Операционные системы

Управление процессами[1]

Соловьев А. В.

ПетрГУ – КИИСиФЭ

(Rev. 2018 06 07)

[1] По материалам «Таненбаум Э., Бос Х. Современные операционные системы. СПб.: Питер, 2015.»

Процессы

Модель процесса (1)

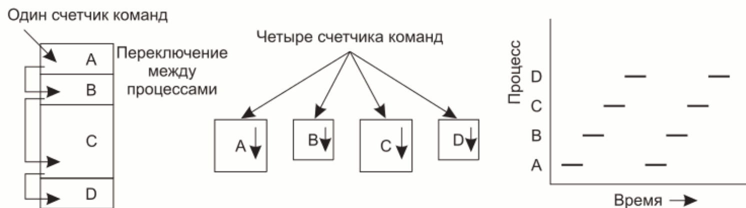
Далее предполагается, что в системе имеется единственный (одноядерный) ЦП.

Всё выполняемое ПО (в т.ч. ОС) – это ряд *последовательных процессов*.

Процесс – это экземпляр выполняемой программы, включая текущие значения счётчика команд, регистров и переменных.

Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. На самом деле настоящий ЦП постоянно переключается между процессами. Такое постоянное переключение между процессами называется *мультипрограммированием*, или *многозадачным режимом работы*.

Модель процесса (2)



В многозадачном режиме скорость, с которой процесс выполняет свои вычисления, не будет одинаковой и, чаще всего, не сможет быть вновь показана, если тот же процесс будет запущен ещё раз.
Поэтому процессы не должны программироваться с использованием каких-либо жёстко заданных предположений относительно времени их выполнения.

Создание процесса (1)

Зачем создаются процессы?

- При старте системы – запуск оболочек и демонов.
- Работающий процесс создаёт вспомог. процессы (стратегия I/O).
- Запрос пользователя на создание нового процесса.
- Инициация пакетного задания.

Технически новый процесс создается за счет уже существующего процесса, который выполняет системный вызов (*fork* или *CreateProcess*).

Список процессов.

Создание процесса (2)

UNIX:
fork создает точную копию вызывающего процесса. Два процесса (родительский и дочерний) имеют единый образ памяти, единые строки описания конфигурации и одни и те же открытые файлы. Обычно после этого дочерний процесс изменяет свой образ памяти, выполняя системный вызов типа *execve*. Т.о. дочерний процесс может управлять своими файловыми дескрипторами после разветвления, но перед выполнением *execve* с целью выполнения перенаправления stdin, stdout, stderr.

Win32:
CreateProcess создается процесс, и в него загружается нужная программа. У этого вызова имеется 10 параметров, включая выполняемую программу, параметры командной строки для этой программы, различные параметры безопасности, биты, управляющие наследованием открытых файлов, информацию о приоритетах, спецификацию окна, создаваемого для процесса (если оно используется), и указатель на структуру, в которой вызывающей программе будет возвращена информация о только что созданном процессе.

Создание процесса (3)

После создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами. Если какой-нибудь процесс изменяет слово в своем адресном пространстве, другим процессам эти изменения не видны.

В UNIX первоначальное состояние адресного пространства дочернего процесса является копией адресного пространства родительского процесса. Также используется **copy-on-write**.

В Windows адресные пространства родительского и дочернего процессов различаются с самого начала.

Завершение процесса

UNIX	Win32	Причина
exit	*ExitProcess*	Процесс выполнил свою задачу и добровольно выходит.
		Процесс обнаружил ошибку и решает добровольно закончить работу.
kill	*TerminateProcess*	В процессе возникает фатальная ошибка и ОС принудительно завершает процесс [UNIX: сигналы (SIGSEGV, SIGILL). Win32: UAE].
		Другой процесс решает принудительно завершить данный процесс.

