Inter process communication.

На лекциях проблема взаимодействия процессов рассматривалась теоретически.

Средства взаим-я процессов могут предоставляет либо ОС, либо яз (+м.б. JVM)

В UNIX - inter process communication

> С был написан для системного программирования (написания ОС)

Открытые исходные коды ядра ОС

⇓

можно было переписывать всё, в т.ч. сисм. вызовы

⇓

из ранних вар-тов выросли 2 коммерчески значимые ветки: System V и UNIX BSD;
five

⇓

ОС стали лицензионными

⇓

Разработчикам нужно покупать лицензию

...

Возникла монополия

Portable OS Interface for UNIX

POSIX – гос. американский стандарт;

POSIX.1 – 1-я версия

Стандарт POSIX описывает набор интерфейсов, необходимых для достижения переносимости программы.

Принцип разделения цены на аппаратное и программное обеспечение

На рынке важно кол-во продаж, а не завышение цен (это даёт наиб. доход)

„Маркс был неправ" :)

POSIX.1 FIPS,

Federal Information Processing Standard

Потом появился POSIX.2, расш. сист. вызовы;

Но этого было недостаточно;

В Европе появился X/Open – это сделало рынок более свободным;

Большее кол-во людей и компаний могут разрабатывать ПО

Организованность процесса ↓
Качество ПО ↑

IPC System V.

- Сигналы;
- Семафоры;
- Прогр. каналы (именованные и неименованные);
- Очереди сообщений;
- Сегменты разделяемой памяти;

Здесь не рассм. RPC (Remote Procedure Call) (средства межмашинного взаимодействия)

Сигналы

Сигналы информируют один процесс или группу процессов;

Далее будет рассм. материал не по Linux, а по классическому UNIX!

Сигнал – программное средство информирования процессов о событиях, которые могут происходить как внутри самих процессов (синхронные события), так и вне процессов (асинхронные события относительно процессов).

Синхронный – при выполнении процесса возникает событие, сопровождаемое сигналом;

Источник сигнала – процесс;
Он может перехватить свой сигнал, тогда это будет синхронным событием;

Асинхронное событие – то, которое происходит вне процесса, т.е. независимо от того, в какой точке находится процесс;

Все процессы ОС – асинхронные, т.е. выполняются со своей скоростью, независимо от того, как выполняются другие процессы.

Асинхронность – базовое понятие ОС.

Как правило, получение процессом сигнала указывает ему на необходимость завершить выполнение.

Реакция процесса на получаемый сигнал зависит от того, как сам процесс определяет свою реакцию на этот сигнал.

Процесс может
1) проигнорировать полученный сигнал;
2) реагировать на полученный сигнал по умолчанию (так, как определено в ОС);
3) определить собственную реакцию на полученный сигнал (для этого в коде процесса должен быть написан собственный обработчик сигнала);

MAC OS основан на UNIX BSD

Все ОС поддерживают POSIX, но интерфейсы разные (хотя суть одна и та же);

POSIX оговаривает только систем. вызовы, т.е. API (интерфейс), а ф-ции ядра нигде не регламентируются ⟹ ⟹ везде разные;

В классическом UNIX не м.б. >20 сигналов. Рассмотрим некоторые из них.

```
#define NSIG 20
#define SIGHUP 1  //SIGHUP сопровождает разрыв связи с упр. терминалом
#define SIGINT 2  //Прерывание (завершение) выполнения процесса на Ctrl+C
#define SIGKILL 9 //Уничтожение процесса (сист. вызов kill)
#define SIGSEGV 11 //нарушение сегментации, выход за пределы сегмента
                   (virtual)
//Рассматриваем вирт. адр. пр-во в x86 (из методички по опт. fork):
//В 32-разр. ОС процесс мог иметь виртуальное адр. пр-во размером 4 Гб
//В старших адресах - mapping системы (1 Гб);
//Оставшиеся 3 Гб - занимаемое адресное пр-во
```

// В Windows: 2 Гб защищ. адр. пр-во + 2 Гб mapping
// Таненбаум: объёмы кода UNIX и Windows отличаются на порядок;
объём кода - минус ОС, т.к. чем больше кода, тем больше ошибок;

// В Windows 2000 нет выделенного модуля планировщика,
// т.е. задачи планирования время от времени решают
// разные участки кода, т.е. код Windows не структурирован (в отличие от UNIX/Linux)
// Не смотря на то, что в совр. ОС осуществляется
// упр-е памятью страницами по запросу, понятие
// сегмента существует, так как оно эффективно и позволяет
// разделить назначение участков кода, выделять память (виртуальную); а в классическом UNIX было упр-е памятью сегментами по запросу; это понятие сохраняется;

#define SIGSYS 12 // ошибка арг-я сист. вызова
#define SIGPIPE 13 // Запись в канал есть чтения нет
// - это невозможно, т.к. это может привести к заполнению канала ⟹ записывающий процесс будет блокирован - крайне негативная ситуация
#define SIGALARM 14 // Сигнал будильника (для этого надо повеситься на таймер)
#define SIGTERM 15 // Сигнал завершения
#define SIGUSR1 16 // Пользовательские сигналы
#define SIGUSR2 17
#define SIGCLD 18 // В Linux это SIGCHLD (сопровождает завершение потомка)
#define SIGPWR 19 // Сигнал отключения питания
// ОС перехватывает падение напряжения, чтобы сохранить свою работоспособность (сохранить критические данные для возобновления своей работы)
// SIGPWR может перехватываться приложениями,
// чтобы сохранить все важные данные в файле

