Многопоточное программирование на C++

Сигналы

В этой лекции

Различные сигналы и их назначения;

Когда ядро может послать сигнал процессу и как одному процессу послать сигналы другому процессу.

Как процессы реагируют на сигналы по-умолчанию, как процессу изменить поведение в ответ на сигнал.

Использование маски для блокировки сигналов, ожидающие сигналы.

Как процесс может приостановить выполнение и ждать прихода сигнала.

Сигнал

Сигнал - оповещение процессу, что произошло некоторое событие.

Сигналы схожи с прерываниями, в том смысле что они прерывают нормальное исполнение программы. Почти во всех случаях нельзя предсказать, когда точно придет сигнал.

Источники сигналов

Хардварное исключение - означает, что железо обнаружило ошибку, которую передала ядру, которое в свою очередь послало сигнал процессу. Это может быть неправильная машинная инструкция, деление на 0, обращение к недоступной области памяти.

Юзер нажал специальную комбинацию клавиш, которая генерирует сигнал, например Ctrl+C (interrupt) или Ctrl-Z (suspend).

Произошло программное событие. Например, появились данные на файловом дескрипторе, окно терминала изменило размер, произошло событие таймера, израсходован лимит времени CPU, умер порожденный процесс.

Сигналы

Каждый сигнал - уникальное число (небольшое), начиная с 1.

Определены в <signal.h> именами вида SIGxxxx.

Т.к. на разных системах могут быть разные числа для сигналов, нужно пользоваться символическими именами.

Например сигнал interrupt (2) - SIGINT.

Сигналы

Две основные категории:

Стандартные - используемые ядров для уведомления процессов.

Реалтаймовые -

Сигналы

Сигналы ***генерируются*** каким-либо событием.

Затем, сигнал ***доставляется*** до процесса.

Процесс выполняет какие-то действия, реагирует на сигнал.

Между временем, когда сигнал был сгенерирован и когда она был доставлен, сигнал находится в ***ожидании*** (pending).

Сигналы

Обычно, ожидающий сигнал доставляется процессу сразу, как только он будет запланирован на выполнение шедулером или же моментально, если процесс уже выполняется (например послал сигнал сам себе).

Иногда требуется быть уверенным, что сегмент кода не будет прерван доставкой сигнала, тогда мы можем добавить сигнал в сигнальную маску процесса (signal mask) - набор сигналов, доставка которых сейчас заблокирована.

Если сигнал был сгенерирован пока его доставка заблокирована, он будет в режиме ожидания до момента разблокировки.

Дефолтные реакции на сигнал

Сигнал игнорируется.

Процесс уничтожается.

Генерируется core dump, затем процесс уничтожается.

Процесс останавливается - выполнение процесса приостанавливается.

Процесс продолжает выполнение после приостановки.

Можно менять реакцию на сигнал

Следует предпринять действие по-умолчанию.

Сигнал игнорируется.

Выполняется обработчик сигнала (signal handler).

Обработка сигналов

Невозможно изменить обаботку сигналов ***terminate*** и ***dump core***.

Сигналы

SIGABRT (6) - Завершение с дампом памяти. Сигнал посылаемый функцией abort()
SIGALRM (14) - Завершение. Сигнал истечения времени, заданного alarm()
SIGBUS (10) - Завершение с дампом памяти. Неправильное обращение в физическую память.
SIGCHLD (18) - Игнорируется. Дочерний процесс завершен или остановлен
SIGCONT (25) - Продолжить выполнение. Продолжить выполнение ранее остановленного процесса
SIGILL (4) - Завершение с дампом памяти. Недопустимая инструкция процессора
SIGINT (2) - Завершение. Сигнал прерывания (Ctrl-C) с терминала
SIGKILL (9) - Завершение
SIGSEGV (11) - Завершение с дампом памяти. Нарушение при обращении в память
SIGUSR1 (16) - Завершение Пользовательский сигнал № 1
SIGUSR2 (16) - Завершение Пользовательский сигнал № 2

Обработка сигналов

signal()

sigaction()

```
#include <signal.h>

void ( *signal(int sig, void (*handler)(int)) ) (int);
```

возвращает предыдущий обработчик сигнала при успешном вызове, либо SIG_ERR.

