения; б — сообщение копируется в задание-получатель; в — получатель производит изменения в странице

Отправитель может указывать в дескрипторе типа флаг освобождения. В этом случае ядро не использует методику копирования при чтении. Оно просто копирует вхождения карты адресации в карту хранения во время вы-

полнения процедуры `msg_copyin()` и удаляет их из карты адресации отправителя. При запросе сообщения функция `msg_copyout()` производит копирование вхождений в карту адресации получателя и удаляет временную карту хранения. В результате происходит перемещение страницы из адресного пространства отправителя в пространство получателя без какого-либо копирования данных.

6.7.3. Управление нитью

Передача сообщений происходит по двум возможным путям — медленному и быстрому. Медленный путь применяется в том случае, если получатель не ожидает сообщение в момент его отправления. В таком случае отправитель помещает сообщение в очередь и затем заканчивает свою часть работы. После того как получатель вызовет `msg_rcv`, ядро удалит сообщение из очереди и скопирует его в адресное пространство получателя.

Каждый порт обладает настраиваемым ограничением, называемым *журналом заказов* (*backlog*), в котором указывается максимальное количество сообщений, хранящихся в очереди. Если их число достигнет предельного значения, порт будет считаться заполненным, а все новые отправители будут блокированы до тех пор, пока из очереди не будет получено сколько-нибудь сообщений. Каждый раз, когда происходит запрос одного сообщения из порта, приостановившего работу отправителей, выполнение одного из них будет продолжено. Следовательно, если из порта будут выбраны все сообщения, то будут разбужены все блокированные отправители.

По быстрому пути передача происходит тогда, когда получатель уже ожидает поступления сообщения. В этом случае функция `msg_send` не будет помещать сообщение в очередь порта. Вместо этого она разбудит получателя и передаст ему сообщение напрямую. В системе Mach реализовано средство под названием *автоматического (hand-off) планирования* [8], при использовании которого одна нить может освобождать процессор непосредственно для другой определенной нити. Код быстрой отправки сообщений использует это средство для переключения на нить-получатель, которая завершает вызов `msg_rcv`, используя функцию `msg_copyout()` для копирования сообщения в свое адресное пространство. Такой подход предупреждает перегрузки, которые могут возникнуть при постановке сообщения в очередь и удалении из нее. Это также ускоряет контекстное переключение, так как новая выполняемая нить выбирается непосредственно.

6.7.4. Уведомления

Уведомления — это асинхронные сообщения, посылаемые ядром для информирования задания об определенных событиях в системе. Эти сообщения от-

правляются в *порт уведомления* заданий. В Mach IPC применяются три типа уведомлений:

<code>NOTIFY_PORT_DESTROYED</code>	Это сообщение посыпается владельцу резервного порта, если его основной порт был удален (в случае существования резервного порта). Удаление портов и резервные порты будут обсуждаться в следующем разделе
<code>NOTIFY_PORT_DELETED</code>	Рассыпается всем заданиям, обладающим правом отправки на порт при его удалении
<code>NOTIFY_MSG_ACCEPTED</code>	Уведомление может быть запрошено отправителем, если он посыпает сообщение в заполненную полностью очередь (используя опцию <code>SEND_NOTIFY</code>). Уведомление создается ядром после того, как сообщение удаляется из очереди

Последний тип уведомлений требует дополнительных уточнений. При использовании опции `SEND_NOTIFY` сообщение будет передано даже в том случае, если очередь будет заполнена полностью. Ядро возвращает статус `SEND_WILL_NOTIFY`, который указывает отправителю на необходимость приостановить передачу дальнейших сообщений до того, как им будет получено уведомление `NOTIFY_MSG_ACCEPTED`. Такой подход позволяет отправителю передавать сообщения без блокировки.

6.8. Операции порта

В этом разделе будут описаны некоторые операции, производимые над портами.

6.8.1. Удаление порта

Порт удаляется при освобождении прав на получение из него. Обычно это происходит при завершении работы задания, являющегося владельцем порта. Если порт необходимо аннулировать, то непосредственно после этой операции все задания, обладающие правами на отправку в этот порт, получают уведомление `NOTIFY_PORT_DELETED`. При удалении порта удаляются и все сообщения, находящиеся в его очереди, будятся все заблокированные отправители и получатели, им отправляются уведомления `SEND_INVALID_PORT` и `RCV_INVALID_PORT` соответственно.

Операция удаления порта является сложной процедурой, так как сообщения, находящиеся в его очереди, могут содержать права на другие порты. Если это права на получение, будут упразднены и порты, на которые они ссылаются. Фактически злоумышленник может отправить права на получение для определенного порта через сообщение на каждый порт. Такое случается нечасто, но если это происходит, результатом может стать возникновение взаимных блокировок и неограниченной рекурсии. К сожалению, в ОС Mach 2.5 эти проблемы так и не были решены.

6.8.2. Резервные порты

Системный вызов `port_set_backup` устанавливает *резервный порт* для указанного порта. Если порт, имеющий резерв, аннулируется, ядро системы не освобождает его, а передает все права на получение резервному порту.

Этот процесс показан на рис. 6.13. Порт П1 назначен в качестве резервного для порта П2. При удалении П1 (возможно, что это произошло из-за завершения работы задания, владеющего им) ядро системы посылает сообщение `NOTIFY_PORT_DESTROYED` владельцу П1. При этом порт П1 не освобождается и становится владельцем прав на отправку в порт П2. Все сообщения, адресуемые П1, автоматически перенаправляются на П2, откуда они могут быть затребованы его владельцем.

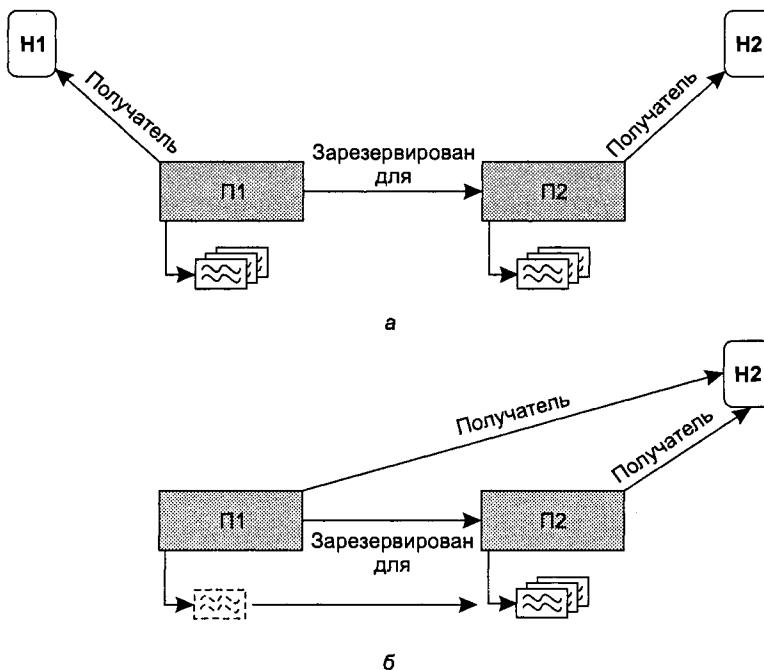


Рис. 6.13. Реализация резервных портов: а — назначение резервного порта, б — после удаления порта П1

6.8.3. Наборы портов

Набор портов состоит из группы портов, чьи индивидуальные права на получение заменены единственным групповым правом получения (рис. 6.14). Таким образом, задание может получать сообщения из набора портов. Более того, оно может получать их выборочно от определенных портов набора. В противоположность описанной возможности, сообщения могут отправляться толь-

ко на индивидуальные порты, а не на их наборы. Задания могут обладать правами отправки на определенные порты набора. При получении сообщения в нем будет содержаться информация о том порте, с которого оно было послано.

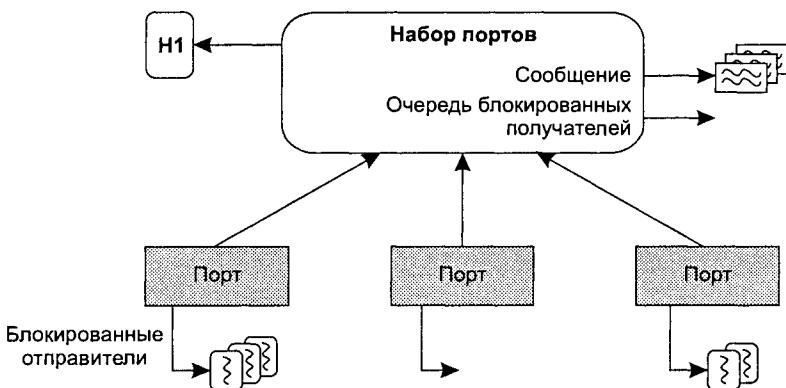


Рис. 6.14. Набор портов

Наборы портов удобны при обработке сервером сразу нескольких объектов. Серверы могут ассоциировать с каждым объектом отдельный порт, а затем объединить эти порты в набор. После этого сервер будет получать сообщения, отправленные на любой из портов набора. Каждое сообщение содержит в себе запрос, адресованный объекту, ассоциированному с портом. Так как в заголовке сообщения имеется информация о порте-получателе, сервер знает, с каким из объектов необходимо произвести манипуляции.

Возможности наборов портов сходны с вызовом системы UNIX `select`, позволяющим процессу проверить ввод на нескольких дескрипторах. Однако существует и важное отличие, заключающееся в том, что *время, потраченное на запрос сообщения из набора портов или на отправку сообщения в порт из набора, не зависит от общего количества портов в этом наборе*.

Объект ядра порта содержит указатель на набор, к которому относится этот порт, а также указатели на двунаправленный связанный список всех портов набора. Если порт не входит в набор, значения этих указателей равны `NULL`. Объект набора портов содержит единую очередь сообщений. Очереди входящих в набор портов не используются. Однако каждый порт продолжает поддерживать собственный счетчик и ограничение на количество сообщений, а также очередь блокированных отправителей.

Когда нить отправляет сообщение в порт, являющийся членом набора, ядро проверяет, не превышен ли лимит порта (если это так, отправитель будет добавлен в очередь блокированных отправителей порта) и увеличивает количество сообщений, находящихся в очереди. Затем ядро записывает имя порта в сообщение и переносит его в общую очередь набора портов. При вызове получателем функции `msg_rcv` ядро запрашивает первое сообщение из очереди набора независимо от конкретного порта, которому оно было адресовано.

6.8.4. Передача прав

Mach позволяет заданию произвести замену прав, относящихся к иному заданию, правами другого порта или получить право от другого задания. Такая возможность поручительства придала дополнительную гибкость управлению заданиями отладчиками и эмуляторами.

На рис. 6.15 показан возможный сценарий взаимодействия между отладчиком и отслеживаемым им заданием (жертвой). Сначала отладчик переносит право отправки жертвы на свой порт `task_self` при помощи вызова `task_extract_send` и забирает его себе. Затем он вызывает `task_insert_send` для передачи права отправки порту `P1`, владельцем которого является отладчик замен права `task_self` жертвы. Точно так же отладчик использует вызовы `task_extract_receive` и `task_insert_receive` для передачи права на получение порта жертвы на свой порт `task_self` и подстановки права получения на другой порт `P2`, на который у отладчика имеются права отправки.

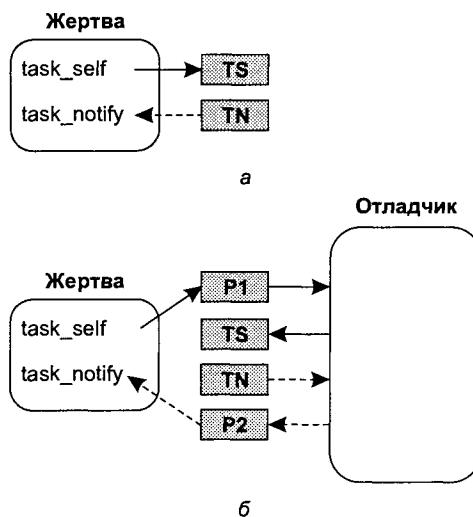


Рис. 6.15. Переназначение портов отладчиком: а — задача-жертва перед передачей прав; б — после передачи прав порта

После завершения этих действий ядро перехватывает все сообщения, отправляемые управляемым им заданием на порт `task_notify` (то есть все системные вызовы Mach). Отладчик обрабатывает вызов и проверяет отправку ответа на порт ответа, указанного в исходном сообщении. Отладчик может имитировать вызов и отправить ответ самостоятельно. Вместе с тем возможен и более стандартный ход: обработчик пересыпает сообщение ядру при помощи изначально установленного порта `task_notify` задания (на который отладчик обладает правами отправки). При пересылке сообщения ядру ответ может быть послан напрямую заданию или на порт, указанный обработчиком. В последнем случае произойдет перехват ответа ядра.

С другой стороны, отладчик в состоянии перехватывать и все уведомления, передаваемые управляемому заданию, после чего он принимает решение о самостоятельной обработке уведомления или передаче его заданию. Из вышесказанного следует, что обработчик (или иное другое задание) может управлять заданием, если он передаст свои права на получение из порта `task_self` управляемого им задания.

6.9. Расширяемость

Средства IPC системы Mach разрабатывались с учетом возможности прозрачного расширения на распределенные среды. Для расширения Mach IPC на сеть используется программа пользовательского уровня `netmsgserver`, дающая возможность взаимодействия с заданиями на удаленных машинах точно так же, как будто эти задания выполняются на локальном компьютере. Приложения ничего не знают об удаленном соединении, так как они продолжают использовать тот же интерфейс и набор вызовов, применяемых для локальных коммуникаций. Приложение обычно не обладает информацией о том, является ли задание, с которым оно взаимодействует, локальным или удаленным.

В системе Mach имеются две очень важные особенности, позволившие прозрачно расширить возможности IPC на сеть. Во-первых, права портов используют независимое от местонахождения пространство имен. Отправитель посыпает сообщение на локальное имя порта (право отправки), при этом он не знает, какой из объектов представлен этим портом — локальный или удаленный. Карты преобразований локальных имен портов в объект ядра поддерживаются на уровне ядра системы.

Во-вторых, отправители являются анонимными. Сообщения их не идентифицируют. Отправитель вправе переслать право на отправку в порт ответа прямо в сообщении. Ядро преобразует это таким образом, что получателю будет видно только локальное имя, действительное только для данного задания. Это означает, что получатель не может определить, кто является отправителем сообщения. Более того, получатель даже не обязан быть владельцем права на порт ответа, ему достаточно обладать правом на отправку в этот порт. Указывая порт ответа, владельцем которого является другое задание, отправитель перенаправляет ответ на это задание. Перечисленные возможности также могут применяться отладчиками, эмуляторами и другими программами.

Функции `netmsgserver` достаточно просты. Типичный сценарий работы программы показан на рис. 6.16. Программа `netmsgserver` загружается на каждой машине в сети. Если клиенту на узле А необходимо установить соединение с сервером на сервере Б, программа `netmsgserver` на узле А установит порт-посредник, на который отправитель будет отсылать сообщения. Затем она запросит сообщения, посланные на этот порт, и перешлет их программе `netmsgserver` узла Б, которая в свою очередь передаст их на порт сервера.

Если клиент ожидает ответа, он указывает в исходном сообщении порт ответа. Программа netmsgserver на узле А получит права отправки в ответный порт. После этого netmsgserver на узле Б создаст порт-посредник для ответного порта и отправит право на порт-посредник сервера. Сервер передает ответное сообщение на свой порт-посредник, которое далее будет передано через два netmsgserver (сервера и узла) на порт ответа клиента.

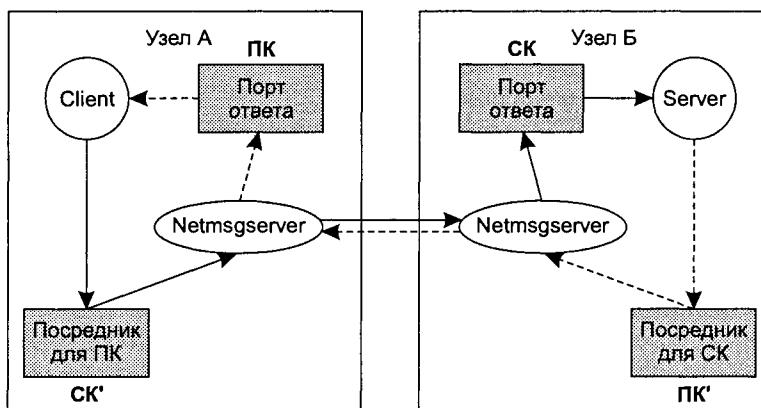


Рис. 6.16. Удаленные коммуникации с использованием программ netmsgserver

Серверы регистрируют себя в локальной программе netmsgserver и затем передают ей право отправки на порт, прослушиваемый сервером. Программа поддерживает распределенную базу данных таких зарегистрированных сетевых портов и предоставляет для них набор служб (защиту, уведомления и т. д.), сходный с набором, выделяемым ядром для локальных портов машины. Таким образом, программа поддерживает глобальную службу поиска для всех имен портов. Задания используют эту службу для получения прав отправки на порты, зарегистрированные удаленными программами netmsgserver. В отсутствии сети программа работает как локальная служба имен.

