1-15(3)

*1.  Потоки исполнения (нити), их характеристики (атрибуты) и применение.*

Лекция 14

В традиционных операционных системах у каждого процесса есть адресное пространство и единственный поток управления. Фактически это почти что определение процесса. Тем не менее нередко возникают ситуации, когда неплохо было бы иметь в одном и том же адресном пространстве несколько потоков управления, выполняемых квазипараллельно, как будто они являются чуть ли не обособленными процессами, за исключением общего адресного пространства.

У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя поток может выполняться в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться отдельно.

Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний:

- выполняемый

- заблокированный

- готовый

- завершенный.

Переходы между состояниями потока аналогичны переходам между состояниями процесса.

*Интерактивное приложение*

Текстовый процессор может быть написан как двухпоточная программа. Один из потоков взаимодействует с пользователем, а другой занимается формированием представления в фоновом режиме. Как только предложение с первой страницы, например, будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за представление, переформатировать документ. Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления. Может так случиться, что переформатирование закончится до того, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена. Многие текстовые процессоры автоматически сохраняют рабочий файл на диск каждые несколько минут — еще один поток может заниматься созданием резервных копий на диске.

*Веб-сервер*

Поступают запросы на страницы, и запрошенные страницы отправляются обратно клиентам. Некоторые страницы запрашиваются чаще других. Это обстоятельство для повышения производительности за счет размещения содержания часто используемых страниц в основной памяти, чтобы исключить необходимость обращаться за ними к диску (кеширование).

Один из потоков — диспетчер — читает входящие рабочие запросы из сети. После анализа запроса он выбирает простаивающий (то есть заблокированный) рабочий поток и передает ему запрос, возможно, путем записи указателя на сообщение в специальное слово, связанное с каждым потоком. Затем диспетчер пробуждает спящий рабочий поток, переводя его из заблокированного состояния в состояние готовности.

При пробуждении рабочий поток проверяет, может ли запрос быть удовлетворен из кэша веб-страниц, к которому имеют доступ все потоки. Если нет, то он, чтобы получить веб-страницу, приступает к операции чтения с диска и блокируется до тех пор, пока не завершится дисковая операция. Когда поток блокируется на дисковой операции, выбирается выполнение другого потока, возможно, диспетчера, с целью получения следующей задачи или, возможно, другого рабочего потока, который находится в готовности к выполнению.

*Обработка большого объема данных*

Еще одним примером, подтверждающим пользу потоков, являются приложения, предназначенные для обработки очень большого объема данных.

При обычном подходе блок данных считывается, после чего обрабатывается, а затем снова записывается. При доступности лишь блокирующих вызовов процесс блокируется и при поступлении данных, и при их возвращении. Простой центрального процесса при необходимости в большом объеме вычислений слишком расточителен и его по возможности следует избегать. Проблема решается с помощью потоков. Структура процесса может включать входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные во входной буфер. Обрабатывающий поток извлекает данные из входного буфера, обрабатывает их и помещает результат в выходной буфер. Выходной буфер записывает эти результаты обратно на диск. Таким образом, ввод, вывод и обработка данных могут осуществляться одновременно.

*2.  Преимущества и недостатки потоков по сравнению с процессами. Использование объектов потоками и общий программный интерфейс (create(), exit(), join(), yield()).*

Лекция 15 16-дальше

Преимущества:

- потоки довольно просто обмениваются данными по сравнению с процессами;

- создавать потоки для операционной системы гораздо проще и быстрее, чем создавать процессы.

Недостатки:

- при программировании мультипоточного приложения необходимо обеспечить т.н. «потоковую безопасность» функций — thread safety. К приложениям, выполняющимся в виде множества процессов, таких требований не выдвигается.

- ошибка в одном из потоков может повредить остальные, поскольку потоки делят общее адресное пространство. Процессы более изолированы друг от друга.

- потоки конкурируют друг с другом в адресном пространстве. Стек и локальная память потока захватывают часть виртуального адресного пространства процесса и делают его недоступным для других потоков. Для встраиваемых устройств такое ограничение может иметь существенное значение.



В случае многопоточности, процесс обычно начинается с использования одного потока. Этот поток может создавать новые потоки, вызвав библиотечную процедуру thread\_create(). В параметре thread\_create() обычно указывается имя процедуры, которая запускается в новом потоке. Создающий поток обычно получает идентификатор потока, который дает имя новому потоку.

Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотечной процедуры, к примеру thread\_exit(). После этого он прекращает свое существование и больше не фигурирует в работе планировщика.

В некоторых случаях какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока, для чего используется thread\_join(). Эта процедура блокирует вызывающий поток до тех пор, пока не будет осуществлен выход из другого (указанного) потока. В этом отношении создание и завершение работы потока очень похожи на создание и завершение работы процесса при использовании примерно одних и тех же параметров.

Другой распространенной процедурой, вызываемой потоком, является thread\_yield(). Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока. Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности. Для потоков важно проявлять вежливость и время от времени добровольно уступать центральный процессор, чтобы дать возможность выполнения другим потокам.

Есть и другие вызываемые процедуры, которые позволяют одному потоку ожидать, пока другой поток не завершит какую-нибудь работу, а этому потоку — оповестить о том, что он завершил определенную работу, и т. д.

