Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лекции по ОС

Москва — 2022 г.

Оглавление

1 Лабораторная по Unix №2 **3**

1.1 fork 3

1.2 exec 4

1.3 wait 5

1.4 Процесс-зомби 6

1.5 Программные каналы 7

1.6 Сигналы 8

2 Процесы-демоны **10**

3 Семафоры **11**

3.1 Основная информация 11

3.2 Средства разделяемой памяти. 14

4 Мониторы **16**

4.1 Простой монитор 16

4.2 Кольцевой монитор 17

4.3 Монитор читатели-писатели 17

4.4 Алгоритм-булочная (Бейкери-алгоритм) 19

5 Очереди сообщений **20**

6 Очереди сообщений **22**

1 Лабораторная по Unix №2

Типы файлов:

— исполняемый

1.1 fork

Системный вызов fork() создает новый процесс, который находится в отношении потомок-предок. Это отношение поддерживается в структуре proc соотв. указателями. Процесс-потомок является копией процесса-предка: процесс-потомок наследует код (программу) предка, дескрипторы открытых файлов (в том числе программных каналов), сигнальную маску, маску режима создадия файлов, окружение.

//доп.

Другими словами, программа, которую выполняет процесс-потомок является программой родительского процесса. Процесс потомок и процесс-родитель разделяют одно и то же адрусное пр-во. ?области данных и стека. Процесс-потомок начинает работу в режиме задачи после возвращения из системного вызова fork(). Тот факт, что образы памяти, переменные, регистры и все остальное и у родительского процесса, и у дочернего идентичны, могло бы привести к невозможности различить эти процессы, однако, системный вызов fork() возвращает дочернему процессу число 0, а родительскому — PID (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса обычно проверяют возвращаемое значение и действуют соответственно.

//доп.

Если вызов fork() завершается аварийно, он возвращает -1. Обычно это происходит из-за ограничения числа дочерних процессов, которые может иметь родительский процесс (CHILD_MAX), в этом случае переменной errno будет присвоено значение EAGAIN. Если для элемента таблицы процессов недостаточно места или не хватает памяти, переменная errno получит значение ENOMEM.

В старых Unix-системах код предка копировался в адресное пространство потомка. Для потомка создавалось собственное адресное пространство(таблицы страниц). В результате создавалось несколько копий одной программы, что не эффективно. В современных системах исользуется оптимизированный fork(): для процесса потомка создаются собственные карты трансляции адресов (таблицы страниц), но дескрипторы этих страниц ссылаются на адресное пространство процесса-предка (на страницы предка). При этом для страниц адресного пространства предка права доступа меняются с read-write на only-read и (в дескрипторе каждой страницы) устанавливается флаг copy-on-write. Если предок или потомок попытаются изменить страницу, возникнет исключение по правам доступа. Обработчик исключения обнаружит установленный флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. Таким образом код процесса-предка не копируется полностью, а создаются только копии страниц, которые редактируются. Такая ситуация в системе сохраняется до тех пор, пока потомок не вызовет exit или exec.

1.2 exec

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство программы, которая передана exec в качестве параметра. Говорят, что системный вызов exec() создает так называемый низкоуровневый процесс: на самом деле процесс не создаётся (у него нет идентификатора и дескриптора), но создаются таблицы страниц для адресного пространства программы, указанной в exec().

— Разбирает путь к файлу (для системы это строка символов) и осуществляет доступ к нему.

— Проверяются права на выполнение файла.

— Если файл существ., проверяется что он исполняемый (по заголовку файла).

— Высвобождает старое адресное пр-во.

— Создаёт таблицы страниц для (виртуального) адресного пространства программы, переданной exec в качестве параметра.

— Заменяет в дескрипторе процесса старый адрес таблицы страниц на новый

— Сбрасывает все обработчики сигналов в дей-я, определённые по умолчанию.

— Инициализируется аппаратный контекст. При этом большинство регистров сбрасывается в 0.

Чтобы программа выполнилась, адрес точки входа надо загрузить в instruction_pointer. Для Intel: в регистр CR3 нужно загрузить адрес программы, переданной exec в качестве параметра (? без этого невозможна адресация). ? В других системах - program_counter.

//доп.

В результате флаг copy-on-write сбрасывается. Сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

1.3 wait

Системный вызов wait() блокирует родительский процесс до момента завершения дочернего. При этом процесс-предок получает статус завершения процесса-потомка.