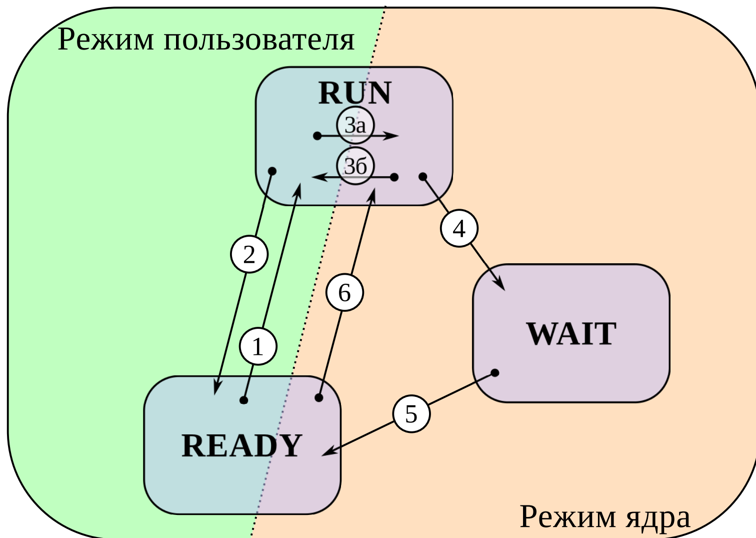
Контейнер завершённого процесса (в UNIX называется «зомби») хранит статус выхода.

Иерархия процессов

В UNIX процесс, все его дочерние процессы и более отдаленные потомки образуют *группу процессов*.
Сигнал (например SIGINT – прерывание с клавиатуры) может посылаться всем процессам группы. Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию (обычно – завершиться).

В Windows не существует понятия иерархии процессов, и все процессы являются равнозначными. (Но существуют дескрипторы процессов.)

Состояния процессов



Реализации процессов

Процессы				
0	1	…	$n-2$	$n-1$
Планировщик				

Планировщик – самый нижний уровень ОС (обработка прерываний, запуск, останов и планирование процессов). Вся остальная часть ОС может быть структурирована в виде последовательных процессов.

Таблица процессов обычно содержит:

- Поля, необходимые для управления процессом: РОН, счётчик команд, специальные регистры, указатель стека, состояние процесса, приоритет, параметры планирования, PID, PPID, группа процесса, сигналы, время запуска процесса, использованное время и др.
- Поля, необходимые для управления памятью: дескриптор текстового сегмента, дескриптор сегмента данных, дескриптор сегмента стека.
- Поля, необходимые для управления файлами: корневой каталог, рабочий каталог, дескрипторы открытых файлов, UID, GID.

Реализации процессов: обработка прерывания

1. ЦП помещает в стек счётчик команд.
2. ЦП загружает новый счётчик команд из вектора прерывания.
3. Планировщик сохраняет контекст и устанавливает новый стек.
4. Планировщик запускает процедуру, обслуживающую данное прерывание.
5. Планировщик принимает решение, какой процесс запускать следующим.
6. Управление возвращается в новый текущий процесс.

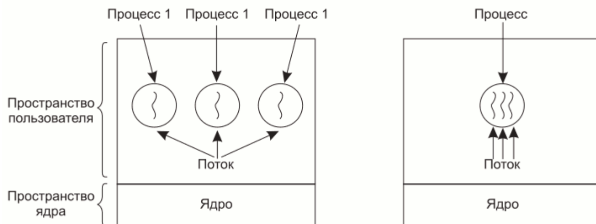
Потоки (Threads, LWP – Lightweight processes)

Потоки: применение

В одном и том же адресном пространстве несколько потоков управления выполняются (квази)параллельно, или несколько потоков одного процесса имеют общее адресное пространство.

- Во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в (квази)параллельном режиме.
- Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов («потоки – легковесные процессы»).
- Потоки позволяют эффективнее использовать возможности многопроцессорных систем, где есть реальная возможность параллельных вычислений.

Классическая модель потоков (1)

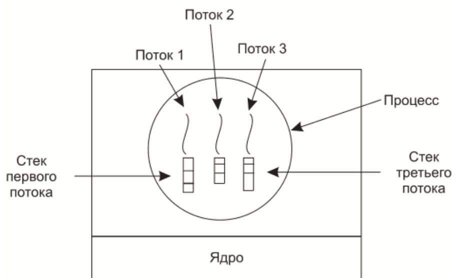


Процессы используются для группировки в единое образование различных ресурсов (адресное пространство, открытые файлы, сигналы, учётная информация и т. д.). Процесс – это примитив–контейнер ресурсов.

