Лекция 30
каналы

Перенаправление ввода/вывода

- Для перенаправление ввода или вывода после fork() необходимо открыть нужный файл и потом скопировать его файловый дескриптор в стандартные файловые дескрипторы 0, 1 или 2.
- Копирование файлового дескриптора: dup2
  - int dup2(int oldfd, int newfd);
- Пример: redir.c

Соответствие с флагами open

- [n]> FILE
  - fd = open(FILE, O_WRONLY | O_CREAT | O_TRUNC, 0666); dup2(fd, n); close(fd);
- [n]>>FILE
  - fd = open(FILE, O_WRONLY | O_CREAT | O_APPEND, 0666); dup2(fd, n); close(fd);
- [n]<FILE
  - fd = open(FILE, O_RDONLY, 0); dup2(fd, n); close(fd);
- [n]>&[m]
  - dup2(m, n);

Неименованные каналы

- Канал — механизм синхронной однонаправленной потоковой передачи данных между процессами
  - Синхронной — процесс-писатель и процесс-читатель могут синхронизировать работу по write/read
  - Потоковой — при передаче в канале не сохраняются границы записанных блоков, последовательность данных, записанных несколькими write, может быть прочитана за один read

Передача данных

- Посылка данных — write
- Прием данных — read
- Read и write могут быть не синхронизированы по времени — в ядре ОС находится буфер для временного хранения данных
- Буфер имеет **ограниченный** и **фиксированный** размер
- Например, Linux — 64 KiB. Пример: pipesize.c

Создание канала

int pipe(int fds[2]);

- Создаются два связанных файловых дескриптора, которые возвращаются в массиве fds
- fds[0] — файловый дескриптор для чтения из канала
- fds[1] — файловый дескриптор для записи в канал

Запись в канал: write

- Если все ф. д. чтения из канала закрыты, при попытке записи процессу посылается SIGPIPE и write возвращает код ошибки EPIPE
- Если размер данных <=PIPE_BUF
  - Если в канале достаточно места, данные записываются в канал атомарно
  - Если в канале места недостаточно, записывающий процесс блокируется либо до освобождения места, либо до закрытия всех ф. д. чтения
- Если размер данных > PIPE_BUF, атомарность записи не гарантируется

Чтение из канала: read

- Если в канале нет данных и все ф. д. записи в канал закрыты, read возвращает 0 — признак конца файла
- Если в канале есть данные, то read завершается немедленно и возвращает минимум из запрошенного и имеющегося в канале размера
- Если в канале нет данных, процесс блокируется до наступления одного из первых двух условий

Взаимная блокировка, тупик (deadlock)

- Проблема синхронизации процессов, когда ни один из взаимодействующих процессов не может продолжить выполнение, так как ожидает выполнения действия другим процессом

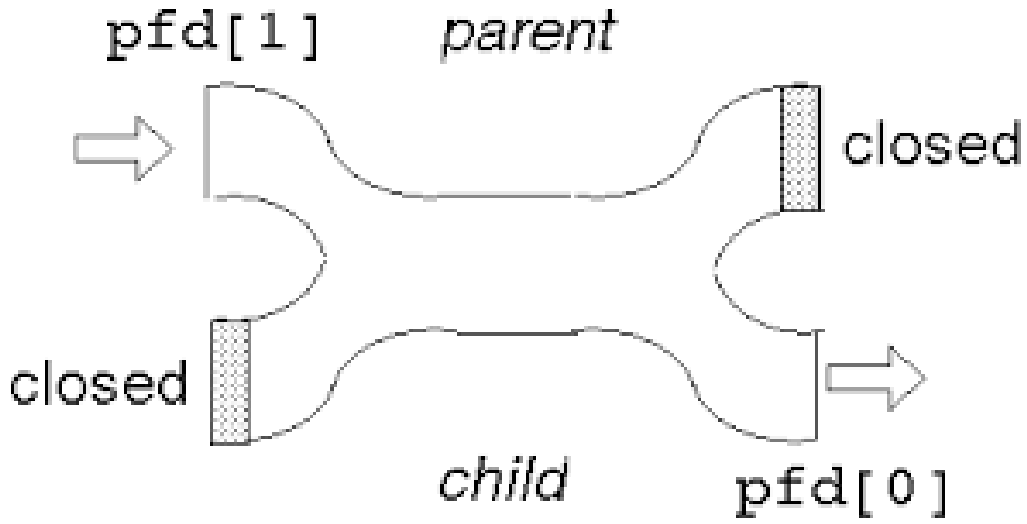
```
int fds1[2], fds2[2];
pipe(fds1);
pipe(fds2);
```

```
read(fds1[0],b,sizeof(b));
write(fds2[1],b,sizeof(b));
```

```
read(fds2[0],b,sizeof(b));
write(fds1[1],b,sizeof(b));
```

