Operating Systems 2024/2025

TP Class 06 – Signals

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Some slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

operating system
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

Source: Dictionary.com

1 2 9 0
Departamento de
Engenharia Informática
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

Updated on
06 February 2025

Overview

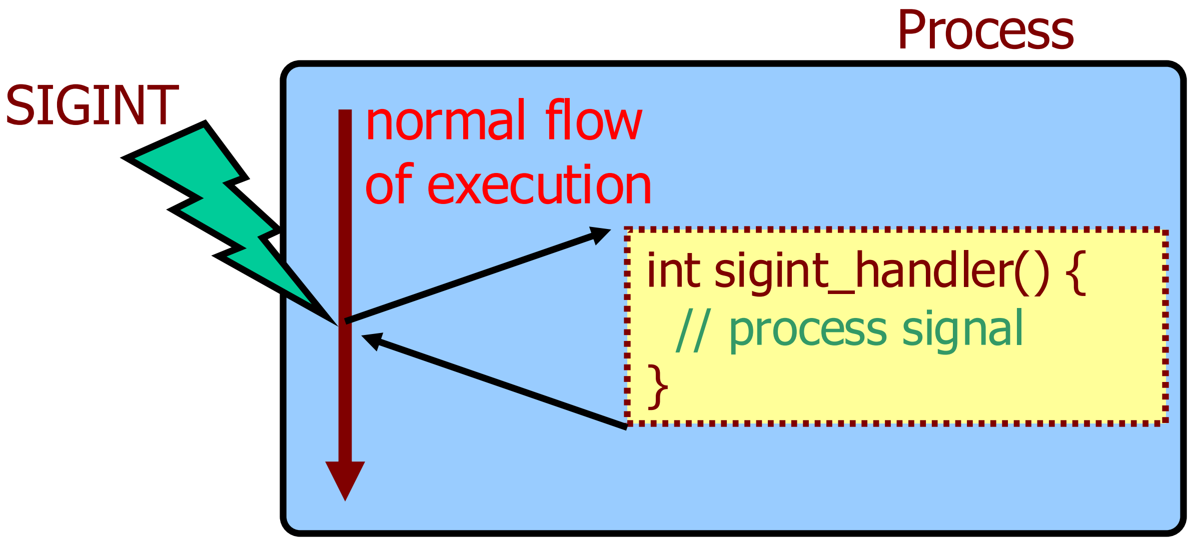
- A signal is a software interrupt;
- Signals notify a process that an event has occurred;
- HW can also create events (e.g., as result from a division by 0)
  - This events are detected by the HW, received by the kernel and sent to the process as signals;

- Signals have 2 main categories:
  - Standard POSIX reliable signals (POSIX.1 standard) ;
  - POSIX real-time signals (POSIX.1b standard) - will not be studied in our class!

  - Some advantages of POSIX real-time signals
    - They provide additional signals to be used by applications;
    - Signals are queued – if multiple instances of a signal are sent, the signal will be received many times;
    - Data may be sent together with the signal.

Overview (2)

- A signal represents an asynchronous event which an application must (should? can?) process
  - The programmer can register a routine to handle such events
- Examples:
  - The user hits Ctrl+C -> SIGINT
  - The system requests the application to terminate -> SIGTERM
  - The program tried to write to a closed channel -> SIGPIPE



```
int sigint_handler() {
  // process signal
}
```

Overview (3)

- Each signal is identified by a positive integer, defined in <signal.h> (or in one of the header files it includes)
  - 0 is a special case – is defined as *null signal* by POSIX.1
- Symbolic signal names start with SIGxxx
  - E.g., SIGINT, SIGSEGV

- A signal is generated by an event and delivered to a process. In between the signal is said to be pending.
- A pending signal is delivered to a process as soon as it is scheduled to run, or immediately if it is already running.

Delivery

- When a signal is delivered to a process one of the following default results occur:
  - The signal is **ignored** – is discarded by the kernel and has no effect on the process;
  - The process is **terminated**;
  - A coredump file is generated (able to be debugged) and the process is **terminated**;
  - The process is **stopped**.

- E.g.

