

Межпроцессное взаимодействие

АКОС, МФТИ

07 ноября, 2024



Зачем передавать данные между процессами?



Например, чтобы подсвечивать код в IDE. Хотя это встречается везде.

Можно общаться через файл:

```
1  int send(const char* msg) {  
2      int len = strlen(msg);  
3      return write(file, msg, len);  
4  }
```

Отправка сообщения

```
1  void receive(char* buf, int len) {  
2      int res = 0;  
3      while(true) {  
4          res = read(file, buf, len);  
5          if (res != 0) break;  
6          sleep(1);  
7      }  
8      return res;  
9  }
```

Приём сообщения

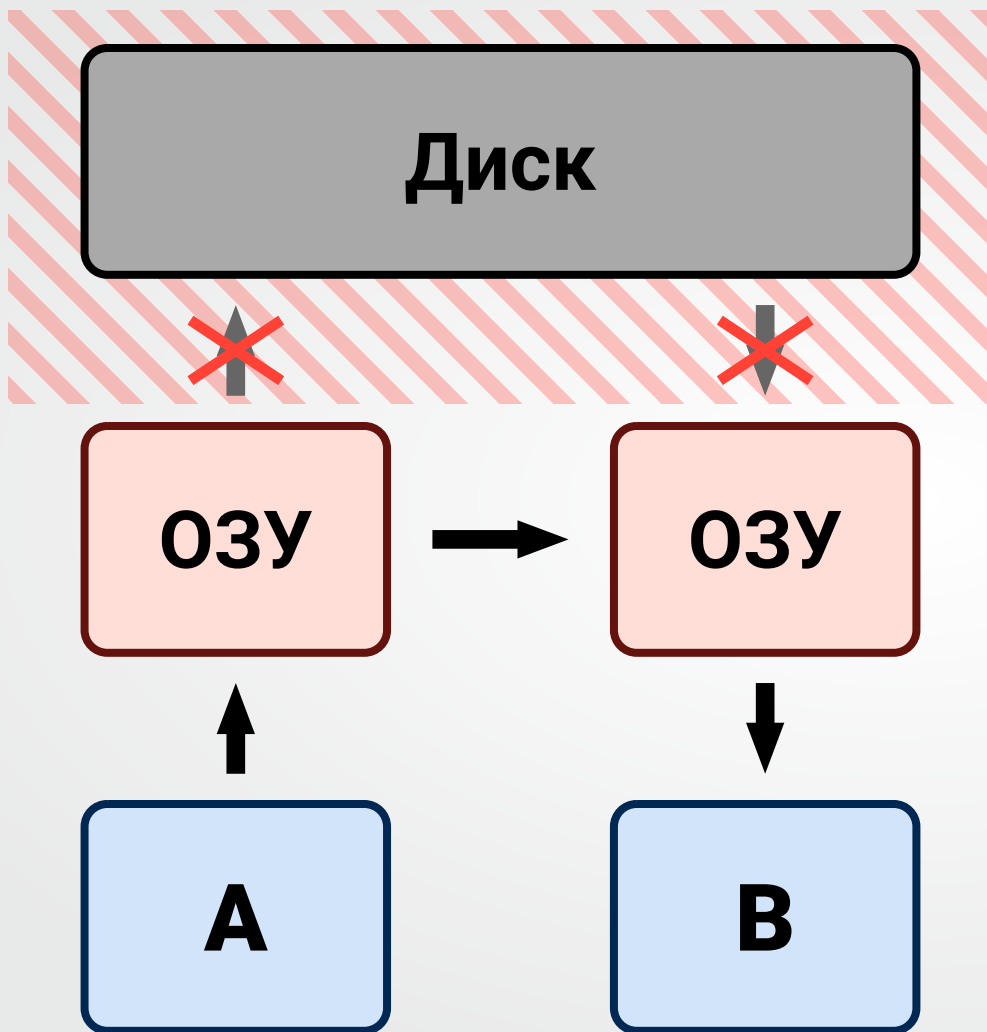
Вопрос

Чем плох такой подход?

