**16. Системные вызовы read(), write() и lseek()** *Лекция 04 – 57-59, 54-55*

*ssize\_t read(int fd, void \*buf, size\_t count)*

Пытается записать count байт из файла, с которым связан файловый дескриптор fd в буфер, адрес которого начинается с buf. Если количество count равно нулю, то read() возвращает это нулевое значение и завершает свою работу. Если count больше, чем SSIZE\_MAX, то результат не определен. При успешном завершении вызова возвращается количество байт, которые были считаны (возврат нуля означает конец файла EOF), а позиция файла увеличивается на это значение. Если количество прочитанных байт меньше, чем количество запрошенных, то это не считается ошибкой — данные, например, могли быть почти в конце файла, в канале, на терминале, или read() был прерван сигналом. В случае ошибки возвращаемое значение равно -1, а переменной errno присваивается номер ошибки. В этом случае позиция файла не определена.

*ssize\_t write(int fd, const void \*buf, size\_t count)*

Записывает до count байт из буфера buf в файл, на который ссылается файловый дескриптор fd. В случае успешного завершения возвращается количество байт, которые были записаны (ноль означает, что не было записано ни одного байта). В случае ошибки возвращается -1, а переменной errno присваивается соответствующее значение. Если count равен нулю, а файловый описатель ссылается на обычный файл, то будет возвращен ноль.

*off\_t lseek(int fd, off\_t offset, int whence)*

Устанавливает смещение для файлового дескриптора fd в значение аргумента offset в соответствии с параметром whence который может принимать одно из следующих значений:

SEEK\_SET Смещение устанавливается в offset байт от начала файла.

SEEK\_CUR Смещение устанавливается как текущее смещение плюс offset байт.

SEEK\_END Смещение устанавливается как размер файла плюс offset байт.

SEEK\_DATA Подогнать файловое смещение к следующему расположению, большему или равному значению offset, по которому в файле есть данные. Если значение offset указывает на данные, то файловое смещение устанавливается в offset.

SEEK\_HOLE Подогнать файловое смещение к следующему промежутку, большему или равному значению offset. Если значение offset указывает в середину промежутка, то файловое смещение устанавливается в offset. Если перед offset нет промежутка, то файловое смещение подгоняется к концу файла (т.е., это скрытый промежуток, который есть в конце любого файла).

Если offset указывает на конец файла, в обоих вышеуказанных случаях, lseek() завершится с ошибкой. При успешном выполнении lseek() возвращает получившееся в результате смещение в байтах от начала файла. При ошибке возвращается значение (off\_t) -1 и в errno записывается код ошибки.

**17. Сегменты общей памяти UNIX** *Лекция 10 – 17-19; 09 – 10*

Общая память представляет собой область, которая может совместно использоваться несколькими процессами. Чтобы упорядочить обмен данными, процессы должны пользоваться мьютексами или семафорами. Процесс должен «открыть» сегмент общей памяти и получить указатель на нее.

Сегмент общей памяти уникально идентифицируется положительным целым (shmid) и имеет связанную структуру данных struct shmid\_ds.

struct shmid\_ds {

struct ipc\_perm msg\_perm;

size\_t shm\_segsz;

pid\_t shm\_cpid;

pid\_t shm\_lpid;

shmatt\_t shm\_nattch;

time\_t shm\_atime;

time\_t shm\_dtime;

time\_t shm\_ctime;

};

Работа с совместно используемой памятью начинается с того, что процесс при помощи системного вызова shmget(2) создает совместно используемый сегмент, указывая первоначальные права доступа к этому сегменту (чтение и/или запись), а также его размер в байтах. Чтобы затем получить доступ к совместно используемому сегменту, его нужно присоединить посредством системного вызова shmat(2), который разместит сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их. Когда разделяемый сегмент становится ненужным, его следует отсоединить, воспользовавшись системным вызовом shmdt(2). Для выполнения управляющих действий над разделяемыми сегментами памяти служит системный вызов shmctl(2). После того, как последний процесс отсоединил разделяемый сегмент, следует выполнить управляющее действие по удалению сегмента из системы.