*3.  Потоки (нити) исполнения POSIX и их атрибуты. Понятие потокобезопасности и уступчивости. Обработчики очистки.*

Лекция 15

В POSIX.1 определён набор интерфейсов (функции, заголовочные файлы) для работы с потоками, более известными как потоки POSIX или Pthreads. В одном процессе может быть несколько потоков, которые выполняют одну программу. Эти потоки работают с общей глобальной памятью (сегментами данных и кучи), но у каждого потока есть собственный стек (автоматические переменные).

Также, в POSIX.1 требуется, чтобы потоки имели общий диапазон других атрибутов (т. е., эти атрибуты процесса, а не потоков):

- идентификатор процесса;

- идентификатор родительского процесса;

- идентификатор группы процессов и сеанса;

- управляющий терминал;

- идентификаторы пользователя и группы;

- открытые файловые дескрипторы;

- обычные блокировки (смотрите fcntl(2));

- обработчики сигналов;

- маска создания режима доступа к файлу (umask(2));

- текущий каталог (chdir(2)) и корневой каталог (chroot(2));

- интервальные таймеры (setitimer(2)) и таймеры POSIX (timer\_create(2));

- значение уступчивости (setpriority(2));

- ограничения по ресурсам (setrlimit(2));

- измерители потребления времени ЦП (times(2)) и ресурсов (getrusage(2));

Как и для стека, в POSIX.1 определены другие атрибуты, которые уникальны в каждом потоке:

- идентификатор потока (тип данных pthread\_t);

- маска сигналов (pthread\_sigmask(3));

- переменная errno (определена как макрос, задающий для каждого потока собственное l-value);

- альтернативный стек сигнала (sigaltstack(2));

- алгоритм и приоритет планирования реального времени (sched(7));

Потокобезопасная функция — это функция, которую можно безопасно (т. е., это приведёт к единым результатам независимо от окружения) вызывать из нескольких потоков одновременно.

Безопасная асинхронная отменяемая функция (async-cancel-safe function) — это функция, которую можно безопасно вызывать в приложении, в котором разрешено асинхронная отмена (pthread\_setcancelstate(3)).

В POSIX.1 определено, что некоторые функции должны, а несколько других могут было точками отмены. Если поток отменяем, его тип отменяемости откладывается, и ожидается запрос отмены потока, затем поток отменяется, когда он вызывает функцию, которая является точкой отмены.

Уступчивость

В ядре работают специальные алгоритмы, которые управляют переключением процессов/потоков и распределяют между ними системные ресурсы. Есть возможность влиять на этот механизм.

При распределении системных ресурсов ядро по умолчанию рассматривает процессы/потоки как равные. Но можно сообщить ядру, что некоторый процесс/поток желает уступить часть своего права на системные ресурсы. В таком случае говорят, что уступчивость процесса увеличивается. Так же встречается понятие «приоритет процесса/потока», характеризующий величину, обратную уступчивости.

Как измеряется уступчивость процесса, зависит от реализации. В Linux уступчивость процесса измеряется целыми числами в диапазоне от –20 до 19 По умолчанию процессы выполняются с уступчивостью 0 Запрос на получение пониженной уступчивости (повышение приоритета) из отрицательного диапазона (от –20 до –1) может выполнять только суперпользователь. Диапазон от 0 до 19 доступен всем пользователям. Получается, что обычный пользователь может только повышать уступчивость процессов. Для изменения уступчивости запускаемого процесса предназначена команда nice(1,1p).

Функции очистки ключей гарантируют, что в случае завершения или отмены потока не произойдет потери ресурсов. Но иногда возникает необходимость в создании функции, которая будет связана не с ключом, дублируемым между потоками, а с обычным ресурсом. Такая функция называется *обработчиком очистки*.

Обработчик очистки — это временная мера, требуемая только тогда, когда поток завершается или отменяется, не закончив выполнять определенный участок кода. При нормальных обстоятельствах ресурс должен удаляться явно. Для регистрации обработчика следует вызвать функцию pthread\_cleanup\_push(), передав ей указатель на обработчик и значение его аргумента. Каждому такому вызову должен соответствовать вызов функции pthread\_cleanup\_pop(), которая отменяет регистрацию обработчика.

*4.  Организация программы, использующей потоки.*

Лекция 14-15

Новый поток создается функцией pthread\_create(), которой передается имя функции, с которой начнется выполнение потока, — потоковая функция. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции. Поток завершает выполнение задачи когда:

- потоковая функция выполняет return и возвращает результат произведенных вычислений;

- в результате вызова завершения исполнения потока pthread\_exit();

- в результате вызова отмены потока pthread\_cancel();

- один из потоков совершает вызов exit();

- основной поток в функции main() выполняет return, и в таком случае все нити процесса резко сворачиваются.

Если основной поток в функции main() выполнит pthread\_exit() вместо просто exit() или вместо return, остальные потоки продолжат исполняться.

Ожидание потока

Чтобы синхронизировать потоки, используется функция pthread\_join(). Она ожидает завершения указанного потока. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение. Для досрочного завершения потока можно воспользоваться функцией pthread\_cancel(). Функция pthread\_cancel() возвращается сразу, но этот не означает, что поток будет в обязательном порядке завершен досрочно — поток не только может самостоятельно выбрать момент завершения в ответ на вызов pthread\_cancel(), но и его проигнорировать. Вызов функции pthread\_cancel() следует рассматривать как запрос на выполнение досрочного завершения потока. Поэтому, если важно, чтобы поток был удален, нужно дождаться его завершения функцией pthread\_join().