Программы netmsgserver взаимодействуют между собой при помощи сетевых протоколов низкого уровня, не применяя при этом сообщений IPC.

6.10. Новые возможности Mach 3.0

В системе Mach 3.0 были расширены некоторые возможности средств IPC [7]. Были преодолены проблемы, возникавшие в ОС версии 2.5. Изменения коснулись как интерфейсной части, так и внутренних структур данных и используемых алгоритмов. В этом разделе показаны основные усовершенствования подсистемы IPC.

Главной проблемой Mach 2.5 является передача прав. Средства IPC тесно интегрируются с технологией «клиент-сервер». Обычно клиент посыпает сооб-

щение, содержащее право отправки в порт ответа, владельцем которого он является. После генерации ответа серверу нет необходимости далее поддерживать посланное ему право. С другой стороны, сервер не может после обработки запроса освободить порт, так как существует вероятность использования этого порта другой нитью серверного приложения, обрабатывающей одновременно другой запрос от того же задания. Ядро преобразует одно и то же право отправки для следующих нитей в одно и то же имя, следовательно, сервер обладает единым правом отправки на указанный порт. Если нить, обработавшая первый запрос, освободит право, то вторая нить уже не сможет отправить на этот порт ответное сообщение.

Означенная проблема решена серверами просто: они никогда не освобождают присланные им права отправки. Однако это приводит к необходимости решения уже новых проблем. Во-первых, появляется определенный риск безопасности работы: клиент может не желать, чтобы сервер постоянно имел право на отправку сообщений в его порт. Также существует проблема усложнения обработки «раздутых» таблиц преобразований, что влияет на общую производительность системы. И наконец, при удалении порта клиентом серверу приходится обрабатывать не нужные ему уведомления об этом событии. Ядро обязано разослать такие уведомления всем серверам, которые имеют права на отправку в удаленный порт, серверы, в свою очередь, должны получить и обработать их, даже если судьба порта их не интересует.

В системе Mach 3.0 введены новые средства, позволяющие решить эту проблему: права на однократную отправку, запросы уведомлений и пользовательские ссылки на права отправки.

6.10.1. Права на однократную отправку

Право на однократную отправку в порт — это, как и следует из его названия, право на отправку, которое может быть использовано только один раз. Такое право создается заданием-владельцем права на получение из порта и в дальнейшем может передаваться между заданиями. Оно дает гарантию на получение ответного сообщения именно из указанного в нем порта. Обычно такое право применяется для использования порта в качестве получателя ответного сообщения и отзывается сразу после передачи на него ответа. Если право аннулируется вследствие других причин (например, при завершении работы его владельца), то вместо ответа ядром будет отослано однократное уведомление.

В системе Mach права на однократную отправку используются для указания портов ответа. Такое право удаляется сразу после получения ответного сообщения, следовательно, теперь сервер не продолжает удерживать не нужные ему права. Такой подход защищает от необходимости обработки излишних уведомлений, создаваемых при удалении порта клиентом.

6.10.2. Уведомления в Mach 3.0

В системе Mach 2.5 уведомления отправлялись заданиями асинхронно в ответ на различные системные события. Задания не могли контролировать процесс получения таких уведомлений. В большинстве случаев информация о произошедших событиях не нужна заданиям, поэтому уведомления просто уничтожаются. Однако существование таких излишних уведомлений влияет на общую производительность системы.

Более того, использование единого порта для уведомления всех нитей задания приводит к определенным ограничениям. Одна из нитей может перехватить и изъять уведомление, ожидаемое другой нитью задания. Применение пользовательских библиотек увеличило вероятность такого варианта развития событий, так как основные программы и библиотеки используют уведомления независимо друг от друга и обрабатывают их различными способами.

В системе Mach 3.0 уведомления отправляются только тем заданиям, которые запросили их при помощи вызова `mach_port_request_notification`. При этом запросе используется право на однократную отправку в порт уведомлений. Таким образом, в ОС Mach 3.0 появилась возможность использования нескольких портов уведомлений внутри одного задания. Каждый программный компонент или нить может теперь указать собственный порт уведомлений, защитив себя тем самым от вышеуказанных проблем.

6.10.3. Подсчет прав на отправку

Если задание дает несколько прав отправки для одного порта, ядро использует для всех них единое имя в пространстве имен портов, комбинируя тем самым все права в единое право на отправку. Если одна из нитей освободит право, ни одна другая нить больше не сможет воспользоваться им, даже если эта нить затребовала данное право ранее самостоятельно.

Применение прав на однократную отправку помогло решить проблему, но только частично. Этого рода права используются по большей мере для задания портов ответа серверам. Права на отправку можно получить несколькими способами. Если клиенту необходимо начать взаимодействие с сервером, он запросит право на отправку к нему сообщений (через сервер имен). Если несколько нитей клиента независимо друг от друга инициализируют общение с сервером, каждая из них получит право на получение, которое затем будет объединено одним общим именем. Если одна из нитей освободит это имя, то это действие повлияет и на все другие нити: никто из них не сможет в дальнейшем пользоваться портом.

В системе Mach 3.0 эта проблема была решена при помощи ассоциации пользовательской ссылки с каждым правом отправки. Теперь ядро увеличивает значение счетчика ссылок на единицу при каждом запросе заданием одного и того же права и точно так же уменьшает его при каждом освобождении права. После освобождения последней ссылки ядро удаляет право.

6.11. Дискуссия о средствах Mach IPC

В системе Mach возможности IPC являются не только средствами осуществления коммуникаций между процессами, но и фундаментальным компонентом структуры ядра. Например, подсистема виртуальной памяти использует их для реализации технологии копирования при записи [15], а ядро применяет IPC для управления заданиями и нитями. Основные объекты системы Mach, такие как задания, нити и порты, используют для взаимодействия друг с другом средства передачи сообщений.

Архитектура ОС Mach имеет интересные свойства. Средства IPC построены таким образом, что при помощи программы `netmsgserver` можно расширить возможности взаимодействия между процессами на распределенные системы, позволяя заданиям управлять объектами на удаленных узлах. Это свойство системы Mach позволило реализовать такие средства, как удаленная отладка, распределенная разделяемая память и другие клиент-серверные программы.

Однако если посмотреть на архитектуру системы с другой точки зрения, мы увидим, что интенсивная передача сообщений ведет к ухудшению производительности. Разработчики направили свои усилия на построение операционных систем с микроядром, в которых большинство возможностей реализовывалось на уровне пользовательских серверных заданий, взаимодействующих друг с другом при помощи IPC. Над созданием подобных систем трудились многие производители, но проблемы производительности практически свели на нет их усилия: такие системы не получили широкого распространения.

Сторонники архитектуры Mach считали, что производительность IPC не является важным фактором, который следует рассматривать при разработке ОС с микроядром [4]. Они объясняли это следующими причинами:

- ◆ достижения в улучшении производительности IPC и так велики по сравнению с другими компонентами операционной системы;
- ◆ с увеличением доверия к аппаратным кэшам стоимость системных служб будет сильно зависеть от их емкостей. Так как код подсистемы IPC является хорошо локализированным, он может быть легко настроен для оптимального использования кэш-памяти;
- ◆ некоторый объем передачи данных можно осуществить другими путями, например, используя механизмы разделяемой памяти;
- ◆ перевод некоторых возможностей ядра на уровень серверов пользовательского уровня позволил уменьшить количество переключений режима и выходов за границы защиты, являющихся весьма затратными операциями.

Разработчики потратили много усилий на увеличение производительности средств IPC [3], [4]. Но несмотря на это подсистема IPC получила лишь ограниченное распространение среди коммерческих ОС. Даже в системе Digital UNIX, разработанной на основе Mach, в большинстве компонентов ядра средства IPC Mach не применяются.

6.12. Заключение

В этой главе описывалось сразу несколько вариантов механизмов взаимодействия процессов (IPC). Сигналы, каналы и трассировка процессов — это универсальные средства, поддерживаемые всеми ранними версиями системы UNIX. В наборе System V IPC были представлены новые возможности, такие как разделяемая память, семафоры и очереди сообщений, которые после появились во всех современных вариантах UNIX. В ОС Mach средства IPC используются для взаимодействия всех объектов ядра друг с другом. Границы взаимодействия объектов Mach могут быть легко расширены при помощи программы *netmsgserver*, дающей возможность построения распределенных клиент-серверных приложений.

Некоторые другие средства IPC будут описаны в книге позже: в главе 8 вы увидите рассказ о защите файлов, в главе 14 прочтете о файлах, отображаемых в память, а глава 17 расскажет о каналах библиотеки STREAMS.

6.13. Упражнения

1. Назовите ограничения *ptrace* как инструмента, используемого для создания отладчиков.
2. Аргумент *pid* функции *ptrace* должен указывать на идентификатор процесса, являющегося потомком вызвавшего ее. Зачем необходимо выполнение этого требования? Почему процессы не могут использовать *ptrace* для взаимодействия с любыми другими процессами системы?
3. Сравните коммуникационные возможности, предоставляемые каналами и очередями сообщений. Какими преимуществами и недостатками обладает каждое из упомянутых средств? В каких случаях предпочтительнее использовать каналы, а в каких — очереди сообщений?
4. Во многих системах UNIX возможно подключение одной и той же разделяемой области памяти сразу на несколько адресов в адресном пространстве. Это недоработка или, наоборот, преимущество систем? В каких случаях описанная возможность может быть использована? К возникновению каких проблем это может привести?
5. О чём должен помнить программист при выборе адреса для подключения к нему разделяемой области памяти? От каких ошибок способна при этом защитить сама операционная система?
6. Каким образом можно задействовать флаг *IPC_NOWAIT* для предупреждения возникновения взаимных блокировок при использовании семафоров?
7. Создайте программы, позволяющие взаимодействующим процессам эксклюзивно использовать ресурс:
 - на основе канала;
 - FIFO;

- системного вызова `mkdir`;
- системных вызовов `flock` и `lockf`.

Сравните созданные программы и их производительность.

8. С какими возможными побочными эффектами может столкнуться программист при создании всех вариантов, предложенных в предыдущем вопросе?

9. Можно ли реализовать блокировку ресурса, используя:

- только сигналы;
- сигналы и разделяемую память?

Какой производительностью будут обладать эти средства?

10. Создайте программы, передающие большой объем данных между процессами, используя

- каналы;
- FIFO;
- очереди сообщений;
- разделяемую память совместно с семафорами (для синхронизации).

Сравните созданные программы и их производительность.

11. Какие проблемы безопасности существуют в средствах IPC System V? Каким образом может вредоносная программа вмешаться во взаимодействие процессов или перехватить информацию?

12. Семафоры создаются при помощи `semget`, но для их инициализации необходимо применить вызов `semctl`. Следовательно, инициализация и создание семафора не могут быть проведены как единая неделимая операция. Опишите ситуацию, при которой такой подход может привести к состязательности. Предложите способ решения создавшейся проблемы.

13. Можно ли реализовать очереди сообщений System V на основе Mach IPC? Какие проблемы могла бы решить такая реализация?

14. По каким причинам может быть удобно применение прав на однократную отправку?

15. Каким образом наборы портов помогают разработчикам в создании клиент-серверных приложений для ОС Mach?

6.14. Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid R. F., Tevanian, A., Jr., and Young, M. W., «Mach Kernel Interface

- Manual», Department of Computer Science, Carnegie-Mellon University, Jan. 1990.
3. Barrera, J. S., «A Fast Mach Network IPC Implementation», Proceedings of the USENIX Mach Symposium, Nov. 1991, pp. 1–12.
 4. Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M., «Light-weight Remote Procedure Call», ACM Transactions on Computer Systems, Vol. 8, No 1, Feb. 1990, pp. 37–55.
 5. Bershad, B. N., «The Increasing Irrelevance of IPC Performance for Micro-kernel-Based Operating Systems», USENIX Workshop on Micro-Kernels and Other Kernel Architectures, Apr. 1992, pp. 205–212.
 6. Dijkstra, E. W., «Solution of a Problem in Concurrent Programming Control», Communications of the ACM, Vol. 8, Sep. 1965, pp. 569–578.
 7. Draves, R. P., «A Revised IPC Interface», Proceedings of the First Mach USENIX Workshop, Oct. 1990, pp. 101–121.
 8. Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W., «Using Continuations Implement Thread Management and Communication in Operating Systems», Technical Report CMU-CS-91-115R, School of Computer Science, Carnegie Mellon University, Oct. 1991.
 9. Faulkner, R. and Gomes, R., «The Process File System and Process Model in UNIX System V», Proceedings of the 1991 Winter USENIX Conference, Jan. 1991, pp. 243–252.
 10. Presotto, D. L., and Ritchie, D. M., «Interprocess Communications in the Ninth Edition UNIX System», UNIX Research System Papers, Tenth Edition, Vol. II, Saunders College Publishing, 1990, pp. 523–530.
 11. Rashid, R. F., «Threads of a New System», UNIX Review, Aug. 1986, pp. 37–49.
 12. Salus, P. H., «A Quarter Century of UNIX», Addison-Wesley, Reading, MA, 1994.
 13. Stevens, R. W., «UNIX Network Programming», Prentice-Hall, Englewood Cliffs, NJ, 1990.
 14. Thompson, K., «UNIX Implementation», The Bell System Technical Journal, Vol. 57, No. 6, Part 2, Jul.–Aug. 1978, pp. 1931–1946.
 15. Young, M., Tevanian, A., Rashid, R. F., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., «The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System», Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 63–76.

Глава 7

Синхронизация.

Многопроцессорные

системы

7.1. Введение

Желание увеличить производительность компьютерных систем привело к появлению новых аппаратных архитектур. Основным направлением развития вычислительной техники стало создание многопроцессорных систем. Такие системы могут содержать от двух и более центральных процессоров, при этом остальные ресурсы, такие как память, используются ими совместно. Многопроцессорные технологии обладают некоторыми бесспорными преимуществами. Они позволяют гибко расширять объем ресурсов, используемых программой, которая может начать выполнение на одном процессоре и в дальнейшем, при увеличении объема производимых вычислений, использовать другие процессы системы. Большинство вычислительных задач зависят, прежде всего, от производительности CPU. Для таких задач мощность процессора (или процессоров) является самым критичным параметром, в отличие от работы с шиной ввода-вывода или памятью, используемых менее интенсивно. Применение многопроцессорных систем позволяет увеличивать вычислительные мощности без дублирования остальных ресурсов. Следовательно, такое решение для вычислительных задач является самым эффективным по себестоимости.

Многопроцессорные системы обладают повышенной надежностью: если на одном из процессоров произойдет сбой, система сможет продолжать работу дальше. Но, с другой стороны, многопроцессорные архитектуры таят в себе и определенные проблемы, связанные с большим количеством потенциально опасных точек сбоев. Чтобы гарантировать определенное количество *среднего времени наработка на отказ* (mean time before failure, MTBF), многопроцессорные системы должны разрабатываться на основе отказоустойчивого оборудования и программного обеспечения. В частности, многопроцессорные системы должны уметь восстанавливаться после сбоя одного из процессоров.

Существуют различные операционные системы, созданные с учетом многопроцессорных архитектур. Одна из первых многопроцессорных реализаций UNIX работала на машинах AT&T 3B20A и IBM 370 [1]. На сегодняшний день большинство вариантов систем UNIX поддерживают многопроцессорную обработку изначально (DEC UNIX, Solaris 2.x), либо для этой цели имеются специализированные версии ОС (SVR4/MP, SCO/MPX).

Теоретически производительность системы должна увеличиваться пропорционально умножению числа процессоров. Однако в реальности все происходит немного иначе. На это существует несколько причин. Так как остальные компоненты системы не дублируются, именно они и становятся узким местом, тормозящим ее работу. Доступ к разделяемым структурам данных требует определенной синхронизации. Для многопроцессорных систем необходимо обеспечивать синхронизацию при помощи дополнительных средств, увеличивающих загрузку процессоров и уменьшающих тем самым общую производительность. Операционная система обязана по возможности уметь минимизировать такие перегрузки и оптимизировать использование ресурсов CPU.