Поток выполнения включает счётчик команд, регистры, стек. Поток – это примитив–единица исполнения. Поток выполняется в рамках какого-нибудь процесса. Более того, в рамках одного процесса может выполнятся несколько потоков.

Классическая модель потоков (2)

Элементы, специфичные для процесса	Элементы, специфичные для потока
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные сигналы	
Обработчики сигналов	
Учетная информация	



Общее адресное пространство – один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует.

Иногда потоки имеют иерархическую структуру (родитель–ребёнок), но чаще всего такие взаимоотношения отсутствуют и все потоки считаются равнозначными.

Потоки в POSIX (pthreads)

pthread_create	Создание нового потока
pthread_self	Получение собственного идентификатора потока
pthread_exit	Завершение работы потока
pthread_join	Ожидание выхода из указанного потока
pthread_detach	Не сохранять статус завершения потока
pthread_yield	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока
pthread_cancel	Прекращение работы указанного потока
pthread_setcancelstate	Разрешение или запрещение cancel
pthread_setcanceltype	Момент срабатывания cancel (deferred, asynchronous)
pthread_testcancel	Проверка, не вызван ли cancel

Реализация потоков в режиме пользователя

User-level threads (ULT), или модель N:1

- Ядро управляет обычными, однопотоковыми процессами. Потоки реализуются с помощью библиотеки. Можно реализовать в ОС, которая не поддерживает потоки.
- Процедура, которая сохраняет состояние потока, и планировщик – это всего лишь локальные процедуры, поэтому их вызов намного более эффективен, чем вызов ядра (не требуется переключение контекста, кэш в памяти не нужно сбрасывать на диск и т. д.) Благодаря этому планировщик потоков работает очень быстро.
- Каждый процесс может иметь собственные настройки алгоритма планирования для потоков.
- Проблема блокирующих системных вызовов (решается «обёртками»).
- Проблема задействования нескольких ядер в SMP.

Реализации потоков (сравнение)

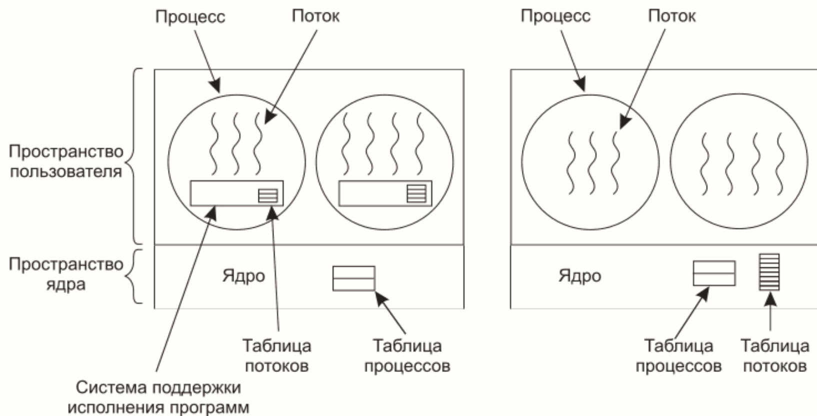


Рис.: В режиме пользователя (слева) и в режиме ядра (справа)

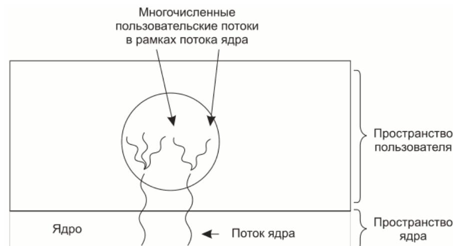
Реализация потоков в режиме ядра

Kernel-level threads (KLT), или модель 1:1

- Блокирующие системные вызовы для KLT: когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же самого процесса, либо поток из другого процесса. Для ULT система поддержки исполнения программ работает с запущенными потоками собственного процесса до тех пор, пока ядро не заберёт у неё центральный процессор (или не останется ни одного готового к выполнению потока).
- Поскольку создание и уничтожение потоков в ядре требует относительно более весомых затрат, некоторые KLT-системы используют свои потоки повторно. При уничтожении потока он помечается как неспособный к выполнению, но это не влияет на его структуру данных, имеющуюся в ядре. Чуть позже, когда должен быть создан новый поток, вместо этого повторно активируется старый поток, что приводит к экономии времени. Повторное использование потоков допустимо и в ULT-системах, но для этого нет достаточно веских оснований, поскольку издержки на управление потоками там значительно меньше.

