**24. Потоки (нити) исполнения POSIX. Системные вызовы pthread\_create(), pthread\_join(),pthread\_cancel(). Подсоединение к потоку и отсоединение потока.**

*ЛЕКЦИЯ 15*

В POSIX.1 определён набор интерфейсов 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).

**int pthread\_create**(*pthread\_t \*restrict thread*, // здесь возвращается ID

*const pthread\_attr\_t \*restrict attr*, // атрибуты потока

*void \*(\*start\_routine)(void\*)*, // функция, начинающая поток

*void \*restrict arg)*; // аргумент для start\_routine

Функция pthread\_create() запускает новый поток в вызывающем процессе.

Новый поток начинает выполнение с функции start\_routine(), при этом этой функции передается единственный аргумент arg.

***int pthread\_join(pthread\_t thread, void \*\*retval);***

Функция pthread\_join() ожидает завершения потока, указанного потоком. Если этот поток уже

завершился, pthread\_join() немедленно возвращается. Поток, указанный в pthread\_join(), должен быть подсоединяемым.

***int pthread\_cancel(pthread\_t thread);***

Функция pthread\_cancel() отправляет запрос отмены потоку thread. Отреагирует ли целевой

поток на запрос отмены, и когда, зависит от двух атрибутов, находящихся под контролем этого потока: его состояния отмены (cancelability state) и типа.

Для отсоединения потока используется вызов pthread\_detach(). После это перехватить поток с помощью вызова pthread\_join() становится невозможно, также невозможно получить статус его завершения и прочее. Отменить отсоединенное состояние также невозможно. Если завершение потока не перехватить вызовом pthread\_join() в результате получим зомби-поток. Если завершился отсоединенный поток, все будет нормально.

Любому потоку по умолчанию можно присоединиться вызовом pthread\_join() и ожидать его завершения. Однако в некоторых случаях статус завершения потока и возврат значения нам не интересны.

**25. Завершение нитей POSIX. Системные вызовы pthread\_cancel(), pthread\_testcancel().**

*ЛЕКЦИЯ 15*

***void pthread\_exit(void \*retval);***

При компиляции и компоновке неоходимо использовать опцию -pthread.

Функция pthread\_exit() завершает вызывающий поток и возвращает значение через retval, которое (если поток присоединяется) доступно другому потоку в том же процессе, который вызывает pthread\_join(3). Если завершение потока не перехватить вызовом pthread\_join() в результате получим зомби-поток. Если завершился отсоединенный поток, все будет нормально.

***int pthread\_cancel(pthread\_t thread);***

Функция pthread\_cancel() отправляет запрос отмены потоку thread. Отреагирует ли целевой

поток на запрос отмены, и когда, зависит от двух атрибутов, находящихся под контролем этого потока: его состояния отмены (cancelability state) и типа.

***void pthread\_testcancel(void)***

Создает точку отмены внутри вызывающий поток, так что поток, который в противном случае выполняется код, который не содержит точек отмены, будет реагировать на запрос на отмену. Если возможность отмены отключена (с помощью pthread\_setcancelstate(3)), или запрос на отмену не ожидается, то вызов pthread\_testcancel() не действует.

**26. Системные вызовы POSIX управления процессами getpgrp(), setprgp(), getpgid(), setpgid(), getsid(), setsid().**

*ЛЕКЦИЯ 3(55 и до конца)*

*pid\_t getpgid(pid\_t pid) -* Возвращает ID группы вызывающего процесса.

*int setpgid(pid\_t pid, pid\_t pgid)* - Устанавливает PGID процесса, указанного в pid, равным pgid. Если pid равен нулю, то идентификатор процесса вызывающего процесса использовал. Если pgid равен нулю, то PGID процесса, указанного параметром pid делается таким же, как его идентификатор процесса.

В целом вызовы getpid и setpgid одинаковы с getpgrp и setpgrp. Отличия только в системе, на которой они были реализованы.

*int setpgid(pid\_t pid, pid\_t pgid);*

*pid\_t getpgid(pid\_t pid);*

*pid\_t getpgrp(void);* /\* POSIX.1 version \*/

*pid\_t getpgrp(pid\_t pid);* /\* BSD version \*/

*int setpgrp(void);* /\* System V version \*/

*int setpgrp(pid\_t pid, pid\_t pgid);* /\* BSD version \*/

**27. Искусственная генерация сигналов. Системные вызовы abort(), raise(), kill(), pthread\_kill().**

*Лекция 6 – виды сигналов*

*sighandler\_t signal(int SIGNUM, sighandler\_t ACTION)* — установить обработчик сигнала

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

SIGNUM — номер сигнала, который мы хотим контролировать.