Ядро традиционных вариантов UNIX разрабатывалось для однопроцессорных машин. Для того, чтобы такие системы работали на многопроцессорных архитектурах, требуется внести в дизайн их ядра значительные изменения. Необходимо заново переписать такие компоненты, как реализация синхронизации, параллельного выполнения задач и правил планирования. Синхронизация используется для управления доступом к разделяемым данным и ресурсам системы. Традиционный подход, основанный на приостановке и возобновлении работы процессов, совмещенный с блокировкой прерываний, является непригодным для многопроцессорных сред и должен быть заменен более производительными методиками.

Параллельность позволяет эффективно использовать объекты синхронизации для управления доступом к разделяемым ресурсам. Эта технология предлагает решения, позволяющие регулировать степень детализации и место блокировок, защищать от возникновения взаимоблокировок и т. д. Некоторые из перечисленных проблем будут описаны в разделе 7.10. Для оптимизации использования всех процессоров системы необходимо разработать новые правила планирования. Некоторые аспекты реализации планирования в многопроцессорных системах будут обсуждаться в разделе 7.4.

Эта глава начинается с описания механизмов синхронизации традиционных систем UNIX и анализа их ограничений. Затем вы познакомитесь со многопроцессорными архитектурами. В конце главы приведено описание технологий синхронизации, применяемых в современных вариантах UNIX. Эти методы работают одинаково хорошо как на однопроцессорных, так и на многопроцессорных системах.

В традиционных вариантах UNIX базовым объектом диспетчеризации является процесс, который обладает единственным управляемой нитью. В главе 3 описывались современные системы UNIX, которые позволяют процессу иметь

несколько нитей выполнения, поддерживаемых на уровне ядра. Многонитевые системы существуют как для однопроцессорных, так и для многопроцессорных архитектур. В таких системах нити соревнуются за обладание разделяемыми ресурсами и блокируют их. В дальнейшем (до окончания главы) мы будем считать нить базовой единицей планирования. Для однонитевых систем нить является синонимом процесса.

7.2. Синхронизация в ядре традиционных реализаций UNIX

Ядро UNIX является реентерабельным. Это означает, что в одно и тоже время в ядре системы могут работать сразу несколько процессов, иногда выполняя при этом одинаковые задачи. На однопроцессорных системах в один момент времени может выполняться только один процесс. Система постоянно переключается от выполнения одного процесса к другому, создавая иллюзию параллельного функционирования. Такая возможность получила название *многозадачной работы*. Так как все процессы используют единое ядро, ему необходимо как-то синхронизировать доступ к своим структурам данных для предупреждения возможности их повреждения. В разделе 2.5 подробно рассказывалось о технологиях синхронизации, применяемых в традиционных системах UNIX. В этой главе мы подведем краткие итоги.

Первым защитным механизмом традиционного ядра UNIX является его невытесняемость. Любая нить будет продолжать работу в режиме ядра до тех пор, пока сама не будет готова освободить его либо приостановить выполнение в ожидании ресурса, даже в том случае, если эта нить исчерпала выделенный ей квант времени. Такой подход дает возможность кодам ядра управлять различными структурами данных без обеспечения какой-либо защиты содержащейся в них информации, поскольку заранее известно, что никакая другая нить не сможет получить доступ к ним до тех пор, пока текущая нить не закончит манипуляции данными и не переведет ядро в непротиворечивое состояние.

7.2.1. Блокировка прерываний

Непрерываемость ядра традиционных версий UNIX является свойством, обеспечивающим мощный инструментарий синхронизации, но, к сожалению, она обладает и некоторыми ограничениями. Текущую выполняемую нить невозможно вытеснить, но ее работу можно прервать. Прерывания являются внутренними событиями системы и должны обрабатываться как можно быстрее. Обработчик прерывания в состоянии манипулировать структурами данных, принадлежащих текущей нити, что может стать причиной их поврежде-

ния. Следовательно, ядро системы должно как-то синхронизировать доступ к данным, используемым его подпрограммами и обработчиками прерываний.

В системе UNIX вышеописанная проблема решена при помощи блокировки (или маскирования) прерываний. Каждому прерыванию присваивается *уровень приоритета* (interrupt priority level, *ipl*). Система поддерживает текущий уровень *ipl* и проверяет его после возникновения прерывания. Если приоритет окажется выше текущего значения, то такое прерывание будет обработано немедленно (то есть будет приостановлена обработка прерывания с более низким уровнем *ipl*). В противоположном случае ядро заблокирует прерывание до тех пор, пока текущий уровень *ipl* не снизится до необходимого уровня. Сразу перед загрузкой обработчика ядро системы устанавливает *ipl* в значение текущего прерывания, после завершения его обработки ядро восстанавливает предыдущее сохраненное значение. Ядро обладает возможностью установить значение *ipl* в принудительном порядке с целью временной блокировки прерываний в течение выполнения критических участков кода.

Например, одной из процедур ядра необходимо удалить дисковый буфер из очереди буферов. Эта очередь доступна и обработчику прерываний диска. Участок кода, производящий действия с очередью, является *критическим*. До начала его выполнения процедура поднимает уровень *ipl* до значения, достаточного для блокировки дисковых прерываний. После завершения действий над очередью процедура восстановит предыдущее значение *ipl*, разрешая тем самым обработку дисковых прерываний. Поддержка уровней *ipl* позволяет эффективно решить проблему синхронизации ресурсов, используемых совместно ядром и обработчиками прерываний.

7.2.2. Приостановка и пробуждение

Во многих случаях нить должна использовать какой-либо ресурс системы эксклюзивно даже в том случае, если она приостанавливает выполнение. Например, нити необходимо прочесть блок информации с диска и записать ее в дисковый буфер. Для этого нить запрашивает буфер и начинает производить действия с диском. Ей необходимо дождаться завершения ввода-вывода, что означает возможность освобождения процессора на некоторое время для другой нити. Если новая текущая нить попытается получить доступ к тому же буферу и использовать его для своих целей, результатом может стать изменение или повреждение данных, находящихся в нем. Это означает, что система должна поддерживать некие методики защиты блокированных ресурсов.

В UNIX защита ресурсов реализована при помощи флагов *locked* (ресурс занят) и *wanted* (ресурс необходим). Если нити необходимо получить доступ к разделяемому ресурсу (например, буферу), то в первую очередь ей нужно проверить состояние флага *locked*. Если флаг не установлен, нить установит

его и начнет манипуляции с ресурсом. Если другая нить попытается получить доступ к тому же ресурсу, она обнаружит флаг *locked* и приостановит работу («заснет») до тех пор, пока ресурс не станет доступным. Перед блокировкой нить установит для ресурса флаг *wanted*. Приостановка работы нити означает помещение ее в очередь спящих нитей и изменение информации о ее статусе, оповещающей о приостановке работы в ожидании определенного ресурса. После этого нить освобождает процессор и передает его следующей работоспособной нити системы.

Через некоторое время первая нить закончит манипуляции с ресурсом, сбросит флаг *locked* и проверит флаг *wanted*. Если флаг окажется установленным, то это означает, что по крайней мере еще одна нить находится в режиме ожидания того же ресурса. В этом случае текущая нить проверит очередь спящих нитей и разбудит все нити, ожидающие освобожденный ею ресурс. Пробуждение нити приводит к удалению ее из очереди спящих нитей, изменению ее статуса в состояние «работоспособная» и помещению ее в очередь планировщика. В какой-то момент времени нить станет текущей. Первым ее действием при получении процессорного времени будет являться проверка флага *locked*. Если флаг не установлен, нить сможет продолжить манипуляции с необходимым ей ресурсом.

7.2.3. Ограничения традиционного ядра UNIX

Традиционная модель синхронизации корректно работает на однопроцессорных машинах, но она, к сожалению, не избавлена от проблем, связанных с производительностью системы. В многопроцессорных системах использование традиционной модели невозможно, что и будет более подробно показано в разделе 7.4.

Блокировка ресурсов и очереди приостановленных нитей

В некоторых ситуациях организация очередей приостановленных нитей может являться причиной заметного ухудшения производительности системы. В системах UNIX нить блокируется при ожидании освобождения ресурса или возникновении определенного события. Каждый ресурс или событие ассоциированы с каналом ожидания, представляющим собой 32-разрядную переменную, обычно указывающую на адрес ресурса. Система поддерживает набор очередей ожидания, соединение канала с одной из очередей реализовано при помощи хэширования (рис. 7.1). Нить приостанавливает работу, помещая себя в одну из очередей ожидания, при этом ссылка на соответствующий канал ожидания сохраняется в структуре *proc*.

Применение такой методики блокировки приводит к двум последствиям. Во-первых, на каждый канал может ссылаться более чем одно событие. Например, одна из нитей может заблокировать буфер, инициализировать проведение над ним определенных действий и приостановить работу до тех пор,

пока эти действия не будут закончены. Если другая нить попытается получить доступ к тому же буферу, то обнаружит его заблокированным. Следовательно, ей придется приостановить выполнение в ожидании освобождения ресурса. Оба события ссылаются на один и тот же канал, указывающий на необходимый обеим нитям буфер. После завершения процедур ввода-вывода обработчик прерываний разбудит обе нити, несмотря на то, что последняя из них ожидает еще не произошедшее событие.

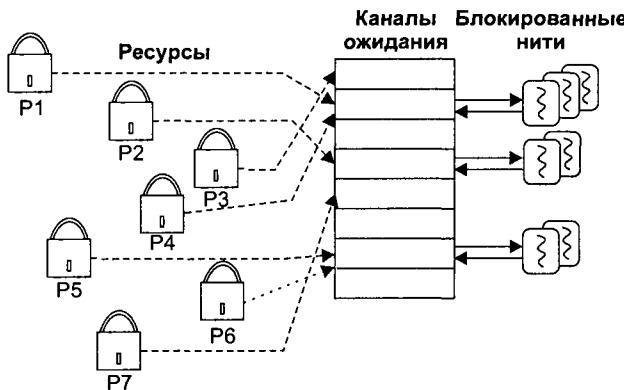


Рис. 7.1. Отображение ресурсов в глобальных очередях приостановленных нитей

Во-вторых, количество структур хэширования обычно меньше, чем различных каналов ожидания (то есть ресурсов и событий), следовательно, несколько каналов могут ссылаться на одно и то же место. Таким образом, оказаться хэшированными могут нити, ожидающие нескольких различных каналов. Процедуре `wakeup()` необходимо проверять каждый из них и делать работоспособными только те нити, которые ожидают определенный ресурс. Общее время работы `wakeup()` зависит не от количества процессов, находящихся в режиме ожидания данного канала, а от общего количества хэшированных процессов. Это приводит к непредусмотренным задержкам, появление которых нежелательно для систем, поддерживающих приложения реального времени, требующих соблюдения определенных верхних границ задержек планировщика (см. раздел 5.5.4).

Одним из решений проблемы является ассоциация отдельной очереди ожидания каждому ресурсу или событию (рис. 7.2), что позволит оптимизировать задержки при пробуждении процессов, но приведет к необходимости выделения большего объема памяти для хранения большего количества очередей. Обычный заголовок очереди наряду с другой информацией имеет два указателя (на предыдущую и следующую очередь). Общее количество объектов синхронизации может оказаться слишком большим, так что поддержка отдельной очереди ожидания для каждого из них является неприемлемой.

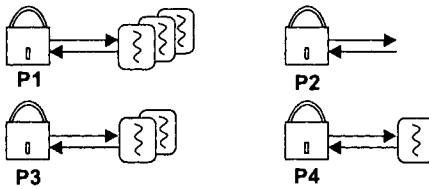


Рис. 7.2. Отдельные очереди для каждого ресурса системы

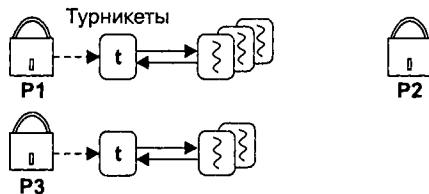


Рис. 7.3. Блокировка нитей с применением турникетов

В системе Solaris 2.x представлено более эффективное решение [7]: каждый объект синхронизации поддерживает двухразрядное поле, указывающее на структуру турникета, которая содержит очередь ожидания и другую необходимую информацию (рис. 7.3). Ядро выделяет турникеты только для тех ресурсов, которые ожидают заблокированные нити. Для ускорения процедуры выделения ядро поддерживает набор турникетов, размер которого больше, чем общее количество активных нитей в системе. Такой подход удобен для приложений реального времени, так как дает минимальные задержки. Более подробно технология синхронизации системы Solaris 2.x описана в разделе 5.6.7.

Разделяемый и эксклюзивный доступ

Механизм приостановки и возобновления выполнения является удобным в том случае, если в один момент времени ресурс используется только одной нитью. Однако он не позволяет использовать более сложные протоколы, такие как синхронизация читающих-пишущих нитей. Иногда необходимо предоставить ресурс сразу нескольким нитям на чтение, но потребовать для его изменения эксклюзивных полномочий. Некоторые ресурсы, такие как файлы или каталоги, будут использоваться более эффективно при применении этой технологии.

7.3. Многопроцессорные системы

Многопроцессорные системы обладают тремя важнейшими свойствами. Во-первых, это модель памяти, определяющая способ ее разделения между всеми процессорами. Второй отличительной особенностью является аппаратная поддержка синхронизации. Под третьим свойством подразумевается программная архитектура, определяющая взаимодействие процессоров, подсистем ядра и пользовательских процессов.

7.3.1. Модель памяти

С точки зрения аппаратуры многопроцессорные системы можно разделить на три категории в зависимости от доступа к памяти и взаимосвязи компонентов (рис. 7.4):

- ◆ с унифицированным доступом к памяти (Uniform Memory Access, UMA);
- ◆ с неунифицированным доступом к памяти (Non-Uniform Memory Access, NUMA);
- ◆ с невозможностью доступа к удаленной памяти (No Remote Memory Access, NORMA).

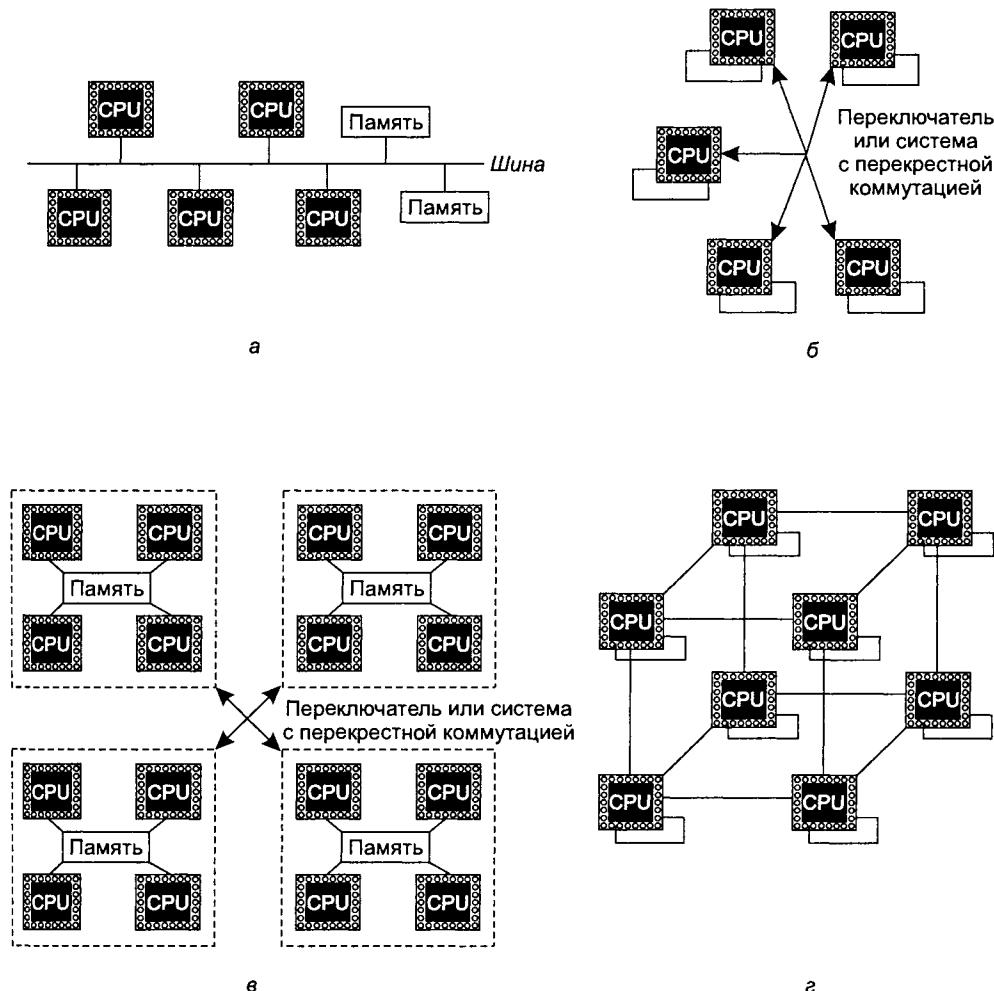


Рис. 7.4. Категории систем: а — UMA, б — NUMA, в — гибридная архитектура NUMA, г — NORMA

Наиболее распространенные системы основаны на методе UMA или разделяемой памяти (рис. 7.4, а). В таких системах все процессоры обладают равнозначным доступом к основной памяти¹ и устройствам ввода-вывода, которые обычно соединяются между собой единой шиной. С точки зрения разработки операционных систем эта модель является наиболее простой. Ее основным недостатком является масштабируемость. Архитектура UMA поддерживает небольшое количество процессоров. При увеличении числа CPU возникают конфликты в общей для всехшине. Одной из самых крупных систем, основанных на UMA, является SGI Challenge, поддерживающая до 36 процессоров на однойшине.

