OPERATING SYSTEMS CSCI-SHU215

Lesson 07 - Signals

Introduction to signals

Most basic inter-process communication mechanism

A signal is information transmitted to a program during its execution

- The system communicates with process users via this mechanism:
  - upon errors (memory violation, I/O error)
  - at the request of the user via the keyboard (ctrl-C, ctrl-Z...)
  - upon disconnection of the line/terminal, etc.
- Given the right (UID) permissions, a process may send a signal to another process
- Upon reception, signal handling induces a default behavior

Introduction to signals

Each signal is associated with a strictly positive integer value that is smaller than **NSIG** (non POSIX constant)

Each value is mapped to a name constant

- **/usr/include/signal.h**
- **List of signals:**

```
$ kill -l
```

- **Use the name instead of the number**

  Example: SIGKILL (= 9), SIGINT (= 2), etc.

```
$ kill -KILL <num. proc>; kill -INT <num. proc>
```

- Sending a signal comes down to sending an integer value to a process

Signals - Terminology

- Signal delivery

  A signal is delivered to a process when the process takes into account and performs the action associated with it.

- Pending signal

  Signal that was sent to a process but has yet to be delivered.
  - Signal reception is stored in the receiver's PCB
  - "One signal may hide another"

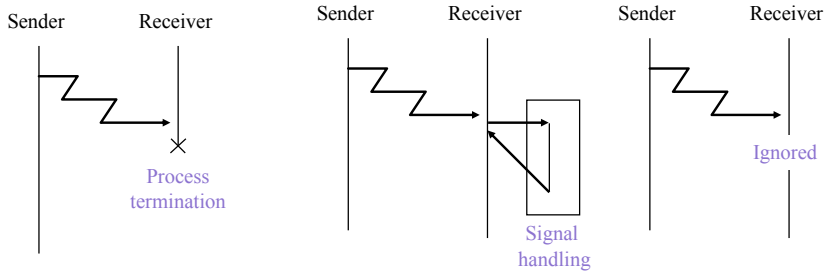
    Reception of a new signal on top of a pending signal with the same value causes the loss of the new signal

- Masked or blocked signal

  Deliberate adjournment of delivery for a signal value

Behavior upon delivery of a signal

Default behaviors are associated with the delivery of each signal value



Do not mix up with interrupts

Hardware: clock, disk, etc.

Main POSIX signals

Name	*Event*	*Behavior*
Termination		
SIGINT	ctrl-C	termination
SIGQUIT	<QUIT> ctrl-\	termination + core
SIGKILL	Kill a process	termination
SIGTERM	Termination signal	termination
SIGCHLD	Terminate or suspend a child process	ignored
SIGABRT	Abnormal termination	termination + core
SIGHUP	Disconnect terminal	termination

Main POSIX signals

Name	*Event*	*Behavior*
Suspension / Resumption		
SIGSTOP	Suspend execution	suspension
SIGTSTP	Suspend execution (ctrl-Z)	suspension
SIGCONT	Continuation of a suspended process	resumption
Error notifications		
SIGFPE	Arithmetical error	termination + core
SIGBUS	Bus error	termination + core
SIGILL	Illegal instruction	termination + core
SIGSEGV	Protected memory violation	termination + core
SIGPIPE	Write error on a pipe without reader	termination

Main POSIX signals

Name	*Event*	*Behavior*
Other		
SIGALRM	Timeout	termination
SIGUSR1	Reserved for the user	termination
SIGUSR2	Reserved for the user	termination
SIGTRAP	Trace / breakpoint trap	termination + core
SIGIO	Asynchronous I/O	termination

9

Sending and Receiving Signals

Sending a signal

Source code

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

pid

>0	process whose PID is *pid*
0	all processes in the same group as the sender
-1	not specified by POSIX (most systems: broadcast to all processes)
<-1	all processes in group whose GID is |*pid*|

signal

value between 0 and NSIG (0 sends no signal, tests the existence of pid)

Command line

`$ kill -l`	list all signals
`$ kill -sig pid`	send signal `sig` to process `pid`