Вызов возвращает PID дочернего процесса. Обычно это дочерний процесс, который завершился. Сведения о состоянии позволяют родительскому процессу определить статус завершения дочернего процесса, т.е. значение, возвращенное из функции main потомка или переданное функции exit(). Если stat_loc не равен пустому указателю, информация о состоянии будет записана в то место, на которое указывает этот параметр.

— WIFEXITED

— WEXITSTATUS

— WIFSIGNALED

— WTERMSIG

— WIFSTOPPED

— WSTOPSIG

pid_t waitpid(pid_t pid, int *stat_loc, int options);

Аргумент pid — конкретный дочерний процесс, окончания которого нужно ждать. Если он равен –1, waitpid() возвращает информацию о любом дочернем процессе. Как и вызов wait(), он записывает информацию о состоянии процесса в то место, которое указывает аргумент stat_loc, если последний не равен пустому указателю. Аргумент options позволяет изменить поведение waitpid. Наиболее полезная опция WNOHANG мешает вызову waitpid() приостанавливать выполнение вызвавшего его процесса. Она применяется для определения, завершился ли какой-либо из дочерних процессов, и если нет, то можно продолжить выполнение. Остальные опции такие же, как в вызове wait().

И так для того, чтобы родительский процесс периодически проверял, завершился ли конкретный дочерний процесс, можно использовать следующий вызов:

waitpid(child_pid, (int *)0, WNOHANG);

Он вернет ноль, если дочерний процесс не завершился и не остановлен, или child_pid, если это произошло. Вызов waitpid вернет -1 в случае ошибки и установит переменную errno. Это может произойти, если нет дочерних процессов (errno равна ECHILD), если вызов прерван сигналом (EINTR) или аргумент options неверный (EINVAL).

1.4 Процесс-зомби

Когда дочерний процесс завершается, связь его с родителем сохраняется до тех пор, пока родительский процесс в свою очередь не завершится нормально, или не вызовет wait(). Следовательно, запись о дочернем процессе не исчезает из таблицы процессов немедленно. Становясь неактивным, дочерний процесс все еще остается в системе, поскольку его код

завершения должен быть сохранен, на случай если родительский процесс в дальнейшем вызовет wait().

Процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Это сделано для того, чтобы процесс-предок, вызвавший системный вызов wait(), не был заблокирован навсегда.

Можно увидеть переход процесса в состояние зомби, если дочерний процесс завершиться первым, то он будет существовать как зомби, пока процесс-предок или вызовет системный вызов wait(), или родительский процесс завершиться.

1.5 Программные каналы

Программные каналы – базовое средство взаимодействия между процессами. У каналов односторонняя (симпликсная) связь. Любой прог. канал – буфер в области данных ядра типа FIFO. Процессы в мультизадачных ОС являются защищёнными (имеют защищённые адресные пространства). Пр-с не может обратиться в адресное пр-во др. процесса, они могут взыимодействовать только через адресное пространство ядра.

Виды ПК:

— именованный (mknode). У него есть имя, дескриптор, номер.

— неименованный (pipe). Нет имени, есть дескриптор(inode). Неименованные программные каналы могут использоваться для обмена сообщениями между процессами-родственниками. Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключения – массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают.

Прог. каналы буферезуются на 3-х уровнях:

— В ядре системы. При переполнении системы ... переписывается на диск, при этом исп. станд. ф-и read-write. Если пр-с записывает в ПК

> 4096 байт, то он блокируется до тех пор, пока из канала всё не прочитают. -> мы не пишепм размер канала(он всегда 4096 байт - 1 страница, чтобы быстрее передавать данные).

—

—

1.6 Сигналы

Сигнал – способ информирования процесса ядром о происшествии какого-то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло.

Обычно, получение процессом сигнала предписывает ему завершиться. Вместе с тем процесс может установить собственную реакцию на получаемый сигнал. Например, получив сигнал процесс может его проигнорировать, или вызвать на выполнение некоторую программу, а после ее завершения продолжить выполнение с точки получения сигнала.