*5.  Стандартные сигналы. Базовая обработка и функция signal().*

Лекция 6

Стандартные сигналы — особые виды сигналов со стандартными названиями и значениями. В этом разделе перечислены названия различных стандартных типов сигналов и описано, какие события они означают. Каждое имя сигнала — это макрос, обозначающий положительное целое число — «номер сигнала» для данного типа сигнала. Программы никогда не должны делать никаких предположений о числовом коде для определенного вида сигнала, а вместо этого всегда обращаться к ним по именам, определенным ниже. Это связано с тем, что номер для данного вида сигнала может варьироваться от системы к системе, но значения названий стандартизированы и довольно единообразны.

Имена сигналов определены в заголовочном файле signal.h.

Значение этой символьной константы — общее количество определенных сигналов. Поскольку номера сигналов назначаются последовательно, «NSIG» также на единицу больше, чем наибольший определенный номер сигнала.

- сигналы программных ошибок;

- сигналы завершения;

- аварийные сигналы (Alarm Signals);

- асинхронные сигналы ввода/вывода;

- сигналы управления заданиями;

- сигналы ошибок обслуживания;

- прочие сигналы.

Функция signal() предоставляет простой интерфейс для установки действия для определенного сигнала. Функция и связанные с ней макросы объявлены в заголовочном файле signal.h.

sighandler\_t — тип функции-обработчика сигналов

#include <signal.h>

typedef void (\*sighandler\_t)(int); // ANSI C

sighandler\_t signal(int signum, sighandler\_t handler);

Обработчики сигналов принимают один целочисленный аргумент, определяющий номер сигнала, и имеют тип возвращаемого значения void. Поэтому функции-обработчики должны выглядеть, как:

void handler\_function(int signum) { ... }

Имя sighandler\_t является расширением GNU.

signal() — установить обработчик сигнала

#include <signal.h>

sighandler\_t signal(int SIGNUM, sighandler\_t ACTION)

Функция signal() устанавливает ACTION как действие для сигнала SIGNUM.

Первый аргумент, SIGNUM, определяет сигнал, поведение которого вы хотите контролировать, и должен быть номером сигнала. Правильный способ указать номер сигнала — использовать одно из символических имен сигналов. Не следует использовать явное число, потому что числовой код для данного вида сигнала может варьироваться от одной ОС к другой. Второй аргумент, ACTION, указывает действие, используемое для сигнала SIGNUM.

Программа, как правило, не должна игнорировать сигналы, которые представляют серьезные события или обычно используются для запроса завершения. Cигналы SIGKILL или SIGSTOP не возможно игнорировать. Можно игнорировать сигналы об ошибках программы, такие как SIGSEGV, но игнорирование такой ошибки не позволит программе продолжить осмысленное выполнение. Игнорирование пользовательских запросов, таких как SIGINT, SIGQUIT и SIGTSTP, является неприемлемым. Если нежелательно, чтобы сигналы передавались во время определенной части программы, нужно их заблокировать, но не игнорировать. HANDLER ‒ указывает адрес функции обработчика в программе, чтобы указать запуск этого обработчика, как способ доставки сигнала.

*6.  Расширенная обработка сигналов и функция sigaction().*

Лекция 6

Функция sigaction() имеет тот же основной эффект, что и signal() — указывает, как должен обрабатываться сигнал процессом. Однако sigaction() предлагает больший контроль за счет большей сложности. В частности, sigaction() позволяет указать дополнительные флаги для управления, когда генерируется сигнал и как вызывается обработчик. Для определения всей информации о том, как обрабатывать конкретный сигнал, в функции sigaction() используются структуры типа struct sigaction. Функция sigaction() объявляется следующим образом:

#include <signal.h>

int sigaction(int sig,

const struct sigaction \*restrict act,

struct sigaction \*restrict oldact)

Аргумент act используется для установки нового действия для сигнала sig, а аргумент oldact используется для возврата информации о действии, ранее связанном с этим сигналом. Параметр oldact имеет ту же цель, что и возвращаемое значение функции signal() — можно проверить, какое старое действие действовало для сигнала, и, если необходимо, восстановить его позже. Либо act, либо oldact могут быть нулевым указателем (NULL). Если нулевым указателем является oldact, это просто подавляет возврат информации о старом действии. Если нулевым указателем является act, действие, связанное с сигналом sig, не изменяется. Такое поведение позволяет узнать, как обрабатывается сигнал, без изменения этой обработки. Функция sigaction() возвращает: 0 в случае успеха и -1 в случае неудачи.

Cтруктура sigaction определяется примерно так:

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

Член sa\_restorer не предназначен для использования приложением. (POSIX не определяет sa\_restorer). Некоторые дополнительные сведения о назначении этого поля можно найти в sigreturn(2). Эта структура содержит как минимум члены sa\_handler, sa\_mask и sa\_flags.