Использование каналов для конвейера (pipeline.c)

```
int main(void)
{
  pipe(pfd);
  if (!(pid1 = fork())) {
    dup2(pfd[1], 1); close(pfd[1]);
    execlp("/bin/ls","/bin/ls", "-l", NULL);
  }
  if (!(pid1 = fork())) {
    dup2(pfd[0], 0);
    close(pfd[0]);
    execlp("/usr/bin/wc",
      "/usr/bin/wc", "-l", NU
  }
  wait(0); wait(0);
  return 0;
}
```



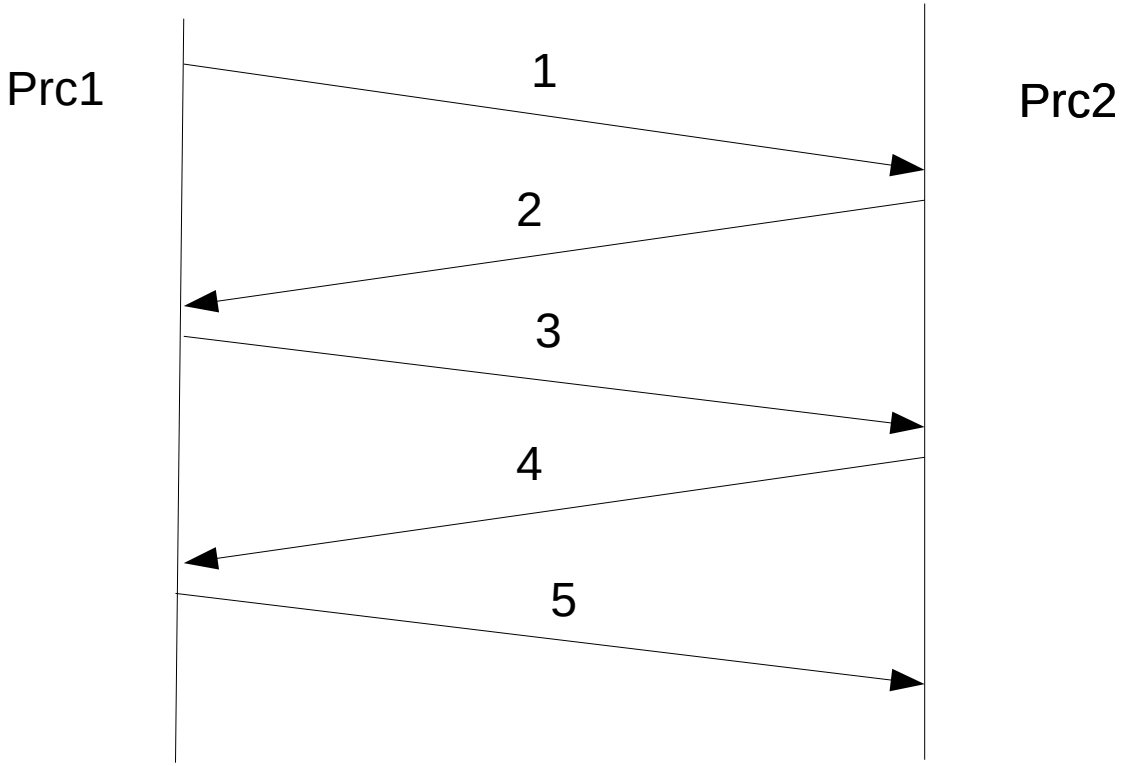
Использование каналов для конвейера

```
int main(void)
{
  pipe(pfd);
  if (!(pid1 = fork())) {
    dup2(pfd[1], 1); close(pfd[1]); close(pfd[0]);
    execlp("/bin/ls","/bin/ls", "-l", NULL);
  }
  if (!(pid1 = fork())) {
    dup2(pfd[0], 0); close(pfd[0]); close(pfd[1]);
    execlp("/usr/bin/wc","/usr/bin/wc", "-l", NULL);
  }
  close(pfd[0]); close(pfd[1]);
  wait(0); wait(0);
  return 0;
}
```

Все ненужные файловые дескрипторы должны быть закрыты!