Проблемы передачи данных через файл:

- Поллинг **добавляет задержку** и **нагружает CPU**.
- Реализовать без поллинга **сложно** (нужен `inotify`).
- Задействуем файловую систему – **медленно**.
- На скорость передачи будет влиять скорость диска, но даже с быстрым SSD это будет **медленно**.





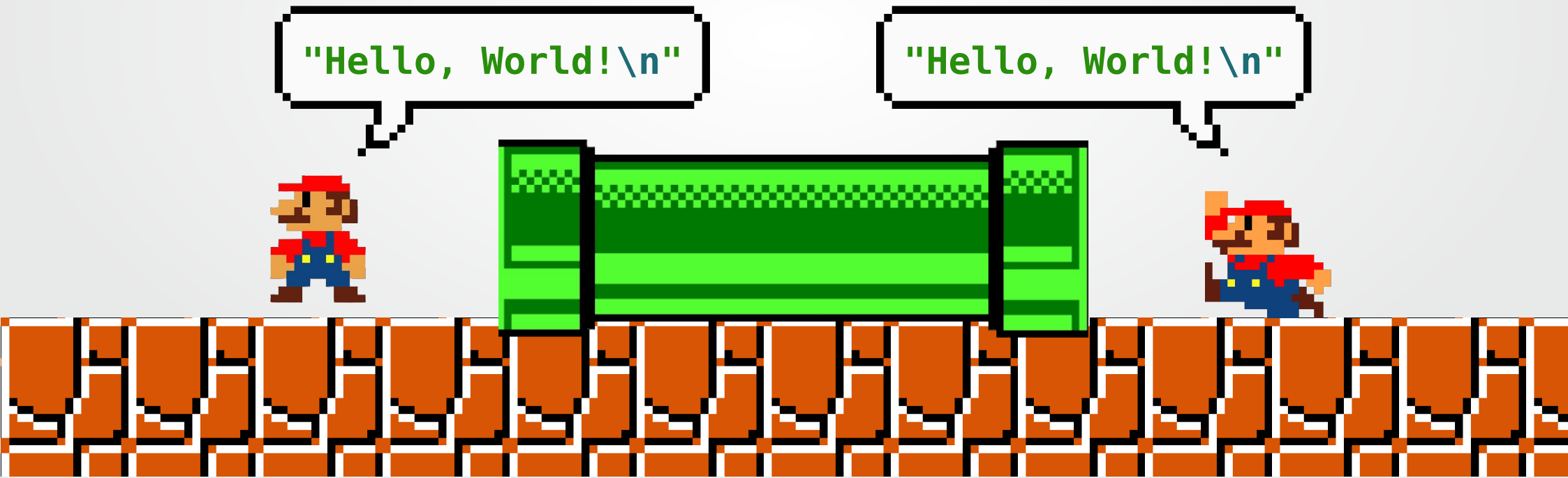
Давайте жить в ОЗУ!

Нам не нужен диск как таковой.
Можно срезать путь, если
передавать данные напрямую,
не покидая оперативную
память.

```
pipe(int fds[2])
```

Создает неименованный канал (два парных дескриптора).

- Байты, записанные во второй дескриптор, можно будет прочитать из первого;
- Канал однонаправленный, имеет ограниченный размер буфера.



Общаемся через трубу канал

```
1  int pipefd[2] = {};  
2  pipe(pipefd);  
3  
4  if(fork() != 0) {  
5      // Если мы - процесс-родитель, то кричим в трубу  
6      write(pipefd[1], "Friendly message\n", 18);  
7      wait(NULL);  
8  } else {  
9      // Если мы - процесс-ребёнок, то слушаем трубу:  
10     char buffer[32] = {};  
11     read(pipefd[0], buffer, 31);  
12     printf("Received %s", buffer); // "Received Friendly message\n"  
13 }  
14  
15 close(pipefd[0]); // Дескрипторы каналов тоже нужно закрывать  
16 close(pipefd[1]);
```

Заменяем стандартный ввод/вывод

```
dup2(int fd, int new_fd)
```

Копирует файловый дескриптор по номеру `fd` в номер `new_fd`

- Если целевой дескриптор уже существует, он будет предварительно закрыт.
- Таким образом можно **перенаправлять потоки данных**.
- Например, заменить стандартный вывод одного процесса на канал, ведущий в стандартный ввод другого процесса.
- Еще есть `dup(int fd)`. Это как `dup2`, но `new_fd` выбирается системой.