Гибридная реализация

Модель M:N



Некоторое количество (M) ULT отображаются на некоторое число (N) сущностей ядра или «виртуальных процессоров» (KLT). Вообще говоря является более сложной для реализации: два уровня планировщиков. В модели M:N библиотека потоков отвечает за планирование ULT на имеющихся планируемых сущностях. При этом переключение контекста потоков делается очень быстро, поскольку модель позволяет избежать системных вызовов. Тем не менее, увеличивается сложность, вероятна неоптимальность планирования без обширной (и дорогой) координации между пользовательским планировщиком и планировщиком ядра.

Примеры реализаций

ULT (N:1)

- POSIX Threads на старом ядре Linux ($>$ 1.3.56).
- GNU Portable Threads

KLT (1:1)

- Light Weight Kernel Threads (LWKT) в различных версиях BSD
- Linux
- Windows

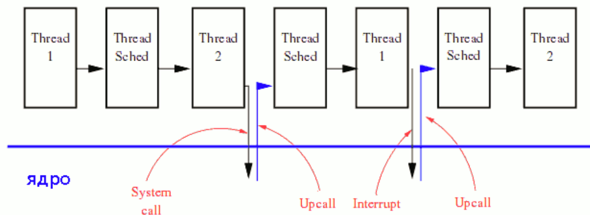
Гибридные (M:N)

- Некоторые Java VM
- Windows 7

Файберы (fibers).

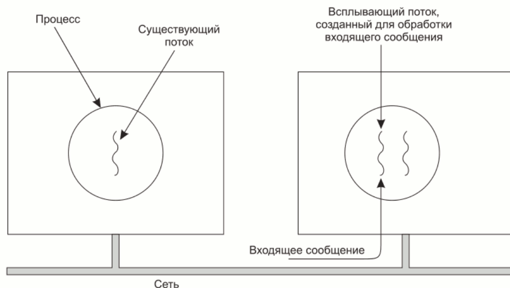
Методы повышения производительности потоков (1)

- *Активация планировщика (scheduler activations)*. Ядро назначает каждому процессу определенное количество виртуальных процессоров, а системе поддержки исполняемых программ (ULT) разрешается распределять потоки по процессорам. Когда ядро знает, что поток заблокирован, оно уведомляет принадлежащую процессу систему ULT, передавая через стек в качестве параметров номер данного потока и описание произошедшего события. Уведомление осуществляется «вызовом наверх» *upcall* (похоже на обработчик сигналов в UNIX). Пример: NetBSD.



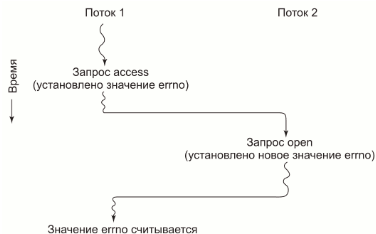
Методы повышения производительности потоков (2)

- *Всплывающие потоки (pop-up threads)*. Поступление сообщения (события) вынуждает систему создать новый поток для его обработки. Такие потоки не имеют «прошлого» (регистров, стека, контекста), создаются с чистого листа, и поэтому быстро. В результате использования всплывающих потоков задержку между поступлением и началом обработки сообщения можно свести к минимуму. Может быть запущен в пространстве ядра (pros & cons).



Проблемы многопоточного кода (1)

- Несинхронизованный доступ потоков к глобальным переменным может приводить к нарушению согласованности их состояния (см. рис.).
- Нереентерантные процедуры.



Библиотечные процедуры могут вызываться одновременно несколькими потоками. Не все процедуры к этому готовы (например, используют статические переменные). Большинство стандартных функций языка Си используют статические буферы и поэтому нереентерантны! Сравните: localtime() и localtime_r(), strtok() и strtok_r(), gethostbyname() и gethostbyname_r().

