

# Взаимодействие процессов: сигналы

АКОС #12 2019-2020

# Завершение работы процесса

- Добровольное:
  - выход из функции **main**
  - вызов функции **exit**
  - системный вызов **\_exit**
- Подходит для детерминированного выполнения: у программы есть начало и конец

# Завершение работы процесса

- Принудительное - отправкой сигнала:
  - команда **kill**
  - команда **killall**
  - запуск через **timeout**
  - кнопки Ctrl+C
  - закрытие вкладки терминала
  - выключение или перезагрузка
- Это всё - штатные ситуации, которые нужно учитывать
- Событие может произойти **асинхронно**

# Завершение работы процесса

- Ошибка, за которую можно поплатиться:
  - нарушение сегментации
  - запись в закрытый канал или сокет
  - деление на ноль
  - недопустимая инструкция
  - нарушение assertion
- Действие по умолчанию - кто-то прибавляет процесс отправкой **сигнала**

# Сигнал

- Асинхронное событие, которое может отправить процессу:
  - ядро
  - другой процесс, если у него есть право на это
- Что может сделать процесс:
  - игнорировать
  - самовыпилиться (иногда с Core Dump для gdb)
  - изменить состояние: **sTopped** | **R**unning
  - сделать что-то ещё

# Сигналы

Номер	Имя	По умолчанию	Описание
1	SIGHUP	Term	обрыв соединения
2	SIGINT	Term	Ctrl+C
3	SIGQUIT	Core	Ctrl+\
4	SIGILL	Core	плохая инструкция
6	SIGABRT	Core	abort()

**man 7 signal**

*Демонстрации: 1) посылка сигналов; 2) do\_abort.c*

# Core Dump

- Снимок памяти процесса
- Полезен для отладки
- В современных системах управляется systemd

```
/usr/sbin/sysctl kernel.core_pattern  
ulimit -c
```

Номер	Имя	По умолчанию	Описание
9	SIGKILL	Term	убийство
11	SIGSEGV	Core	что-то плохо с памятью
13	SIGPIPE	Term	Broken pipe
15	SIGTERM	Term	завершение работы
17	SIGCHLD	Ign	завершился дочерний проц.
18	SIGCONT	Cont	Команда fg
19	SIGSTOP	Stop	Ctrl+Z
23	SIGURG	Ign	Socket urgent data

*Демонстрация: Ctrl+Z*



# Обработка сигналов [deprecated]

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

```
// Тип sighandler_t – только в Linux!  
typedef void (*sighandler_t)(int);
```

```
sighandler_t  
signal(int signum, sighandler_t handler);
```

# Немного о стандартах

- Сигналы System-V (Solaris)
- Сигналы BSD (в том числе Linux)

gcc -std=c99    v.s.    gcc -std=gnu99

~~#define \_BSD\_SOURCE~~

#define \_DEFAULT\_SOURCE

#define \_GNU\_SOURCE

# Обработка сигналов [deprecated]

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

```
// Тип sighandler_t – только в Linux  
typedef void (*sighandler_t)(int);
```

```
// Тип sig_t – только в *BSD  
typedef void (*sig_t)(int);
```

```
sighandler_t  
signal(int signum, sighandler_t handler);
```

# Сигналы BSD v.s. System-V

- В System-V обрабатывается один раз, в BSD - до отмены обработчика
- В System-V во время обработки сигнала может быть вызван обработчик другого сигнала, в BSD - блокируется до завершения обработки
- В System-V блокирующие системные вызовы завершают свою работу (EINTR в errno), в BSD - автоматически перезапускаются (кроме sleep, pause, select)

# Обработка сигналов [modern-way]

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

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

```
int  
sigaction(int signum,  
          const struct sigaction *act,  
          struct sigaction *oldact /* might be NULL */);
```

# Системный вызов `signal`

- Можно использовать только с обработчиками `SIG_IGN` (`=1`) и `SIG_DFL` (`=0`)
- В Linux системный вызов ведёт себя в стиле System-V
- В стандартной библиотеке Си, если объявлен макрос `_BSD_SOURCE` или `_DEFAULT_SOURCE`, то `signal` - это функция-оболочка поверх `sigaction`, а не `signal`

# Обработчик сигнала

- Может быть вызван в **произвольный момент времени**
  - использует текущий стек (хотя это можно настроить)
  - для x86\_64 гарантируется RedZone размером 128 байт
  - может использовать ограниченный набор функций: только асинхронно-безопасные (AS-Safe)

# Async Signal Safety

- Классы функций:
  - Небезопасные (Unsafe)
  - Поточно-безопасные (MT-Safe)
  - Асинхронно-безопасные (AS-Safe)
- Множества функций AS-Safe и MT-Safe не совпадают! Пример: **fwrite**
- Полный список функций:  
`man 7 signal-safety`



# Обработчик сигнала

- Тип данных **sig\_atomic\_t**
  - целочисленный
  - **int** для большинства платформ
  - гарантируется «атомарность» чтения/записи, но только на уровне обработки сигнала
- Ключевое слово **volatile**
  - указывает компилятору, что ни в коем случае нельзя оптимизировать использование переменной

# Обработчик сигнала

- Может использовать только асинхронно-безопасные функции
- Должен выполняться как можно быстрее
- Правильная стратегия: проставить флаг о том, что поступил сигнал, а затем его проверить, не нарушая логики работы программы

# Примеры обработчиков сигналов

- Для SIGTERM и SIGINT - корректно завершить свою работу
- Для SIGINT возможно запросить «Вы уверены?»
- Для SIGHUP - перечитать файл с настройками
- Для SIGCHLD - прочитать код возврата дочернего процесса
- Для SIGSEGV - попытаться (без каких-либо гарантий) сохранить важные данные