Склеиваем ввод и вывод

```
1  int pipefd[2] = {};  
2  pipe(pipefd);  
3  
4  if(fork() != 0) { // Если мы - процесс-родитель:  
5      dup2(pipefd[1], STDOUT_FILENO); // - Подключаем STDOUT к началу канала;  
6      printf("FriendlyMessage\n"); // - Пишем в STDOUT (канал);  
7      fflush(stdout); // - Убеждаемся, что данные улетели  
8      wait(NULL); // - Ждем ребёнка;  
9  } else { // Если мы - процесс-ребёнок:  
10     dup2(pipefd[0], STDIN_FILENO); // - Подключаем STDIN к концу канала;  
11     char buffer[32] = {}; //  
12     scanf("%31s", buffer); // - Читаем STDIN (канал);  
13     printf("Received %s", buffer); // - "Received FriendlyMessage\n";  
14 }  
15  
16 close(pipefd[0]); // Дескрипторы каналов тоже нужно закрывать  
17 close(pipefd[1]);
```

Оператор `|` из вселенной `$ bash`

Превращает вывод предыдущей команды в ввод следующей команды.

```
$ ps
```

Список всех процессов

```
$ ps | grep java
```

Список всех процессов, содержащих `java`

```
$ cat file
```

Вывести содержимое файла

```
$ cat file | sort
```

Вывести отсортированные строки файла

```
$ cat file | sort -u
```

Вывести уникальные строки файла

```
$ cat file | sort -u | wc -l
```

Посчитать уникальные строки файла

Под капотом он точно так же создаёт `pipe` и делает `dup2`. Это достаточно дорого.

Поэтому вместо `$ cat file | grep pattern` лучше написать `$ grep pattern file`.

Чтение из `pipe`


Данные есть:

 Прочитать их

Данных нет, писатели есть:

 Ждать

Данных нет, писателей нет:


 Вернуть ноль

Запись в `pipe`

Есть место, есть читатели:

 Записать данные в буфер канала

Места нет, есть читатели:

 Ждать

Читателей нет:

 Вернуть ошибку **Broken Pipe**

Задача

Придумайте, как можно поймать deadlock

Привет из многопоточки!

Если дескриптор канала утечёт в другой процесс, то чтение может ждать вечно.

```
1  int pipefd[2] = {};  
2  pipe(pipefd);  
3  
4  if (fork() == 0) { // Порождаем дочерний процесс  
5      execve(...);  
6      return -1;  
7  }  
8  
9  write(pipefd[1], msg, sizeof(msg));  
10 close(pipefd[1]);  
11  
12 char buffer[32] = {};  
13 // Этот read() будет ждать вечно, т.к pipefd[1] не закрыт в дочернем процессе  
14 while (read(pipefd[0], buffer, sizeof(buffer)) != 0);
```

Мораль – закрывайте дескрипторы!

Настройки каналов

```
fcntl(fd, F_SETPIPE_SZ, 65536)
```

- Изменить размер буфера канала.

```
fcntl(fd, F_SETFL, old_flags | O_NONBLOCK)
```

- Запретить блокирующее чтение, даже если в буфере канала пусто.
- Вместо ожидания **read()** вернёт ошибку и установит **errno = EWOULDBLOCK**.
- Нужно получить старые флаги через **fcntl(fd, F_GETFL)**.

```
PIPE_BUF
```

- Максимальный размер атомарной записи.
- Записи большего размера могут быть разбиты на несколько.
- Эта настройка изменяется перекомпиляцией ядра.

```
mkfifo(const char* path, mode_t mode)
```

Создаёт именованный канал.

- Именованный канал имеет путь. Его можно открыть через `open()`;
- Только один процесс может открыть `fifo` на чтение;
- Открытие `fifo` **блокирующее**:
 - Открытие на чтение ждёт, пока появится писатель;
 - Открытие на запись ждёт, пока появится читатель.
- Именованные каналы позволяют взаимодействовать любым процессам, имеющим доступ к этому пути, а не только родителю с детьми.



Сигналы

Сигнал

Простое асинхронное сообщение, которое можно отправить процессу.