Sending a signal - example


```
#define _XOPEN_SOURCE 700
#include <sys / types.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main (int arg, char ** argv) {
    printf ("begin program\n");
    / * Send a SIGINT to oneself * /
    kill (getpid(), SIGINT);
    printf ("end program\n");
    return EXIT_SUCCESS;
}
```

Signal delivery

Delivery occurs when the process **switches from active kernel mode to active user mode**

ie. return from a system call, return from a hardware interrupt, election by the scheduler

Default behaviors upon reception

- Process termination
- Termination of the process + production of a *core* file
- Signal ignored
- Suspension of the process (*stopped* or *suspended*)
- Resumption of the process

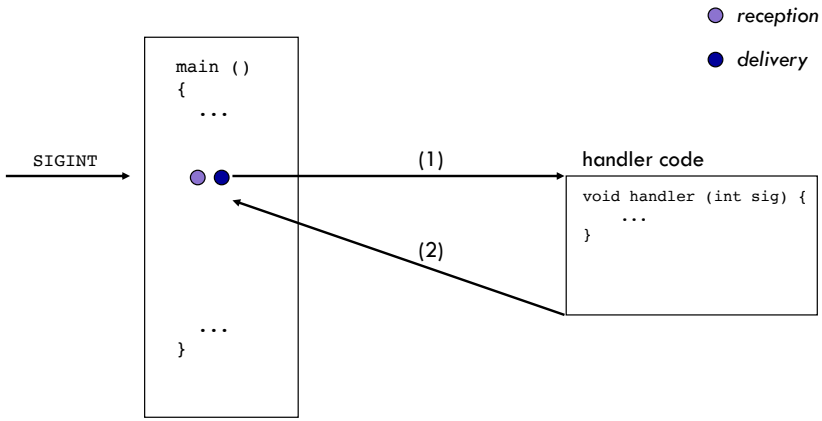
Behavior can be redefined by installing a new handler

- `SIG_IGN` (Ignore the signal)
- User-defined **handler function**
- `SIG_DFL` (Restore the default behavior)

Applicable to all signals ***except*** `SIGKILL`, `SIGSTOP`

User-defined signal delivery

13



Redefinition of the default handler

14

```
struct sigaction {
            void (* sa_handler) ();    / * Function * /
            sigset_t sa_mask;          / * Mask signals * /
            int sa_flags;              / * Options * /
};
```

Structure sigaction holds the data that defines signal reception behavior

- *sa_handler*
  function to execute, SIG_DFL (default behavior), or SIG_IGN (discard signal)
- *sa_mask*
  list of signals added to the current mask during the execution of the handler
  - *sa_mask U {sig}*
  - the value of the signal being delivered is **automatically** masked
- *sa_flags*
  behavioral options

Enforcing a new handler

15

```
int sigaction (   int sig,
                  struct sigaction * act,
                  struct sigaction * old    );
```

Redefines the handler *act* for the signal *sig*

- *act* and *old* point to a *struct sigaction*
- Delivery of signal *sig* triggers the function *act.sa_handler*
- *old:* if not NULL, references the previous handler

