# Семинар 8 (06.12.2021)

## ?System V?

### Сигналы

Механизм сигналов позволяет процессам реагировать на события, которые могут произойти внутри процесса или вне его. Как правило, получение некоторым процессом сигнала, указывает ему на необходимость завершить свою работу. Вместе с тем реакция процесса на сигнал зависит от того, как сам сигнал определяет своё поведение в случае приёма какого-то определённого сигнала.

Процесс может реагировать на сигнал стандартным образом, то есть в соответствии с существующим в системе обработчиком сигнала, может игнорировать сигнал или вызвать на выполнение свой обработчик сигнала.

Начиная с классического Unix сигналы имеют числовой идентификатор, а также мнемоническую форму записи идентификатора сигнала, которые хранятся в библиотеке <signal.h>. (Мнемоника – запоминания, мнемоническая форма – форма удобная для запоминания)

В классическом UNIX было определенно 20 сигналов.

Укажем некоторые, которые встречаются в наших работах:

#define NSIG 20

#define SIGHUB 1 //разрыв связи с терминалом

#define SIGINT 2 //сигнал завершения программы Ctrl+C

#define SIGUIT 3 // Ctrl+слэш

//...

#define SIGKILL 9

//...

#define SIGSEGV 11 // Нарушение сегментации, выход за пределы сегмента

//...

#define SIGPIPE 13 // ?Запись в канал есть, чтения нет? - недопустимая операция с каналом

#define SIGALARM 14 // Прерывание от таймера

#define SIG 15 // команда kill которая выполняется в ... режиме

#define SIGUSR1 16

#define SIGUSR2 17

#define SIGCHL 18 //сигнал который получает предок при завершении процесса потомка

#define SIG\_DFL(int(\*)()) 0

#define SIG\_IGN(int(\*)()) 1;

Средством посылки и приёма сигналов служат два системных вызова: signal и kill.

int kill**(**int pid**,** int sig**);**

//вызов

kill**(**pid**,** sig**);**

//сигнал sig будет послан процессу с pid и всем потомкам процесса

В UNIX процессы объеденяются в группы. Например, может быть такая ситуация когда в первом параметра указывается pid<= 1: в этом случае сигнал будет послан группе процессов. При pid == 0 сигнал будет послан всем процессам с группой = процессу вызвавшему kill.

//Примеры:

kill**(**37**,** SIGKILL**);** // Приказывает процессу с pid 37 безусловно завершиться

kill**(**getpid**(),** SIGALARM**);** //Процесс вызвавший kill, получит сигнал побудки

//getpid() - получает **собственный** идентификатор

signal() не является стандартным в POSIX 1. Но он определён во всех Си, соответственно во всех Unix подобных системах. Поскольку он не portable, что такое POSIX? POSIX 1– аббревиатура: Portable Operating System Interface.  
  
POSIX 1 FIPS (Federal Information Processing Standard)  
Portable Operating System Interface

В настоящее время существует POSIX 2 (Он включает в себя 1 и еще кое-что…)

(Антимонопольный закон… В Европе создан стандарт X/OPEN)

X/OPEN основан на … POSIX1, POSIX2.

Поскольку signal() не входит в POSIX – его не рекомендуется использовать в переносимом ПО. … поведение отличается от поведения в BSD

signal() – возвращает указатель на предыдущий обработчик данного сигнала, и его можно использовать для восстановления данного сигнала. Еще можно восстановить обработчик сигнала с помощью DFL.

#include <signal.h>

int main**()**

**{**

void**(\***old\_handler**)(**int**)** **=** signal**(**SIGINT**,** SIG\_IGN**);**

/\*действия\*/

signal**(**SIGINT**,** old\_handler**);**

**return** 0**;**

**}**

Поскольку системный вызов signal() не входит в POSI, есть с.в. который входит в него. Это sigaction

int sigaction**(**int sig\_num**,** struct sigaction **\***action**,** struct sigaction **\***old\_action**);**

//struct sigaction определена в <signal.h>

Важнейшей особенностью сигналов, является возможность определить процессу собственную реакцию на сигнал. => можно изменить ход программы.  
sigsetjmp() – устанавливает одну или несколько позиций в программе  
siglongjmp() – осуществляет переход на одну из выделенных позиций

### Pipe (Пайпы – букв. перевод труба)

Изначально были созданы неименованные программные каналы, потом именованные. В отличии от разделяемой памяти, для которой в системе существует таблица разделяемых сегментов, программные каналы поддерживаются файловой подсистемой (Это значит, что у программных каналов есть дескриптор файла). То есть каналы, являются специальными файлами.

Существует два типа каналов:

1. Неименованные – только ?I\_NODE? дескриптор
2. Именованные – имеют имя и ?I\_NODE?

Неименованные каналы создаются системным вызовом pipe(), который возвращает его файловый дескриптор, если удалось его создать. Но поскольку нету имени, а только дескриптор, то пользоваться им могут только процессы родственники, потому что процессы потомки в результате системного вызова fork() наследуют от процесса предка дескрипторы открытых файлов. Потоковая … (как вода в трубах), это симплексная (односторонняя) связь. Для двухсторонней (дуплексной) связи необходимо как минимум две трубы.