Thread Local Storage.

Доступ к библиотечным функциям через примитивы синхронизации (mutex).

Проблемы многопоточного кода (2)

- Что делает вызов fork() в многопоточном процессе?

  POSIX/IEEE 1003.1 в редакции 2004 г. предусматривает:

  - В дочернем процессе будет существовать только один поток.
  - Состояние примитивов синхронизации потоков копируется как есть. Т. о. если в родительском процессе мьютекс был заблокирован каким-то потоком, в дочернем он также будет заблокирован возможно несуществующим потоком (т. е. будет испорченным).
  - Предложен вызов thread_atfork(), устанавливающий обработчики, которые срабатывают в родительском и дочернем процессах до и после вызова fork(), способные выполнить корректирующие действия с примитивами синхронизации, чтобы привести их в пригодное состояние.
  - В дочернем процессе гарантируется безопасный вызов exec().

- Работа с сигналами в многопоточном процессе.

  - Обработчики сигналов общие для всех потоков процесса.
  - У каждого потока может быть своя маска блокируемых сигналов.
  - При возникновении сигнала он доставляется случайному потоку процесса, который не блокирует этот сигнал.

Взаимодействие процессов – IPC (Inter-Process Communication)

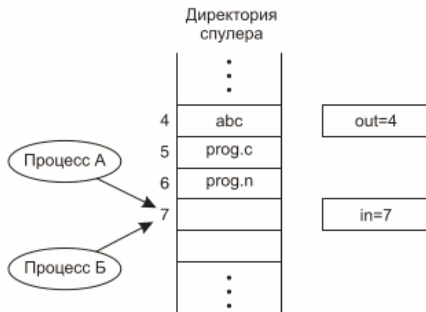
Взаимодействие процессов – IPC

- Как передавать информацию между процессами?
- Как синхронизовать доступ процессов к критическому ресурсу?
  (Актуально также для потоков)
- Как обеспечить правильную последовательность выполнения процессов (задача «читатель–писатель»)?
  (Актуально также для потоков)

Состязательная ситуация

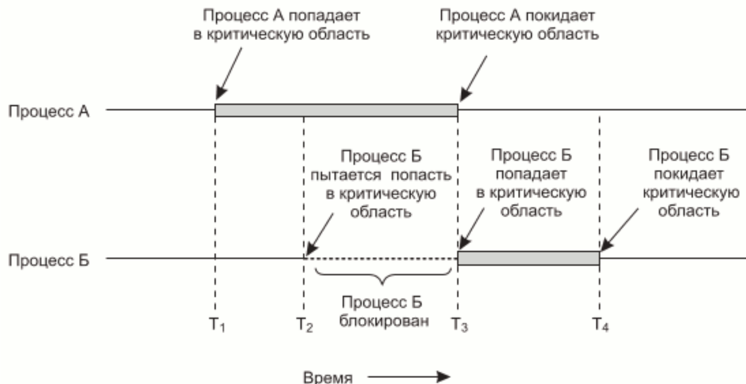
Race conditions
(«Гонки»)

Два или более процесса считывают или записывают какие-нибудь общие данные, окончательный результат зависит от того, какой процесс и когда именно выполняется.



Критические области

Взаимное исключение: если общие данные (файл) используются одним процессом, возможность их использования всеми другими процессами исключается.
Та часть программы, в которой используется доступ к общим данным, называется *критической областью*, или *критической секцией*.
Чтобы избежать «гонок», никакие два процесса не должны находиться одновременно в своих критических областях.



Методы взаимного исключения: запрещение прерываний

Запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области.

- Самое простое решение!
- Требуются привилегии (плохое решение для пользовательских процессов, но пригодно для ядра в 1-процессорных ОС).
- Исключаются также процессы, которые не собираются входить в критическую секцию.
- Непригодно для многопроцессорных систем.

Методы взаимного исключения: блокирующие переменные

```
int lock = 0;
...
while (lock != 0); /* ожидание */
lock = 1;          /* начало критической секции */
...
lock = 0;          /* конец критической секции */
```

Это неправильная реализация.
Она также подвержена «гонкам»!

