Синхронизация и межпроцессное взаимодействие, ч. 2

Сигналы

- Асинхронное событие, которое может произойти после *любой инструкции*
- Используются для уведомления процессов о каких-то событиях
- Сигнал можно либо обрабатывать, либо игнорировать
- Однако `SIGKILL` и `SIGSTOP` нельзя обработать или проигнорировать
- `SIGKILL` сразу убивает процесс (если он не находится в состоянии `D`)
- `SIGSTOP` переводит процесс в состояние остановки – `T`
- `SIGCONT` – обратный ему

Сигналы: примеры

- Нажатие `^C` (Ctrl+C) в терминале генерирует `SIGINT`
- Нажатие `^\` (Ctrl+Backslash), в терминале генерирует `SIGQUIT`
- Вызов `abort()` приводит к `SIGABRT` и откладыванию coredump
- Обращение к несуществующей памяти генерирует `SIGSEGV`

Сигналы

- У каждого сигнала есть своё действие по-умолчанию
- `Ign`, `Term`, `Core`, `Stop`, `Cont`
- Действия по-умолчанию для каждого сигнала можно увидеть в `man 7 signal`

Coredump

- Когда программа аварийно завершается, состояние памяти может быть испорчено
- Coredump'ы используются, чтобы увидеть какое было состояние памяти на момент завершения программы
- Coredump представляет из себя ELF файл, где полностью записан образ процесса (его память)
- Coredump'ы откладывает ядро (`/proc/sys/kernel/core_pattern`)

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

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 или специальный тип `sig_atomic_t`
- `sig_atomic_t` атомарен относительно прерывания сигналами, но не является атомарном в смысле работы с памятью
- `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

- `pid` имеет такое же значение, как и в `wait*`
- Если `sig == 0`, то сигнал не будет никому отправлен, а будет только осуществлена проверка прав доступа
- Возврат из kill не гарантирует, что сигнал обработался в получателях, он лишь гарантирует доставку

```c
#include <signal.h>

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

Маски сигналов

- Маска сигналов — bitset всех сигналов
- У процесса есть две маски сигналов: *pending* и *blocked*
- pending — сигналы, которые должны быть доставлены, но ещё не обработаны
- blocked — это те сигналы, которые процесс блокирует
- Если сигнал заблокирован, это значит, что он не будет доставлен *вообще*, если проигнорирован — то у него просто пустой обработчик

```c
#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);
```

Маски сигналов

- `SIG_SETMASK` — установить маску заблокированных сигналов
- `SIG_BLOCK` — добавить сигналы из `set` в заблокированные
- `SIG_UNBLOCK` — удалить сигналы из `set` из заблокированных
- Если `oset != NULL`, то туда будет записана предыдущая маска

```
int sigprocmask(int h, sigset_t *set, sigset_t *oset)
```

Обработка сигналов:

- 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

```c
#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`

- 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

```c
#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: `siginfo_t`

```
#include <signal.h>

struct 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;
};
```

SIGCHLD

- Ещё один способ уведомлять процессы о завершении дочерних
- `si_status` хранит информацию о том, как завершился процесс (`CLD_KILLED` , `CLD_STOPPED` , etc)
- `si_pid` хранит PID дочернего процесс
- Если выставить `SIG_IGN` , то зомби не будут появляться (по-умолчанию сигнал не доставляется)

Доставка сигналов во время системных вызовов

- Syscall restart: если использован `sigaction` и в `sa_flags` есть `SA_RESTART`, то после того, как обработчик завершится, сисколл продолжит свою работу
- Если не указан, то сисколл вернёт ошибку `EINTR`
- Если `read` / `write` что-то уже записал, то `EINTR` не будет возвращён

Ожидание сигналов

- `pause` блокируется до первой доставки сигналов (которые не заблокированы)
- `sigsuspend` атомарно заменяет маску заблокированных сигналов на `mask` и ждёт первой доставки сигналов
- `sigwaitinfo` ждёт один из указанных сигналов

```
int pause(void);
int sigsuspend(const sigset_t* mask);
int sigwaitinfo(const sigset_t* set, siginfo_t* info);
```

Ожидание сигналов

- Linux-only
- `mask` – принимаемые сигналы (надо их заблокировать перед этим)
- Возвращает файловый дескриптор, на котором можно делать `read`

```c
#include <sys/signalfd.h>

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

Как устроены сигналы изнутри?

- Ядро проверяет, нужно ли доставить сигнал в текущий процесс
- Конструируется специальный стек
- В начало стека кладётся фрейм, который содержит информацию о прерванной инструкции
- Ядро «прыгает» в обработчик события
- Обработчик завершается и вызывает `sigreturn`
- Процесс возвращается в предыдущий контекст

```
typedef struct {
    void *ss_sp;
    int ss_flags;
    size_t ss_size;
} stack_t;

int sigaltstack(const stack_t* ss, stack_t* old_ss);
int sigreturn(...);
```

Наследование сигналов

- `fork` сохраняет маску сигналов и назначенные обработчики
- `execve` сохраняет **только** маску 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
- Старайтесь не использовать сигналы для IPC!