Установить реакцию на поступление сигнала можно с помощью системного вызова signal(): func = signal(snum, function);

Где snum - номер сигнала, а function - адрес функции, которая должна быть выполнена при поступлении указанного сигнала. Возвращаемое значение – адрес функции, которая будет реагировать на поступление сигнала. Вместо function можно указать ноль или единицу. Если был указан ноль, то при поступлении сигнала snum выполнение процесса будет прервано аналогично вызову exit. Если указать единицу, данный сигнал будет проигнорирован, но это возможно не для всех процессов.

Системный вызов signal() возвращает указатель на предыдущий обработчик данного сигнала. Его можно использовать для восстановления обработчика сигнала.

С помощью системного вызова kill() можно сгенерировать сигналы и передать их другим процессам. kill(pid, snum); где pid - идентификатор процесса, а snum - номер сигнала, который будет передан процессу. Обычно kill() используется для того, чтобы принудительно завершить (“убить”)

процесс.

Pid состоит из идентификатора группы процессов и идентификатора процесса в группе. Если вместо pid указать нуль, то сигнал snum будет направлен всем процессам, относящимся к данной группе (понятие группы процессов аналогично группе пользователей). В одну группу включаются процессы, имеющие общего предка, идентификатор группы процесса можно изменить с помощью системного вызова setpgrp. Если вместо pid указать -1, ядро передаст сигнал всем процессам, идентификатор пользователя которых равен идентификатору текущего выполнения процесса, который посылает сигнал.

2 Процесы-демоны

3 Семафоры

3.1 Основная информация

Семафор – неотриц. защищённая переменная, на которой определены 2 операции: P(S) - захват. P(V) - освобождение.

Виды семафоров:

— Бинарный

— Считающий ($xN0$) (контр. заполненность буфера)

— Множественный (наборы (массивы) считающих семафоров)

Какие-то переменные:

— sc

— sf

— sb - число заполненных ячеек буфера

При использовании семафора мы используем системные вызовы -> переходим в режим ядра. Переход в режим ядра – плата за переход в режим активного ожидания. Алгоритм булочная – очередь типа фифо. В нём два процесса одновременно входят в дверь. В этом алг. есть акт. ожидание. Процесс ждёт, пока все проц. с большими идент. освободят...

Семафоры поддерживаются системой. Семантически они представляются в виде массивов -> в наборе семафоры обозначаются индексами(здесь это номер семафора в наборе, а не смещение). В адресном прве ядра есть таблица семафоров. В ней каждый набор семафоров имеет дескриптор.

В дескрипторе есть:

— идентификатор(его присв. процесс, создавший набор). Другие процессы могут получить этот ид. для доступа к семафору

— uid и ид. его группы . Эффективный uid которого совп. с uid создателя может ... и изменять его управляющие параметры.

— права доступа (для user, group, others)

— кол-во семафоров

— время изм. семафоров последним процессом

— время после изменения упр. параметров

— указатель на массив семафоров.

О каждом семафоре известно:

— значение семафора

— id процесса, который оперировал семафором в последний раз

— кол-во процессов, заблокированных на данном семафоре

Особенности семафоров:

— У семафора нет хозяина -> освободить его может любой процесс(в отл. от ...)

В SystemV для каж. семафора определены сист. вызовы:

— semget - получение семафора

— semctl - контроль

— semop - операция на семафоре

Одной неделимой операцией можно изменить несколько семафоров набора.

```
1 int semop(int semfd, struct sembuf *opPts, int len);
2 // len -- колво- семафороввнаборе
3 struct sembuf
4 {
5     ushost sem_num;
6     short sem_op;
7     short sem_flag;
8 }
```

В отличае от семафоров Дейкстры, у Unix определено 3 операции:

— semop $>$ 0 освобождение процессом захваченного ресурса(ыход из крит. секции). Если к семафору есть очередь процессов, после этой операции 1-й семафор в оч. сможет получить доступ к ресурсу.

— semop = 0 перевод процессса в сост. ожидания освоб. семафора и его ресурса без захвата семафора (только как инф.)

— захват. Если процесс не может вып. декремент, то он будет блокирован на данном сем., т.е. поставлен к нему в очередь.