Методы взаимного исключения: алгоритм Петерсона

Пример для двух процессов (потоков):

```
int turn = 0;
bool flag[2] = {false, false};
void enter_region(int process) {
    int other = 1 - process;
    flag[process] = true;
    turn = process;
    while (turn == process && flag[other]);
}
void leave_region(int process) {
    flag[process] = false;
}
```

- Масштабируется на N процессов.
- Ожидание за счёт холостых циклов.
- Корректно работает, только если ЦП не переупорядочивает исполнение инструкций.

Методы взаимного исключения: инструкции TSL, XCHG

TSL (Test and Set Lock) – «проверь и установи блокировку».
ЦП считывает содержимое слова памяти в указанный регистр, а по адресу памяти записывает ненулевое значение. При этом гарантируются неделимость операций чтения слова и сохранение в нём нового значения – никакой другой процесс не может получить доступ к слову в памяти, пока команда не завершит свою работу. ЦП, выполняющий команду TSL, блокирует шину памяти, запрещая другим ЦП доступ к памяти до тех пор, пока не будет выполнена эта команда.

```
enter_region:
    TSL REG,LOCK
    CMP REG,#0
    JNE enter_region
    RET
leave region:
    MOVE LOCK,#0
    RET
```

Для процессоров Intel x86 существует команда XCHG (а с i486 – CMPXCHG).

```
enter_region:
    MOV AX,1
    XCHG AX,[LOCK]
    CMP AX,0
    JNE enter_region
    RET
leave region:
    MOVE [LOCK],0
    RET
```

Семафоры (1)

Активное ожидание, или *спин-блокировка* впустую тратит процессорное время, порождает проблему *инверсии приоритетов*.

Э.В.Дейкстра (1965 г.): *Семафор* – счётчик с двумя атомарными операциями: *Proberen* («пытаться») и *Verhogen* («поднимать выше»).

P: Если счётчик больше нуля (семафор открыт), это значение уменьшается на 1, работа продолжается. Если значение равно 0 (семафор закрыт), процесс приостанавливается.

V: Значение счётчика увеличивается на 1. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции P, система выбирает один из них и позволяет ему завершить его операцию P.

Семафоры (2)

Гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена или заблокирована.
Обеспечивается либо запретом прерываний (подходит для 1-процессорных систем), либо спин-блокировками (на основе TSL/XCHG).

Семафоры, инициализированные значением 1 и используемые двумя или более процессами для исключения их одновременного нахождения в своих критических областях, называются *двоичными семафорами*.

Пример использования семафоров: задача «производитель–потребитель».

Мьютексы (mutex)

Мьютекс (MUTual EXclusion) – упрощённая разновидность двоичного семафора.
Реализуется примерно так же, как спин-блокировка, только вместо активного ожидания используется системный вызов с передачей управления планировщику.

Фьютекс (Fast User-space muTEX) – разновидность мьютекса, реализованная в пользовательском пространстве.

Мьютексы в pthreads:

pthread_mutex_init	Создание мьютекса
pthread_mutex_destroy	Уничтожение существующего мьютекса
pthread_mutex_lock	Овладение блокировкой или блокирование потока
pthread_mutex_trylock	Овладение блокировкой или выход с ошибкой
pthread_mutex_unlock	Разблокирование

Условные переменные (УП)

УП (condition variable) позволяют потокам блокироваться до выполнения конкретных условий.
Основания для блокирования и ожидания не являются частью протокола ожиданий и отправки сигналов.
УП позволяют осуществлять ожидание и блокирование как неделимые операции.

Реализация в pthreads:

pthread_cond_init	Создание УП
pthread_cond_destroy	Уничтожение УП
pthread_cond_wait	Блокировка в ожидании сигнала
pthread_cond_signal	Сигнализирование другому потоку и его активизация

УП всегда должна быть ассоциирована с каким-то мьютексом, чтобы избежать гонок, когда один поток ещё только готовится ждать, а другой поток уже сигнализирует.
wait: 1) разблокирует мьютекс, 2) ожидает сигнализации, 3) блокирует мьютекс.
Вызов signal должен быть защищён этим мьютексом.