Name	Description	Default action
SIGALRM	Timer expired (alarm)	terminate
SIGBUS	Memory access error	core dumped + terminate
SIGINT	Terminal interrupt character (Ctrl+C)	terminate
SIGTSTP	Terminal stop character (Ctrl+Z)	Stop process

Delivery (2)

- A process can change the disposition of the signal (i.e., the action that occurs when the signal is received). It can:
  - Ignore the signal
    - Signals are discarded - it is as if they had never existed.
  - Block the signal
    - Signals are stored in a queue (not always) until the process unblocks them - then, they are delivered.
  - Handle (catch) the signal;
    - Signals are redirected to a signal handler which is called.
  - None of the above - let the default action apply
    - Non handled/blocked/ignored signals - upon arrival, they cause program termination.

Delivery (3)

- Some signals cannot be ignored or handled
  - SIGKILL - sure kill signal – always terminates a process
  - SIGSTOP – sure stop signal (for job-control), always stops a process

- When a process starts, signals are on their “default behaviour”.
  - Some are ignored, most are in the “non-handled, non-blocked nor ignored state”. If a signal occurs, the process will die.

- Multiple occurrences of the same signal while a process has blocked it, may not result in multiples deliveries when the signal is unblocked
  - POSIX.1 allows the delivery of the signal either once or more than once
  - Only queued signals guarantee the delivery of multiple occurrences
    - Signals are always queued when POSIX real-time extensions are used

Basic signal functions

- signal function

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

prototype of the handler routine; it receives an integer and returns nothing

  - signo is the signal number or the symbolic signal name
  - handler can be a function, the constant SIG_IGN or SIG_DFL
    - function – address of a function that will handle the signal specified in signo (handler function)
    - SIG_IGN – ignores the signal specified in signo
    - SIG_DFL – restores the default handler of the signal specified by signo
  - Returns the previous handler of the signal specified in signo or SIG_ERR on error
  - It is the original API for setting the disposition of a signal and has a simple interface. However, there are variations in the implementation of signal() across different versions of Unix. The function sigaction() does not have that problems and has more functionalities - use it instead of signal(). In some systems signal() is based on sigaction() – in this case it presents no problems.

Basic signal functions (2)

Example: Handling a signal (demo01.c)

```
void sigint(int signum) {
  char option[2];
  printf("\n ^C pressed. Do you want to abort? ");

  scanf("%1s", option);
  if (option[0] == 'y') {
    printf("Ok, bye bye!\n");
    exit(0);
  }
}

int main()
{
  // Redirects SIGINT to sigint()
  signal(SIGINT, sigint);

  // Do some work!
  while (1) {
    printf("Doing some work...\n");
    sleep(1);
  }
  return 0;
}
```

Basic signal functions (3)

Special constants in `signal()`

```
signal(SIGINT, SIG_IGN)
```

Ignores SIGINT

```
signal(SIGINT, SIG_DFL)
```

Restores SIGINT to its "default" handling

Basic signal functions (4)

- `kill` function

```
#include <signal.h>
int kill(pid_t pid, int signo);
```

  - Sends a signal to a certain process identified by a PID=`pid`.
    - If `pid` is 0, sends to all processes in the current sender process group, including the sender process
  - If `signo` is 0 (*null signal*) no signal is sent, `kill()` only checks if a process can be signaled – it can be used to test if a process exists;
  - Returns 0 on success, or –1 on error

- `raise` function

```
#include <signal.h>
int raise(int signo);
```

  - Sends a signal to the process itself. Is equivalent to `kill(getpid(),signo);`
  - If using **pthreads** raise can be implemented as:
    `pthread_kill(pthread_self(),signo);`
  - Returns 0 on success or nonzero on error

Basic signal functions (5)

- Waiting for a signal - `pause`

```
#include <unistd.h>
int pause(void);
```

  - Suspends the execution of the process until a signal is received.
  - Always returns -1 with *errno* set to EINTR

Additional signal functions

- Signal sets

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

  - initialize the set and exclude all signals

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

  - initialize the set and include all signals

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

  - add signal to set

```
int sigaddset(sigset_t *set, int signo);
```

  - delete signal from set

```
int sigdelset(sigset_t *set, int signo);
```

  - A signal set represent multiple signals
  - Is used in signal functions that need signal sets
  - Returns 0 if OK, −1 on error