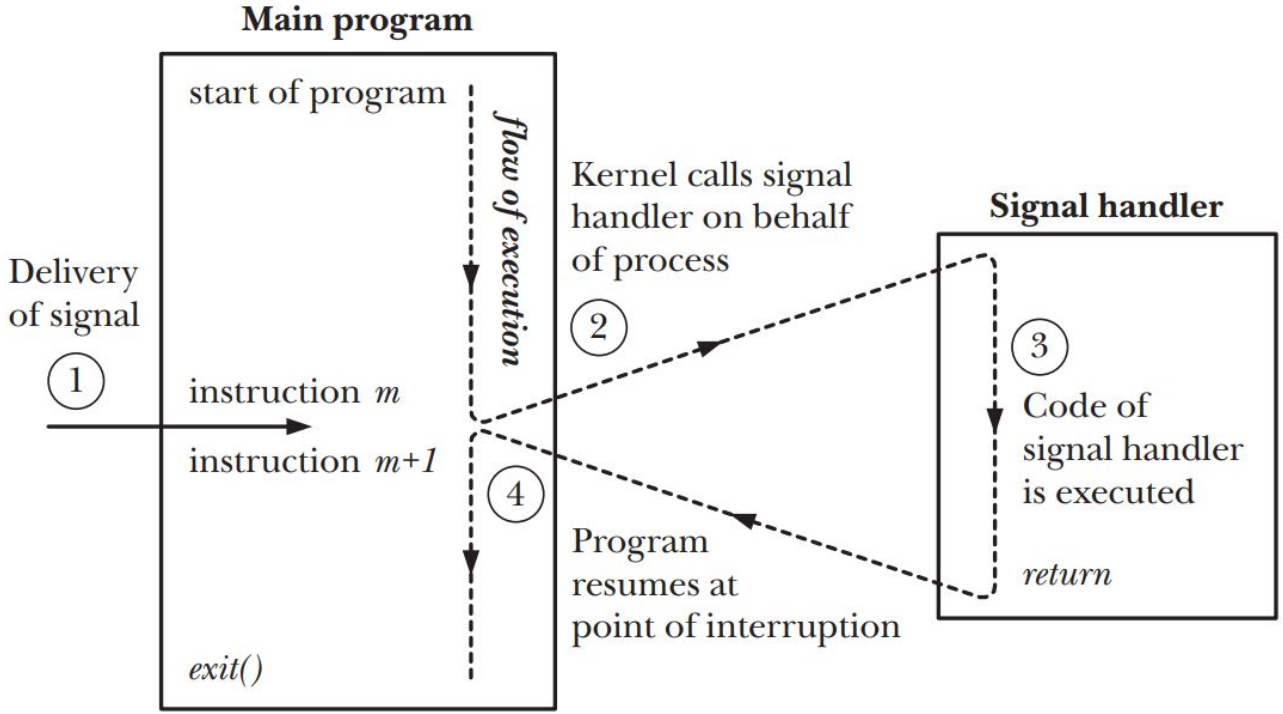
```
void handler(int sig)
{
/* Code for the handler */
}
```

```
void (*oldHandler)(int);

oldHandler = signal(SIGINT, newHandler);

if (oldHandler == SIG_ERR)
    errExit("signal");

/* Что-то делаем */

if (signal(SIGINT, oldHandler) == SIG_ERR)
    errExit("signal");
```

SIG_DFL - дефолтный обработчик

SIG_IGN - игнорировать сигнал

Обработка сигналов



```
static void
sigHandler(int sig)
{
    printf("Ouch!\n"); /* UNSAFE */
}
```

Посылка сигналов. kill

```
#include <signal.h>

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

Посылка сигналов. kill

pid > 0 - сигнал процессу с пидом pid;

pid == 0 - сигнал каждому процессу в той же группе процессов.

pid < -1 - сигнал посылается всем процессам из группы процессов процесса с пидом ранвым abs(pid)

pid == -1 - сигнал посылается всем процессам, для которых у посылающего процесса достаточно прав (кроме init и самого себя).

Проверка существования процесса

kill - послать 0 вместо sig.

raise(), killpg()