Член sa\_handler используется так же, как второй аргумент handler функции signal() — он определяет действие, которое должно быть связано с сигналом, и может быть SIG\_DFL для действия по умолчанию, SIG\_IGN для игнорирования этого сигнала или указателем на функцию обработки сигнала. Эта функция получает номер сигнала в качестве единственного аргумента.

sa\_mask определяет набор сигналов, которые должны быть заблокированы во время работы обработчика. Доставленный сигнал автоматически блокируется по умолчанию перед запуском его обработчика независимо от значения параметра sa\_mask. Если необходимо, чтобы этот сигнал не блокировался в его обработчике, следует в обработчике написать код для его разблокировки.

Член sa\_flags структуры sigaction определяет различные флаги, которые могут влиять на поведение сигнала. Значение sa\_flags интерпретируется как битовая маска. Таким образом, флаги, которые необходимо установить, следует указать вместе по ИЛИ и сохранить результат в члене sa\_flags структуры sigaction. Каждому номеру сигнала соответствует свой набор флагов — каждый вызов sigaction() влияет на один конкретный номер сигнала, и указанные в вызове флаги применяются только к этому конкретному сигналу.

*7.  Каналы и конвейеры FIFO. Характеристики и особенности их использования.*

Лекция 11

Каналы и FIFO (также известные как именованные каналы) обеспечивают однонаправленный канал связи между процессами. Канал имеет конец для чтения и конец для записи. Данные, записанные в конец канала для записи, могут быть прочитаны из конца канала для чтения. Канал создается с помощью pipe(2), которая создает новый канал и возвращает два файловых дескриптора, один из которых относится к концу канала для чтения, а другой — к концу для записи. Каналы можно использовать для создания канала связи между связанными процессами. FIFO имеет в файловой системе имя, которое создается с помощью mkfifo(3), при этом FIFO открывается с помощью open(2). Любой процесс может открыть FIFO, если права доступа к файлу позволяют это сделать. Чтобы FIFO работал, он должен быть открыт как на запись, так и на чтение. Конец для чтения открывается с помощью флага O\_RDONLY. Конец для записи открывается с помощью флага O\_WRONLY.

Хотя FIFO имеют имя пути в файловой системе, ввод-вывод в FIFO не включает операции на базовом устройстве (если оно есть) ‒ это специальный файл.

Единственная разница между каналами и FIFO заключается в том, как они создаются и открываются. Как только эти задачи выполнены, ввод-вывод по каналам и FIFO имеет одинаковую семантику. Если процесс попытается прочитать из пустого канала, read(2) заблокируется до тех пор, пока в канале не станут доступны данные. Если процесс пытается записать в заполненный канал, то блокируется write(2) до тех пор, пока из канала не будет прочитано достаточно данных, чтобы запись можно было завершить. При использовании операции fcntl(2) F\_SETFL для включения флага состояния открытия файла O\_NONBLOCK возможен неблокирующий ввод-вывод. Канал связи, предоставляемый каналом, представляет собой поток байтов ‒ понятия границ сообщения для него нет. Если все файловые дескрипторы, относящиеся к концу канала для записи, были закрыты, то попытка read(2) из канала увидит конец файла (read(2) вернет 0). К каналу невозможно применить вызов lseek(2).

Канал имеет ограниченную емкость. Если канал заполнен, то write(2) заблокируется или завершится ошибкой, в зависимости от того, установлен ли флаг O\_NONBLOCK. Различные реализации имеют разные ограничения на емкость канала. Приложения не должны полагаться на конкретную емкость ‒ приложение должно быть спроектировано таким образом, чтобы процесс чтения потреблял данные сразу, как только они становятся доступными ‒ это предотвращает блокировку процесса записи.

Единственными флагами состояния открытого файла, которые можно осмысленно применить к каналу или FIFO, являются O\_NONBLOCK и O\_ASYNC.

Установка флага O\_ASYNC для конца канала для чтения вызывает генерацию сигнала (по умолчанию SIGIO), как только в канале становится доступным новый ввод. Цель доставки сигналов должна быть установлена с помощью команды fcntl(2) F\_SETOWN.

*8.  Очистка при завершении программы. Функции стандартной библиотеки exit() и atexit().*

Лекция 3 32-40

Существует два способа корректного завершения процесса в программах, написанных на языке C.

1) процесс корректно завершается по достижении конца функции main() или при выполнении оператора return в функции main();

2) второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы.

Для этого применяется функция exit(3) из стандартной библиотеки функций для языка C. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние «закончил исполнение». Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции exit(3) — кода завершения процесса — передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0

Функция atexit() регистрирует функцию, на которую указывает func, для вызова без аргументов при нормальном завершении программы. Регистрации функции atexit() отличаются от регистраций at\_quick\_exit(), поэтому приложениям может потребоваться вызывать обе функции регистрации с одним и тем же аргументом. Реализация обычно поддерживает регистрацию не менее 32 функций. Функция atexit() возвращает ноль, если регистрация прошла успешно, и ненулевое значение, если она не удалась.