Additional signal functions

- Signal sets (example)

`sigset_t block_signals;`								…
`sigfillset(&block_signals);`	1	1	1	1	1	1	1	…
`sigemptyset(&block_signals);`	0	0	0	0	0	0	0	…
`sigaddset(&block_signals, 2);`	0	0	1	0	0	0	0	0
`sigaddset(&block_signals, 3);`	0	0	1	1	0	0	0	0
`sigdelset(&block_signals, 2);`	0	0	0	1	0	0	0	0

Additional signal functions (2)

- `sigprocmask` function

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set,
sigset_t *restrict oset);
```

  - The signal mask of a process is the set of signals currently blocked from delivery to that process.
  - `sigprocmask()` changes the process signal mask (`set` parameter), retrieves the existing mask (`oset` parameter), or both.
  - `how` determines the changes:
    - SIG_BLOCK - signals specified in the signal set pointed to by set are added to the existing signal mask;
    - SIG_UNBLOCK - signals in the signal set pointed to by set are removed from the existing signal mask;
    - SIG_SETMASK - The signal set pointed to by set replaces the existing signal mask
  - Returns 0 on success, or –1 on error
  - This function is for use with processes; to manipulate threads signal masks use `pthread_sigmask()`

Additional signal functions (3)

- `sigaction` function

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
struct sigaction *oact);
```

  - Sets or examines the action associated to the signal defined by `signo`.
  - If the `act` (action) pointer is non-null, we are modifying the action. If the `oact` (old action) pointer is non-null, the system returns the previous action for the signal through the `oact` pointer.
  - Returns 0 on success, or –1 on error

Additional signal functions (4)

- `sigaction` function (cont.)

```
struct sigaction {
  void (*sa_handler)(int);  // addr of signal handler(or SIG_IGN/SIG_DFL)
  sigset_t sa_mask;         // signals to block during handler
  int sa_flags;             // signal options (see manual)
  void (*sa_restorer)(void);  // not for application use
  };
```

  - `sa_handler`
    - specifies the action to be associated with signum and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.
  - `sa_mask`
    - Defines the signals to be blocked during the invocation of the handler `sa_handler`.
    - The signal that caused the handler to be invoked is automatically added to the mask (is also blocked);
    - When the handler returns the `sa_mask` is removed;
    - It specifies the set of signals that cannot interrupt this handler.

Additional signal functions (5)

- Other functions (not covered in our classes)
  - `sigismember()` : test for membership of a set.
  - `sigandset()` : intersects sets (and)
  - `sigorset()` : union of sets (or)
  - `sigpending()` : to determine which signals are pending for a process
  - `killpg()` : send signal to a process group
  - …
- And for threads…
  - `pthread_sigmask()` : signal mask for threads
  - `sigwait()` : wait for a signal
  - `pthread_kill()` : send a signal to a thread
- NOTE: In PL classes…
  - You may use `signal()` to handle signals in processes and `sigprocmask()` together with set functions to block them.

Example using sigaction()

- Example of setting up a handler to delete temporary files when certain fatal signals happen, using `sigaction()`

```
#include <signal.h>

void termination_handler (int signum) {
  struct temp_file *p;
  for (p = temp_file_list; p; p = p->next) unlink (p->name);
}

int main (void) {
  …
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;   // sets the new handler
  sigemptyset (&new_action.sa_mask);             //exclude all signals from set
  new_action.sa_flags = 0;

  sigaction (SIGINT, NULL, &old_action);   // find action associated with SIGINT
  if (old_action.sa_handler != SIG_IGN) sigaction (SIGINT, &new_action, NULL);
  sigaction (SIGHUP, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN) sigaction (SIGHUP, &new_action, NULL);
  sigaction (SIGTERM, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN) sigaction (SIGTERM, &new_action, NULL);
  …
}
```

Source: https://www.gnu.org

Example: Blocking a signal

demo02.c

```c
void sigint(int signum) {
  ...
}

int main()
{
  signal(SIGINT, sigint);

  sigset_t block_ctrlc;
  sigemptyset (&block_ctrlc);
  sigaddset (&block_ctrlc, SIGINT);

  // Do some work!
  while (1) {

    sigprocmask (SIG_BLOCK, &block_ctrlc, NULL);
    printf("Doing some work...\n");
    sleep(5);
    printf("End of job.\n");
    sigprocmask (SIG_UNBLOCK, &block_ctrlc, NULL);
  }
  return 0;
}
```

The problem with signals

- They make programming extremely hard
  - It’s completely asynchronous: you never know when you are going to get a signal
  - This means that you must protect all calls!