Передача сообщений

Семафоры и мьютексы пригодны только для систем с общей памятью (SMP/UMA), но не подходят для систем с распределённой памятью (MPP, кластеры, grid). Кроме того, не позволяют осуществлять информационный обмен между процессами.

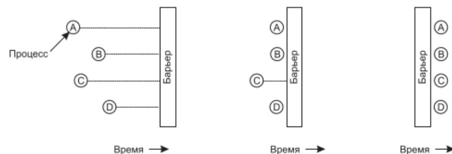
Передача сообщений – метод взаимодействия процессов с использованием примитивов (системных вызовов) send() и receive(). Проблемы системы передачи сообщений:

- Гарантированность доставки.
- Идентификация отправителя и получателя.
- Аутентификация.
- Синхронизация производительности отправителя и получателя (через буфер или рандеву).

Другие способы синхронизации

Монитор – коллекция переменных и структур данных, сгруппированных вместе в специальную разновидность модуля или пакета процедур. В любой момент времени в мониторе может быть активен только один процесс. Пример – synchronized в Java.

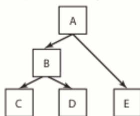
Барьер – примитив для синхронизации группы процессов. Ни один из процессов не может перейти к следующей фазе, пока все процессы не будут готовы перейти к следующей фазе. В конце фазы ставится барьер. Когда процесс достигает барьера, он блокируется до тех пор, пока этого барьера не достигнут все остальные процессы. Пример: OpenMP.



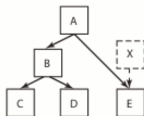
Работа без блокировок: RCU (Read – Copy – Update)

Обеспечивается, чтобы каждый считывающий процесс читал либо старую, либо новую версию данных, но не некую непонятную комбинацию из старой и новой версий.

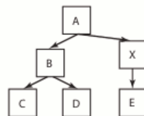
Добавление узла:



(а) исходное дерево

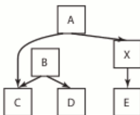


(б) инициализация узла X и подключение E к X. На любых читателей, находящихся в A и E, это не повлияет

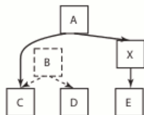


(в) после полной инициализации X подключение X к A. Читатели, находящиеся в это время в E, прочитают старую версию дерева, а читатели, находящиеся в A, получат новую версию

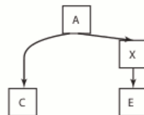
Удаление узлов:



(г) отсоединение B от A. Заметьте, что в B все еще могут быть читатели. Все читатели в B будут видеть старую версию, а все читатели, находящиеся в это время в A, увидят новую версию



(д) ожидание полной уверенности в том, что все читатели покинули B и C. Больше доступа к этим узлам не будет



(е) теперь B и D можно спокойно удалить

Планирование (scheduling)

Термины и вопросы планирования

Планировщик (scheduler) решает, какой из процессов в состоянии готовности займёт ЦП (*алгоритм планирования*).

Когда планировать?

- При создании нового процесса.
- При завершении процесса.
- При блокировании процесса.
- При возникновении аппаратного прерывания.

Категории алгоритмов планирования (по реакции на прерывание от таймера):

- неприоритетные алгоритмы (кооперативная, или невытесняющая многозадачность) – подходят для пакетных систем и некоторых СРВ;
- приоритетные алгоритмы (вытесняющая многозадачность) – подходят для интерактивных систем и СРВ.

Задачи алгоритма планирования

- Равнодоступность – предоставление каждому процессу справедливой доли времени ЦП.
- Принуждение к определенной политике – наблюдение за выполнением установленной политики.
- Баланс – поддержка загруженности всех составных частей системы.
- (ПС) Производительность – выполнение макс. кол-ва заданий в час.
- (ПС) Оборотное время – минимизация времени между представлением задачи и её завершением.
- (ПС) Использование ЦП – поддержка постоянной загруженности ЦП.
- (ИС) Время отклика – быстрый ответ на запросы.
- (ИС) Пропорциональность – оправдание пользовательских надежд.
- (СРВ) Соблюдение предельных сроков – предотвращение потери данных.
- (СРВ) Предсказуемость – предотвращение ухудшения качества в мультимедийных системах.