Функция exit() вызывает обычное завершение программы. Функции, зарегистрированные функцией at\_quick\_exit(), не вызываются. Если программа вызывает функцию выхода более одного раза или вызывает функцию quick\_exit() в дополнение к функции выхода, поведение не определено. Сначала вызываются все функции, зарегистрированные функцией atexit(), в порядке, обратном их регистрации. Каждая функция вызывается столько раз, сколько она была зарегистрирована, и в правильном порядке по отношению к другим зарегистрированным функциям. Затем сбрасываются все открытые потоки с незаписанными буферизованными данными, все открытые потоки закрываются, а все файлы, созданные функцией tmpfile(), удаляются. Наконец, управление возвращается в хост-среду. Если значение status равно нулю или EXIT\_SUCCESS, возвращается определяемая реализацией форма успешного завершения состояния. Если значением статуса является EXIT\_FAILURE, возвращается определенная реализацией форма статуса неудачного завершения. В противном случае возвращаемый статус определяется реализацией. Не возвращает управление вызывающей стороне.

*9.  Алгоритмы замещения страниц.*

Я НАЗЫВАЮ ЭТОТ БИЛЕТ СМЕРТЬ, ПРОЩЕ ОТЧИСЛИТЬСЯ, ЧЕМ ЗАПОМНИТЬ

Просто похороны…. Тут я даже хуй какая это лекция

При возникновении ошибки отсутствия страницы операционная система должна выбрать выгружаемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть перезаписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. В этом случае загружаемая страница просто пишется поверх выгружаемой. Самый простой вариант — при ошибке отсутствия страницы для выселения выбирается произвольная страница — ведет к высоким затратам на замещение. Производительность системы была бы намного выше, если бы выбор выгружаемой страницы падал на редко востребованную страницу. При удалении интенсивно используемой страницы высока вероятность того, что она в скором времени будет загружена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в экспериментальной областях.

*САМЫЙ ОПТИМАЛЬНЫЙ АЛГОРИТМ (ХАХАХ НЕРЕАЛЬНО)*

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (например, эти команды содержатся на странице). К другим страницам обращения может не быть и через 10, 100 или, возможно, даже 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице. Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении двадцати млн. команд, а другая какая-нибудь страница не будет использоваться на протяжении пяти млн. команд, то удаление первой из них приведет к ошибке #PF в самом отдаленном будущем.

*Алгоритм FIFO*

Выталкивание первой пришедшей (самой старой) страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память.

Для замещения выбирается самая старая страница.

К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла.

В случае замещения активных страниц все сработает корректно, но тут же за этим сразу последует #PF. Поэтому принцип FIFO в чистом виде используется довольно редко.

*Алгоритм NRU*

Замещается не использовавшаяся в последнее время страница.

Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния R и M, связанных с каждой страницей. Бит R устанавливается при каждом обращении к странице (при чтении или записи). Бит M устанавливается, когда в страницу ведется запись (то есть когда она модифицируется). Эти биты присутствуют в каждой записи таблицы страниц и они должны обновляться при каждом обращении к памяти, поэтому необходимо, чтобы их значения устанавливались аппаратной частью. После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой. У x86\_64 для каждой 4k страницы есть биты A — Accessed и D — Dirty, устанавливаемые аппаратурой процессора. Если у аппаратуры нет таких битов, они должны быть созданы искусственно с помощью механизмов операционной системы ошибки отсутствия страницы и прерывания таймера. При запуске процесса все записи в его таблице страниц помечаются отсутствующими в памяти. Как только произойдет обращение к странице, возникнет ошибка отсутствия страницы операционная система устанавливает бит R (в своих внутренних таблицах), изменяет запись в таблице страниц, чтобы она указывала на правильную страницу, с режимом доступа только для чтения (READ ONLY) и перезапускает команду. Если впоследствии страница модифицируется, возникает другая ошибка страницы, позволяющая операционной системе установить бит M и изменить режим доступа к странице на чтение-запись (READ/WRITE).Биты R и M могут использоваться для создания простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0 Время от времени (например, при каждом прерывании по таймеру) бит R сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были. При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов R и M делит их на четыре категории:

1 Класс 0 — в последнее время не было ни обращений, ни модификаций.

2 Класс 1 — обращений в последнее время не было, но страница модифицирована.

3 Класс 2 — в последнее время были обращения, но модификаций не было.

4 Класс 3 — в последнее время были и обращения, и модификации.

*Алгоритм «вторая попытка»*

Подобен FIFO, но если R=1, то страница переводится в конец очереди, если R=0, то страница выгружается. В таком алгоритме часто используемая страница никогда не покинет память. Но в этом алгоритме приходится часто перемещать страницы по списку. Является простой модификацией алгоритма FIFO, исключающей проблему удаления часто востребованной страницы. Заключается в проверке бита R самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется.

Если бит R имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается. Этот алгоритм занимается поиском ранее загруженной в память страницы, к которой за только что прошедший интервал времени таймера не было обращений. Если обращения были ко всем страницам, то алгоритм «вторая попытка» превращается в простой алгоритм FIFO — если у всех страниц установлен бит R, операционная система поочередно перемещает все страницы в конец списка, очищая бит R. В результате она возвращается к самой старой странице, у которой бит R теперь уже сброшен. И эта страница выгружается. Таким образом, алгоритм всегда завершает свою работу.

*Алгоритм «часы»*

При всей своей логичности алгоритм «вторая попытка» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Лучше содержать все страничные блоки в циклическом списке как бы в виде часов. Стрелка (указатель) указывает на самую старую страницу. При возникновении ошибки отсутствия страницы #PF проверяется та страница, на которую указывает стрелка. Если ее бит R имеет значение 0, страница выгружается, на ее место в списке вставляется новая страница и указатель смещается вперед на одну страницу. Если значение бита R равно 1, то он сбрасывается и указатель перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с R = 0

*Алгоритм LRU (Least Recently Used)*

Выталкивание дольше всего не использовавшейся страницы.