- В сообщении хранится только номер сигнала - число от 1 до 64.
- Сигнал может отправить как процесс, так и ядро.
- Процесс, получивший сигнал, может отреагировать по-разному:

Term

Завершить работу

Core

Завершить работу и сгенерировать [Core Dump](#)

Ign

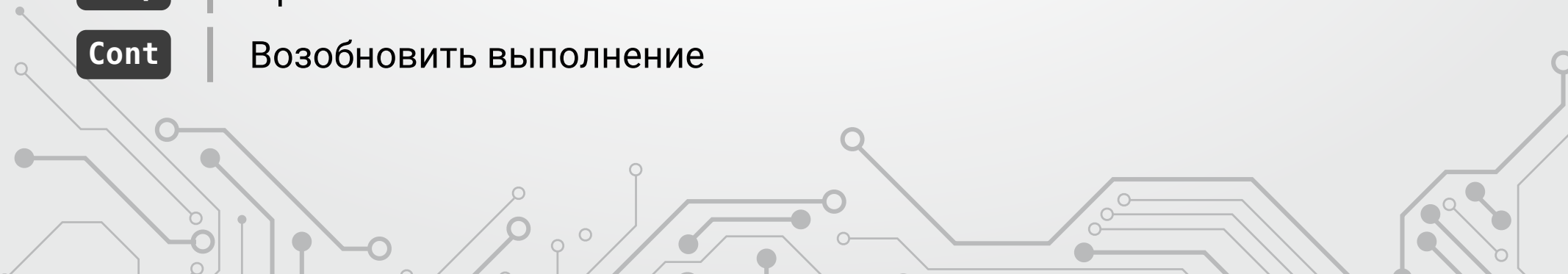
Проигнорировать сигнал

Stop

Приостановить выполнение

Cont

Возобновить выполнение



Сигнал	Действие	Описание
SIGKILL	Term	Завершение работы (нельзя проигнорировать)
SIGABRT	Core	Завершение работы (нельзя проигнорировать)
SIGTERM	Term	Мягкое завершение работы
SIGSEGV	Core	Ошибка сегментации
SIGCHLD	Ign	Завершился дочерний процесс
SIGSTOP	Stop	Ctrl+Z (нельзя проигнорировать)
SIGCONT	Cont	Возобновление работы
SIGURG	Ign	Нужно прочесть что-то важное
SIGINT	Term	Ctrl+C
SIGQUIT	Core	Ctrl+\
SIGALRM	Term	Сработал таймер <code>alarm()</code>
SIGPIPE	Term	Получена ошибка <code>Broken Pipe</code>

Это некоторые из существующих POSIX-сигналов.

alarm(unsigned int seconds)

Заводит таймер, по истечении которого процесс получает сигнал SIGALRM

- По умолчанию процесс завершит работу, получив SIGALRM, но можно установить свой обработчик.
 - Вызов **alarm(0)** отменяет таймер.
-

abort()

Провоцирует немедленное получение сигнала SIGABRT.

- Вызывается, если сделать **assert(false)**
- SIGABRT делает Core Dump. Через это **abort()** может помогать в отладке.

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

Отправляет сигнал процессу. Тот случай, когда название сбивает с толку.

```
pid_t pid
```

- **Номер процесса, которому нужно отправить сигнал.**
- **0:** Отправить сигнал своей группе процессов.
- **-1:** Отправить сигнал всем, кому можно (кроме init-процесса).
- **-n:** Отправить группе процессов с номером n

```
int sig
```

- **Номер отправляемого сигнала.**
- **-0:** Не отправлять сигнал, но проверить разрешения.

Сигналы можно отправлять:

- Процессам с тем же пользователем
- Если вы - суперпользователь, то кому угодно, кроме init-процесса

Как узнать, завершился ли процесс сигналом?

Системный вызов `waitpid(...)` записывает **битовую маску**, в которой закодирован статус возврата:

`WIFEXITED(status)`

- **true** при нормальном завершении;
- Код возврата – `WEXITSTATUS(status)`.