Почему сигналы — это плохо?

Если в процессе несколько тредов, какой из них получит сигнал?

```
...
A process-directed signal may be delivered to any one of the
threads that does not currently have the signal blocked.  If more
than one of the threads has the signal unblocked, then the kernel
chooses an arbitrary thread to which to deliver the signal.
...
```

Почему лучше всегда использовать sigaction?

- Война поведений: BSD vs System-V
- В отличие от BSD, в System-V обработчик сигнала выполяется 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 (`_BSD_SOURCE` или `-std=c99`): `SA_RESTART`
- System-V (`_GNU_SOURCE` или `-std=gnu99`): `SA_NODEFER|SA_RESETHAND`
- Всегда лучше использовать `sigaction` для однозначности поведения программы!

Real-time signals

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

```c
#include <signal.h>

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

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

Использование сигналов: nginx

- Если поменялся конфиг nginx, то как его подхватить заново?
- Постоянный траффик от клиентов, $\Rightarrow$ перезапуск не возможен
- Сигналы приходят на помощь!
- `SIGHUP` сигнализирует nginx о том, что конфиг поменялся и его нужно перечитать и применить
- `SIGTERM` сигнализирует nginx о том, что нужно остановить обработку запросов как можно скорее и завершиться
- `SIGUSR1` используется для hot upgrade

Использование сигналов: Go

- Реализация preemptive multitasking
- Внутри Go реализованы лёгкие потоки — горутины (cooperative multitasking)
- Иногда горутины могут зависать (если, например, много вычислений) и их нужно уметь принудительно вытеснять
- Отдельный тред `sysmon`, который следит за остальными тредами, исполняющими горутины
- Если какая-то горутина зависат больше, чем на 10 мс, посылается `SIGURG` и её выполнение прерывается

Пайпы

- *Пайпы* или *каналы* позволяют передавать данные в одном направлении в пределах ОС
- Пайп представляет буфер из которого можно читать и писать
- Два файловых дескриптора – один на чтение (`pipefd[0]`), другой на запись (`pipefd[1]`)

```c
#include <unistd.h>

int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

Пайпы: чтения

- `read` читает данные из пайпа или зависает, если данных нет
- Чтения блока данных, меньшие `PIPE_BUF` байт, обрабатываются атомарно
- Если парный файловый дескриптор закрыт (пишущая сторона), то `read` вернёт `0`

Пайпы: записи

- `write` записывает данные в пайп или зависает, если в пайпе недостаточно места
- Если парный файловый дескриптор закрыт (читающая сторона), то присылается сигнал `SIGPIPE` или возвращается ошибка `EPIPE`
- `write` может записать не все данные, которые ему передали

Неблокирующий режим

- Если открыть пайп (`pipe2`) с флагом `O_NONBLOCK` , то вместо зависаний `read` / `write` будут возвращать ошибку `EAGAIN`
- Также флаг можно установить на конкретный файловый дескриптор с помощью `fcntl`

```c
#include <fcntl.h>

int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Файловые дескрипторы: дублирование

- Семество функций `dup*` дублирует файловые дескрипторы
- В отличие от *копирования*, дублирование не копирует сами объекты под файловыми дескрипторами, поэтому изменения положения или флагов будут видны между всеми дублями

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);

// Псевдокод:
int dup2(int oldfd, int newfd) {
    // ...
    current_process->fdtable[newfd] = oldfd
    // ...
}
```

Именованные пайпы

- Такие пайпы существуют не только в виде файловых дескрипторов, но и в виде имён файлов
- Специальный тип файлов
- `open` на именовоном канале может заблокиролваться до того момента, пока не появится пишущая сторона (и наоборот)
- В Linux `open` всегда завершается успехов в блокирующем режиме
- `write` в неблокирующем режиме возвращает `ENXIO`

```c
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char* pathname, mode_t mode);
```

Пайпы: применение

- Самое важное применение пайпов – перехват ввода-вывода дочерних процессов
- Command chaining (`cat ... | grep -v | grep -v | less`) реализован с помощью пайпов
- Очень важно закрыть ненужные концы пайпов, иначе могут появиться дедлоки!

```
int p[2];
pipe(p);
pid_t pid = fork();
if (pid == 0) {
    dup2(p[1], 1);
    close(p[0]);
    close(p[1]);
    execve(...);
} else {
    close(p[1]);
    // ...
}
```

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 (экземляра 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***: write-конец пайпа должен быть с флагом `O_NONBLOCKING`, иначе возможен дедлок

धन्यवाद!

«Спасибо» на хинди