

Семинар 18: POSIX signals

25 апреля, 2020

Сигналы

- ▶ Аналог процессорных прерываний, но для программ
- ▶ Используются для уведомления процессов о каких-то событиях
- ▶ Сигналы можно обрабатывать или игнорировать
- ▶ Однако, SIGKILL и SIGSTOP нельзя обработать или проигнорировать

Сигналы: примеры

- ▶ Нажатие `^C` в терминале генерирует `SIGINT`
- ▶ Нажатие `^\` в терминале генерирует `SIGQUIT`
- ▶ Запись в пайп только с `write`-концом генерирует `SIGPIPE`
- ▶ Вызов `abort()` приводит к `SIGABRT`
- ▶ Обращение к несуществующей памяти генерирует `SIGSEGV`

Доставка сигналов

- ▶ Сигналы могут быть доставлены от ядра (например, SIGKILL, SIGPIPE)
- ▶ Либо от другого процесса (например, SIGHUP, SIGINT, SIGUSR)
- ▶ Или процесс может послать сигнал сам себе (SIGABRT)
- ▶ Сигналы могут быть доставлены *в любой момент* выполнения программы
- ▶ Они прерывают выполнения программы на время своего выполнения
- ▶ У каждого сигнала есть действие по-умолчанию: некоторые просто игнорируются, а некоторые проводят к завершению процесса

Signal safety

- ▶ Во время обработки сигнала процессы могут быть в критической секции
- ▶ Поэтому в обработчиках сигналов нельзя использовать, например, `printf`
- ▶ Можно использовать только `async-signal-safe` функции
- ▶ `man 7 signal-safety`

Обработка сигналов

```
void signal_handler(int sig) {  
    // ...  
}  
  
int main() {  
    signal(SIGINT, signal_handler);  
    signal(SIGTERM, signal_handler);  
    signal(SIGSEGV, SIG_IGN);  
    signal(SIGABRT, SIG_DFL);  
}
```

Посылка сигналов

```
#include <signal.h>
```

```
int raise(int sig);
```

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

Посылка сигналов: аргументы `kill`

- ▶ Если `pid == 0`, то сигнал будет доставлен текущей группе процессов
- ▶ Если `pid > 0`, то сигнал будет доставлен процессу `pid`
- ▶ Если `pid == -1`, то сигнал будет всем процессам, которым текущий процесс может его отправить
- ▶ Если `pid < -1`, то сигнал будет доставлен группе процессов `-pid`
- ▶ Если `sig == 0`, то сигнал не будет никому отправлен, а будет только осуществлена проверка ошибок (dry run)
- ▶ Возврат из `kill` не гарантирует, что сигнал обработался в получателях!

Доставка сигналов: маски сигналов

- ▶ Маска сигналов – bitset всех сигналов
- ▶ У процесса есть две маски сигналов: pending и blocked

Доставка сигналов: маски сигналов

- ▶ Маска сигналов – bitset всех сигналов
- ▶ У процесса есть две маски сигналов: pending и blocked
- ▶ pending – это те сигналы, которые должны быть доставлены, но ещё не успели
- ▶ Из этого следует, что если несколько раз отправить сигнал в один процесс, то он может быть обработан лишь единожды

Доставка сигналов: маски сигналов

- ▶ Маска сигналов – bitset всех сигналов
- ▶ У процесса есть две маски сигналов: pending и blocked
- ▶ pending – это те сигналы, которые должны быть доставлены, но ещё не успели
- ▶ Из этого следует, что если несколько раз отправить сигнал в один процесс, то он может быть обработан лишь единожды
- ▶ blocked – это те сигналы, которые процесс блокирует (блокировать и игнорировать – разные вещи)
- ▶ Если сигнал заблокирован, это значит, что он не будет доставлен *вообще*, если проигнорирован – то у него просто пустой обработчик

Доставка сигналов: маски сигналов

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int signum);  
int sigdelset(sigset_t *set, int signum);  
int sigismember(const sigset_t *set, int signum);
```

Доставка сигналов: sigprocmask

- ▶ `int sigprocmask(int h, sigset_t *set, sigset_t *oset);`
- ▶ SIG_SETMASK – установить маску заблокированных сигналов
- ▶ SIG_BLOCK – добавить сигналы set в заблокированные
- ▶ SIG_UNBLOCK – удалить сигналы set из заблокированных
- ▶ Если oset != NULL, то туда будет записана предыдущая маска

Доставка сигналов: маски сигналов

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);

struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int       sa_flags;
};
```

Обработка сигналов: `sigaction`

- ▶ Выставляет обычный обработчик `sa_handler`
- ▶ Или «расширенный»: `sa_sigaction`
- ▶ При выполнении сигнала `signum` в заблокированные сигналы добавятся сигналы из `sa_mask`, а также сам сигнал
- ▶ `sa_flags` – флаги, меняющие поведение обработки
- ▶ Чтобы использовать `sa_sigaction`, нужно выставить `SA_SIGINFO`
- ▶ Если выставить `SA_RESETHAND`, то обработчик сигнала будет сброшен на дефолтный после выполнения
- ▶ Если выставить `SA_NODEFER`, то если сигнал не был в `sa_mask`, обработчик может быть прерван самим собой

Обработка сигналов: siginfo_t

```
#include <signal.h>
```

```
siginfo_t {  
    int      si_signo;      int      si_overrun;  
    int      si_errno;     int      si_timerid;  
    int      si_code;      void      *si_addr;  
    int      si_trapno;    long      si_band;  
    pid_t    si_pid;       int      si_fd;  
    uid_t    si_uid;       short     si_addr_lsb;  
    int      si_status;    void      *si_lower;  
    clock_t  si_utime;     void      *si_upper;  
    clock_t  si_stime;     int      si_pkey;  
    sigval_t si_value;     void      *si_call_addr;  
    int      si_int;       int      si_syscall;  
    void     *si_ptr;      unsigned int si_arch;  
}
```