# Маска сигналов, ожидающих доставки

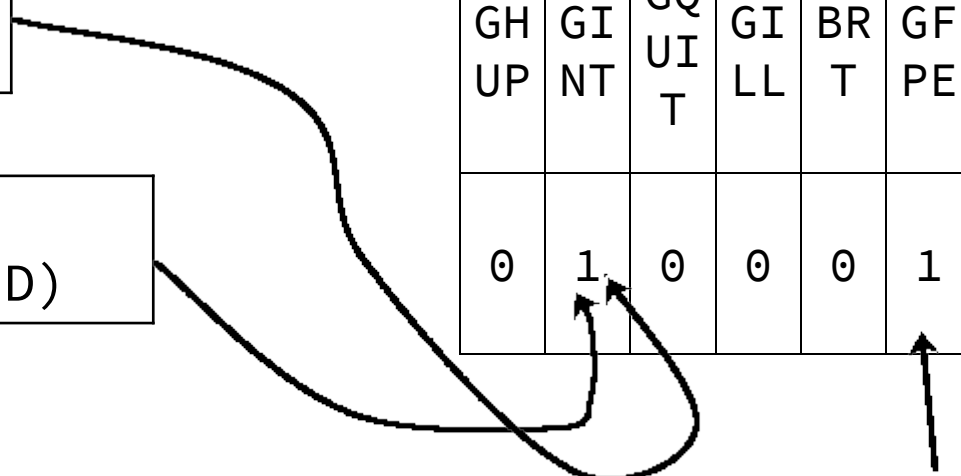
- Один из атрибутов процесса
- Не наследуется при клонировании системным вызовом `fork`

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

**Process 3**  
`kill(SIGFPE, PID)`

SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
0	1	0	0	0	1	0



# Доставка сигнала

- Произвольный процесс или ядро устанавливают нужный флаг в маске другого процесса
- Выполнение продолжается до тех пор, пока не планировщик не выберет другой процесс
- Когда планировщик заданий добирается до того процесса, который получил сигнал, то первым делом проверяется маска сигналов, ожидающих доставки

# Маска сигналов, ожидающих доставки

**Учитывается только факт наличия сигнала,  
но не их количество!**

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
0	1	0	0	0	1	0

**Process 3**  
`kill(SIGFPE, PID)`

# Маски сигналов

(у каждого thread'a - своя)

Какие сигналы будут обработаны →

Маска [заблокированных] сигналов →

**Process 1**  
`kill(SIGINT, PID)`

**Process 2**  
`kill(SIGINT, PID)`

0	0	0	0	0	1	0
1	0	1	1	1	1	1
SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
0	1	0	0	0	1	0

**Маски сигналов,**  
в отличии от маски ожидающих доставки,  
**наследуются при fork**

**Process 3**  
`kill(SIGFPE, PID)`

# Изменение маски сигналов

- Установка маски для процесса  
`sigprocmask(int how,  
              const sigset_t *mask,  
              sigset_t *old_mask)`
- Установки маски для отдельной нити  
(если используется многопоточность)  
`pthread_sigmask(int how,  
                 const sigset_t *mask,  
                 sigset_t *old_mask)`



# Множества сигналов

- Стандартом не регламентируется содержимое **sigset\_t**
- Для инициализации и модификации используются функции (man sigsetops)
  - **sigemptyset**
  - **sigfillset**
  - **sigaddset**
  - **sigdelset**
  - **sigismember**

# Маски сигналов

(у каждого thread'a - своя)

Какие сигналы будут обработаны →

Маска [разрешенных] сигналов →

0	0	0	0	0	1	0
1	0	1	1	1	1	1
SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
0	1	0	0	0	1	0

**Process 1**

`kill(SIGINT, PID)`

**Process 2**

`kill(SIGINT, PID)`

**Process 3**

`kill(SIGFPE, PID)`

*Заблокированные сигналы не исчезают, а "откладываются" до тех пор, пока не будут разблокированы*

# Ожидание поступления сигнала

- **pause** ( ) - приостановить выполнение до прихода и обработки **любого незаблокированного** сигнала
- **sigsuspend** ( **sigset\_t** \*set ) - приостановить выполнение до прихода **инвертированного множества** сигналов, **игнорируя все остальные**

# Файловый дескриптор signalfd

*Только Linux*

```
int signalfd(int old_signal_fd_or_minus_1,  
              const sigset_t *mask,  
              int flags)
```

- создает файловый дескриптор для "чтения" событий о поступающих сигналах из множества mask
- можно читать объекты типа **struct signalfd\_siginfo**

# Обычные сигналы

- Имеют стандартное назначение для большинства UNIX-подобных систем
- Процесс может проверить факт того, что во время его паузы ему были доставлены сигналы, но не их количество
- Обработать поступившие сигналы процесс имеет право в произвольном порядке

# Сигналы реального времени

POSIX.1b real-time extensions; реализованы в Linux

- Имеют номера от `SIGRTMIN` до `SIGRTMAX`
- Могут быть использованы как обычные сигналы (доставка через `kill`)
- Могут быть доставлены через очередь доставки (в этом случае гарантируется порядок доставки и количество)
- Могут нести дополнительную информацию - целое число в поле `si_value` структуры `siginfo_t`

# Отправка сигналов реального времени

## POSIX:

```
sigqueue(int pid,  
         int signum,  
         const union signal value)
```

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

## LINUX:

```
sys_rt_sigqueueinfo(  
    int pid,  
    int signum,  
    const siginfo_t  
        *uinfo  
)
```