```
#include <signal.h>

int raise(int sig);

int killpg(pid_t pgrp, int sig);
```

Signal set

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int sig);

int sigdelset(sigset_t *set, int sig);

int sigismember(const sigset_t *set, int sig);
```

Signal Mask

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t
*oldset);

how - SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
```

Pending Signals

```
#include <signal.h>

int sigpending(sigset_t *set);
```

sigaction

```
#include <signal.h>

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

struct sigaction {
    void (*sa_handler)(int); /* Address of handler */
    sigset_t sa_mask; /* Signals blocked during handler invocation */
    int sa_flags; /* Flags controlling handler invocation */
    void (*sa_restorer)(void); /* Not for application use */
};
```

Ожидание сигнала

```
#include <unistd.h>

int pause(void);
```

Reentrant and Async-Signal-Safe Functions

Не все системные вызовы и библиотечные функции могут быть безопасно вызваны из обработчика сигнала.

Две концепции:

reentrant functions

async-signal-safe functions

async-signal-safe функции

Table 21-1: Functions required to be async-signal-safe by POSIX.1-1990, SUSv2, and SUSv3

_Exit() (v3)	*getpid()*	*sigdelset()*	*dup2()*	*recv() (v3)*	*tcgetpgrp()*
_exit()	*getppid()*	*sigemptyset()*	*execle()*	*recvfrom() (v3)*	*tcsendbreak()*
abort() (v3)	*getsockname() (v3)*	*sigfillset()*	*execve()*	*recvmsg() (v3)*	*tcsetattr()*
accept() (v3)	*getsockopt() (v3)*	*sigismember()*	*fchmod() (v3)*	*rename()*	*tcsetpgrp()*
access()	*getuid()*	*signal() (v2)*	*fchown() (v3)*	*rmdir()*	*time()*
aio_error() (v2)	*kill()*	*sigpause() (v2)*	*fcntl()*	*select() (v3)*	*timer_getoverrun() (v2)*
aio_return() (v2)	*link()*	*sigpending()*	*fdatasync() (v2)*	*sem_post() (v2)*	*timer_gettime() (v2)*
aio_suspend() (v2)	*listen() (v3)*	*sigprocmask()*	*fork()*	*send() (v3)*	*timer_settime() (v2)*
alarm()	*lseek()*	*sigqueue() (v2)*	*fpathconf() (v2)*	*sendmsg() (v3)*	*times()*
bind() (v3)	*lstat() (v3)*	*sigset() (v2)*	*fstat()*	*sendto() (v3)*	*umask()*
cfgetispeed()	*mkdir()*	*sigsuspend()*	*fsync() (v2)*	*setgid()*	*uname()*
cfgetospeed()	*mkfifo()*	*sleep()*	*ftruncate() (v3)*	*setpgid()*	*unlink()*
cfsetispeed()	*open()*	*socket() (v3)*	*getegid()*	*setsid()*	*utime()*
cfsetospeed()	*pathconf()*	*sockatmark() (v3)*	*geteuid()*	*setsockopt() (v3)*	*wait()*
chdir()	*pause()*	*socketpair() (v3)*	*getgid()*	*setuid()*	*waitpid()*
chmod()	*pipe()*	*stat()*	*getgroups()*	*shutdown() (v3)*	*write()*
chown()	*poll() (v3)*	*symlink() (v3)*	*getpeername() (v3)*	*sigaction()*	
clock_gettime() (v2)	*posix_trace_event() (v3)*	*sysconf()*	*getpgrp()*	*sigaddset()*	
close()	*pselect() (v3)*	*tcdrain()*			
connect() (v3)	*raise() (v2)*	*tcflow()*			
creat()	*read()*	*tcflush()*			
dup()	*readlink() (v3)*	*tcgetattr()*			

Сигналы во время системных вызовов

Если во время системного вызова (например блокирующего read) пришел сигнал, то после обработки сигнала системный вызов возвратиться с ошибкой EINTR.

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue; /* Do nothing loop body */
```