Доставка сигналов во время системных вызовов

- ▶ Если сигнал прервал выполнение блокирующего сисколл, то есть два поведения
- ▶ Если использован `sigaction` и в `sa_flags` есть `SA_RESTART`, то после того, как обработчик завершится, сисколл продолжит свою работу (syscall restarting)
- ▶ Если не указан, то сисколл вернёт ошибку и `errno == EINTR`

Ожидание сигналов: pause и sigsuspend

- ▶ `int pause(void)`
- ▶ Блокируется до первой доставки сигналов (которые не заблокированы)

Ожидание сигналов: `pause` и `sigsuspend`

- ▶ `int pause(void)`
- ▶ Блокируется до первой доставки сигналов (которые не заблокированы)
- ▶ `int sigsuspend(const sigset_t *mask)`
- ▶ Атомарно заменяет маску заблокированных сигналов на `mask` и ждёт первой доставки сигналов

Сигналы: fork и execve

- ▶ fork сохраняет blocked mask и назначенные обработчики
- ▶ execve сохраняет **только** маску заблокированных сигналов

Почему сигналы – это плохо?

- ▶ Почти невозможно обработать сигналы без race condition'ов
- ▶ Обработчики сигналов могут вызываться во время работы других обработчиков
- ▶ Посылка нескольких сигналов может привести к посылке только одного
- ▶ Не используйте сигналы для IPC!
- ▶ Допустимые использования – graceful program termination или live configuration reload

Почему лучше всегда использовать sigaction?

- ▶ Война поведений: BSD vs System-V
- ▶ Стандарты определяются через макросы препроцессора: `_BSD_SOURCE` или `_GNU_SOURCE` (устаревший метод)
- ▶ Либо через опции компилятора: `-std=gnu99` или `-std=gnu11`
- ▶ В отличие от BSD, в System-V обработчик сигнала выполняется единожды, после чего сбрасывается на обработчик по умолчанию
- ▶ В BSD обработчик сигнала не будет вызван, если в это время уже выполняется обработчик того же самого сигнала
- ▶ В BSD используется `syscall restarting`, в System-V – нет

Почему лучше всегда использовать sigaction?

- ▶ Война поведений: BSD vs System-V
- ▶ Стандарты определяются через макросы препроцессора: `_BSD_SOURCE` или `_GNU_SOURCE` (устаревший метод)
- ▶ Либо через опции компилятора: `-std=gnu99` или `-std=gnu11`
- ▶ В отличие от BSD, в System-V обработчик сигнала выполняется единожды, после чего сбрасывается на обработчик по умолчанию
- ▶ В BSD обработчик сигнала не будет вызван, если в это время уже выполняется обработчик того же самого сигнала
- ▶ В BSD используется `syscall restarting`, в System-V – нет
- ▶ BSD: `SA_RESTART`
- ▶ System-V: `SA_NODEFER | SA_RESETHAND`

Почему лучше всегда использовать sigaction?

- ▶ Война поведений: BSD vs System-V
- ▶ Стандарты определяются через макросы препроцессора: `_BSD_SOURCE` или `_GNU_SOURCE` (устаревший метод)
- ▶ Либо через опции компилятора: `-std=gnu99` или `-std=gnu11`
- ▶ В отличие от BSD, в System-V обработчик сигнала выполняется единожды, после чего сбрасывается на обработчик по умолчанию
- ▶ В BSD обработчик сигнала не будет вызван, если в это время уже выполняется обработчик того же самого сигнала
- ▶ В BSD используется `syscall restarting`, в System-V – нет
- ▶ BSD: `SA_RESTART`
- ▶ System-V: `SA_NODEFER | SA_RESETHAND`
- ▶ Всегда лучше использовать `sigaction` для однозначности поведения программы!

Бонус: RT signals

- ▶ При посылке сигналов учитывается их количество и порядок
- ▶ Отделены от обычных: начинаются с SIGRTMIN, заканчиваются SIGRTMAX
- ▶ Вместе с сигналом посылается специальная метайнформация (правда, только число), которую можно как-то использовать
- ▶ Получить эту дополнительную информацию можно через `siginfo_t->si_value`

Бонус: RT signals

```
#include <signal.h>
```

```
union sigval {  
    int    sival_int;  
    void*  sival_ptr;  
};
```

```
int sigqueue(pid_t pid, int signum, const union sigval value);
```

Бонус: обработка сигналов с помощью пайпов

- ▶ Иногда не хочется возиться с атомарными счётчиками или хочется выполнить какие-то нетривиальные действия в обработчике
- ▶ Или в обработчике нет какого-то нужного контекста (экземпляра класса итд)
- ▶ Помогает трюк с пайпами
- ▶ В обработчике будем писать номер сигнала (или какую-то другую информацию) в пайп
- ▶ В основной программе будем делать read на другой конец пайпа

Бонус: обработка сигналов с помощью пайпов

- ▶ Иногда не хочется возиться с атомарными счётчиками или хочется выполнить какие-то нетривиальные действия в обработчике
- ▶ Или в обработчике нет какого-то нужного контекста (экземпляра класса итд)
- ▶ Помогает трюк с пайпами
- ▶ В обработчике будем писать номер сигнала (или какую-то другую информацию) в пайп
- ▶ В основной программе будем делать read на другой конец пайпа
- ▶ Важно, чтобы write-конец пайпа был с флагом `O_NONBLOCKING`!
- ▶ Иначе это может привести к дедлоку

Спасибо!