Задача ping-pong

- Модельная задача взаимодействия двух процессов
- Поочередный обмен сообщениями между процессами



Ping-pong

- Чтобы данные процессов не перемешивались, два варианта
  - Использовать два pipe:
    - (1) от первого процесса ко второму,
    - (2) от второго процесса первому
  - Использовать один pipe, но арбитрировать доступ к pipe другими средствами
- Пример: pingpong.c

Низкоуровневый и высокоуровневый ввод-вывод (C)

- fdopen — создает структуру FILE по файловому дескриптору

  FILE *fdopen(int fd, const char *mode);

- fileno — получить файловый дескриптор по структуре FILE
  int fileno(FILE *f);

- Если из pipe создать FILE *, то он по умолчанию будет **полностью буферизован**

- Пример: fdopen.c

Лекция 31
сигналы

Сигналы

- Средство асинхронного взаимодействия процессов
- Посылаются:
  - Одним процессом другому процессу
  - Ядром ОС процессу для индикации событий, затрагивающих процесс
  - Ядром ОС процессу в ответ на некорректные действия самого процесса
  - Процессом самому себе

Виды сигналов

- Асинхронные — могут поступить процессу и вызвать обработку в произвольный момент времени
- Синхронные — поступают процессу и вызывают обработку в определенные моменты времени (например, в результате вызова kill, или в результате попытки выполнения некорректной инструкции)

Стандартные сигналы

- Взаимодействие процессов (асинхронное):
  - SIGINT — завершение работы процесса, посылается при нажатии Ctrl-C
  - SIGTERM — завершение работы процесса
  - SIGKILL — завершение работы процесса (нельзя предотвратить)
  - SIGQUIT — завершение работы процесса с выдачей core dump
  - SIGUSR1, SIGUSR2 — произвольного назначения (определяется пользователем)
  - SIGSTOP — приостановка работы процесса (нельзя предотвратить)

Стандартные сигналы

- Ядро процессу (асинхронные)
  - SIGHUP — отключение от терминала
  - SIGALRM — срабатывание таймера
  - SIGCHLD — завершение работы сыновнего процесса

Стандартные сигналы

- Ядро процессу в ответ на ошибочное действие (синхронные)
  - SIGILL — недопустимая инструкция
  - SIGFPE — ошибка вычислений с плавающей точкой (обычно — целочисленное деление на 0)
  - SIGSEGV — ошибка доступа к памяти
  - SIGPIPE — запись в канал, закрытый на чтение

Стандартные сигналы

- Процесс самому себе (синхронные сигналы)
  - SIGABRT

- Перечислены не все сигналы. См. список: man 8 signal
- Процесс может послать другому процессу любой сигнал (в т. ч., например, SIGSEGV)
- Все сигналы, посылаемые одним процессом другому, асинхронны
- Сигналы, посылаемые процессом самому себе синхронны

Способы обработки сигнала

- Стандартная реакция на сигнал (реакция по умолчанию)
  - Завершение работы процесса (большинство сигналов)
  - Завершение работы процесса с записью core dump (SIGSEGV, SIGABRT...)
  - Пустая реакция (ничего не делать) (SIGCHLD)
  - Приостановка работы процесса (SIGSTOP)
- Игнорирование сигнала (кроме SIGSTOP, SIGKILL)

Способы обработки сигнала

- Пользовательская обработка — назначение функции, которая будет вызвана для обработки поступившего сигнала
- Функция обработчик:
  void handler(int signal);
- Не возвращает значения, принимает номер сигнала, который обрабатывает — одна и та же функция-обработчик может использоваться для обработки нескольких сигналов

Специальные сигналы

- Сигналы SIGKILL и SIGSTOP не могут быть перехвачены, заблокированы или проигнорированы
  - SIGKILL снимает процесс с выполнения ВСЕГДА!

Посылка сигнала

- Системный вызов kill

```
int kill(pid_t pid, int sig);
```

- pid > 0 — посылка указанному процессу
- pid == 0 — посылка всем процессам текущей группы
- pid == -1 — посылка всем процессам, которым процесс имеет право послать сигнал
- pid < -1 — посылка всем процессам в группе -pid

Доставка сигнала

Процесс 1000

```
kill(1234, 5);
```

Процесс 2000

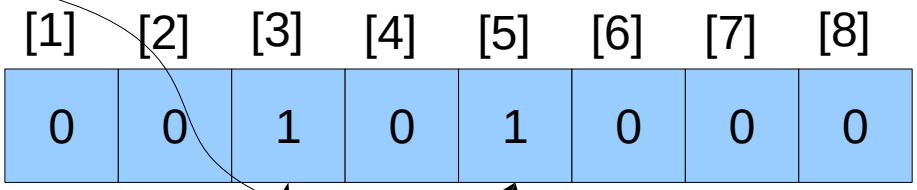
```
kill(1234, 3);
```

Процесс 2009

```
kill(1234, 5);
```

Процесс pid == 1234

Маска сигналов, ожидающих доставки (pending)