**18. Работа с общей памятью. Системные вызовы shmget(), shmctl(), shmat(), shmdt()**

*Лекция 10 – 20-33*

*int shmget(key\_t key, int size, int shmflg (флаги создания))*

Возвращает идентификатор сегмента общей памяти, соответствующий значению аргумента key. Его можно использовать для получения идентификатора ранее созданного общего сегмента памяти (shmflg = 0 и key ≠ IPC\_PRIVATE), а также для создания нового. Если key = IPC\_PRIVATE, создается новый сегмент общей памяти размером size (округленным до размера, кратного PAGE\_SIZE). Новый сегмент общей памяти создается, если значение key ≠ IPC\_PRIVATE и нет идентификатора, соответствующего key, причем shmflg должен содержать флаг IPC\_CREAT. Аргумент size имеет смысл только в том случае, если создается новый сегмент. Вновь созданный сегмент памяти заполняется нулями. Чтобы получить возможность пользоваться сегментом памяти, нужно обратиться к вызову shmat(), который вернет указатель. При удачном завершении вызова возвращается идентификатор сегмента shmid, и -1 при ошибке.

Поле shmflg может содержать: IPC\_CREAT — создания нового сегмента. Если флага нет, то функция shmget() будет искать сегмент, соответствующий ключу key и затем проверит, имеет ли пользователь права на доступ к сегменту; IPC\_EXCL — используется совместно с IPC\_CREAT для создания нового сегмента вызовом. Если сегмент уже существует, то вызов завершается с ошибкой.

*int shmctl( int shmid, int cmd, // команда struct shmid\_ds \*buf)*

shmctl() позволяет пользователю: получать информацию о общих сегментах памяти; устанавливать владельца, группу общего сегмента, права на него; удалить сегмент. При удачном выполнении возвращается 0, а при ошибке -1.

shmid – идентификатор общего сегмента памяти, полученный при помощи shmget().

cmd – команда: IPC\_STAT – копирование информации о сегменте в буфер buf (предоставляемая пользователем структура данных); IPC\_SET – применение пользовательских изменений к содержимому полей uid, gid или mode в структуре shm\_perms; IPC\_RMID – пометка сегмента как удаленного (сегмент будет удален после отключения).

*void \*shmat( int shmid, const void \*shmaddr, int shmflg)*

Функция shmat() подсоединяет сегмент общей памяти shmid к адресному пространству вызывающего процесса. При завершении работы процесса сегмент будет отсоединен. Один и тот же сегмент может быть присоединен в адресное пространство процесса несколько раз. При успешном завершении системного вызова shmat() результат равен адресу, который получил присоединенный сегмент. В случае неудачи возвращается -1.

shmid – идентификатор сегмента общей памяти, полученный от shmget().

shmaddr – адрес присоединения сегмента (адрес в виртуальном пространстве пользователя, который получит начало сегмента). Если shmaddr = NULL, то система выбирает для присоединяемого сегмента подходящий (неиспользованный) адрес в адресном пространстве процесса. Если shmaddr ≠ NULL, присоединение производится по адресу shmaddr (с учетом значения shmflg).

shmflg — параметр используется для передачи системному вызову shmat() флагов: SHM\_RND – адрес shmaddr следует округлить до некоторой системно-зависимой величины; SHM\_RDONLY – присоединяемый сегмент будет доступен только для чтения.

*int shmdt( const void \*shmaddr)*

Функция shmdt() отсоединяет сегмент общей памяти, находящийся по адресу shmaddr, от адресного пространства вызывающего процесса. Эта функция освобождает занятую ранее этим сегментом область памяти в адресном пространстве процесса. Отсоединяемый сегмент должен быть среди присоединенных ранее функцией shmat(). При успешном завершении системного вызова shmdt() результат равен нулю; в случае неудачи возвращается -1. После того, как последний процесс отсоединил сегмент общей памяти, этот сегмент вместе с идентификатором и ассоциированной структурой данных следует удалить с помощью системного вызова shmctl().