В системах NUMA (рис. 7.4, б) каждый процессор имеет некоторый объем локальной памяти и обладает возможностью доступа к памяти других CPU. Скорость доступа к удаленной памяти ниже, чем локальный обмен. Также существуют гибридные системы (рис. 7.4, в), в которых группы процессоров разделяют между собой некоторый объем локальной памяти и при этом обладают возможностью доступа к памяти других групп CPU. Модель NUMA весьма сложна для практической реализации, так как программисту требуется немало усилий для того, чтобы скрыть подробности аппаратной архитектуры от пользовательских приложений.

В системах NORMA каждый процессор обладает прямым доступом только к локальной для него области памяти. Доступ к удаленной памяти осуществляется посредством передачи сообщений. Аппаратура должна поддерживать высокую скорость взаимодействия, обладать большой пропускной способностью, достаточной для осуществления удаленного доступа к памяти. При построении систем на основе архитектуры NORMA необходимо реализовать в ОС управление кэш-памятью, поддержку модели на уровне планировщика, а также создать компиляторы, умеющие оптимизировать коды программ конкретно для данной архитектуры.

В этой главе мы будем рассказывать только о системах, основанных на модели UMA.

7.3.2. Поддержка синхронизации

Синхронизация в многопроцессорных системах сильно зависит от ее аппаратной поддержки. Представим, как проходит простейшая операция блокировки ресурса при помощи флага *locked*, обрабатываемого в области разделяемой памяти. Процедура может состоять из указанной ниже последовательности операций.

1. Чтение флага.
2. Если значение флага равно нулю (следовательно, ресурс свободен), то блокируем ресурс путем установки флага в единицу.

¹ При этом кэширование данных, инструкций и преобразования адресов происходит в каждом процессоре отдельно.

3. Возвращаем TRUE в случае удачной блокировки ресурса или FALSE в противоположном случае.

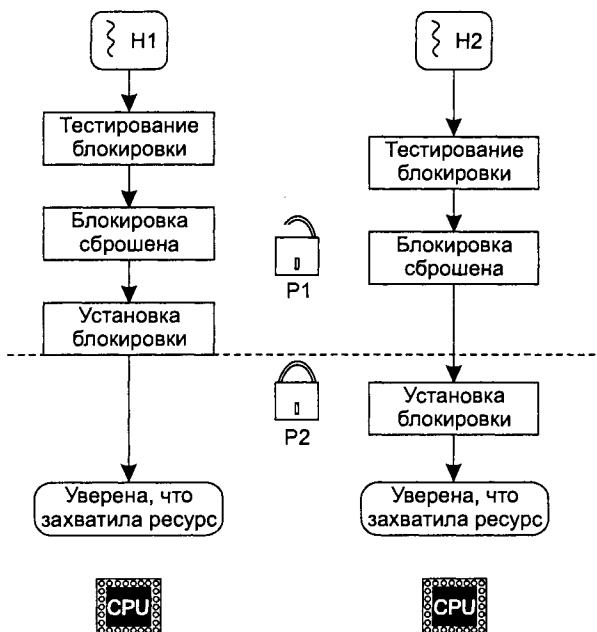


Рис. 7.5. Возникновение состязательности при невыполнении условия неделимости операции тестирования и установки

В многопроцессорной системе две нити, выполняющиеся на двух различных процессорах, могут производить эти операции одновременно. При этом обе нити будут считать, что обладают эксклюзивным правом на ресурс (как это показано на рис. 7.5). Для предупреждения возникновения подобных ситуаций аппаратура компьютера должна предлагать более мощную конструкцию, объединяющую все три операции. Существуют две реализации этой задачи, основанные на применении инструкции тестирования и установки, либо хранения по условию.

Неделимая команда тестирования и установки

Неделимая операция *проверки и установки* (*test-and-set*) обычно используется для действий над одним битом. Команда проверяет его значение, устанавливает в единицу и возвращает предыдущее значение. Таким образом, после завершения ее работы бит равняется 1 (ресурс занят), а возвращаемая величина показывает предыдущее значение этого бита. Если две нити, выполняющиеся на двух различных процессорах, попытаются одновременно произвести действия над одним и тем же битом, то свойство неделимости операции даст возможность сделать это только последовательно, друг за другом. Эта

действие также неделимо с точки зрения прерываний. Если оно возникнет во время выполнения команды, то будет обработано только после ее завершения.

Операция тестирования и установки идеально подходит для простых случаев блокировки ресурсов. Если команда возвращает единицу, вызвавшая ее нить занимает ресурс. Если возвращает ноль, то требуемый ресурс уже занят другой нитью. Освобождение ресурса производится путем сброса бита в 0. Примерами практической реализации операции тестирования и установки являются **BBSSI** (Branch on Bit Set and Set Interlocked) на машине VAX-11 [5] и **LDSTUB** (LoaD and STore Unsigned Byte) на SPARC.

Инструкции взаимосвязи с загрузкой и хранения по условию

В некоторых процессорах применяется пара специальных инструкций загрузки и хранения, используемых для неделимых операций чтения, изменения и записи. Команда *взаимосвязи с загрузкой* (*load-linked*, иногда называемая *load-locked*) загружает значение из памяти в регистр и устанавливает флаг, заставляющий аппаратуру проверять это значение. Если какой-либо из процессоров запишет данные в эту область памяти, аппаратное обеспечение сбросит флаг. Инструкция *хранения по условию* (*store-conditional*) сохраняет новую величину переменной в область памяти, предоставленную установленным флагом. Кроме этого, команда установит значение еще одного регистра, указывающего на то, что была проведена операция сохранения.

7.3.3. Программная архитектура

С точки зрения программного обеспечения существуют три типа многопроцессорных систем: основанные на связке ведущий-ведомый, несимметричные и симметричные. Модель типа *ведущий-ведомый* (*master-slave*) является несимметричной [8], так как один из процессоров в системе играет роль *ведущего* (*master*), в то время как остальные являются *подчиненными* (*slave*). В таких системах возможна ситуация, при которой только ведущий процессор обладает правом обработки ввода-вывода и получения прерываний от различных устройств. В некоторых случаях коды ядра выполняются только на ведущем процессоре, при этом остальные CPU могут быть заняты исключительно пользовательскими программами. При соблюдении таких условий упрощается разработка систем, но преимущества многопроцессорных архитектур используются далеко не в полной мере. Результаты тестов показали [1], что в системах UNIX тратится более 40% общего процессорного времени на выполнение в режиме ядра. Следовательно, наиболее приемлемым вариантом является распределение подпрограмм ядра между несколькими процессорами.

В функционально несимметричных многопроцессорных системах различные подсистемы выполняются на отдельных процессорах. Например, на одном из процессоров могут работать сетевые процедуры, в том время как другой

занят обработкой ввода-вывода. Такой вариант более подходит для специализированных систем, чем для систем общего применения, к которым относится UNIX. Одной из удачных реализаций модели является файловый сервер Auspex NS5000 [10].

Симметричная многопроцессорная обработка (*symmetric multiprocessing*, SMP) является наиболее популярной моделью выполнения. В системах, основанных на этой архитектуре, все процессоры равнозначны и разделяют между собой единственную копию кодов и данных ядра, а также обладают доступом к различным системным ресурсам, таким как память или периферийные устройства. В режиме ядра может функционировать каждый процессор. Пользовательский процесс может быть направлен диспетчером на выполнение на любой CPU системы. В этой главе мы будем описывать только системы, основанные на архитектуре SMP (если не указано иное).

Оставшаяся часть главы посвящена современным механизмам синхронизации, применяемым в однопроцессорных и многопроцессорных системах.

7.4. Особенности синхронизации в многопроцессорных системах

Одним из основных условий, на которых построена традиционная модель синхронизации, является непрерываемое использование ядра нитью до тех пор, пока та не выйдет из привилегированного режима самостоятельно или не будет заблокирована в ожидании ресурса (кроме случаев возникновения прерываний). Такое свойство однопроцессорных ОС совершенно не подходит для многопроцессорных систем, в которых коды ядра могут выполняться одновременно на нескольких CPU. Данные, не требовавшие защиты от повреждения при использовании единственного процессора, теперь необходимо защищать. Рассмотрим проблему на примере организации доступа к таблице ресурсов IPC (см. раздел 6.3.1). Таблица ресурсов недоступна со стороны обработчиков прерываний и не поддерживает операции, которые могут повлечь за собой блокировку процесса. Следовательно, в однопроцессорных системах ядро может производить действия с этой структурой данных без какой-либо ее защиты. При применении многопроцессорных архитектур две нити, выполняющиеся на двух различных CPU, могут одновременно запросить доступ к таблице. В последнем случае перед использованием таблицы нити необходимо заблокировать ее тем или иным образом, запретив к ней доступ другим нитям на время проведения манипуляций.

Необходимо также пересмотреть основные элементы, использующиеся при осуществлении блокировки ресурсов. В традиционных системах ядро просто проверяет флаг *locked* и устанавливает его для блокировки объекта. В многопроцессорных архитектурах может случиться так, что один и тот же флаг

`locked` начнет проверяться двумя нитями одновременно. В этом случае обе нити обнаружат флаг сброшенным и сделают вывод о том, что ресурс свободен. Затем нити установят флаг и начнут использовать объект, что может привести к совершенно неожиданным результатам. Следовательно, многопроцессорные системы должны поддерживать неделимые операции проверки и установки флагов, гарантирующие занятие ресурса одновременно только одной нитью.

Еще одним примером необходимости изменений является приостановка обработки прерываний. В многопроцессорных системах нить обычно имеет право блокировать прерывания, относящиеся только к тому процессору, на котором она выполняется. Чаще всего не существует возможности приостановки прерываний на всех CPU системы, так как некоторые процессоры могут уже получить прерывания, приводящие к возникновению конфликта. Существует возможность повреждения структур данных нити, выполняющейся на одном процессоре, в том случае, если обработчик стартует на другом CPU. Еще одной сопутствующей проблемой является невозможность использования обработчиком модели синхронизации, основанной на приостановке и возобновлении выполнения, так как большинство реализаций ОС не позволяет обработчикам производить блокировку нитей. Система должна поддерживать некий механизм, позволяющий приостанавливать обработку прерываний, возникающих на других процессорах. Одним из возможных решений проблемы является использование глобальных уровней `ipl`, поддерживаемых на программном уровне.

7.4.1. Проблема выхода из режима ожидания

В многопроцессорных системах механизм ожидания-возобновления выполнения работает некорректно. Пример возникновения состязательности при применении этой технологии показан на рис. 7.6. Нить `H1` заблокировала ресурс `P1`. Нить `H2`, выполняющаяся на другом процессоре, пытается получить тот же ресурс, но обнаруживает, что он уже занят. Тогда нить `H2` вызывает функцию `sleep()`, чтобы перейти в режим ожидания ресурса `P1`. Между операциями проверки занятости ресурса и вызовом `sleep()` происходит освобождение `P1`, при этом `H1` инициирует пробуждение всех нитей, находящихся в режиме ожидания этого ресурса. Так как нить `H2` еще не успела переместиться в очередь ожидания, она не может получить сигнал о необходимости возобновления работы. В результате возникает ситуация, когда ресурс уже свободен, но при этом нить `H2` продолжает ожидать его разблокирования. Если в дальнейшем ни одна нить не запросит ресурс `P1`, то `H2` будет оставаться в режиме ожидания постоянно. Описанная ситуация получила название проблемы потери сигнала выхода из режима ожидания. Для ее решения необходимо объединить процедуру проверки ресурса на занятость и последующий вызов `sleep()` в единую, неделимую операцию.

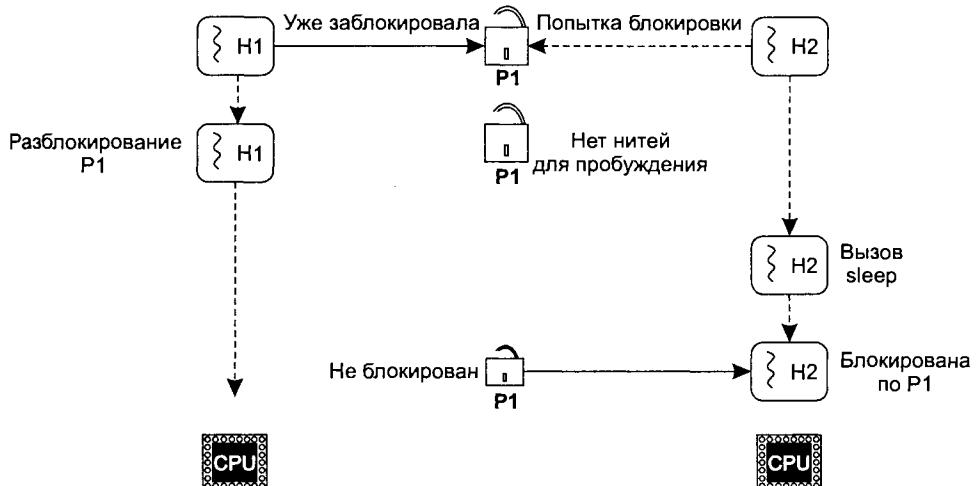


Рис. 7.6. Проблема потери сигнала выхода из режима ожидания

Приведенный пример показывает необходимость привлечения совершенствованием механизмов работы с ресурсами для многопроцессорных систем. Если при их разработке учесть и другие проблемы традиционных реализаций UNIX, то результатом станет создание наилучших решений, что повлияет и на производительность систем.

7.4.2. Проблема быстрого роста

При освобождении ресурса нитью происходит пробуждение всех остальных нитей, ожидающих объекта. При этом только одна из них может занять ресурс, в то время как остальные нити обнаружат его заблокированным и снова перейдут в режим ожидания. Такой подход приводит к дополнительной загрузке системы, тратящей определенное время на возобновление работы нитей и контекстные переключения.

Описанная проблема не фатальна для однопроцессорных систем, поскольку в один момент времени текущим является только одна нить, которая может освободить ресурс. Однако в многопроцессорных архитектурах вероятно возникновение ситуации, когда несколько нитей, ожидающих освобождения ресурса, могут быть направлены планировщиком на выполнение одновременно на различных процессорах, что приведет к новой попытке получения этого ресурса всеми нитями. Такая ситуация получила название проблемы быстрого роста.

В случае если только одна нить оказалась заблокированной в ожидании ресурса, все равно между операциями восстановления работоспособного состояния нити и началом ее выполнения происходит определенная задержка. За этот промежуток времени ресурс может успеть занять совершенно другая

нить, что приведет к повторной блокировке первой нити. Если такая ситуация будет возникать часто, то она способна повлечь устаревание нити.

В текущих разделах вы увидели краткое описание проблем традиционной модели синхронизации, влияющих на корректную работу системы и ее производительность. Оставшаяся часть главы посвящена описанию механизмов синхронизации, подходящих как для однопроцессорных, так и для многопроцессорных систем.

7.5. Семафоры

Синхронизация в ранних реализациях системы UNIX для многопроцессорных машин строилась в основном на использовании *семафоров Дейкстры* (Dijkstra's semaphores) [6]. Их иногда также называют *семафорами со счетчиком* (counted semaphores). Семафоры — это переменные целого типа, поддерживающие две основные операции, *P()* и *V()*. Операция *P()* декрементирует (уменьшает на единицу) значение семафора и блокирует процесс, если результат меньше нуля. Операция *V()* инкрементирует переменную семафора. Если результат меньше или равен нулю, то она разбудит процесс, заблокированный по нему (если таковой существует). Листинг 7.1 показывает пример использования этих операций, а также функцию *initsem()*, применяемую для инициализации семафора, и функцию *CP()*, являющуюся версией *P()* и производящую блокировку.