На семафоре определены флаги (в sem_flag):

— IPC_NOWAIT - информарует ядро о нежелании процесса переходить в сост. ожидания. Нужно, чтобы избежать блокирования всех процессов. В силу того, что kill нельзя перехватить, пр-с не сможет освободить захваченный семафор. Чтобы избежать этой ситуациии исп. флаг sem_undo он указывает ядру на необходимость отслеживать изм. значения семафора в результате вызова sem_op, чтобы сист. при завершении процесса смогла отменить сделанные изм. , чтобы процессы не были забл. невечно.

```
1  int sem_ctl(int semfd, int num, int end, union semun
       avg);
2
3  #include <sys/types.h>
4  #include <sys/ipe.h>
5  #include <sys/sem.h>
6
7  struct sembuf sbuf[2] = {{0, -1, SEM_UNDO| IPC_NOWAIT},
       {1, 0, 1}};
8  int main() {
9      int pems = S_SIRWXV | S_IRWXG | S_IRWX0;
10     int fd = semget(100, 2 IPC_CTREATE, | pems);
11     if (fd == -1) {
12         perror("semget");
```

```
13            exit(1);
14        }
15        if (semop(fd, buf, 2) == -1) {
16            perror("semget");
17            return 0;
18        }
19    }
```

Id набора 100 - ключ доступа к сегменту. Если такого небора нет, будет выполнен semop. .. 1-й семафор будет захвачен

3.2 Средства разделяемой памяти.

Семафор – средство взяимоисключения. Каналы – средство передачи инф. Средства разделяемой памяти (сегменты) – это буфер (не FIFO) в области данных.

На этот участок памяти пр-с получает указатель. Здесь не определены ср-ва взаимоискл. (в отл. от pipe). Разд. сегменты созданы таким образом, что они не требуют копирования данных из адр. пространства пр-са в пр-во ядра. и наоборот за счёт того, что пр-с получает адрес разделяемой памяти. -> реализация монопольного достпупа лежит на программисте. В ядре системе есть таблица разд. сегментов памяти.

- shmget
- shmctl
- shmat
- shmdt

```
1 #include <string.h>
2 #include <sys/shm.h>
3
4 struct sembuf sbuf[2] = {{0, -1, SEM\_UNDO|
    IPC\_NOWAIT}, {1, 0, 1}};
```

```
5  int main() {
6      int pems = S_SIRWXU | S_IRWXG | S_IRWX0;
7      int fd = shmget(100, 1024, IPC_CTREATE, | pems); //
         созд. сег. разд. памятисид . 100
8      if (fd == -1) {
9          perror("shmget");
10         exit(1);
11     }
12     char *adr = (char*) shmat(fd, 0, 0);
13     if (adr == (char*)-1) {
14         perror("shmmat");
15         exit(1);
16     }
17     strcpy(adr, "aaaa");
18     if (shmdt(adr) == -1) {
19         perror("shmmat");
20         return 0;
21     }
22 }
```

Создаёт сегмент разделяемой памяти с идентификатором 100. Если его удалось создать, он получает указатель. Если удалось получить адрес разделяемой памяти, он... Информация, записанная в разделяемый сегмент остаётся. Её сможет прочитать любой другой пр-с по идентификатору.

Системные ограничения для разденяемых сегментов:

- SHMMNI - максисальное возможное число разделяемых сегментов в ядре. При попытке создать больше, процесс перейдёт в состояние ожидания до удаления какого-либо сегмента;
- SHMMIN - максимальный размер разделяемого сегмента;
- SHMMAX - максимальный размер разделяемого сегмента.

4 Мониторы

Проблемы семафоров решают мониторы. Это средства более высокого уровня, чем примитивы ядра.

Примитив – низкоуровневые средства, предоставляемые в распоряжение процессов.

NB! команда test_and_set Команды lock – unlock wait – post wait – signal

Идея монитора – унификация взаимодействия процессов. Монитор может быть реализован в ОС или в библиотеке языка (например, Cuncurrent Pascal)

Монитор – это набор процедур и данных (монитор защищает свои данные, т.к. к ним можно обратиться только через процедуры). Пр-с, вызвавший процедуру, наз. процессом, находящимся в мониторе. Процедура может одновременно использовать только один процесс. Остальные рпоцессы ставятся к очередь к монитору.

Обычно монитор оперирует переменными типа Conditional (событие) с помощью ф-й wait (блокировка) и signal (разблокировка).

Виды мониторов:

— Простой монитор;

— Кольцевой монитор;

— Монитор читатели-писатели.

4.1 Простой монитор

Обеспечивает выделение ресурса произвольному числу процессов. Когда пр-с захватывает ресурс, вызывается acquire.