Алгоритм позволяет сократить количество страничных нарушений. В основе лежит использование прошлого для аппроксимации (экстраполяции) будущего — замещается страница, которая не использовалась в течение самого долгого времени. В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое время не востребованные страницы наверняка еще долго так и останутся невостребованными. Из этого следует алгоритм — при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется замещением наименее востребованной страницы. LRU хороший, но труднореализуемый алгоритм — необходимо вести связанный список всех страниц, находящихся в памяти. В начале этого списка должна быть только что востребованная страница, а в конце — наименее востребованная. Сложность в том, что этот список должен обновляться при каждом обращении к памяти. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возло-

жено на аппаратное обеспечение (если предположить, что такое оборудование можно создать). Существуют способы реализации LRU с использованием специального оборудования. Для этого аппаратное обеспечение должно иметь 64-разрядный счетчик C, значение которого автоматически увеличивается после каждой команды. Кроме этого каждая запись в таблице страниц должна иметь довольно большое поле, чтобы содержать значение этого счетчика. После каждого обращения к памяти текущее значение счетчика C сохраняется в записи таблицы страниц, относящейся к той странице, к которой было это обращение. При возникновении ошибки отсутствия страницы операционная система проверяет все значения счетчика в таблице страниц, чтобы найти наименьшее из них. Та страница, к чьей записи относится это значение, и будет наименее востребованной.

Алгоритм NFU

Выталкивание редко используемой страницы. Алгоритм NFU(Not Frequently Used) — программная реализация алгоритма, близкого к LRU. Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает. Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Возможна небольшая модификация алгоритма, которая позволяет ему «забывать». Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения. Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

*10. Спинлоки и их использование.*

Лекция 17 (26- 31)

Большинство программ должны использовать мьютексы вместо спин-блокировок. Спин-блокировки в первую очередь полезны в сочетании с политиками планирования в реальном времени. Функция pthread\_spin\_init() выделяет любые ресурсы, необходимые для использования спин-блокировки, на которую ссылается lock, и инициализирует спинлок в разблокированном состоянии. Аргумент pshared должен иметь одно из следующих значений: PTHREAD\_PROCESS\_PRIVATE — спин-блокировка должна использоваться только потоками в том же процессе, что и поток, вызывающий pthread\_spin\_init(). Попытка совместно использовать спин-блокировку между процессами приводит к неопределенному поведению. PTHREAD\_PROCESS\_SHARED — спин-блокировка может использоваться любым потоком в любом процессе, который имеет доступ к памяти, содержащей блокировку, т.е. спин-блокировка может находиться в объекте совместно используемой памяти, который совместно используется несколькими процессами.

Спиновые блокировки следует использовать в сочетании с политиками планирования в реальном времени (SCHED\_FIFO или, возможно, SCHED\_RR). Использование спин-блокировок с недетерминированными политиками планирования, такими как SCHED\_OTHER, вероятно, указывает на ошибку проектирования. Проблема в том, что если поток, работающий в соответствии с такой политикой, запланирован в состоянии вне ЦП, в то время как он удерживает спин-блокировку, другие потоки будут тратить время на ожидание, пока владелец блокировки не будет еще раз перепланирован и не снимет блокировку. Если потоки создают ситуацию взаимоблокировки при использовании спин-блокировок, эти потоки будут крутиться бесконечно, потребляя процессорное время. Спин-блокировки в пользовательском пространстве неприменимы в качестве общего решения для блокировок. Они по определению склонны к инверсии приоритета и неограниченному времени ожидания. Программист, использующий спин-блокировки, должен быть чрезвычайно осторожен не только в коде, но и в отношении конфигурации системы, размещения потоков и назначения приоритетов.

*11. Блокировки чтения/записи и их использование в программах. Протокол обновления файла в многопользовательской среде.*

*(2ую часть вопроса не нашел)*

Лекция 17 31-далее

Во многих ситуациях данные читаются чаще, чем изменяются или записываются. В этих случаях можно, удерживая блокировку, разрешить потокам читать одновременно, и разрешить только одному потоку удерживать блокировку для изменения данных. Таким поведением обладает блокировка с несколькими «читателями» и одним «писателем» (или блокировка чтения/записи). Блокировка чтения-записи устанавливается либо для чтения, либо для записи, а затем снимается. Поток, который получает блокировку чтения-записи, должен быть тем, который ее снимает.

*12. Условные переменные и их использование потоками — ожидание и сигнализация.*

Лекция 18 полная лекция

Условная переменная — примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков в состоянии ожидания выполнения некоторого условия до момента поступления сигнала от другого потока о том, что данное условие выполнено (или до истечения максимального промежутка времени ожидания). Условные переменные используются вместе с ассоциированным с ними мьютексом и фактически представляют собой совокупность объекта синхронизации cond, предиката P и мьютекса mutex.

Концептуально, условная переменная — это очередь потоков, ассоциированных с совместно используемым объектом данных, которые ожидают выполнения некоторого условия, накладываемого на состояние данных. Таким образом, каждая условная переменная C связана c некоторым утверждением PC. Когда поток находится в состоянии ожидания условной переменной, он не считается владеющим данными, в то же время другой поток может изменить совместно используемый объект и просигнализировать ожидающим потокам в случае выполнения утверждения.