*Abort()* — разблокировать SIGABRT и инициировать этот сигнал для вызывающего процесса (как если бы была вызвана функция raise(3)). Это приводит к аварийному завершению процесса, если только сигнал SIGABRT не перехватывается и обработчик сигнала не возвращается.

*raise(int sig)* — функция посылает сигнал вызывающему процессу или потоку.

В однопоточных программах является эквивалентом kill(getpid(), sig). В многопоточных — pthread\_kill(pthread\_self(), sig).

Если сигнал вызывает вызов обработчика, функция raise() вернется только после того, как обработчик сигнала вернется.

*int kill(pid\_t pid, int sig)* — послать сигнал процессу.

Если значение pid является положительным, то сигнал sig посылается процессу с идентификатором pid.

Если значение pid равно 0, то sig посылается каждому процессу, который входит в группу вызывающего процесса.

Если значение pid равно -1, то sig посылается каждому процессу, которым вызывающий процесс имеет право отправлять сигналы, за исключением процесса с номером 1 (init).

Если значение pid меньше -1, то sig посылается каждому процессу, который входит в группу

процессов, чей ID равен -pid.

Если значение sig равно 0, то никакой сигнал не посылается, но выполняется проверка существования и права.

Чтобы процесс мог посылать сигнал, он должен быть привилегированным, либо реальный

или эффективный идентификатор пользователя посылающего процесса должен быть равен реальному или сохранённому идентификатору пользователя процесса, которому отправляется сигнал.

int pthread\_kill(pthread\_t thread, int sig) — послать сигнал потоку. Отправляет sig сигнала в поток, поток в том же процессе, что и вызывающий. Сигнал асинхронно направляется в поток.

**28. Сигналы в UNIX. Типы и жизненный цикл. Блокирование**

*ЛЕКЦИИ 6-7*

Типы сигналов (их принято задавать номерами, как правило, в диапазоне от 1 до 31 включительно или специальными символьными обозначениями) и способы их возникновения в системе жестко регламентированы.

Процесс может получить сигнал от:

1) hardware (при возникновении исключительной ситуации);

2) другого процесса, выполнившего системный вызов передачи сигнала;

3) операционной системы (при наступлении некоторых событий);

4) терминала (при нажатии определенной комбинации клавиш);

5) системы управления заданиями при выполнении команды kill.

Передачу сигналов процессу в случаях его генерации источниками 2, 3 и 5, т.е. в конечном счете каким-либо другим процессом, можно рассматривать как реализацию в UNIX сигнальных

средств связи.

Существует три варианта реакции процесса на сигнал:

1) принудительно проигнорировать сигнал;

2) произвести обработку по умолчанию (проигнорировать, остановить процесс (перевести в состояние ожидания до получения другого специального сигнала), либо завершить работу с образование core файла или без него);

3) выполнить обработку сигнала, указанную пользователем.

Изменить реакцию процесса на сигнал можно с помощью специальных системных вызовов(*signal(), sigaction()).*

С момента, как обработчик начинает выполнение, до момента его завершения, необходимо блокировать все сигналы, которые могут помешать его работе или испортить его данные.

Когда для сигнала вызывается функция-обработчик, этот сигнал в дополнение к любым другим сигналам, которые находятся в маске сигналов процесса, на время работы обработчика автоматически блокируется. Например, если установлен обработчик для «SIGTSTP», то прибытие этого сигнала заставляет дальнейшие сигналы «SIGTSTP» ожидать во время выполнения обработчика.

Однако другие виды сигналов по умолчанию не блокируются, поэтому они могут поступать во время выполнения обработчика.

Надежный способ заблокировать другие виды сигналов во время выполнения обработчика —

использовать элемент «sa\_mask» структуры «sigaction»