Sigaction - example

16

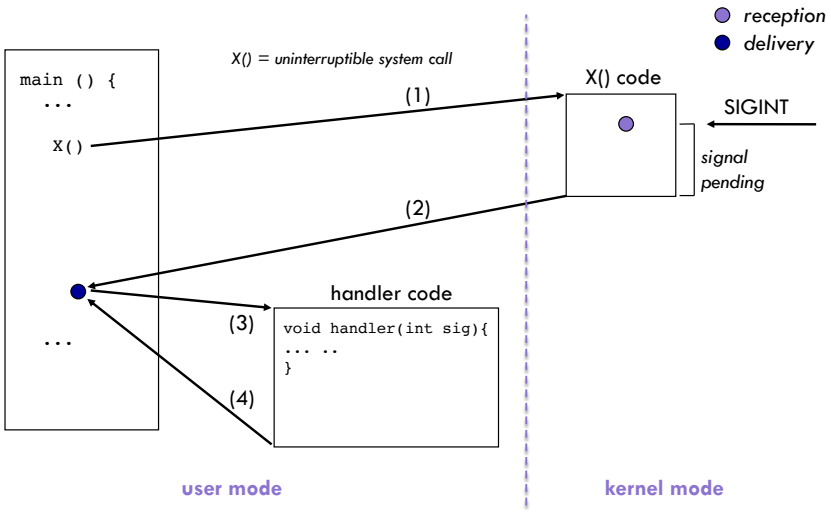
```
void sig_hand(int sig) {
  printf ( "received signal %d \n", sig);
}

int main (int argc, char ** argv) {
    sigset_t sig_proc;
    struct sigaction action;

    sigemptyset (&sig_proc);
    action.sa_mask = sig_proc;
    action.sa_flags = 0;
    action.sa_handler = sig_hand;

    sigaction (SIGINT, &action, 0);

    kill (getpid (), SIGINT);
    printf ("end program \n");

    return EXIT_SUCCESS;
}
```

sigaction-Ex.C

```
> Sigaction-ex
signal received 2
end program
```

Uninterruptible system calls

17



Interruptible system calls

18

A suspended process whose priority level is interruptible awakens upon receiving a signal

- Process goes to the ready state
- The signal gets delivered during the election process
  Execution of the *handler* function
- Examples of interruptible system calls:
  - *pause*,
  - *sigsuspend*,
  - *wait / waitpid*
  - *read, write*,
  - *etc.*

Interruptible system calls

19

The reception of a signal on an interruptible system call disrupts the call

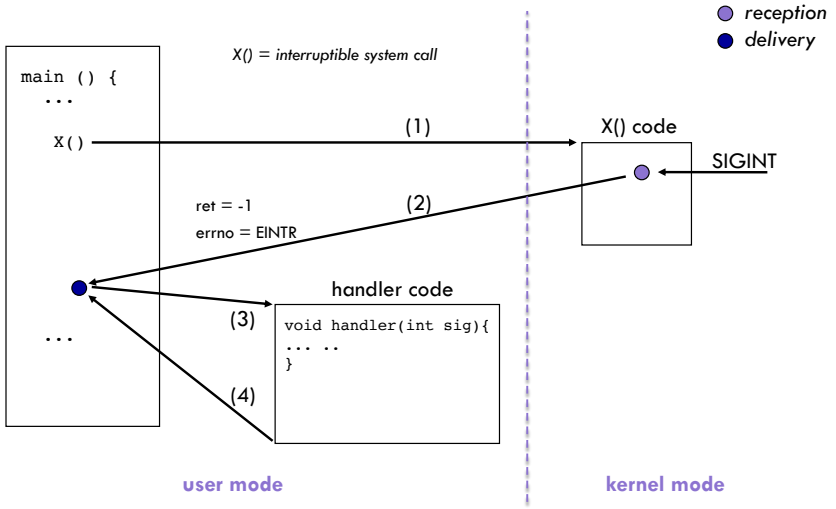
- System call returns -1, indicating a failure
- Error code: errno = EINTR.
- example:

```
ret = read (desc, buffer, BUFFSIZE)
    if ((ret == -1) && (errno == EINTR))
      printf ( "signal interrupt\n");
```

- User-defined handler can trigger automatic recovery

Interruptible system calls

20



21

Postponing Signal Delivery

Postponement of signal delivery

A process can choose to postpone the delivery of some signals

- Specifies a set of signals whose delivery gets deferred (mask)
- *Cannot defer `SIGKILL` and `SIGSTOP`*

```
int sigprocmask (int how, const sigset_t *set, sigset_t *old);
```

how	SIG_BLOCK	union of current mask and set
	SIG_UNBLOCK	unmask set
	SIG_SETMASK	block only signals in set
set	Signal mask	
old	if not NULL, stores value of the previous mask	

Call applies a new mask that replaces the current mask

New mask consists of *set*, or a compound of *set* and the previous mask

Signal set manipulation

The following functions do not change the signals themselves, but manipulate variables that define sets of signals