`WIFSIGNALED(status)`

- **true** при завершении сигналом;
- Номер сигнала – `WTERMSIG(status)`.

```
1  int status = 0;
2  waitpid(pid, &status, 0);
3
4  if (WIFEXITED(status)) {
5      // Процесс завершился нормально
6      int retcode = WEXITSTATUS(status);
7  }
8
9  if (WIFSIGNALED(status)) {
10     // Процесс был завершён сигналом
11     int signum = WTERMSIG(status);
12 }
```

```
sigaction(int sig, const sigaction* act, sigaction* oldact)
```

Устанавливает обработчик **act** на сигнал **sig**. Старый обработчик сохраняет в **oldact**

- **oldact** может быть **NULL**, если он не нужен;
- Собственный обработчик нельзя установить на **SIGSTOP** и **SIGKILL**.

```
struct sigaction
```

Структура, задающая обработчик. Её поля:

void (***sa_handler**) (**int** signo) - Указатель на функцию-обработчик сигнала;

sigset_t **sa_mask** – Какие сигналы блокировать при обработке этого сигнала.

int **sa_flags** – Флаги, о них позже.

... – И ещё некоторые поля для *Real-Time* сигналов

- **sa_handler** можно установить в **SIG_DFL** или **SIG_IGN**. Это **default** и **ignore**.

sigset_t

Набор сигналов. Хранится как битовая маска. Не имеет конструктора и деструктора.

```
sigemptyset(sigset_t* set);
```

Пустое множество

```
sigfillset(sigset_t* set);
```

Полное множество

```
sigaddset(sigset_t* set, int signum);
```

Добавить сигнал

```
sigdelset(sigset_t* set, int signum);
```

Удалить сигнал

```
sigismember(sigset_t* set, int signum);
```

Проверить наличие сигнала



Обрабатываем Ctrl+C

```
1 // Простейший обработчик
2 void handler(int signum) {
3     char message[] = "You've pressed Ctrl+C!\n";
4     write(STDOUT_FILENO, message, sizeof(message));
5 }
6
7 int main() {
8     // Инициализируем sigaction
9     struct sigaction act = {};
10    act.sa_handler = handler;
11    sigemptyset(&act.sa_mask);
12    act.sa_flags = 0;
13
14    // Устанавливаем обработчик на SIGINT
15    sigaction(SIGINT, &act, NULL);
16 }
```



Не стреляйте сигнальной ракетницей по ногам.

- Нужно помнить, что обработчики сигналов могут быть вызваны в любой момент, даже во время исполнения библиотечного кода.
- Многие стандартные функции небезопасны при обработке сигналов. Например:

`malloc()`

`free()`

`printf()`

`fopen()`

`fread()`

`rand()`

`strerror()`

`perror()`

- Список безопасных функций есть в `$ man 7 signal-safety`
- Для глобальных переменных, которые используются обработчиками, нужно использовать атомарные типы. Например, `sig_atomic_t`.
- Если участок кода не хочется прерывать, можно временно заблокировать сигналы. Будет аналогично работе мьютекса.

Блокировка сигналов

Если заблокировать сигнал, процесс не заметит его получения до тех пор, пока не разблокирует его.

- Сигналы блокируются системным вызовом `sigprocmask(...)`
- Заблокировать **SIGSTOP** и **SIGKILL** нельзя.

Зачем?

Обработка сигналов прерывает исполнение. Можно временно заблокировать сигналы, чтобы обезопасить выполнение какой-то опасной секции кода.



```
int sigprocmask(int how, sigset_t* set, sigset_t* old_set)
```

Обновляет маску заблокированных сигналов. Старую маску сохраняет в `old_set`.

- `old_set` может быть `NULL`, если он не нужен.

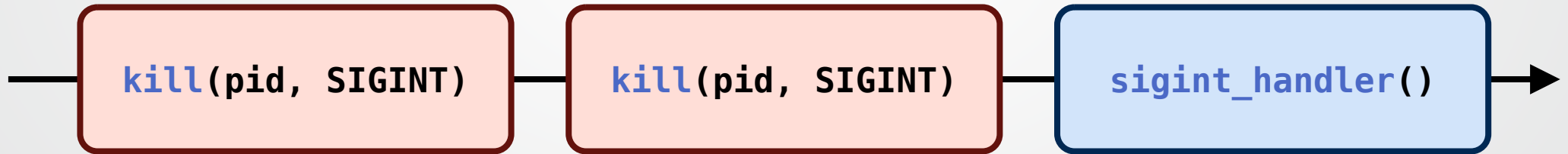
`int how` :

= <code>SIG_SETMASK</code>	Установить маску;
= <code>SIG_BLOCK</code>	Заблокировать из маски;
= <code>SIG_UNBLOCK</code>	Разблокировать из маски.