- After calling a standard function, it may return -1 indicating an error
  - errno==EINTR means that a certain routine was interrupted and has to be tried again.
  - Other routines return other things.
  - It you are using signals, you must protect them against all that!

The problem with signals (2)

- For instance, simply to try to read a “struct person” from disk…

```
struct person p;
...
int n, total = 0;
while (total < sizeof(p))
{
  n = read(fd, (char*)p + total, sizeof(p)-total);
  if (n == -1)
  {
    if (errno == EINTR)
      continue;
    else
    {
      // True error!
    }
  }
  total+= n;
}
```

And you have to do something like this
for all calls being done in your program!

Example: Sending a signal

demo03.c

- It’s just a question of calling kill() with the PID of the target process…

```
void master(pid_t pid_son)
{
  printf("Master sleeping for a while...\n");
  sleep(3);
  printf("Master says: Hello son!\n");
  kill(pid_son, SIGUSR1);
}

int main() {
  pid_t son;

  // Creates a worker process
  if ((son=fork()) == 0) {
    worker();
    exit(0);
  }

  // The master
  master(son);
  wait(NULL);
  return 0;
}
```

Example: Sending a signal (1)

demo03.c

- The code of the child process

```
void dady_call(int signum)
{
  printf("Dady has just called in!\n");
}

void worker()
{
  // Redirect "user signal 1" to a handler routine
  signal(SIGUSR1, dady_call);

  // Do some work
  printf("Child process, life is good...\n");
  for (int i=0; i<10; i++)
  {
    printf("Child doing some work\n");
    sleep(1);
  }

  printf("Child saying bye bye!\n");
}
```

Danger!!!

- What do you think it will happen if you receive a signal inside a signal handler??
  - In most systems, upon entering a signal handling routine, all signals of that type become blocked (i.e. they are queued). [Well, for “normal” signals, a finite set of them are queued (typically 1); for “real time signals”, all are...]
  - The other signals are still processed asynchronously if they arrive.
  - This behaviour is not consistent across systems. In fact, in some systems, that signal type resets to its default behaviour. This means that if, meanwhile, the program receives a signal of the same type it may die! On that type of system, the first thing that you must do is to once again set the signal handler.

```
void dady_call(int signum)
{
  signal(SIGUSR1, dady_call);
  printf("Dady has just called in!\n");
}
```

  - Well... doesn’t really solve the problem, it just makes it less likely.
  - The new POSIX routines address this – use them. Also, most system nowadays do not reset the signal handler.

Beware!

- Signal numbers vary across operating systems and architectures. Do not rely on them, use symbolic constants!