- `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 (sigset_t *set, int sig);`
  sigismember returns 1 if the signal is a member of the set, 0 otherwise

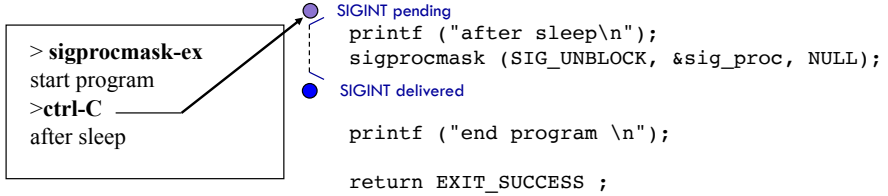
A process can list blocked signals that are pending

```
int sigpending (sigset_t *set);
```

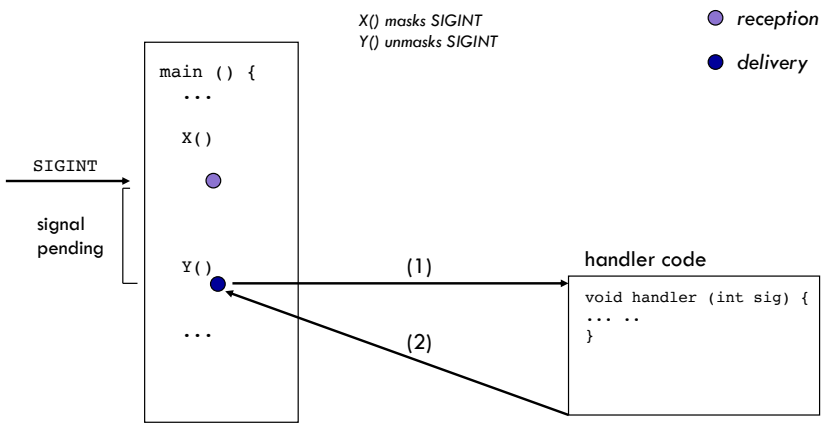
Masking signals - example

sigprocmask-ex.c

```
#define _XOPEN_SOURCE 700
#include <signal.h> // and others
int main (int argc, char ** argv) {
    sigset_t sig_proc;
    printf ("start program \n");

    sigemptyset (&sig_proc);
    sigaddset (&sig_proc, SIGINT);
    sigprocmask (SIG_BLOCK, &sig_proc, NULL);

    sleep (10);
    printf ("after sleep\n");
    sigprocmask (SIG_UNBLOCK, &sig_proc, NULL);

    printf ("end program \n");

    return EXIT_SUCCESS ;
}
```



Masking signals



Pending signals – example


```
int main (int argc, char * argv []) {

     sigset_t sig_set; /* List of blocked signals */

     sigemptyset (&sig_set);
     sigaddset (&sig_set, SIGINT);

     /* Hide SIGINT */
     sigprocmask (SIG_SETMASK, &sig_set, NULL);

     kill (getpid (), SIGINT);

     /* Get the list of pending signals */
     sigpending (& sig_set);

     if (sigismember (&sig_set, SIGINT))
         printf ("SIGINT is pending\n");

     return EXIT_SUCCESS;
}
```


Waiting for a Signal

Waiting for a signal

A process can suspend its execution voluntarily in order to wait for a signal

- switches to the suspended state
- gets awakened upon reception of an unmasked signal

`int pause (void)`

- Suspends process until reception of an unmasked signal

`int sigsuspend (sigset_t *p_ens)`

- Temporarily replaces current mask by *p_ens*
- If p_ens unmasks pending signals, these get delivered
- Otherwise process is suspended until reception of an unmasked signal
- Reinstates previous mask upon return of the function