Листинг 7.1. Операции с семафорами

```
void initsem(semaphore *sem, int val);
{
    *sem = val.
}

void P(semaphore sem) /* запрос семафора */
{
    *sem -= 1;
    while (*sem<0)
        sleep.
}
void V(semaphore *sem) /* освобождение семафора */
{
    *sem += 1;
    if (*sem <= 0)
        будим процесс, заблокированный по sem:
}
boolean_t CP(semaphore *sem) /* попытка получить семафор без блокировки */
{
```

```
if (*sem > 0) {  
    *sem -= 1;  
    return TRUE;  
} else  
    return FALSE;  
{
```

Неделимость операций над семафорами обеспечивается ядром, даже в случае использования многопроцессорных систем. Таким образом, если две нити одновременно попытаются произвести какие-либо операции над одним и тем же семафором, то действия второй начнутся только после того, как произойдет завершение или блокировка процедур первой нити. Операции $P()$ и $V()$ аналогичны *sleep* и *wakeup*, но имеют отличающуюся от них семантику. Команда $SP()$ позволяет запрашивать семафоры без блокировки и может быть использована обработчиками прерываний или другими функциями, для которых блокировка нежелательна. Операцию $SP()$ также можно использовать для предупреждения взаимоблокировки в тех случаях, при которых использование стандартной операции $P()$ может привести к их возникновению.

7.5.1. Семафоры как средство взаимного исключения

Пример, приведенный в листинге 7.2, показывает, как можно использовать семафоры для взаимного исключения по ресурсу. Семафор можно ассоциировать с совместно используемыми ресурсами, например взаимосвязанным списком, и присвоить ему значение 1 при инициализации. Для блокировки ресурса нить выполняет операцию $P()$, для его освобождения — $V()$. Первое применение $P()$ установит значение семафора в ноль, следовательно, все последующие вызовы этой операции приведут к блокировке. При вызове $V()$ значение семафора будет инкрементировано, следовательно, одна из заблокированных ранее нитей будет разбужена.

Листинг 7.2. Применение семафоров для эксклюзивного использования ресурса

```
/* при инициализации */  
semaphore sem;  
initsem (&sem, 1),  
  
/* при каждом применении семафора */  
P(&sem);  
...  
использование ресурса  
...  
V(&sem);
```

7.5.2. Семафоры и ожидание наступления событий

Листинг 7.3 демонстрирует пример использования семафоров для организации ожидания события. Для этого семафор должен быть установлен при инициализации в значение «ноль». Тогда нить, производящая операцию `P()`, будет заблокирована. При наступлении события каждая приостановленная нить должна выполнить `V()`. Это можно реализовать при помощи вызова `V()`, производимого единожды после возникновения ожидаемого события, и применения той же операции `V()` каждой нитью после пробуждения.

Листинг 7.3. Применение семафоров для организации ожидания наступления событий

```
/* при инициализации */
semaphore event;
initsem (&event, 0); /* инициализацию можно произвести при загрузке */

/* выполняется нитью, которой необходимо ждать наступления события */
P(&event); /* блокировка в случае, если событие еще не произошло */
/* после наступления события */
V(&event); /* дает возможность другой нити возобновить выполнение */
/* продолжение кода нити */

/* выполняется при наступлении события */
V(&event); /* будет одну нить */
```

7.5.3. Семафоры и управление исчисляемыми ресурсами

Семафоры можно использовать для размещения различных исчисляемых ресурсов, таких как заголовки блоков сообщений в библиотеке STREAMS. Как показано на примере в листинге 7.4, для этого семафор при инициализации получает значение, равное допустимому количеству экземпляров ресурса. При его запросе нить вызывает операцию `P()`, при освобождении — `V()`. Таким образом, значение семафора показывает текущее количество доступных экземпляров ресурса. Если значение отрицательно, то абсолютное значение семафора равно количеству ожидающих запросов ресурса (или блокированных нитей). Представленный алгоритм является решением проблемы взаимосвязи производителей-потребителей ресурсов.

Листинг 7.4. Применение семафоров для подсчета доступного количества экземпляров ресурса

```
/* при инициализации */
semaphore counter;
initsem(&counter, resourceCount);
```

```
/* выполняется на стадии использования ресурса */
P(&counter); /* блокировка нити, пока ресурс не станет доступен */
использование ресурса; /* ресурс доступен в текущий момент */
V(&counter); /* освобождение ресурса */
```

7.5.4. Недостатки семафоров

Семафоры представляют собой достаточно гибкие, расширяемые компоненты, при помощи которых можно решать различные проблемы синхронизации, но они обладают некоторыми недостатками, не позволяющими применять их в некоторых ситуациях. Во-первых, семафоры являются компонентами высокого уровня, основанными на элементах более низкого уровня, обладающих свойством неделимости и механизмами блокировки. Для того чтобы сохранить неделимость операций P() и V() на многопроцессорных системах, необходимо гарантировать их выполнение на низком уровне, обладая при этом эксклюзивным доступом к объекту семафора. Блокировка и возобновление работы требуют проведения контекстных переключений и манипуляций с очередями сна и планирования, что делает выполнение этих операций очень медленным. Такая скорость может быть приемлема для ресурсов, удерживаемых на большие промежутки времени, однако совершенно не подходит для объектов, требуемых на малое время.

Семафоры не предоставляют информации о том, какая конкретная нить блокируется вследствие проведения операции P(). Чаще всего это не важно, но в некоторых случаях отсутствие информации о блокируемой нити является критичным. Например, буфер кэша в системе UNIX использует функцию getblk() для просмотра конкретного дискового блока, размещенного в кэше. Если необходимый блок обнаружен, вызов getblk() попытается заблокировать его путем проведения операции P(). Если выполнение P() приведет к переходу нити в режим ожидания (так как блок окажется уже занятым), то при возобновлении ее работы может возникнуть ситуация, когда требуемый буфер содержит уже совершенно другой блок. За время ожидания необходимый блок может оказаться уже в совершенно ином месте буфера. Таким образом, после окончания работы P() вполне вероятно, что нить заняла совершенно другой буфер. Описанная проблема может быть решена и при помощи семафоров, но такое решение является весьма неэффективным и громоздким, следовательно, более разумным является применение в этом случае других элементов и конструкций системы [15].

7.5.5. Конвой

Если сравнить семафоры с традиционными механизмами ОС, основанными на приостановке-возобновлении выполнения, то станет видно, что первые обладают определенным преимуществом, связанным с возобновлением выполнения

дополнительных процессов. Если процесс был разбужен в результате операции $P()$, то он гарантированно получит необходимый ресурс. Семантика вызова гарантирует передачу пробуждаемой нити прав владения семафором до начала его выполнения. Если в этот промежуток времени иная нить попытается запросить тот же семафор, то она потерпит неудачу. Однако такое свойство семафоров приводит к возникновению проблемы, названной *конвоированием* семафоров [12]. Конвой возникает в случае частого обращения к семафору. Это может уменьшить производительность любого механизма блокировки, но особенность семантики семафоров значительно усложняет возникающую проблему.

На рис. 7.7 показан пример формирования конвоев. Р1 – это критический участок кода, защищенный семафором. На стадии *a* семафор удерживается нитью Н2, а нить Н3 находится в режиме его ожидания. Нить Н1 выполняется на другом процессоре системы, в то время как Н4 находится в очереди планирования. Теперь представьте, что Н2 завершает выполнение критической секции и освобождает семафор. Тогда она разбудит нить Н3 и перенесет ее в очередь планировщика. Теперь семафор удерживает нить Н3 (как это показано на рис. 7.7, *b*).

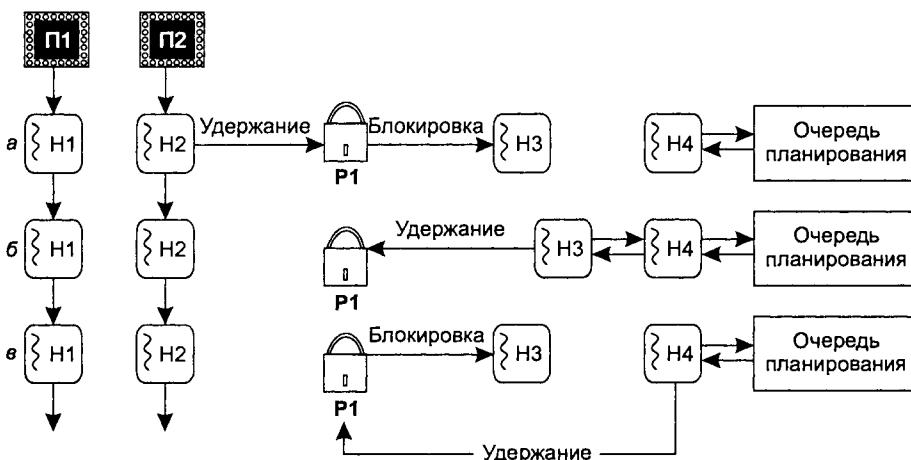


Рис. 7.7. Пример формирования конвоев

Затем выполнение критического участка кода необходимо начать нитью Н1. Так как семафор в текущий момент удерживается нитью Н3, Н1 будет заблокирована и освободит процессор Р1. Система направит нить Н4 на выполнение на этом процессоре. Таким образом, на этапе *в* семафор удерживает нить Н3, Н1 ожидает его освобождения, а нити Н2 и Н4 могут выполняться до тех пор, пока сами не освободят занимаемые ими процессоры.

Проблема возникает именно на этой стадии. Семафор был присвоен нити Н3, но она в данный момент не выполняется и, следовательно, не находится в критической области. В результате нить Н1 должна быть заблокирована в ожидании семафора, несмотря на то, что ни одна из нитей не выполняет

критичный участок кода. Семантика команд работы с семафорами предоставляет ресурсы по принципу: «кто первым пришел, тот и будет обслужен». Такой подход приводит к большому количеству ненужных переключений контекста. Представьте ситуацию, в которой вместо семафора используется *объект mutex* (взаимное исключение). Тогда на стадии б нить Н2 должна освободить объект и разбудить Н3. Но последняя не является на этой стадии владельцем объекта. Следовательно, на следующем этапе объект будет пытаться получить нить Н1, что исключит переключение контекста.

ПРИМЕЧАНИЕ

Mutex, или взаимное исключение (сокращение от *mutual exclusion lock*), является общим термином, обозначающим любой примитив, приводящий к семантике эксклюзивного доступа.¹

Из сказанного можно сделать вывод, что наиболее предпочтительным является применение вместо единого монолитного объекта высшего уровня простых в обращении элементов низшего уровня. Именно этой идеей руководствуются разработчики ядра современных многопроцессорных систем. Следующие разделы будут посвящены описанию механизмов низшего уровня, создающих вместе гибкую систему синхронизации.

7.6. Простая блокировка

Простейшим элементом блокировки является объект, называемый *циклической блокировкой* (spin lock), а иногда — *простой блокировкой* (simple lock) или *простым взаимным исключением* (simple mutex). Если ресурс защищен

¹ Согласно POSIX, как семафоры, так и взаимные исключения являются объектами синхронизации, но первые представляют собой *минимальный примитив*, на котором основываются комплексные механизмы синхронизации, а вторые — не что иное как *объекты синхронизации*, используемые для обеспечения *последовательного доступа множества нитей к разделяемым данным*. Последнее определение не отличается в своей основе от такового для циклической (spin) блокировки. Следуя логике примечания и примеру в 7.5.1, семафоры также можно считать взаимными исключениями. Отсюда нетрудно сделать вывод, что семафоры и простые блокировки можно причислить к объектам mutex. И наоборот, применяемые в иных ОС (VxWorks) целочисленные семафоры (со счетчиком) нельзя назвать семафорами, если ориентироваться на POSIX (следование которому — дело добровольное). И уж тем более такими нельзя назвать multiple wait (muxwait) семафоры OS/2, аналогичные наборам семафоров в UNIX, используемые для управления именованными каналами. В этой же ОС имеется понятие семафора взаимного исключения (все тот же mutex). Или посмотрите далее в начале раздела 7.6... Таким образом, реализация объекта, то есть семантика обращений к нему, и является главным его идентификатором. Например, если в System V для управления несколькими семафорами требуется один вызов, а не несколько, то это — один объект. В этом отношении POSIX недостаточно учитывает ситуацию на местах, что логично, поскольку «авторы стандартов не пишут компьютерных программ». Но с другой стороны, программисты часто, пытаясь «договориться между собой», используют жаргонные термины, например «мьютекс», поэтому дальше мы будем придерживаться более грамотного «взаимного исключения». — Прим. ред.

при помощи этого механизма, то нить, пытающаяся получить доступ к такому ресурсу, перейдет в режим занятого ожидания (выполнения определенного цикла) до тех пор, пока объект не будет освобожден. Взаимоблокировка обычно представляет собой переменную, имеющую значение 0, если ресурс доступен, и значение 1, если он занят. Отслеживание переменной происходит внутри цикла, основанного на одной из неделимых операций, поддерживаемых конкретной аппаратной архитектурой, например при помощи команды тестирования и установки. Ниже представлен пример использования взаимоблокировки. Он основан на том, что операция `test_and_set()` возвращает предыдущее значение объекта.

Листинг 7.5. Применение циклической блокировки

```
void spin_lock (spinlock_t *s) {
    while (test_and_set(s)!= 0) /* ресурс занят */
        ; /* выполнение цикла до тех пор, пока ресурс не освободится */
}
void spin_unlock (spinlock_t s) {s = 0; }
```

Несмотря на простоту алгоритма, в нем имеются определенные недостатки. На многих машинах операция `test_and_set()` блокирует шину памяти. Таким образом, длительный цикл может привести к занятию шины одной нитью – и, следовательно, к существенному уменьшению производительности системы. Решить проблему можно при использовании двух циклов: внешний цикл предназначен для проверки переменной, если проверка не проходит успешно, внутренний цикл будет ждать обнуления значения. Простейшая проверка условия во внутреннем цикле не требует занятия шины памяти. Реализация взаимоблокировки при помощи двух циклов показана в листинге 7.6.

Листинг 7.6. Более корректный способ применения простой блокировки

```
void spin_lock (spinlock_t *s)
{
    while (test_and_set(s)!= 0) /* ресурс занят */
        while (*s != 0)
            ; /* ожидание освобождения ресурса */
}
void spin_unlock (spinlock_t s) {s = 0; }
```

7.6.1. Применение простой блокировки

Наиболее важным свойством простой блокировки является продолжение выполнения нити (или использования процессорного времени) во время ожидания освобождения ресурса. Следовательно, такую блокировку можно применять только для коротких промежутков ожидания. В частности, эти операции не следует производить совместно с блокировкой нитей. Рекомендуется также приостановить обработку прерываний до вызова объекта простой блокировки, что гарантирует уменьшение общей продолжительности ожидания.

Главным условием применения простой блокировки является использование ресурса другой нитью (выполняющейся на другом процессоре), в то время как первая нить находится в режиме ожидания ресурса. Следовательно, использование таких объектов возможно только в многопроцессорных архитектурах. На однопроцессорной машине нить, находящаяся в цикле ожидания освобождения ресурса, не выйдет из этого цикла никогда. Однако многопроцессорные алгоритмы должны работать корректно при любом количестве процессоров в системе, в том числе и в случае одного CPU. Для этого необходимо ограничить применение правила занятия процессора нитью при ожидании ресурса. В частности, на однопроцессорных машинах ни одна нить не должна иметь право входить в режим ждущего цикла при применении взаимоисключения.

Основным достоинством объектов простой блокировки является невысокое использование ресурсов системы. Блок не содержит никакой информации внутри себя, а операции занятия и освобождения объекта требуют выполнения всего лишь одной команды для каждой из них. Простая блокировка идеально подходит для блокировки структур данных, к которым производятся частые обращения, например для удаления элемента из двунаправленного связанного списка или проведения операции загрузки, изменения и сохранения переменной. Из сказанного можно сделать вывод, что применение объектов простой блокировки наиболее целесообразно для защиты тех структур данных, которые не нуждались в таковой в однопроцессорных системах. Однако простые объекты блокировки применяются и для более сложной защиты, что будет продемонстрировано в следующих разделах главы. Например, они могут быть использованы совместно с семафорами для подтверждения неделимости операции (листинг 7.7.).

Листинг 7.7. Применение простых блокировок для организации доступа к двунаправленному связанному списку

```
spinlock_t list;
spin_lock (&list);
item->forw->back = item->back;
item->back->forw = item->forw;
spin_unlock (&list);
```

7.7. Условные переменные

Условные переменные представляют собой более сложный механизм блокировки, основанный на логике *предикатов* (выражения, имеющие значения TRUE и FALSE). Такие переменные позволяют нити приостановить работу в ожидании совместно используемых ресурсов и возобновить выполнение только одной или сразу всех ожидающих нитей при изменении значения выражения. Использование условных переменных является более приемлемым, чем применение простых блокировок.