Ожидание условия, будь то ограниченное по времени или нет, является точкой отмены ‒ функции pthread\_cond\_[timed]wait() являются точками, где может быть замечен запрос отмены. Причина такого положения состоит в том, что в этих точках возможно неопределенное ожидание — какое бы событие ни ожидалось, даже если программа полностью верна, оно может никогда не произойти. Например, некоторые ожидаемые входные данные могут никогда не быть отправлены. Сделав условие ожидания точкой отмены, поток можно отменить и выполнить свой обработчик очистки, даже если поток может застрять в некотором неопределенном ожидании. Если для типа отменяемости потока задано значение PTHREAD\_CANCEL\_DEFERRED, побочным эффектом действия по запросу отмены, если поток заблокирован по условной переменной, является повторное получение мьютекса до вызова первого обработчика очистки. Эффект такой, как если бы поток был разблокирован, он выполняется до точки возврата из вызова [timed]wait(), но в этот момент он замечает запрос на отмену и вместо возврата управления стороне запускает действия по отмене потока, включая вызов обработчиков очистки отмены. Такое поведение гарантирует, что обработчик очистки отмены выполняется в том же состоянии, что и критический код, который находится как до, так и после вызова функции ожидания условия. Это особенно важно при взаимодействии с потоками POSIX из C ++, реализация которого может выбрать отображение отмены на исключение (exception). Оно гарантирует, что каждый обработчик исключений, оборачивающий критическую секцию, всегда может безопасно зависеть от того факта, что связанный мьютекс уже заблокирован, независимо от того, где именно в критической секции возникло исключение. Без этого не было бы единого правила, которому могли бы следовать обработчики исключений в отношении блокировки, и поэтому кодирование было бы очень громоздким.

Функция pthread\_cond\_broadcast() разблокирует все потоки, заблокированные в данный момент для указанной условной переменной cond. Функция pthread\_cond\_signal() разблокирует по крайней мере один из потоков, которые заблокированы по указанной условной переменной cond (если есть какие-либо потоки, которые заблокированы по cond). Если на условной переменной заблокирован более чем один поток, порядок, в котором потоки разблокируются, определяет политика планирования. Когда каждый поток, разблокированный в результате pthread\_cond\_broadcast() или pthread\_cond\_signal(), возвращается из своего вызова, он будет владеть мьютексом, с которым он вызывал pthread\_cond\_wait() или pthread\_cond\_timedwait(). При этом разблокированные потоки «борятся» за мьютекс в соответствии с политикой планирования, как если бы каждый из них вызвал pthread\_mutex\_lock().

*13. Методы ввода/вывода — блокирующий, неблокирующий, мультиплексированный и асинхронный.*

Лекция 19 3-9

Блокирующий метод

Любая пользовательская программа запускается внутри процесса, а код выполняется в контексте потока. Предположим, мы пишем программу, которой нужно читать информацию из файла. С блокирующим вводом-выводом мы просим ОС «усыпить» читающий поток и «разбудить» его после того, как данные из файла будут доступны для чтения. То есть блокирующий ввод-вывод называется так, потому что поток, который его использует, блокируется и переходит в режим ожидания, пока ввод-вывод не будет завершён.

Неблокирующий метод

Проблема блокирующего метода заключается в том, что поток будет спать, пока ввод-вывод не завершится. Поток не сможет выполнять никаких других задач, кроме ожидания завершения ввода-вывода. Иногда программе больше и не надо ничего делать. В противном же случае во время ожидания ввода-вывода было бы полезно выполнять другие задачи. Один из способов осуществить это — использовать неблокирующий ввод-вывод. Идея заключается в том, что когда программа делает вызов на чтение файла, ОС не будет блокировать поток, а просто вернёт программе либо готовые данные, либо информацию о том, что ввод-вывод ещё не завершен. В этом случае поток не будет заблокирован, но программе придётся позже проверять, завершён ли ввод-вывод. Это означает, что можно по-разному реагировать в зависимости от того, завершён ли ввод-вывод и выполнять другие задачи. Когда же программе снова понадобится ввод-вывод, она сможет повторно попробовать прочесть содержимое файла, и если ввод-вывод завершён, то получит содержимое файла. В противном случае ПО снова получит сообщение о том, что операция ещё не завершена и сможет заняться другими задачами.

Мультиплексированный метод

Проблема с неблокирующим вводом-выводом в том, что с ним не удобно работать, если задачи, которые выполняет программа, ожидая ввода-вывода, сами из себя представляют ввод-вывод. Мультиплексированный ввод-вывод: он тоже блокиру-

ет поток на операциях ввода-вывода, но вместо того, чтобы производить блокировку по очереди, можно запланировать все операции ввода-вывода, которые нужно сделать, чтобы продолжить выполнение, и блокировать их все. Операционная система разбудит поток, когда какая-нибудь из операций завершится. В некоторых реализациях мультиплексированного ввода-вывода можно даже точно указать, чтобы поток был разбужен только тогда, когда будет завершён заданный набор операций ввода-вывода