Доставка сигнала (I этап)

- Во множестве сигналов, ожидающих доставки, устанавливается соответствующий бит (возможно, он уже был установлен)
- Если процесс был заблокирован из-за ожидания ввода-вывода, он переносится в очередь процессов, готовых к выполнению, и настраиваются действия пост-обработки

Доставка сигнала (II этап)

- При запуске процесса из очереди готовых процессов на выполнение проверяется множество сигналов, ожидающих доставки и не заблокированных
- Из таких сигналов выбирается некоторый сигнал (обычно с минимальным номером) и доставляется процессу:
  - Удаляется из множества сигналов, ожидающих доставки
  - Производится либо стандартная обработка, либо вызов пользовательской функции

Доставка сигнала (II этап)

- При вызове пользовательской функции:
  - Производится настройка стека для обеспечения продолжения работы процесса после завершения обработчика
  - Модифицируется множество заблокированных процессом сигналов (только на время работы обработчика)

Слияние сигналов

- От посылки сигнала процессу до начала выполнения функции обработки может пройти некоторое время
- За это время один и тот же сигнал может быть послан процессу несколько раз
- Обработчик сигнала будет вызван **только один раз**
- Сигналы нельзя использовать там, где требуется учет количества поступлений

Установка обработки сигнала

```
typedef void (*sighnd_t)(int);
sighnd_t signal(int signum, sighnd_t hnd);
```

- SIG_IGN — игнорировать сигнал
- SIG_DFL — установить обработку по умолчанию
- Иначе задается функция-обработчик
- Возвращается старая обработка сигнала
- Обработку сигналов SIGKILL, SIGSTOP изменить нельзя

Особенности обработки сигналов в разных системах

- Переустановка обработчика: в System V обработчик сигнала сбрасывается на обработку по умолчанию, в BSD и Linux — нет

System V

```
void hnd(int signo)
{
    signal(signo, hnd);
    // …
}
```

BSD, Linux

```
void hnd(int signo)
{
    // …
}
```

Если в System V непрерывно посылать процессу сигналы при высокой загрузке системы, процесс не успеет восстановить обработчик сигнала и ядро снимет процесс с выполнения — DOS (Denial of Service) атака

Особенности обработки сигналов в разных системах

- Блокирование сигнала: в BSD и Linux на время обработки сигнала этот сигнал блокируется, в System V не блокируется
- При высокой загрузке системы в System V поток сигналов может привести к повторному входу в обработчик сигнала, что может привести к ошибке

Особенности обработки сигналов в разных системах

- Перезапуск системных вызовов: если сигнал был доставлен в процесс в тот момент, когда процесс находится в состоянии ожидания:
  - В System V системный вызов завершается с ошибкой EINTR, эту ошибку в процессе необходимо обработать и при необходимости перезапустить системный вызов
  - В BSD, Linux системный вызов перезапускается автоматически
  - Не перезапускаются: sleep, pause, select

Обработка сигналов в Linux

- На время выполнения обработчика сигнала повторное поступление сигнала блокируется. Если сигнал поступил в это время, обработчик сигнала будет перезапущен как только доработает до конца.
- Настройки обработки сигналов сохраняются при вызове обработчика. Обработчик достаточно установить один раз.
- Если процесс ожидал обмена данными (вызовы read, write, open, accept, wait, …), после завершения обработчика процесс продолжит ожидание обмена.
- Если процесс ожидал прихода сигнала (sleep, pause, usleep, nanosleep, sigsuspend, select, pselect, …), после завершения обработчика обработка системного вызова завершится

Особенности обработки сигналов в разных системах

- Схема обработки сигналов BSD и Linux более удобна для программиста и более надежна
- В дальнейшем будем предполагать, что используется эта схема
- Системный вызов sigaction позволяет устанавливать обработчик произвольным образом комбинируя свойства

Обработчики сигналов

- В обработчиках сигналов можно использовать только асинхронно-безопасные (async signal safe) стандартные функции
- Большинство системных вызовов (включая fork и exec) — асинхронно-безопасные
- Функции работы с динамической памятью (new, delete, malloc, free), функции работы с потоками (fopen, fprintf, <<) - **не асинхронно-безопасные**