Waiting for a signal – example


```
void sig_hand (int sig) {
     printf ( "received signal %d \n", sig);
}
int main (int argc, char ** argv) {
     sigset_t sig_proc;
     struct sigaction action;
     / * Mask SIGINT * /
     sigemptyset (&sig_proc);
     sigaddset (&sig_proc, SIGINT);
     sigprocmask (SIG_SETMASK, & sig_proc, NULL);
     / * Redefine handler for SIGINT * /
     action.sa_mask = sig_proc;
     action.sa_flags = 0;
     action.sa_handler = sig_hand;
     sigaction (SIGINT, &action, NULL);
     / * Wait for SIGINT * /
     sigfillset (&sig_proc);
     sigdelset (&sig_proc, SIGINT);
     sigsuspend (&sig_proc);
     return EXIT_SUCCESS;
}
```

Timer notification

Goal: interrupt the process after a time limit

- Arm a timer. Upon deadline, the process receives a signal.
- Two system calls
  - ***alarm*** : only real-time, resolution is in seconds, sends SIGALRM
  - ***setitimer*** : set different types of timers, finer resolution than the second, sends SIGALRM, SIGTVALRM, or SIGPROF
- Default behavior is process termination

Timer notification


```
unsigned alarm (unsigned seconds);
```

- Timer in seconds
  - Real time (wall-clock time)
  - Resolution is the second
- Sends SIGALRM upon expiration
- One call at a time per process
  - A new call arms a new timer and cancels the previous one
  - Calling with value 0 does not arm a new timer
- Return value
  - Amount of time left on the timer from a previous call
  - 0 if no alarm is currently set

Timer notification – example


```
void sig_hand (int sig) {
  printf ("received signal %d \n", sig);
  alarm (1);
}

int main (int argc, char ** argv) {
  sigset_t sig_proc; struct sigaction share;
  sigemptyset (& sig_proc);
  action.sa_mask = sig_proc;
  action.sa_flags = 0;
  action.sa_handler = sig_hand;
  sigaction (SIGALRM, & action, 0);
  while (1) {

        alarm(1);
        pause ();
  }
  return EXIT_SUCCESS;
}
```

sig_ALRM.c

```
> sig_ALRM
  received signal 20
  received signal 20
  received signal 20
... ..
```

33

Tricky Situations

Signals vs. Parenthood

34

Signal SIGCHLD notifies a parent process about changes of state incurred by its children

Automatically sent upon

- child termination
- child suspension: child receives SIGSTOP or SIGTSTP.

The default behavior is to ignore SIGCHLD

Signals vs Inheritance

35

A child process

- does not inherit the parent's pending signals
- inherits the parent's signal mask and handlers

`exec()` resets the default handlers

Loss of pending signals

36

"One signal may hide another"

The reception of a signal sets this signal value as pending
The reception of a new signal on top of a pending signal ***never gets delivered***

Loss of pending signals – example

sig_contUSR1.c

```
int cont;


void sig_hand (int sig) {
  if (sig == SIGUSR1)
        cont ++;
  else {
        printf ("nb of SIGUSR1 received: %d \n", cont);
        exit (0);
  }
}
```

Loss of pending signals – example

sig_contUSR1.c

```
pid_t pid_child;

int main (int argc, char ** argv) {
    sigset_t sig_proc; int i;  struct sigaction action;
    sigemptyset (& sig_proc);
    action.sa_mask = sig_proc;
    action.sa_flags = 0;
    action.sa_handler = sig_hand;
    sigaction (SIGUSR1, & action, 0);
    sigaction (SIGINT, & action, 0);
    if ((pid_child = fork ()) == 0) {
        while (1)
            pause ();
    } else {
        for (i = 0; i <20; i ++)
            kill (pid_child, SIGUSR1);
        kill (pid_child, SIGINT);
        wait (NULL);
        return EXIT_SUCCESS;
    }
}
```

```
> sig_contUSR1
  nb of SIGUSR1 received: 4
```

Limitations of signals

A signal notifies an event, but provides little more information than the type of event (signal value)

In particular, POSIX.1 signals do not allow to know

- the number of receptions of a signal value in-between deliveries
- the date of reception of a signal

  Most systems handle deliveries by order of increasing signal value
- the PID of the process that sent the signal