**19. Структура ФС ОС UNIX. Хранение данных** *Лекция 04 – 16-25*

Каждый жесткий диск состоит из одной или нескольких логических частей (групп цилиндров), называемых разделами. В UNIX разделы выступают в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных. Обычно в разделе может располагаться только одна физическая файловая система. Физическая файловая система UNIX занимает раздел диска и состоит из следующих основных компонентов: суперблок (содержит общую информацию, необходимую для монтирования и управления файловой системой), массив индексных дескрипторов (содержит метаданные всех файлов файловой системы), блоки хранения данных (индексный дескриптор содержит ссылки на блоки данных).

Индексный дескриптор содержит информацию о статусе файла и указывает на расположение данных этого файла. Ядро обращается к индексному дескриптору по индексу в массиве. Один дескриптор является корневым для физической файловой системы, через него обеспечивается доступ к структуре каталогов и файлов. Размер массива индексных дескрипторов является фиксированным и задается при создании физической файловой системы (обычно 128 байт). Индексный дескриптор не содержит имени файла и его содержимого.

ОС UNIX обобщает концепцию файла как универсальную абстракцию. Практически любое внешнее устройство может быть представлено на пользовательском уровне, как файл специального типа. Устройства, которые могут быть представлены в виде файлов, делятся на два типа – символьные (или потоковые) и блочные. Иногда используются термины «байт-ориентированные» и «блок-ориентированные» устройства.

В ОС UNIX каталоги хранят только имя файла и некоторый номер, позволяющий идентифицировать соответствующий файл. Вся остальная информация о файле связывается не с его именем, а с этим самым номером, который связан индексным дескриптором файла. Также в линуксе есть единое дерево каталогов.

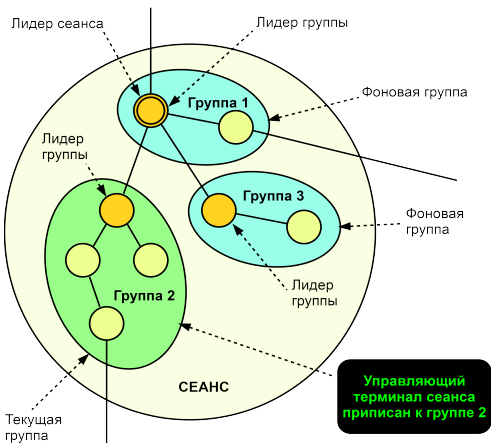
**20. Иерархия процессов в операционной системе. Группы и сеансы. Системные вызовы POSIX порождения и управления процессами**

*Лекция 03 – 3-4, 6-8, 10-11, 25-28, 46-48*

В операционной системе \*NIX все процессы кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве процесса прародителя всех остальных процессов в разных \*NIX-образных системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке, является процесс kernel идентификатором 0.

Все процессы в \*NIX связаны отношениями процесс-родитель – процесс-потомок, образуя генеалогическое дерево процессов. Все эти деревья принято разделять на группы процессов. Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. Каждый процесс обязательно включен в какую-нибудь группу. При рождении нового процесса он попадает в ту же группу, в которой находится его родитель. Процессы могут мигрировать из группы в группу. Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе.

Группы процессов объединяются в сеансы. Понятие сеанса было введено для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе. В связи с этим с каждым сеансом в системе может быть связан терминал, называемый управляющим терминалом сеанса, через который обычно и общаются процессы сеанса с пользователем. Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов. Могут существовать сеансы, вообще не имеющие управляющего терминала. Сеанс – набор процессов, у которых одинаковый ID сеанса.