Как доставляются сигналы

- Если процесс должен обработать сигнал, это фиксируется в битовой маске;
- Маска может хранить факт получения сигнала, но не их количество;
- Обработчик сигнала может вызваться с задержкой.
- Если процесс получит два одинаковых сигнала быстрее, чем успеет их обработать, обработчик будет вызван один раз.



- Даже если процесс получил разные сигналы, порядок их обработки не определён.
- А еще эта маска не наследуется при **fork**. Почему?

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

Отправлять сигналы в файловый дескриптор.

`int fd`

Какой дескриптор использовать. `-1`, чтобы создать новый.

`const sigset_t *mask`

Какие сигналы отслеживать

`int flags`

Флаги: `SFD_NONBLOCK` и `SFD_CLOEXEC`

- Чтение файлового дескриптора `signalfd` по умолчанию заблокируется до сигнала.
- Когда появятся сигналы для обработки, вы сможете читать структуры типа `signalfd_siginfo` из этого дескриптора. В поле `ssi_signo` – номер сигнала.
- `signalfd` доставит вам даже заблокированные сигналы. Их рекомендуется заблокировать, чтобы они не выполняли свои действия по умолчанию.
- Использовать файловый дескриптор для обработки сигналов безопаснее, поскольку это не прерывает исполнение вашего кода.

Читаем сигналы из `signalfd`

```
1  int main() {
2      sigset_t mask = {};
3      sigemptyset(&mask);
4      sigaddset(&mask, SIGINT);
5
6      sigprocmask(SIG_BLOCK, &mask, NULL);
7
8      int sfd = signalfd(-1, &mask, 0);
9
10     while (true) {
11         struct signalfd_siginfo fdsi;
12         ssize_t s = read(sfd, &fdsi, sizeof(fdsi));
13         if (s != sizeof(fdsi)) return -1;
14
15         if (fdsi.ssi_signo == SIGINT) printf("Got SIGINT\n");
16     }
17 }
```

Сигналы реального времени

Обычные сигналы имеют ряд проблем:

- Произвольный порядок доставки;
- Одинаковые сигналы могут склеиться в один;
- Позволяют передать только номер сигнала.

Сигналы реального времени решают их:

- + Порядок доставки равен порядку отправки;
- + Поддерживают очередь сигналов одного типа;
- + Позволяют передать аргумент вместе с сигналом.

Для собственных сигналов можно использовать номера от `SIGRTMIN` до `SIGRTMAX`

```
sigqueue(pid_t pid, int sig, sigval_t value)
```

Отправляет сигнал реального времени. Замена `kill(pid_t pid, int sig)`

- `sigval_t value` - аргумент, который будет отпраавлен вместе с сигналом.
 - Аргумент – **union** с двумя полями: `int sival_int` и `void *sival_ptr`
-

Приём сигнала реального времени:

- Нужно использовать поле `sa_sigaction` вместо `sa_handler` для указателя на обработчик. Такой обработчик должен принимать три аргумента:
- `void (*sa_sigaction)(int signum, siginfo_t* info, void* ucontext)`
- Нужно указать флаг `SA_SIGINFO` при установке обработчика.

Работа с сигналами реального времени

```
1  void handler(int signum, siginfo_t* info, void* ucontext) {
2      sigval_t value = info->si_value;
3
4      // Используем как хотим
5      value.sival_int;
6  }
7
8  int main() {
9      // Инициализируем sigaction
10     struct sigaction act = {};
11     act.sa_sigaction = handler;
12     sigemptyset(&act.sa_mask);
13     act.sa_flags = SA_SIGINFO;
14
15     // Регистрируем обработчик
16     sigaction(SIGRTMIN+1, &act, NULL);
17 }
```


Спасибо за внимание!

 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024