„Структуры могут работать только с файлами, с долговременным хранением"

Средства посылки и восприятия сигналов в UNIX:
2 сист. вызова: kill и signal;

int kill (int pid, int sig);

↳ м.б. ≤0, это обеспечивает получение сигнала группой процессов;

Пример:
kill (getpid(), SIGALARM);
Сигнал побудки будет послан процессу, вызвавшему kill;

При работе с Linux самым важным документом является man!
с Windows - MSDN (поддерживается Microsoft)

CLR очень удобный

Если pid ≤ 1, то сигнал будет послан группе процессов.
Если =0, то будет послан всем процессам с g.id = g.id процесса, вызвавшего kill;
Если =-1, то сигнал будет послан процессу, который имеет тот же uid, что и процесс, вызвавший kill; и т.д.
В UNIX огромное значение имеют группы процессов;
Процессы одной группы могут получать одни и те же сигналы и реагировать на них единообразно.

void signal (*signal (int sig, void (*handler)(int)))(int);
В рез-те получения сигнала вызовется handler;

Signal не входит в POSIX, но входит в ANSI C ⟹ есть в любой поставке, но реакция на сигнал при исп-ии signal отличается в разных ОС ⟹

⟹ signal не рекомендуется использовать в переносимом П.О.;

signal возвращает указатель на старый обработчик. Пример:

```
#include <signal.h>
int main()
{
    void (*old_handler)(int)=signal(SIGINT, SIGIGN);
    /* обработка */
    signal(SIGINT, old_handler); //восстановили реакцию на сигнал
    //---
}
```

Здесь не идёт речь об обработчике, установленном в ОС по умолчанию, т.к. для этого есть SIG_DFL;

int sigaction(int sig_numb, struct sigaction *action, struct sigaction *old_action);
(входит в POSIX)

struct sigaction — заполняются поля структуры, а не „создаётся структура"! ещё она может объявляться и ничего больше! („Это не пустяк, это профессионализм")

(в отличие от signal, работает со структурой)

„Уровень профессионализма — чётко понимать, что вы делаете";

Техника сигналов позволяет изменять ход вып-я программы, т.е. мы можем внешним (асинхронным по отн-ю к нашей программе) событием менять ход её вып-я;

sig setjmp() — устанавливает точки перехода;
sig longjmp() — переходит на конкретную точку перехода

Это ещё более гибкий механизм изменения хода выполнения программы.

Программные каналы (pipe)

Именованные прогр. каналы создаются командой mknod (созд. спец. файл-буфер типа FIFO)
(или может пользоваться любой процесс, который знает имя)

Неименованные – сист. вызовом pipe();
↳ нет имени, но есть дескриптор (struct inode);

ls -i # увидеть номер inode (index node)

это идентификатор файла, а не его имя (см. лабу №1 по UNIX)

Потомки наследуют дескрипторы открытых файлов (struct inode, а не их имена) ⟹ наследуют и дескрипторы программных каналов;

Pipe – модель потоковой передачи данных (симплексная связь)

Программный канал (симв. или неимен.) – буфер типа FIFO;
Буфер – область или диапазон адресов в физической памяти, на которую мы получаем указатель и у которой устанавливается размер

FIFO – первое сообщение первым будет прочитано и исчезнет (ничего не сдвигается, просто ptr перемещается)

При создании буфера с помощью pipe размер не указывается...

„Трубы" (pipe) буферизуются на 3 уровнях:

1) <u>буферизация в системной памяти</u>; при переполнении сист. памяти буфера, имеющие наиб. время сущ-я, переписываются на диск. При этом используются станд. ф-ции работы с файлами: read и write.
2) Если процесс записывает > 4096 б (размер 1 страницы), она <u>буферизуется во времени</u>, приостанавливая процесс до тех пор, пока все данные не будут прочитаны;

⟹ в сист. вызове pipe не указывается размер, т.к. канал создаётся <u>размером в 1 страницу</u>. Это связано с тем, что в ОС перемещение страниц всегда оптимизировано.

Если созданный канал заполняется, то записывающий процесс (вызывающий write и работающий быстрее читающих) будет приостановлен до тех пор, пока все данные не будут прочитаны, <u>и наоборот</u>.

Т.к. это потоковая передача данных, то, для того чтобы обеспечить монопольный доступ к каналу, на нём <u>реализовано взаимоисключение</u>;

Из трубы нельзя читать, если в неё пишут, т.к. это может привести к получению неполного сообщения;

В трубу нельзя писать, если из неё читают — это также может привести к получению неправильной информации

Сама особенность такого буфера потребовала реализации на ней взаимоисключения. Для этого мы объявляем массив критических дескрипторов, чтобы регулировать запись и чтение из трубы.

На экзамене всё это надо будет реализовать в комплексе, собрав всё вместе и приводя примеры из л/р