Листинг 4.1 – Код монитора

```
1 resource : monitor
2 var
3     busy : logical
```

```
4       x     : conditional // event
5
6   procedure acquire
7   begin
8       if busy then wait(x);
9       busy = true; // process get access to resource
10  end
11
12  procedure release
13  begin
14      busy = false;
15      signal(x);
16  end
17
18  Begin
19      busy = false
20  End.
```

4.2 Кольцевой монитор

4.3 Монитор читатели-писатели

Процессы писатели могут изменять данные, поэтому они должны работать в режиме монопольного доступа к раздел. данным. Если писатель изм. анные, другие читатели/писатели не могут к ним обратиться. Процессы-читатели могут только читать данные. При этом они могут читать параллельно. Пример – продажа билетов.

Листинг 4.2 – Код монитора читатели-писатели

```
1   monitor : resource
2   var
3       nr                : integer; // reader
4       wrt               : logical; // writer
5       c_read, c_write   : condition;
```

```
6
7  procedure start_read
8  begin
9      if wrt or turn(c_write) // active writer exists or
           there are waiting readers in queue
10         then wait(c_read);
11     inc(nr);
12     signal(c_read); // other readers in queue become
           active
13 end;
14
15 procedure stop_read
16 begin
17     dec(nr);
18     if nr = 0 then // no readers => can write
19         signal(c_write);
20 end;
21
22 procedure start_write
23 begin
24     if nr > 0 or wrt
25         then wait(c_write);
26     wrt = true;
27 end;
28
29 procedure stop_read
30 begin
31     wrt = false;
32     if turn(c_read) // there are waiting readers in queue
33         signal(c_read);
34     else
35         signal(c_write)
36 end;
37
```

```
38 Begin
39 nr =
40 End.
```

Читатель начинает с вызова srart_write и заканчивавает вызовом stop_write. Данный алгоритм предотвращает бесконечное откладывание.

4.4 Алгоритм-булочная (Бейкери-алгоритм)

Базируется на системе "Take a number".

5 Очереди сообщений

Было придумано ещё одно средство взаимодействия процессов, чтобы обеспечить разные потребности процессов.

Посылка и приём собщений - самый общий способ взаимодействия. В сетях специально упакованные сообщ. наз пакетами. В ядре есть таблица очередей сообщений.

```
1 #include <sys/msg.h>
2 struct msgid_ds;
```

- msgget;
- msgcnt;
- msgset;
- msgrec; // recieve

При посылке сообщения оно копируется из адресного пр-ва процесса(пр-ва пользователя) в адресное пр-во ядра, при получении – наоборот. msg_spot - указатель на текст сообщения.

Когда пр-с помещает сообщение в очередь, ядро созд. для него новую запись и помещает её в конец списка, соотв. сообщ. указанной очереди. В каждой такой записи указывается тип сообщения, число байт в сообщении и указатель на область данных ядра, в которой находится сообщение.

Ядро коп. ... После этого пр-с, отправивший сообщ, может завершитья. (пр-с не должен блокироваться в ожидании получения сооб. другим процессом) Когда к-либо пр-с выбирает сообщ. из очереди, ядро коп. его в адр. пр-во процесса-получателя, после этого сообщ. удаляется из очереди.

Способы выбора соощения.

- Взять самое старое сообщ. (из головы очереди), независимо от его типа.
- Если идентификатор сообщ. совпадает с идентификатором, который указал пр-с. Если сущ. несколько сообщ. с таким ид., берётся самое старое.

— Пр-с может взять сообщени, числ. значение типа которого есть меньше или равное значению типа, кот указал пр-с. Если этому условию соответствеют несколько сообщений, берётся самое старое. (см. диаграмму 3 состояния блокировки пр-са при передачи сообщ.)

Пример.

```
1 #ifndef MSGMAX
2 #define MSGMAX 1024
3 #endif
4
5 struct mbuf {
6 long msgtype;
7 char mtext[MSGMAX];
8 } mobj = {15, "aaa"};
9
10 int main() {
11     int fd = msgget(100, IPC_CREAT | IPC_EXCL | 0642);
12     if (fd == -1 || msgsnd(fd, &mobj,
          strlen(mobj.mtext), + 1, IPC_NOWAIT))
13         perror("message");
14
15     return 0;
16 }
```

— ;

— ;

6 Очереди сообщений