Планирование в пакетных системах

Алгоритм FIFO – «первым пришёл, первым обслужен» (first-come, first-served) (невытесняющий).

- Простота восприятия и реализации.
- Низкая эффективность на смеси процессов, ограниченных скоростью вычислений и скоростью работы у-в вв/выв.

Алгоритм «сначала самое короткое задание» (shortest job first) (невытесняющий).

- Даёт минимальное оборотное время.
- Время выполнения заданий нужно знать заранее.
- Оптимален только в том случае, если все задания доступны одновременно.

Алгоритм «приоритет наименьшему времени выполнения» (shortest operating time) (вытесняющий).

- Выбирается задание с наименьшим оставшимся временем выполнения.
- Время выполнения заданий нужно знать заранее.
- Позволяет быстро обслужить новое короткое задание.

Планирование в интерактивных системах (1)

Алгоритм циклического планирования (round robin) (вытесняющий).

Каждому процессу назначается определённый интервал времени, называемый его *квантом*, в течение которого ему предоставляется возможность выполнения. Если процесс к завершению кванта времени всё ещё выполняется, то ресурс ЦП у него отбирается и передаётся другому процессу. Если процесс переходит в заблокированное состояние или завершает свою работу до истечения кванта времени, то переключение ЦП на другой процесс происходит именно в этот момент.

Выбор длины кванта:

- слишком короткий – слишком частые переключения контекста (снижение эффективности использования ЦП);
- слишком длинный – слишком вялая реакция на короткие интерактивные запросы.

Планирование в интерактивных системах (2)

Алгоритм планирования с приоритетами (вытесняющий).

Каждому процессу присваивается значение приоритета и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.

Необходимо предотвращать бесконечное выполнение высокоприоритетных процессов:

- планировщик понижает уровень приоритета текущего выполняемого процесса с каждым сигналом таймера;
- каждому процессу может быть выделен максимальный квант допустимого времени выполнения, когда квант времени будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет.

Приоритеты могут присваиваться процессам в статическом (пример: nice) или в динамическом режиме (пример: $pri = 1/f$, f – часть последнего кванта времени, использованного этим процессом).

Разделение политики и механизма планирования.

Планирование в интерактивных системах (3)

Алгоритм «следующий – самый короткий» (shortest process next).

- Производится оценка времени выполнения процесса T_i для конкретного терминала на основе предыдущей оценки T_{i-1} и фактически затраченного времени $\hat{T}_{i-1}$: $T_i = \alpha T_{i-1} + (1-\alpha)\hat{T}_{i-1}$. Затем применяется принцип «самый короткий – первым».

Алгоритм гарантированного планирования.

- Для N работающих процессов каждый из процессов получает долю $1/N$ от общего числа процессорных циклов.

Алгоритм лотерейного планирования.

- Процессам раздаются «лотерейные билеты» на доступ к различным системным ресурсам, в том числе к процессорному времени. Когда планировщику нужно принимать решение, в случайном порядке выбирается лотерейный билет, и ресурс отдается процессу, обладающему этим билетом. Более важным процессам, чтобы повысить их шансы на выигрыш, могут выдаваться дополнительные билеты.

Планирование в системах реального времени

СРВ обычно делятся на жесткие СРВ (*системы жесткого реального времени*), в которых соблюдение крайних сроков обязательно, и гибкие СРВ (*системы мягкого реального времени*), в которых нерегулярные несоблюдения крайних сроков нежелательны, но вполне допустимы.

При обнаружении внешнего события планировщик должен так спланировать работу процессов, чтобы были соблюдены все крайние сроки.
События: периодические (происходящие регулярно) или апериодические (происходящие непредсказуемо).
СРВ называется *планируемой*, если удовлетворяет критерию: $\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1$
(P_i – период события i, C_i – время его обработки).

Алгоритмы планирования работы СРВ могут быть статическими (решения по планированию задаются до запуска системы) или динамическими (решения по планированию принимаются во время работы системы).