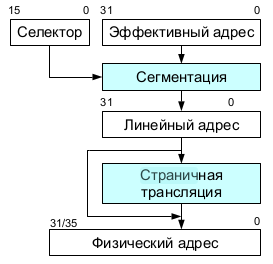
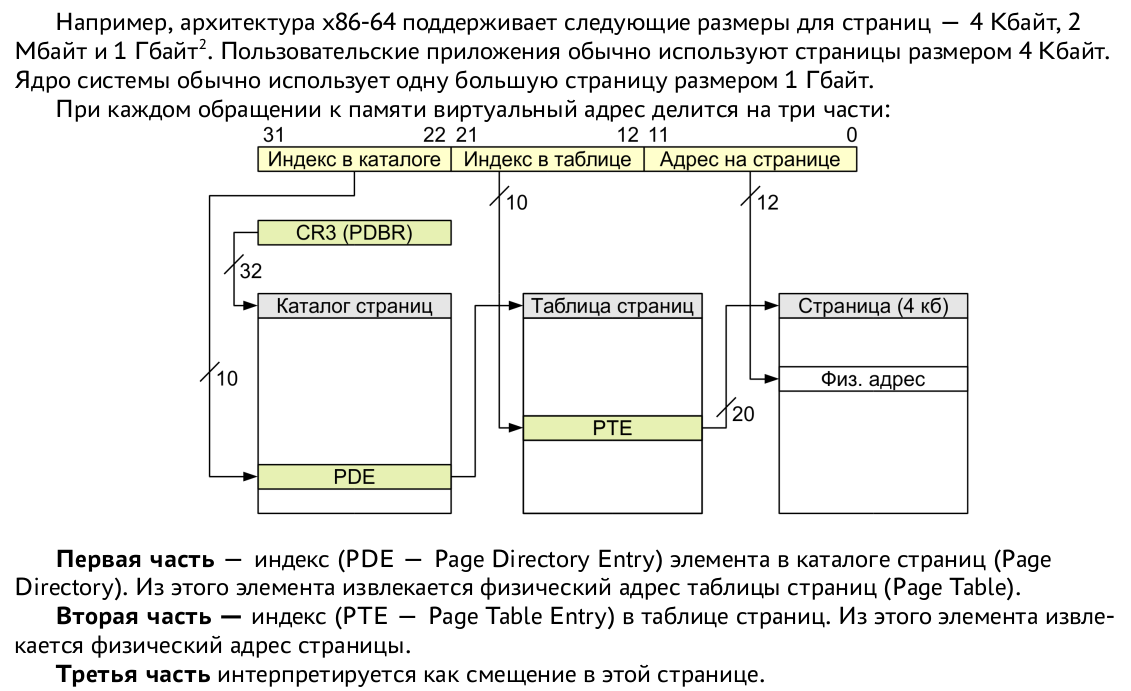
sigaddset(&block\_mask, signum), где block\_mask — маска, используемая для блокирования сигнала, а signum — номер сигнала.

**29. Страничная организация памяти защищенного режима x86. Таблицы и каталоги страниц.**

Дай боже здоровья

В большинстве систем, использующих абстракцию виртуальной памяти, реализована страничная организация памяти. В любой системе программы обращаются к набору адресов памяти. Эти адреса генерируются с помощью базовых регистров, индексной адресации, сегментных регистров или какими-то другими способами.

S\*I + B + Disp -> EA



В защищенном режиме x86, страничная организация памяти осуществляется с помощью иерархической системы таблиц и каталогов страниц. Эта система состоит из нескольких уровней таблиц, которые образуют иерархию.

Основные элементы иерархической системы таблиц и каталогов страниц в защищенном режиме x86 включают следующие:

PML4 (Page Map Level 4): Это самый верхний уровень таблицы страниц. PML4 содержит записи, каждая из которых указывает на PDPT (Page Directory Pointer Table).

PDPT (Page Directory Pointer Table): PDPT находится на втором уровне иерархии. Каждая запись в PDPT указывает на PD (Page Directory).

PD (Page Directory): PD расположен на третьем уровне. PD содержит записи, которые указывают на PT (Page Table).

PT (Page Table): PT находится на нижнем уровне иерархии. PT содержит записи, каждая из которых указывает на физический адрес страницы памяти.

Дескрипторы страниц: Каждая запись в PT содержит дескриптор страницы, который хранит информацию о доступе, правах и атрибутах страницы.

При конвертации виртуального адреса в физический адрес, процессор использует значения из каждого уровня таблицы страниц для определения физического адреса страницы памяти. Процессор выполняет последовательные обращения к таблицам и каталогам страниц, используя части виртуального адреса в качестве индексов для перехода от одного уровня к другому и нахождения соответствующей записи.

Использование иерархической системы таблиц и каталогов страниц позволяет эффективно управлять адресным пространством и обеспечивать изоляцию и защиту памяти между разными процессами в защищенном режиме x86. Это также позволяет использовать большие страницы для улучшения производительности и уменьшения накладных расходов на управление таблицами страниц.

**30. Системные вызовы fork(), exec..(), exit() и их связь с функцией main()**

*ЛЕКЦИЯ 3*

*pid\_t fork(void);*

fork создает процесс-потомок, который отличается от родительского только значениями PID

(идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0.

Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

*int exec..(const char \*filename,*

*char \*const argv[], ..);* - выполняет программу, передавая в неё аргументы и окружение (если надо). exec..() не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы.

*\_Noreturn void exit(int status)*;

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, обрат-

ном их регистрации.

Для корректного вызова программы из другой программы следует выполнить следующие действия:

1) Переопределить прерывания, вызываемые в child-программе.

2) Создать процесс-потомок, в котором будет выполняться другая программа.

3) Перейти в режим ожидания main-процессом и вызвать другую программу child-процессом.

4) Ждать прерывания и реагировать на них.