Программы используют встроенные средства взаимоисключения. То есть в канал нельзя писать, если из него читают. И из канала нельзя читать, если в него пишут.

(Только пользовательские процессы могут иметь динамический пересчёт)

Именованные каналы создаются командой mknod (mknod <pipe> p)

Соответствующий системный вызов можно вызывать из программы.

mknode**(<**имя**>,** IFIFO**|**ACCESS**,**0**);**

//формальные параметры - с типами

Программные каналы определены в системе для взаимодействия процессов. Программные каналы могут создаваться, только в адресном пространстве ядра системы. Программный канал/pipe буферизуется на 3ёх уровнях:

На 1ом уровне буферизуется в области данных ядра системы  
Обычно размер канала не превышает размера одной страницы (4096 б). При переполнении системной памяти программные каналы (буфера), имеющие наибольшее время существования, переписываются во вторичную память. Эта процедура использует стандартные функции управления или работы с файлами. Если процесс пытается записать в трубу >4096 б, то труба буферизуется во времени, приостанавливая процесс, до тех пор, пока все данные не будут прочитаны.

3 уровня:

1. ?.../В области данных ядра системы?
2. На диске
3. Во времени

Ограничение чтобы повысить эффективность обмена т.к. при это не используется обращения к внешней памяти. (которые явл. медленными)

Программный канал обеспечивает потоковую передачу данных. Кроме этого, канал имеет встроенные средства взаимоисключения, чего не имеет разделяемая память. Поэтому разделяемые сегменты используют с семафорами (всё ложится на программиста). Всё что положили в трубу, так там лежать и будет.

### Очереди сообщений

В ядре системы существует таблица очередей сообщений.

Рисунок… (см. тетрадь)

<sys/msg.h>

Struct msgid\_ds

msg\_spot – указатель на выделенную область память в которой находится сообщение.

На очередях сообщений определены следующие системные вызовы:

msgget**()**

msgctl**()**

msgsnd**()**

msgrcv**()**

struct msg\_buf

**{**

long mytype**;** // тип сообщения

char mytext**[**MSGMAX**];** //то что мы пишем в этом сообщении

**}**

**Важные особенности:**

Когда процесс передаёт сообщения в очередь, ядро создаёт для него новую запись и помещает её в конец связного списка записей, соответствующих сообщениям указанной очереди. В каждой такой записи указываются:

* тип сообщения
* длина сообщения (в байтах)
* указатель на область данных ядра системы, в которую копируется сообщение, и в которой он будет фактически находиться

Ядро копирует сообщение из адресного пространства процесса отправителя в область данных ?ядра? для того чтобы процесс отправитель мог завершиться. При этом сообщение остаётся доступным для чтения другими процессами.

Когда какой-то процесс выбирает сообщение из очереди, ядро копирует это сообщение в адресное пространство этого процесса, после этого, сообщение удаляется.

Процесс может выбрать сообщение из очереди следующими способами:

1. Взять самое старое сообщение, независимо от его типа
2. Взять сообщение, если идентификатор сообщения, совпадает с идентификатором, который указал процесс. Если существует несколько сообщений с этим идентификатор, то берётся самое старое из них.
3. Процесс может взять сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом. Если таких сообщений несколько, то берётся самое.

Таким образом мы видим, что процессы могут не блокироваться в ожидании сообщения и при отправке, и при получении. (\*Вспомнить диаграмму состояний процесса при передаче сообщений)

Пример:

//Пример

//#include ...

#ifndef MSGMAX

#define MSGMAX 1024

#endif

struct mbuf

**{**

long mtype**;**

char mtext**[**MSGMAX**];**

**}**mobj**={**15**,** "Hello"**};**

int main **()**

**{**

int fd **=** msgget**(**100**,** IPC\_CREATE**|**IPC\_EXCL**|**0642**);**

**if** **(**fd **==** **-**1 **||** msgsnd**(**fd**,** **&**mobj**,** strlen**(**mobj**.**mtext**))+**1**,** IPC\_NOWAIT**))**

perror**(**"message"**);**

**return** 0**;**

**}**

В примере создаётся новая очередь с иденетфикатором 100 и устанавливаются следующие права доступа, чтение/запись – 6 для user, только чтение для группы и только запись для остальных. Если msgget успешно, то отправляется “Hello”, с типом 15, при этом этот вызов не блокирующий (IPC\_NOWAIT).

Как мы видим, в отличии от разделяемых сегментов, при передаче сообщений, выполняется копирование. При посылке – из программы в область данных ядра, и при чтении – из области данных ядра в буфер программы. Но при этом, если процессу нужно прочитать сообщение, он имеет гибкие возможности и не будет блокирован при чтении. => очереди сообщений позволяют избежать лишних блокировок.

На очередях сообщений определено довольно много флагов. В частности IPC\_EXCL (прочитать самостоятельно, что этот флаг означает в примере?)