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. Расходы на fork сводятся к копированию таблицы страниц родителя и созданию уникальной структуры, описывающей задачу. При успешном завершении родителю возвращается PID процесса-потомка, а процессу-потомку возвращается 0. При неудаче родительскому процессу возвращается -1, процесс-потомок не создается, а значение errno устанавливается должным образом.

Существует два способа корректного завершения процесса в программах, написанных на языке C. В первом случае процесс корректно завершается по достижении конца функции main() или при выполнении оператора return в функции main(). Здесь неявно вызывается функция exit(0). Во втором варианте необходимо 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. Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии «закончил исполнение» либо до завершения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию. Такие процессы называют процессами-зомби.

pid\_t wait(int \*status) и pid\_t waitpid(pid\_t pid, int \*status, int options);

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока какой-нибудь из дочерних процессов не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Функция waitpid приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый «зомби»), то функция немедленно возвращает управление. Возвращает идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался WNOHANG и ни один дочерний процесс пока еще недоступен, или -1 в случае ошибки.

Параметр pid может принимать несколько значений: < -1 – нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid; -1 – ожидание любого дочернего процесса; 0 – ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса; > 0 – ожидание дочернего процесса, чей идентификатор равен pid.

Значение options создается путем логического сложения нескольких следующих констант: WNOHANG – немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение; WUNTRACED — означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено.

Если status ≠ NULL, то функции wait и waitpid сохраняют информацию о статусе в переменной, на которую указывает status. Состояние status можно проверить с помощью макросов, которые принимают в качестве аргумента буфер (типа int).

**21. Жесткие и символические ссылки. Системные вызовы создания и удаления**

*Лекция 04 – 8, 10-11*

Жесткой ссылкой считается элемент каталога, указывающий непосредственно на некоторый индексный дескриптор. Это «настоящее имя файла». Жесткие ссылки очень эффективны, но у них существуют определенные ограничения, так как они могут создаваться только в пределах одной физической файловой системы. Когда создается такая ссылка, связываемый файл должен уже существовать. Кроме того, жесткой ссылкой не могут связываться каталоги из-за вероятности возникновения рекурсии и зацикливания. Таким образом, файл может иметь несколько совершенно равноправных имен. В индексном дескрипторе содержится счетчик жестких ссылок на этот дескриптор. Создание жестких ссылок на файлы, расположенные на других устройствах, не допускается.

Символическая ссылка – файл специального типа, содержащий путь к другому файлу. Указание на то, что данный элемент каталога является символической ссылкой, находится в индексном дескрипторе. Обычные команды доступа к файлу вместо получения данных из физического файла, берут их из файла, имя которого приведено в ссылке. Этот путь может указывать на что угодно – это может быть каталог, он может даже находиться в другой физической файловой системе, более того, указанного файла может и вовсе не быть. В описании пути можно использовать любое сочетание символических и жестких ссылок. Операция открытия символической ссылки на чтение или запись приводит к открытию на чтение или запись того файла, на который она ссылается, а не ее самой. Символическая ссылка в отличие от жесткой имеет свой собственный номер индексного дескриптора и имеет свой тип. Символическая ссылка создается вызовом symlink().

int link(const char \*oldpath, const char \*newpath) – создает новое имя для файла oldpath, если newpath существует, он перезаписан не будет. Новое имя может использоваться точно так же, как и старое, т.к. оба ссылаются на один и тот же файл.