Представьте ситуацию, в которой одна или несколько нитей серверного приложения находятся в ожидании запросов клиентов. Поступающие запросы передаются ожидающим их нитям. Если ни одна из нитей не может в данный момент обработать поступивший запрос, то последний будет поставлен в очередь. Когда очередная нить серверного приложения заканчивает работу с одним запросом и становится готовой обрабатывать новый, то сначала она проверяет очередь. Если в ней будет обнаружено сообщение, нить удалит его из очереди и начнет обработку запроса. Если очередь окажется пустой, нить приостанавливает выполнение в ожидании поступления нового запроса. Обработка сообщений может быть реализована при помощи условных переменных, ассоциируемых с очередью запросов. В этом случае совместно используемыми данными является очередь сама по себе, а проверяемым условием — выражение «очередь не пуста».

Механизм условных переменных схож с каналами сна, при использовании которых нити серверного приложения блокируются по условию и возобновляют работу после получения сообщений. На многопроцессорных системах необходимо защищать эти механизмы от состязательности, которая может привести к возникновению таких проблем, как потеря сигнала выхода из режима ожидания. В качестве примера можно представить ситуацию, при которой сообщение приходит уже после проведения проверки очереди, но до того момента, как нить перешла в режим ожидания. В таком случае нить окажется приостановленной даже при наличии необработанных сообщений. Следовательно, для защиты от подобных проблем необходимо использовать неделимую операцию проверки условия и блокировки нити.

Такая возможность достигается при помощи дополнительных элементов блокировки. Объекты блокировки (обычно это простые блокировки или *spin locks*) защищают совместно используемые данные и предохраняют их от возникновения ситуации «невыхода» из режима ожидания. Для этого нить серверного приложения создает объект блокировки над очередью сообщений и проверяет очередь на отсутствие запросов. Если очередь окажется пустой, нить вызовет функцию *wait()* с условием, по которому происходит удержание блокировки. Объект блокировки является входным аргументом функции *wait()*. Функция производит приостановление выполнения нити и освобождение объекта блокировки. После получения сообщения и возобновления выполнения нити вызов *wait()* снова запрашивает объект блокировки, после чего ее работа завершается. Пример использования условных переменных показан в листинге 7.8.

Листинг 7.8. Применение условных переменных

```
struct condition {
    proc *next; /* двунаправленный связанный список */
    proc *prev; /* приостановленных нитей */
    spinlock_t lostLock; /* используется для защиты списка */
};
```

```
void wait (condition *c, spinlock_t *s)
{
    spin_lock(&c->listLock);
    add self to the linked list;
    spin_unlock(&c->listLock);
    spin_unlock(s); /* освобождение объекта блокировки до приостановки
                      выполнения нити */
    swtch(); /* переключение контекста */
    spin_lock(s); /* повторный запрос объекта блокировки */
    return;
}

void do_signal(condition *c)
/* пробуждение одной нити, находящейся в ожидании по этому условию */
{
    spin_lock(&c->listLock);
    удаление одной нити из связанных списков, если тот не пуст.
    spin_unlock(&c->listLock);
    ...
    если нить удалена из списка, делаем ее работоспособной;
    ...
    return;
}

void do_broadcast(condition *c)
/* пробуждение всех нитей, находящихся в режиме ожидания по заданному
условию */
{
    spin_lock(&c->listLock);
    while (связанный список не пуст) {
        ...
        удаление нити из связанных списков и
        восстановление работоспособности нити
        ...
    }
    spin_unlock(&c->listLock);
}
```

7.7.1. Некоторые детали реализации условных переменных

При рассмотрении механизма условных переменных необходимо сделать несколько важных замечаний. Логическое условие само по себе не является частью переменной, таким образом, оно должно проверяться до вызова `wait()`. Более того, необходимо помнить о том, что при реализации механизма применяются сразу два объекта блокировки. Одним из этих объектов является `listLock`, который используется для защиты списка нитей, находящихся в ожидании изменения условия. Второй объект защищает проверяемые данные. Он не

является частью условной переменной, но передается в качестве аргумента функции `wait()`. Вызов `switch()` и участок кода, меняющий режим ожидающей нити на работоспособный, может использовать для защиты очередей планировщика еще один объект блокировки.

Таким образом, возникает ситуация, при которой нить, пытающаяся получить один объект блокировки, уже удерживает еще один объект такого же типа. Это не представляет опасности, так как объекты блокировки имеют всего лишь одно ограничение: они запрещают приостановку выполнения нити, удерживающей один из таких объектов. Защита от зависания гарантирована при строго определенном порядке следования объектов: блокировка по условию должна запрашиваться до запроса `listLock`.

Очередь ожидающих нитей не обязательно должна быть частью структуры условия. Вместо этого можно применять глобальный набор каналов сна, точно такой же, как в традиционных системах UNIX. В этом случае объект `listLock` в условии заменяется объектом блокировки, защищающим соответствующий канал сна. Оба приведенных метода обладают определенными достоинствами, о которых мы уже рассказывали ранее.

Одним из достоинств условных переменных является существование двух различных методов обработки события. После его возникновения можно возобновить работу либо одной нити (при помощи `do_signal()`), либо всех ожидающих нитей сразу (используя команду `do_broadcast()`). Каждый из этих вариантов может оказаться наиболее подходящим в той или иной ситуации. В случае применения механизма условных переменных серверными приложениями эффективнее всего возобновлять работу только одной нити, так как каждый запрос обычно обрабатывается единственной нитью. Однако существуют и иные ситуации, например, когда программа выполняется сразу же несколькими нитями, использующими совместно одну копию исходных кодов программы. Если код программы, не хранящийся постоянно в памяти, попытаются запросить сразу несколько нитей, результатом станет получение ошибки каждой из них. Первая нить инициирует доступ к странице, расположенной на диске. Остальные нити получают уведомление о начале операции чтения данных и приостановят выполнение в ожидании завершения ввода-вывода. Когда страница кода будет считана в память, наилучшим решением является вызов `do_broadcast()` и возобновление работы всех приостановленных нитей, после чего все они могут иметь доступ к этой странице без возникновения ошибок.

7.7.2. События

Чаще всего логическое условие переменной является простым. Обычно нить ожидает завершения выполнения определенной задачи. Момент завершения легко обозначить при помощи установки глобального флага. Ситуацию можно описать более приемлемым способом при помощи элемента высшего уров-

ия, называемого *событием* (event). Событие включает в себя флаг `done`, объект блокировки, защищающий этот флаг, и условную переменную на единый объект. Событием легко манипулировать при помощи двух простых операций — `awaitDone()` и `setDone()`. Команда `awaitDone()` приведет к блокировке, установленной до тех пор, пока не произойдет определенное событие, а операция `setDone()` помечает событие `event` как уже свершившееся и возобновляет выполнение всех нитей, заблокированных в ожидании этого события. Помимо этих команд интерфейс взаимодействия с объектами-событиями может поддерживать неблокирующие функции `testDone()` и `reset()`, которые снова помечают событие `event` как еще не свершившееся. В некоторых случаях флаг `done` можно заменить переменной, что позволит передавать более подробную информацию при наступлении события.

7.7.3. Блокирующие объекты

Часто возникают ситуации, при которых нити нужно удерживать ресурс на продолжительный период времени, в течение которого объект должен иметь возможность блокировки по другим событиям. При этом нить, которой необходим ресурс, не начинает выполнять цикл ожидания его освобождения, а приостанавливается. Для реализации такого подхода следует использовать элемент, называемый *блокирующим объектом синхронизации* (blocking lock). Такой объект поддерживает две основные операции, `lock()` и `unlock()`, а также дополнительную команду `tryLock()`. Элемент синхронизирует два объекта — флаг `locked`, относящийся к ресурсу, и очередь сна. Неделимость операций гарантирует дополнительное использование объектов простой блокировки. Такие объекты могут быть реализованы при помощи обычных условных переменных, где проверяемым выражением является сброс флага `locked`. С точки зрения производительности объекты блокирующей синхронизации лучше всего реализовывать как базовые элементы. В частности, если каждый ресурс обладает собственным каналом сна, то для защиты флага и канала можно применять единственный объект блокировки.

7.8. Синхронизация чтения-записи

Изменение ресурса требует установки эксклюзивного права на доступ к нему. Это означает, что его модификацию может производить в один момент времени только одна нить. Однако в большинстве случаев целесообразно разрешить сразу нескольким нитям читать совместно используемые данные в течение промежутка времени, пока никто не попытается произвести запись информации. Для реализации такого подхода требуется сложный механизм синхронизации, поддерживающий одновременно два типа доступа к ресурсу, эксклюзивный и совместный. Механизм может быть создан на основе простой

блокировки и условных переменных [2]. Однако перед тем как перейти к рассмотрению деталей реализации этого механизма, обсудим вкратце его семантику. Синхронизация чтения-записи может разрешать либо изменение ресурса (запись) одной нитью, либо его чтение несколькими нитями. Основными операциями над объектом являются `lockShared()`, `lockExclusive()`, `unlockShared()`, `unlockExclusive()`. Кроме этого, объекты можно попытаться сделать эксклюзивно используемыми или разделяемыми при помощи команд `tryLockShared()`, `tryLockExclusive()`, возвращающих значение `FALSE` вместо блокировки нити. Дополнительно необходима поддержка преобразования объекта из «эксклюзивного» в «совместный» и обратно при помощи команд `upgrade()` и `downgrade()`. Операция `lockShared()` должна приводить к блокировке, если существует объект эксклюзивной синхронизации, в то время как `lockExclusive()` блокирует в случае любого занятия ресурса (как в эксклюзивном, так и в режиме совместного доступа).

7.8.1. Задачи, стоящие перед разработчиками

Что должна сделать нить после освобождения ресурса? В традиционных системах UNIX произойдет пробуждение всех нитей, ожидающих этот объект. Такой подход является абсолютно неэффективным. Если нить, производящая изменение информации, снова потребует объект блокировки, остальные читающие и модифицирующие ресурс нити должны будут перейти в режим ожидания повторно. Если объект затребует нить-получатель данных, то все модифицирующие ресурс нити обязаны будут приостановить выполнение. Таким образом, становится очевидной необходимость использовать некий протокол, защищающий от излишних переключений нитей из режима ожидания.

Когда ресурс освобождает нить-получатель, то она не производит никаких дополнительных действий, если существуют другие нити, считывающие данные из этого ресурса. После того как последняя читающая нить освобождает ресурс, она должна разбудить единственную нить, имеющую возможность записи информации.

Если ресурс освобождает пишущая нить, ей необходимо производить выбор между восстановлением работоспособности другой модифицирующей нити либо других читателей (представим, что объект ожидают и считывающие, и записывающие нити). Если отдать предпочтение нитям, изменяющим информацию, то читающие нити могут быть подвержены устареванию. Наиболее приемлемым решением проблемы действий при освобождении объекта эксклюзивной синхронизации является возобновление работоспособности всех ожидающих читающих нитей. Если таковых не существует, то разбудженная будет нить, осуществляющая запись данных.

Такая методика действий может привести к устареванию нитей, осуществляющих запись. Если объект необходимо читать большому количеству нитей, ресурс будет постоянно блокирован на чтение; следовательно, в этом случае

пишущая нить не сможет получить доступ к нему. Для защиты от возникновения подобной ситуации запрос `lockShared()` должен осуществлять блокировку при существовании ожидающих записи нитей даже в том случае, если ресурс защищен только для чтения. Такое решение при определенных условиях может позволить чередовать доступ между несколькими пишущими нитями и большими множествами нитей, осуществляющими только чтение ресурса.

Функция `upgrade()` должна быть защищена от взаимоблокировки. Такая ситуация может возникнуть, если в реализации механизма не предусмотрены определенные правила при изменении статуса запросов от ожидающих пишущих нитей. Если две нити пытаются изменить статус объекта синхронизации, то каждая из них окажется заблокированной, так как другая нить удерживает совместно используемый объект. Одним из методов защиты от возникновения указанной ситуации является освобождение функцией `upgrade()` разделяемого объекта перед блокировкой нити в том случае, если объект для эксклюзивного доступа окажется невозможно получить сразу же. Это приведет к дополнительным проблемам с точки зрения пользователя, так как во время выполнения `upgrade()` другая нить может произвести изменение объекта. Еще одним вариантом выхода из описанной ситуации является возврат функции `upgrade()` с ошибкой и освобождение разделяемого объекта синхронизации в том случае, если другая нить уже находится в ожидании изменения его статуса.

7.8.2. Реализация синхронизации чтения-записи

Пример реализации механизмов синхронизации чтения-записи показан в листинге 7.9.

Листинг 7.9. Реализация синхронизации чтения-записи

```
struct rwlock {
    int nActive; /* количество активных читателей, либо -1, если существует
    активная нить, модифицирующая ресурс */
    int nPendingReads;
    int nPendingWrites;
    spinlock_t s1;
    condition canRead;
    condition canWrite;
};

void lockShared(struct rwlock *r)
{
    spin_lock(&r->s1);
    r->nPendingReads++;
    if (r->nPendingWrites > 0)
        wait (&r->canRead, &r->s1); /* защита от устаревания
    пишущих нитей */
```

продолжение»

Листинг 7.9 (продолжение)

```

while (r->nActive < 0) /* если кто-то удерживает эксклюзивно
    объект синхронизации */
    wait (&r->canRead, &r->s1);
r->nActive++;
r->nPendingReads--;
spin_unlock(&r->s1);
}
void unlockShared(struct rwlock *r)
{
    spin_lock(&r->s1);
    r->nActive--;
    if (r->nActive == 0) { /* нет других читающих нитей */
        spin_unlock(&r->s1);
        do_signal(&r->canWrite);
    } else
        spin_unlock(&r->s1);
}
void lockExclusive(struct rwlock *r)
{
    spin_lock(&r->s1);
    r->nPendingWrites++;
    while (r->nActive)
        wait (&r->canWrite, &r->s1);
    r->nPendingWrites--;
    r->nActive = -1;
    spin_unlock(&r->s1);
}
void unlockExclusive(struct rwlock *r)
{
    boolean_t wakeReaders;
    spin_lock(&r->s1);
    r->nActive = 0;
    wakeReaders = (r->nPendingReads != 0);
    spin_unlock(&r->s1);
    if (wakeReaders)
        do_broadcast (&r->canRead); /* пробуждение всех нитей, ожидающих
            возможности чтения ресурса */
    else
        do_signal (&r->canWrite); /* пробуждение одной нити,
            осуществляющей запись данных */
}
void downgrade(struct rwlock *r)
{
    boolean_t wakeReaders;
    spin_lock(&r->s1);
    r->nActive = 1;
    wakeReaders= (r->nPendingReads != 0);
}

```

```
spin_unlock(&r->s1);
if (wakeReaders)
    do_broadcast (&r->canRead); /* пробуждение всех нитей, ожидающих
        возможности чтения ресурса */
}
void upgrade(struct rwlock *r)
{
    spin_lock(&r->s1);
    if (r->nActive == 1) { /* нет нитей, ожидающих ресурс на чтение */
        r->nActive = -1;
    } else {
        r->nPendingWrites++;
        r->nActive--; /* освобождение совместного объекта синхронизации */
        while (r->nActive)
            wait (&r->canWrite, &r->s1);
        r->nPendingWrites--;
        r->nActive = -1;
    }
    spin_unlock(&r->s1);
}
```

7.9. Счетчики ссылок

Объект блокировки в состоянии защитить данные, находящиеся внутри ресурса. Но существует немало ситуаций, при которых необходима защита ресурса самого по себе. Многие объекты ядра запрашиваются и освобождаются динамически. Если одна из нитей освободит такой объект, другие нити не будут знать об этом и могут попытаться получить доступ к ресурсу при помощи прямого указателя на него. Возможно, что ядро уже произвело передачу участка памяти, занятого объектом, другому ресурсу. Тогда попытка доступа к нему может привести к нарушению работы системы.

Если нить обладает указателем на объект, то такой указатель считается правильным на тот период времени, пока он не будет освобожден нитью. Ядро системы может гарантировать правильность указателя при помощи *счетчика ссылок* на каждый объект. Для этого ядро устанавливает значение счетчика равным единице при первом запросе объекта (создавая при этом указатель). Счетчик инкрементируется при создании каждого нового указателя на объект.

В таком случае при получении указателя на объект нить получает на него ссылку. В дальнейшем нить может освободить ссылку, которая ей уже не нужна, при этом ядро системы произведет декrement счетчика ссылок объекта. Если значение счетчика станет равным нулю, это будет означать, что ни одна нить не обладает ссылкой на объект; тогда ядро осуществит освобождение объекта.

Механизм счетчиков ссылок применяется при управлении файловой системой. Система поддерживает счетчик для объектов vnode, в которых хранит информацию об используемых файлах (см. раздел 8.7). Если пользователь открывает файл, ядро возвращает дескриптор этого файла, содержащий ссылку на vnode. Пользователь передает этот дескриптор системным вызовам `read` или `write`, предоставляя возможность быстрого доступа ядра к файлу без проведения преобразований имен файлов. Если пользователь закрывает файл, ссылка освобождается. Если один и тот же файл открывают сразу несколько пользователей, то они получают ссылки на один и тот же объект vnode. После того как последний пользователь закроет файл, ядро освобождает объект vnode.