Асинхронный метод

Проблема мультиплексированного ввода-вывода в том, что поток всё-таки спит, пока ввод-вывод не будет готов для обработки. Для многих программ это подходит, поскольку у них нет других задач, пока они ждут завершения операций ввода-вывода. Но иногда у них есть и другие задачи. Например, ПО вычисляет цифры числа π и одновременно обрабатывает данные из нескольких файлов. Хотелось бы запланировать все операции чтения файлов, а пока программа ждёт их завершения, вычислять цифры числа π. Когда какой-нибудь из файлов будет прочитан, ПО выполнит обработку и продолжит вычислять цифры числа π дальше, пока ещё один файл не будет прочитан. Чтобы это работало, нужно, чтобы вычисление цифр числа π могло быть прервано вводом-выводом, когда он завершается. Это можно сделать с помощью обратных вызовов, связанных с событиями. Вызов на чтение принимает функцию обратного вызова и возвращается немедленно. Когда ввод-вывод завершается, операционная система остановит ваш поток и выполнит обратный вызов. Когда обратный вызов завершится, система возобновит работу вашего потока.

*14. Циклы опроса poll и select. Функции обратного вызова и очереди завершения.*

*(2ую часть вопроса не нашел)*

Лекция 19

Файловый дескриптор нового сокета может использоваться в системных вызовах select() или poll(). Вызов select() позволяет отслеживать изменения нескольких файловых дескрипторов ожидая, когда один или более файловых дескрипторов станут «готовы» для операции ввода-вывода определённого типа (например, ввода). Файловый дескриптор считается готовым, если к нему возможно применить соответствующую операцию ввода-вывода, например, read() или очень «маленький» write() без блокировки. Вызов poll() выполняет сходную с select() задачу — он ждёт пока какой-нибудь дескриптор из набора зарегистрированных файловых дескрипторов не будет готов выполнить операцию ввода-вывода. Обычно сервер в набор fd\_set системного вызова select() помещает дескриптор своего сокета, который ожидает приема запросов на соединение, а также файловые дескрипторы всех сокетов, которые возвращает системный вызов accept(). Если select() или poll() обнаружат готовность файлового дескриптора серверного сокета, это означает, что сервер должен вызвать accept(), который не будет заблокирован. Готовность других сокетов означает, что от клиента (или от клиентов) поступили новые данные, которые следует прочитать.

Вызов select() позволяет программам отслеживать изменения нескольких файловых дескрипторов ожидая, когда один или более файловых дескрипторов станут «готовы» для операции ввода-вывода определённого типа (например, ввода). Файловый дескриптор считается готовым, если к нему возможно применить соответствующую операцию ввода-вывода, например, read() или очень «маленький» write() без блокировки.

Вызов poll() выполняет сходную с select(2) задачу — он ждёт пока один дескриптор из набора файловых дескрипторов не станет готов выполнить операцию ввода-вывода.

Функции обратного вызова (Callback functions)

Доступен в классической Mac OS, VMS и Windows.

Обладает многими характеристиками сигнального метода, поскольку в основе своей это то же самое, хотя и редко признается таковым.

Разница в том, что каждый запрос ввода-вывода обычно может иметь свою собственную функцию завершения, тогда как сигнальная система имеет единственный обратный вызов.

Этот метод используется некоторыми сетевыми фреймворками из-за простоты реализации, тем не менее, метод может привести к появлению вложенного и беспорядочного кода, получившего название «ад обратных вызовов» (Node.js).

Недостатки

Потенциальная проблема использования обратных вызовов заключается в том, что может неуправляемо расти стек, поскольку очень часто после завершения одного ввода-вывода планируется

другой.

Если операция должно быть немедленно удовлетворена, предыдущий обратный вызов не

«откручивается» из стека, а «закрывается» следующим вызовом.

Системы, предотвращающие такое поведение (например, «промежуточное» планирование новых работ), усложняют работу и снижают производительность.

На практике, однако, это обычно не проблема, потому что новая операция ввода/вывода обычно возвращается, как только запускается новый ввод/вывод, что позволяет стеку "раскручиваться".

Проблему также можно предотвратить, избегая дальнейших обратных вызовов с помощью

очереди до тех пор, пока не вернется первый обратный вызов

*15. Асинхронный ввод/вывод в POSIX и уведомления о завершении операций.*

Лекция 19 21

Интерфейс асинхронного ввода-вывода POSIX (AIO) позволяет приложениям запускать одну или несколько операций ввода-вывода, которые выполняются асинхронно (т. е., в фоновом режиме). Приложение может выбрать каким образом оно будет уведомлено о завершении операции ввода-вывода: с помощью сигнала, созданием новой нити или вообще не получать уведомления.

Функция aio\_return() возвращает окончательное значение завершения запроса асинхронного ввода-вывода, задаваемого указателем на контрольный блок aiocbp. Эта функция должна вызываться один раз для любого запроса в случае, если aio\_error(3) возвращает результат, отличный от EINPROGRESS.

sigevent — структура для уведомления из асинхронных процедур

Структура sigevent используется в различных программных интерфейсах для описания способа, которым нужно уведомлять процесс о событии (например, окончание асинхронного запроса, истечение таймера или поступление сообщения). Определение, приведённое ниже, приблизительно — некоторые поля в структуре sigevent могут быть определены как часть объединения. Программы должны использовать только те поля, которые применимы к значению, заданном в sigev\_notify — способ выполнения уведомления.