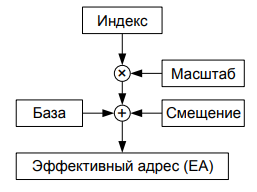
int symlink(const char \*topath, const char \*frompath) – создает символьную ссылку, которая называется frompath и содержит строку topath. Если frompath существует, он перезаписан не будет.

int unlink(const char \*pathname) – удаляет имя и возможно файл, на который оно ссылается

Файл создается с помощью вызовов open() или create() всегда с одним именем. Дополнительные имена можно назначить файлу используя системный вызов link(). Этот вызов является проекцией команды оболочки ln. Файл удаляется функцией unlink(), которая удаляет одну из жестких ссылок. Когда будет удалена последняя, файл считается удаленным и ресурсы, которые он занимает на носителе, освобождаются. Создание и/или удаление символической ссылки никогда не затрагивают ни имени файла, на который она ссылается, ни его индексного дескриптора. Файл может быть удален, а ссылка остается («висящая»). Файл может не существовать в момент создания символической ссылки.

**23. Формирование адреса памяти в x86 (адрес эффективный, логический, линейный (виртуальный), физический).**

*Лекция 10 – 12-16*

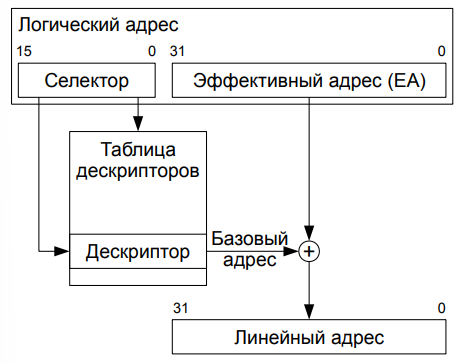
Формирование эффективного адреса (адрес, формируемый процессором из инструкции)

База — может содержаться в любом из РОН;

Индекс — может содержаться в любом из регистров, за исключением ESP;

Смещение — содержится в коде команды;

Масштаб — содержится в коде команды (1, 2, 4, 8).

Формирование линейного адреса (виртуальный в случае страничной трансляции)

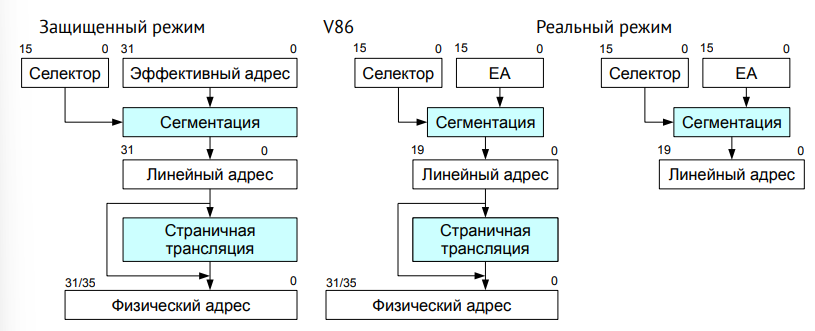
1. Из селектора извлекается поле индекса;

2. По индексу находится соответствующий дескриптор;

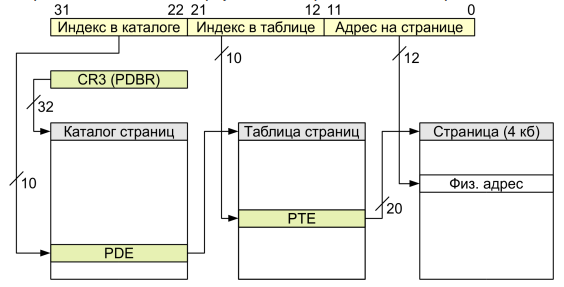
3. Из дескриптора извлекается поле адреса базы.

4. К адресу базы добавляется смещение.

Формирование физического адреса (в системной памяти)



Трансляция виртуального адреса в физический в обычном режиме IA32.



При каждом обращении к памяти виртуальный адрес делится на три части. Первая часть – индекс (PDE – Page Directory Entry) элемента в каталоге страниц (Page Directory). Из этого элемента извлекается физический адрес таблицы страниц (Page Table). Вторая часть – индекс (PTE – Page Table Entry) в таблице страниц. Из этого элемента извлекается физический адрес страницы. Третья часть интерпретируется как смещение в этой странице.