Пример показывает, что механизм счетчиков ссылок больше подходит для однопроцессорных систем, так как он слишком прост для применения в многопроцессорных архитектурах, в которых может возникнуть ситуация, при которой одна из нитей пытается освободить объект, в то время как другая нить, выполняющаяся на другом CPU, продолжает пользоваться им. Для применения механизма в таких системах необходимо реализовать несколько другую методику подсчета ссылок.

7.10. Другие проблемы, возникающие при синхронизации

При разработке и практическом применении сложных механизмов синхронизации для многопроцессорных систем необходимо учитывать несколько важных факторов. Наиболее важные из них будут описаны в этом разделе.

7.10.1. Предупреждение возникновения взаимоблокировки

Во многих случаях нити необходимо удерживать сразу же несколько ресурсов. Например, в реализации условных переменных, описанных в разделе 7.7, применяется два объекта простой блокировки, один из которых защищает данные и логическое условие, в то время как второй используется для защиты списка нитей, находящихся в режиме ожидания изменения значения условия. В этом случае попытка получения сразу же нескольких объектов синхронизации приведет к взаимоблокировке, как это показано на рис. 7.8. Нить H1 удерживает ресурс P1 и пытается получить ресурс P2. В тот же момент времени нить H2 может являться владельцем P2 и добиваться доступа к ресурсу P1. В такой ситуации ни одна из нитей не достигнет своей цели. Обе нити зависнут.

Для предупреждения возникновения взаимоблокировки обычно применяется одна из двух методик, называемых *иерархической блокировкой* (*hierarchical locking*) и *вероятностной блокировкой* (*stochastic locking*). Иерархический метод основан на назначении определенного порядка зависимых друг от друга

объектов синхронизации и требует, чтобы все нити получали такие объекты в указанном порядке. При этом в случае применения, к примеру, условных переменных нить должна занять логическое условие до блокировки связанныго списка. Точное соблюдение порядка следования приводит к невозможности возникновения взаимоблокировки.

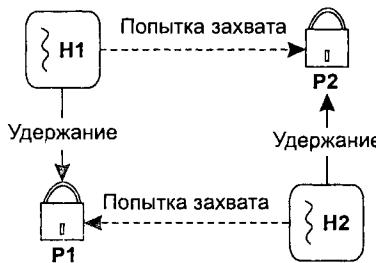


Рис. 7.8. Возникновение взаимной блокировки при применении простых объектов синхронизации

Иногда возникают ситуации, при которых порядок блокировки требуется изменить. Представьте буферный кэш, который обслуживается при помощи двунаправленного связанныго списка дисковых буферов. Список отсортирован в порядке «наиболее давно использовавшихся» (least recently used, LRU) элементов. Все буфера, не используемые в текущий момент времени, занесены в этот список. Защита заголовка очереди, а также указателей на предыдущий и последующий элементы поддерживается при помощи единственного объекта блокировки. Каждый буфер в отдельности использует еще один объект для защиты другой информации, содержащейся внутри буфера. Такой объект необходимо удерживать во время использования буфера.

Если нити необходимо загрузить один из дисковых блоков, сначала она запрашивает буфер (используя для этого хэширование или другие указатели), после чего блокирует его. Затем нить блокирует список LRU для удаления из него полученного блока. Следовательно, обычный порядок блокировок можно описать фразой: «сначала буфер, затем — список».

В некоторых случаях нити нужно получить любой свободный буфер, для чего она обращается к заголовку списка LRU. При этом нить сначала блокирует список и только затем блокирует буфер, оказавшийся в списке первым, после чего удаляет его из списка. Этот случай демонстрирует изменение порядка блокировок на противоположный: нить сначала блокирует список и только затем — буфер.

Можно заметить, что в этом случае возможно возникновение взаимоблокировки. Представьте, что одна нить захватывает буфер из заголовка списка и затем пытается заблокировать сам список. В тот же момент времени другая нить, которая уже заблокировала список, пытается занять буфер из его заголовка. Это приведет к тому, что каждая нить приостановит выполнение в ожидании освобождения блокировки, удерживаемой другой нитью.

Для предупреждения возникновения подобных ситуаций ядро системы использует методику вероятностной блокировки. Если нить пытается получить объект синхронизации в нарушение иерархии, она использует функцию `try_lock()` вместо `lock()`. Операция `try_lock()` применяется для получения объекта, но результатом ее работы в случае обнаружения занятости объекта окажется не блокировка нити, а выход с ошибкой. В приведенном выше примере нить, которой необходимо получить любой свободный буфер, сначала заблокирует список, а затем будет пытаться получить первый свободный объект при помощи операции `try_lock()`, перемещаясь по списку. Листинг 7.10 показывает пример применения функции `try_lock()` над простыми объектами синхронизации.

Листинг 7.10. Пример использования операции `try_lock()`

```
int try_lock(spinlock_t *s)
{
    if (test_and_set(s) != 0) /* объект уже занят */
        return FAILURE;
    else
        return SUCCESS;
}
```

7.10.2. Рекурсивная блокировка

Блокировка называется рекурсивной, если попытка захвата объекта, которым уже обладает нить, окажется успешной и не повлечет приостановку ее выполнения. В каких случаях можно использовать эту возможность? В каких ситуациях нить пытается запросить объект, которым она уже обладает? Обычно такое происходит, если нить, удерживающая ресурс, вызывает процедуру низкого уровня, производящую какие-либо операции с этим ресурсом¹. Процедуры могут быть вызваны и другими командами высшего уровня, не занявшими перед этим ресурс. Становится ясно: процедура низкого уровня может попросту не знать о том, что ресурс является заблокированным. Если процедура попытается произвести блокировку, результатом станет зависание процесса.

Описанные выше ситуации можно предупредить, информируя процедуру о блокировке ресурса при помощи дополнительного входного аргумента. Но, к сожалению, такой подход требует изменения интерфейсов взаимодействия. Он может привести к определенным неудобствам, так как процедуры способны производить вызовы дополнительных функций. В результате обновленный интерфейс потеряет свойство модульности. Альтернативным решением проблемы является применение рекурсивной блокировки. При этом возникают определенные перегрузки, так как объекту блокировки необходимо

¹ Или при рекурсивном вызове самой себя. — *Прим. ред.*

хранить некий идентификатор владельца и проверять его каждый раз при блокировке или запрещении запроса. Однако более важным является тот факт, что рекурсивная блокировка позволяет иметь дело только с общими требованиями, независимо от того, какой конкретный объект блокировки используется нитью. Применение рекурсивной блокировки позволяет строить понятные модульные интерфейсы.

Примером использования методики является осуществление записи каталога файловой системы BSD (*ufs*). Для записи каталогов и отдельных файлов применяется одна и та же процедура *ufs_write()*. Доступ к записываемому файлу обычно осуществляется через элемент файловой таблицы, представляющей указатель на объект *vnode* этого файла. Таким образом, объект *vnode* передается напрямую процедуре *ufs_write()*, которая и должна в дальнейшем осуществить его блокировку. Однако дескриптор *vnode* при осуществлении записи каталога запрашивается посредством процедуры просмотра имен путей, возвращающей этот объект в уже заблокированном состоянии. Если затем произвести вызов *ufs_write()* для записи в каталог, результатом станет взаимоблокировка. Для предупреждения возникновения такого состояния необходимо использовать рекурсивную блокировку.

7.10.3. Что лучше: приостановка выполнения или ожидание в цикле?

Сложная синхронизация объектов может быть реализована как блокирующий объект или как сложный ожидающий объект без проведения изменений в свойствах или интерфейсах обоих типов объектов. Представьте, что ресурс защищен при помощи сложного объекта синхронизации (например, посредством семафора или объекта синхронизации чтения-записи). В реализации большинства механизмов, описываемых в этой главе, если нить пытается получить доступ к объекту и находит его уже занятым, она приостанавливает выполнение в ожидании освобождения этого объекта. Нить может и продолжать выполнение, осуществляя цикл ожидания ресурса.

Выбор между приостановкой выполнения и ожиданием в цикле зависит чаще всего от соображений производительности работы системы. В режиме ждущего цикла занимается процессор, что делает выбор этого варианта нежелательным. Но в системе иногда возникают ситуации, при которых ждущий цикл оказывается удобнее всего. Если нить уже занимает простой объект (*mutex*) синхронизации, он не имеет права блокировки. Если нить попытается получить еще один простой объект, то она перейдет в режим циклического ожидания. В случае необходимости получения сложного объекта синхронизации занимаемый нитью объект-*mutex* будет освобожден (точно по такому же алгоритму происходят действия с условными переменными).

Операции приостановки и возобновления выполнения нити являются громоздкими сами по себе, так как требуют проведения контекстных пере-

ключений и манипуляций с очередями сна и планирования. Однаково неразумным представляется ожидание ресурса, который удерживается в течение продолжительного периода времени, точно так же как и приостановка выполнения нити в ожидании объекта, который вскоре может быть освобожден.

Более того, некоторые ресурсы могут удерживаться как на малый, так и на большой промежуток времени, в зависимости от определенных условий. Например, ядро может хранить в памяти некий список свободных дисковых блоков. Если список становится пустым, его необходимо заполнить новыми данными о свободных блоках. В большинстве случаев список блокируется на небольшие интервалы времени, в течение которых происходит добавление или удаление его элементов. Если при этом необходимо производить операции ввода-вывода, список окажется заблокированным на большой период времени. Следовательно, ни ждущий цикл, ни блокировка не окажется в данном случае идеальным выходом. Одним из вариантов решения проблемы является использование двух объектов синхронизации. При этом блокирующий объект применяется только в случае необходимости пополнения списка. Однако наиболее предпочтительным вариантом в этом случае является использование другого, более гибкого элемента синхронизации.

Один из наиболее приемлемых вариантов решения проблемы заключается в сохранении в объекте синхронизации указания, по которому и определяется, как нужно поступать нити: приостанавливать работу или выполнять цикл ожидания. Указание устанавливается обладателем ресурса и проверяется при неудачном завершении попытки получения объекта синхронизации. Указание может иметь как рекомендательный, так и директивный характер.

Альтернативное решение под названием *адаптивной блокировки* (*adaptive locks*) представлено в системе Solaris 2.x [7]. Если нить T1 необходимо получить адаптивный объект, удерживаемый нитью T2, то сначала производится проверка выполнения T2 на одном из процессоров системы. Если T2 выполняется в текущий момент времени, нить T1 перейдет в режим циклического ожидания; если T2 окажется заблокированной, T1 также приостановит свою работу.

7.10.4. Объекты блокировки

Блокировка применяется для защиты различных типов объектов: данных, логических выражений, инвариантных выражений или операций. К примеру, объекты блокировки чтения-записи используются для защиты данных. Условные переменные обычно ассоциируются с логическими выражениями. Инвариантное выражение имеет определенные сходства с логическим выражением, но отличается в семантике. Если объект защищен инвариантом, то это означает, что его значение будет равно TRUE всегда, кроме случаев удержания объекта кем-либо. В качестве примера можно привести связанный список, который использует единственный объект блокировки при добавлении или удалении записей. Инвариантным выражением, защищающим список, может быть целостность его состояния.

Объекты блокировки можно применять для управления доступом к определенной операции или функции. При этом устанавливается ограничение на код программы, который может выполняться одновременно только на одном процессоре системы. На этом свойстве построена модель синхронизации *управляющих программ* (monitors) [9]. Во многих реализациях UNIX один из процессоров используется для выполнения участков ядра, не поддерживающих параллельность. Такой метод применяется для последовательного выполнения кода, не рассчитанного для работы на многопроцессорных машинах. Это часто приводит к возникновению эффекта «бутылочного горлышка» — следовательно, по возможности от него следует отказаться.

7.10.5. Степень разбиения и длительность

Производительность системы сильно зависит от степени разбиения объектов синхронизации. С одной стороны, в некоторых асимметричных многопроцессорных ОС коды ядра выполняются только на одном процессоре, называемом главным. В этом случае для всего ядра достаточно одного объекта блокировки. С другой стороны, система может применять большое количество таких объектов, предоставляя их каждой переменной данных. Понятно, что ни первое, ни второе решение не является идеальным. Объекты блокировки занимают большие объемы памяти. Производительность системы может снизиться из-за необходимости постоянного получения и освобождения блокировки. Повышается вероятность возникновения взаимоблокировки нитей, так как при большом количестве синхронизируемых ресурсов трудно поддерживать определенный порядок их блокировки.

Идеальное решение, как обычно, находится в районе золотой середины. По этому вопросу до сих пор не достигнут консенсус среди разработчиков ОС. Приверженцы методики крупногранулированной блокировки предлагают использовать сначала небольшое количество объектов, защищающих основные подсистемы, и добавлять их при возникновении эффекта «бутылочного горлышка». Однако в ОС Mach и некоторых других применяется структура с подробной детализацией, в которой блокируются отдельные объекты данных.

Длительность блокировки также требует тщательного анализа. Оптимальным вариантом является удержание ресурса на короткие промежутки времени, что минимизирует степень соперничества при попытке обладания им. Однако в некоторых случаях такой подход может привести к дополнительным операциям занятия и освобождения ресурса. Представьте, что нити требуется произвести две операции над одним и тем же объектом, каждая из которых требует его блокировки. После окончания первой операции нить необходимо совершить некоторое действие, не связанное с объектами. Нить может освободить ресурс после завершения первой операции и снова занять его при проведении второй. В такой ситуации лучшим решением может оказаться блокировка ресурса на весь промежуток времени, особенно если промежу-

точное действие производится быстро. Решения о длительности блокировки объекта должны приниматься в индивидуальном порядке.

7.11. Реализация объектов синхронизации в различных ОС

При описании различных методик синхронизации в разделах 7.5–7.8 было показано, что операционная система может использовать объекты совместно друг с другом, предлагая тем самым комплексный интерфейс синхронизации ресурсов. В этом разделе будут описаны средства синхронизации в основных многопроцессорных вариантах системы UNIX.

7.11.1. SVR4.2/MP

SVR4.2/MP — это версия системы SVR4.2, поддерживающая многопроцессорные архитектуры. ОС обеспечивает четыре типа синхронизации: простую блокировку, блокировку сна, блокировку чтения-записи и переменные синхронизации [17]. Каждый объект должен запрашиваться или освобождаться явно при помощи операций `xxx_ALLOC` и `xxx DEALLOC`, где `xxx_` — это префикс, зависящий от типа используемого объекта. Операции получения объекта имеют входные аргументы, которые могут быть использованы для отладки.

Простые объекты блокировки

Простой блокировкой называется нерекурсивный объект `mutex`, позволяющий производить краткосрочную блокировку ресурсов. Он не может удерживаться при операциях блокировки нитей. Объект задается переменной типа `lock_t`. Для его получения и освобождения используются следующие операции:

```
pl_t LOCK (lock_t *lockp, pl_t new_ipl);  
UNLOCK (lock_t *lockp, pl_t old_ipl);
```

Вызов `LOCK` перед запросом объекта изменяет уровень приоритета до `new_ipl` и возвращает предыдущий уровень `ipl`. Эта переменная передается операции `UNLOCK` для восстановления предыдущего уровня приоритета.

Блокировка чтения-записи

Блокировка чтения-записи — это нерекурсивный метод синхронизации, основанный на семантике одного отправителя и нескольких получателей данных. Он не может удерживаться при операциях блокировки нитей. Объект задается переменной типа `rwlock_t` и поддерживает следующие операции:

```
pl_t RW_RDLOCK (rwlock_t *lockp, pl_t new_ipl);  
pl_t RW_WRLOCK (rwlock_t *lockp, pl_t new_ipl);  
void RW_UNLOCK (rwlock_t *lockp, pl_t old_ipl);
```

Действия с приоритетами прерываний идентичны приведенным для простой блокировки. Операции блокировки изменяют уровень `ipl` до указанного и возвращают предыдущий уровень. Операция `RW_UNLOCK` восстанавливает предыдущее значение уровня `ipl`. Объект блокировки чтения-записи также поддерживает операции `RW_TRYRDLOCK` и `RW_TRYWRLOCK`, не приостанавливающие работу нити.

Блокировка сна

Блокировкой сна является применение нерекурсивных взаимных исключений, поддерживающих длительное удержание ресурсов. Такой объект может удерживаться при операциях блокировки нитей. Он реализован в виде переменной типа `sleep_t` и поддерживает следующие операции:

```
void SLEEP_LOCK (sleep_t *lockp, int pri);
bool_t SLEEP_LOCK_SIG (sleep_t *lockp, int pri);
void SLEEP_UNLOCK (sleep_t *lockp);
```

Параметр `pri` используется для указания приоритета планирования, который будет присвоен процессу после его пробуждения. В случае, когда процесс блокируется в результате вызова `SLEEP_LOCK`, он не может прерываться сигналом. Если процесс блокируется в результате вызова `SLEEP_LOCK_SIG`, поступивший сигнал разбудит процесс. Функция возвратит `TRUE`, когда объект будет получен, или `FALSE`, если сон процесса окажется прерванным. Объекты блокировки сна поддерживают и другие операции, такие как `SLEEP_LOCK_AVAIL` (проверка доступных объектов), `SLEEP_LOCKOWNED` (проверяет, не является ли вызывающий процесс обладателем объекта), `SLEEP_TRYLOCK` (возвращает ошибку вместо блокировки процесса в случае невозможности получения объекта).

Переменные синхронизации

Переменные синхронизации идентичны условным переменным, описанным в разделе 7.7. Они реализованы в виде переменных типа `sv_t`. Условие, которое должно быть защищено простым объектом блокировки, проверяется отдельно пользователями переменной синхронизации. Поддерживаются следующие операции:

```
void SV_WAIT (sv_t *svp, int pri, lock_t *lockp);
bool_t SV_WAIT_SIG (sv_t *svp, int pri, lock_t *lockp);
void SV_SIGNAL (sv_t *svp, int flags);
void SV_BROADCAST (sv_t *svp, int flags);
```

Так же как и в предыдущем случае, аргумент `pri` указывает на приоритет диспетчеризации, который будет присвоен процессу после его пробуждения, а операция `SV_WAIT_SIG` позволяет прерывание сна по сигналу. Аргумент `lockp` используется для передачи указателя на простой объект блокировки, защищающий логическое условие. Перед вызовом `SV_WAIT` или `SV_WAIT_SIG`

процесс должен занять объект `lockp`. Эти операции являются неделимыми и приводят к блокировке вызывающего процесса ядром и освобождении объекта, заданного при помощи `lockp`. После возврата `SV_WAIT` или `SV_WAIT_SIG` объект `lockp` оказывается незанятым. Заданное условие не всегда является истинным выражением в момент возобновления работы вызывающей нити, поэтому при использовании `SV_WAIT` или `SV_WAIT_SIG` необходимо производить его циклическую проверку.

7.11.2. Digital UNIX

Механизмы синхронизации системы Digital UNIX основаны на элементах Mach. Поддерживается два типа синхронизации: простая и комплексная [4]. Простая блокировка выполняется при помощи объекта `mutex`, реализованного на основе неделимой инструкции тестирования и установки конкретного типа аппаратной архитектуры. Объект перед использованием необходимо описать и проинициализировать, при этом он окажется в незанятом состоянии. Блокировка не будет удерживаться при операциях приостановки-возобновления выполнения нитей или при контекстных переключениях.

Сложная блокировка представляет собой элемент высокого уровня, поддерживающий большое количество возможностей, таких как разделяемый и эксклюзивный доступ, блокировка нитей или рекурсивные объекты. Такой элемент можно использовать для организации доступа чтения-записи ресурса. Он поддерживает две опции, `sleep` (режим сна) и `recursive` (рекурсивность). Опция `sna` может быть включена или отключена как на стадии инициализации объекта, так и позже при его использовании. Если она установлена, ядро будет блокировать все запросы в случае неудачного завершения попытки получения объекта. Более того, в случае необходимости блокировки нити во время удержания объекта опция `sna` должна быть обязательно установлена. Рекурсивность задействуется только в случае получения объекта в эксклюзивное пользование и может быть снята только той нитью, которая ранее установила ее.

Поддерживаются неблокирующие варианты различных процедур, возвращающие ошибку при невозможности получения объекта блокировки. Предлагаются функции изменения статуса объекта из *разделяемого* в *эксклюзивный* (`upgrade`) и обратно (`downgrade`). Операция повышения уровня освободит разделяемый объект и возвратит ошибку в том случае, если другая нить находится в режиме ожидания такого же запроса. Неблокирующий вариант операции `upgrade` в этом случае возвращает ошибку, но не освобождает совместный объект блокировки.

Сон и пробуждение

Так как большая часть кода системы была взята из 4BSD, в Digital UNIX оставлена поддержка вызовов `sleep()` и `wakeup()`. Таким образом, блокируе-

мые нити попадают в глобальные очереди сна чаще, чем в очереди отдельных объектов синхронизации. Для корректной работы механизма на многопроцессорных архитектурах алгоритмы блокировки необходимо подвергнуть изменениям. Основная проблема, которой следует уделить внимание, это потеря сигнала необходимости пробуждения нити. С этой целью исходные коды функции `sleep()` были переписаны заново с использованием двух операций низшего уровня, `assert_wait()` и `thread_block()`. Схема работы продемонстрирована на рис. 7.9.

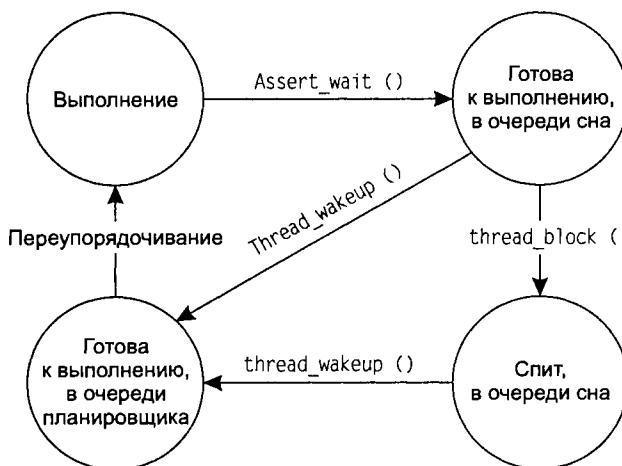


Рис. 7.9. Реализация перехода в режим сна в системе Digital UNIX

Представьте, что нити необходимо ожидать наступления некоторого события, описанного логическим выражением и защищенного при помощи простого объекта синхронизации. Нить запрашивает объект, после чего проверяет условие. Если в дальнейшем необходимо приостановить выполнение нити, происходит вызов `assert_wait()`, который перемещает ее в соответствующую очередь сна. Затем происходит освобождение объекта и вызов функции `thread_block()` для инициализации переключения контекста. Если событие произойдет в промежутке времени между освобождением объекта и контекстным переключением, ядро удалит нить из очереди сна и переместит его в очередь планирования. Следовательно, при применении вышеописанных функций нить не пропустит момент необходимости возобновления выполнения.

7.11.3. Другие реализации систем UNIX

Первая версия SVR4, поддерживающая многопроцессорность, была создана в NCR [3]. В ней была представлена концепция консультативных *объектов блокировки процессора* (*advisory processor locks*, APL), при использовании

которой рекурсивные объекты содержат указание действий над нитями. Рекомендация (или hint) подсказывает, что должна делать нить: приостановить выполнение или перейти в следующий цикл ожидания ресурса. Также можно задать обязательный или рекомендательный характер исполнения указания. Нить, обладающая объектом блокировки, может изменить указание с необходимости приостановки выполнения на ожидание в цикле и обратно. Объекты APL обычно используются для доступа нити к долгосрочно блокируемым ресурсам. Особенностью объектов APL является их автоматическое освобождение и запрос при проведении переключения контекста. Это означает возможность применения традиционного интерфейса сна-пробуждения без проведения каких-либо изменений. Более того, объекты блокировки одного и того же класса защищены от проблемы потери сигнала пробуждения, так как при переходе нити в режим сна происходит освобождение всех ранее удерживаемых объектов. Система также поддерживает нерекурсивные взаимные исключения и объекты APL защиты чтения-записи.

Версия SVR4, представленная NCR, в дальнейшем была усовершенствована консорциумом Intel Multiprocessor Consortium, созданным группой компаний для разработки официальной многопроцессорной реализации системы [14]. Одним из самых важных изменений, произведенных в этой ОС, стал вызов функции, запрашивающей объект APL. Добавился новый входной аргумент – уровень приоритета прерываний. Это позволило изменять уровни *ipl* при манипуляциях с объектами блокировки. Если объект не может быть получен сразу же, цикл занятого ожидания будет иметь изначальный (более низкий) приоритет. Функция возвращает оригинальный уровень *ipl*, значение которого в дальнейшем может быть передано функции освобождения объекта.

Элементами наиболее низкого уровня являются наборы неделимых арифметических и логических операций. Арифметические операции позволяют инкрементировать и декрементировать значения счетчиков ссылок. Логические операции применяются для побитовых манипуляций с полями флагов. Операции обоих типов возвращают оригинальное значение переменной. На следующем, более высоком уровне находятся простые объекты блокировки (*spin locks*), не освобождаемые автоматически при переключениях контекста. Они применяются для простейших операций, таких как добавление или удаление элементов из очереди. На самом высоком уровне находится *блокировка ресурсов*. Это объекты долгосрочного использования, основанные на семантике одной читающей и множества пишущих нитей, которые могут удерживаться при проведении операций блокировки нитей. Система также поддерживает синхронные и асинхронные *межпроцессорные прерывания*, которые могут быть использованы для проведения таких операций, как *распределение тиков таймера и согласование кэша трансляции адресов* (см. раздел 15.9).

В системе Solaris 2.x для увеличения производительности применяются адаптивные объекты блокировки (см. раздел 7.10.3) и турникеты (см. раздел 7.2.3).

ОС поддерживает семафоры, объекты защиты чтения-записи и условные переменные как объекты синхронизации высокого уровня. Для обработки прерываний служат нити ядра, поэтому обработчики прерываний используют те же элементы синхронизации, что и остальная часть ядра, и могут блокировать их по необходимости. Эта возможность подробнее описывалась в разделе 3.6.5.

Реализация любой многопроцессорной системы использует одну из форм простых объектов блокировки для краткосрочной синхронизации на низком уровне. Механизм сна-пробуждения чаще всего поддерживается этими системами (иногда с небольшими изменениями), что защищает от необходимости замены большого количества исходного кода ОС. Основные различия наблюдаются в выборе элементов синхронизации высшего уровня. Первые реализации для IBM/370 и AT&T 3B20A [1] основывались в основном на семафорах. ОС Ultrix [16] использует эксклюзивную блокировку объектов. Ядро системы UTS, созданной Amdahl, основано на условиях [15]. ОС DG/UX применяет для реализации *последовательных объектов блокировки неделимые счетчики событий*, предоставляющие несколько нестандартный способ пробуждения одного процесса.

7.12. Заключение

Проблемы синхронизации в многопроцессорных архитектурах являются более сложными и сильно отличаются от проблем, возникающих при использовании одного процессора. Привлекается большое количество новых решений, таких как механизмы сна-пробуждения, условия, события, объекты защиты чтения-записи и семафоры. Все эти элементы во многом схожи. К примеру, по возможности следует применять семафоры совместно перед условиями и наоборот. Многие из представленных решений не ограничены многопроцессорными системами и могут быть применены при синхронизации на однопроцессорных архитектурах и распределенных системах со слабой связью. Многие многопроцессорные системы UNIX основаны на существующих однопроцессорных вариантах ОС, поэтому для них решение об использовании тех или иных элементов синхронизации сильно зависит от соглашений при переносе на новую платформу. Система Mach и другие, базирующиеся на ней, не зависят от этих соглашений, поэтому разработчики могли выбирать элементы синхронизации на свое усмотрение.

7.13. Упражнения

1. Многие системы поддерживают неделимую функцию, которая выполняет перестановку значения регистра в значение, хранимое в памяти. Как эта функция может быть использована для реализации неделимой операции тестирования и установки?

2. Каким образом можно реализовать неделимую операцию тестирования и установки на машине, поддерживающую связанную загрузку и сохранение по условию?
3. Представьте, что соперничество при попытке обладания критическим участком кода, защищенным семафором, приводит к конвоированию семафора. Если такой участок поделить на две части, каждая из которых будет защищена отдельным семафором, уменьшит ли это вероятность возникновения конвоя?
4. Одним из методов предупреждения возникновения конвоирования является замена семафоров другим механизмом блокировки. Может ли это привести к увеличению степени риска устаревания нитей?
5. Чем отличается счетчик ссылок от совместного объекта блокировки?
6. Создайте объект блокировки ресурса на основе простого объекта mutex и условной переменной, в которой проверке подвергается состояние флага `locked` (см. раздел 7.7.3).
7. Нужно ли удерживать простой объект блокировки, защищающий условие, при сбросе флага (в примере, описанном в предыдущем упражнении)? В работе [15] описывается операция `waitlock()`, использование которой может улучшить алгоритм.
8. Каким образом условные переменные могут предупреждать возникновение проблемы потери сигнала пробуждения нити?
9. Создайте элемент `event` (событие), возвращающий ожидающим нитям значение статуса при совершении события.
10. Представьте объект, к которому часто происходит доступ для чтения и записи. В каких ситуациях целесообразнее защитить его простым объектом mutex, а в каких использовать блокировку чтения-записи?
11. Имеет ли возможность объект защиты чтения-записи блокировать нить? Создайте объект защиты чтения записи, который заставляет нити переходить в режим занятого ожидания при невозможности получить ресурс.
12. Опишите ситуацию, в которой вероятность возникновения взаимоблокировки уменьшается при повышении степени грануляции.
13. Опишите ситуацию, в которой вероятность возникновения взаимоблокировки уменьшается при понижении степени грануляции.
14. Необходимо ли защищать каждый ресурс или переменную ядра многопроцессорной системы объектом блокировки перед доступом к ним? Подсчитайте количество различных типов ситуаций, при которых нить может иметь доступ или изменять объект без его блокировки.
15. Программы управления (*monitors*) — это конструкции, поддерживаемые языками программирования, использующиеся для взаимного ис-

ключения участка кода. При каких ситуациях применение этих механизмов является наиболее естественным?

16. Создайте функции `upgrade()` и `downgrade()` для реализации объектов защиты чтения-записи, представленных в разделе 7.8.2.

7.14. Дополнительная литература

1. Bach, M., and Buroff, S., «Multiprocessor UNIX Operating Systems», AT&T Bell Laboratories Technical Journal, Vol. 63, Oct. 1984, pp. 1733–1749.
2. Birrell, A. D., «An Introduction to Programming with Threads», Digital Equipment Corporation Systems Research Center, 1989.
3. Campbell, M., Barton, R., Browning, J., Cervenka, D., Curry, B., Davis, T., Edmonds, T., Holt, R., Slice, R., Smith, T., and Wescott, R., «The Parallelization of UNIX System V Release 4.0», Proceedings of the Winter 1991 USENIX Conference, Jan. 1991, pp. 307–323.
4. Denham, J. M., Long, P., and Woodward, J. A., «DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation», Digital Technical Journal, Vol. 6, No. 3, Summer 1994, pp. 29–54.
5. Digital Equipment Corporation, «VAX Architecture Reference Manual», 1984.
6. Dijkstra, E. W., «Solution of a Problem in Concurrent Programming Control», Communications of the ACM, Vol. 8, Sep. 1965, pp. 569–578.
7. Eykholt, J. R., Kleinman, S. R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M., and Williams, D., «Beyond Multiprocessing: Multithreading the SunOS Kernel», Proceedings of the Summer 1992 USENIX Conference, Jun. 1992, pp. 11–18.
8. Goble, G. H., «A Dual-Processor VAX 11/780», USENIX Association Conference Proceedings, Sep. 1981.
9. Hoare, C. A. R., «Monitors: An Operating System Structuring Concept», Communications of the ACM, Vol. 17, Oct. 1974, pp. 549–557.
10. Hitz, D., Harris, G., Lau, J. K., and Schwartz, A. M., «Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 285–295.
11. Kelley, M. H., «Multiprocessor Aspects of the DG/UX Kernel», Proceeding of the Winter 1989 USENIX Conference, Jan. 1989, pp. 85–99.
12. Lee, T. P., and Luppi, M. W., «Solving Performance Problems on a Multiprocessor UNIX System», Proceedings of the Summer 1987 USENIX Conference, Jun. 1987, pp. 399–405.

13. National Semiconductor Corporation, «Series 32000 Instruction Set Reference Manual», 1984.
14. Peacock, J. K., Saxena, S., Thomas, D., Yang, F., and Yu, F., «Experiences from Multithreading System V Release 4», The Third USENIX Symposium of Experiences with Distributed and Multiprocessor Systems (SEDM III), Mar. 1992, pp. 77–91.
15. Ruane, L. M., «Process Synchronization in the UTS Kernel», Computing Systems, Vol. 3, Summer 1990, pp. 387–421.
16. Sinkiewicz, U., «A Strategy for SMP ULTRIX», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988, pp. 203–212.
17. UNIX System Laboratories, «Device Driver Reference — UNIX SVR4.2», UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.