Source: `man 7 signal`

	Architecture					
Signal	x86/ARM most others	Alpha/ SPARC	MIPS	Standard that specified the signal	Action	Comment
SIGABRT	6	6	6	P1990	Core	Abort signal from abort(3)
SIGALRM	14	14	14	P1990	Term	Timer signal from alarm(2)
SIGBUS	7	10	10	P2001	Core	Bus error (bad memory access)
SIGCHLD	17	20	18	P1990	Ign	Child stopped or terminated
SIGCLD	-	-	18	-	Ign	A synonym for SIGCHLD
SIGCONT	18	19	25	P1990	Cont	Continue if stopped
SIGEMT	-	7	7	-	Term	Emulator trap
SIGFPE	8	8	8	P1990	Core	Floating-point exception
SIGHUP	1	1	1	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	4	4	4	P1990	Core	Illegal Instruction
SIGINFO	-	29/-	-	-		A synonym for SIGPWR
SIGINT	2	2	2	P1990	Term	Interrupt from keyboard (^C)
SIGIO	29	23	22	-	Term	I/O now possible (4.2BSD)
SIGIOT	6	6	6	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	9	9	9	P1990	Term	Kill signal (cannot be caught, blocked or ignored)
SIGLOST	-	-/29	-	-	Term	File lock lost (unused)
SIGPIPE	13	13	13	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL				P2001	Term	Pollable event (Sys V); synonym for SIGIO
SIGPROF	27	27	29	P2001	Term	Profiling timer expired
SIGPWR	30	29/-	19	-	Term	Power failure (System V)
SIGQUIT	3	3	3	P1990	Core	Quit from keyboard
SIGSEGV	11	11	11	P1990	Core	Invalid memory reference (segmentation violation)
SIGSTKFLT	16	-	-	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	19	17	23	P1990	Stop	Stop process (cannot be caught, blocked or ignored)
SIGSYS	31	12	12	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	15	15	15	P1990	Term	Termination signal
SIGTRAP	5	5	5	P2001	Core	Trace/breakpoint trap
SIGTSTP	20	18	24	P1990	Stop	Stop typed at terminal (^Z)
SIGTTIN	21	21	26	P1990	Stop	Terminal input for background process
SIGTTOU	22	22	27	P1990	Stop	Terminal output for background process
SIGUNUSED	31	-	-	-	Core	Synonymous with SIGSYS
SIGURG	23	16	21	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	10	30	16	P1990	Term	User-defined signal 1
SIGUSR2	12	31	17	P1990	Term	User-defined signal 2
SIGVTALRM	26	26	28	P2001	Term	Virtual alarm clock (4.2BSD)
SIGWINCH	28	28	20	-	Ign	Window resize signal (4.3BSD, Sun)
SIGXCPU	24	24	30	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	25	25	31	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)

P1990 = signal described in POSIX.1-1990 standard

Signals and POSIX threads

- Some conflicts exist between the Unix signal model that was based on processes and POSIX threads model.

- Signal actions are process-wide – if a signal received by a thread has as default action to stop or terminate, then all threads in the process are stopped or terminated.

- Actions are shared between all threads – when an action is changed in a thread, all threads response to the signal is changed.

Signals and POSIX threads (2)

- A signal may be directed to a specific thread (thread-directed):
  - in the case of synchronous signals that result from a specific thread execution, which are received by that same thread (e.g., SIGFPE – floating point exception);
  - by using `pthread_kill()` to enable a thread to send a signal to other thread;
- A signal is process-directed:
  - if sent from a process to other process;
  - in the case of asynchronous signals, which are received by any thread that has not blocked them (pthread_sigmask);
- When a signal is delivered to a multithreaded process just <u>one</u> thread catches it
- Each thread may have its own thread mask

Signals from the Linux shell

- Using Linux command line
  - List signals

```
$ kill -l
```

  - Send a SIGALRM to a process with PID= 76543

```
$ kill -SIGALRM 76543
```

  - Send a signal to all processes that have the same name

```
$ killall -SIGKILL myproc
```

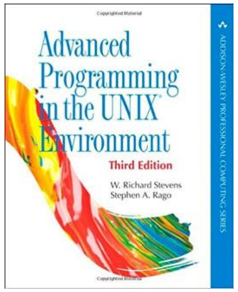
- Special Characters:
  - CTRL+C
    - When '^C' is pressed the terminal will send a SIGINT to the **foreground process group** of the terminal. This is based on the POSIX specification (http://pubs.opengroup.org/onlinepubs/9699919799/).

Note: Test with `demo04.c`

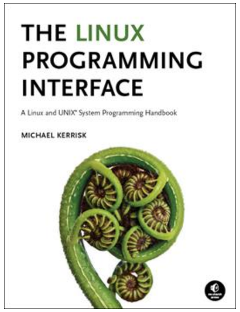
Class demos included

- Demo01 – handling a signal
  `demo01.c`
- Demo02 – blocking a signal
  `demo02.c`
- Demo 03 – sending a signal
  `demo03.c`
- Demo04 – sending a signal to a group of processes
  `demo04.c`
- Demo05 – change handler using “signal”
  `demo05.c`
- Demo06 – change handler using “sigaction”
  `demo06.c`

References



- [Stevens13]
  - Chapter 10 – Signals
  - Chapter 12.8 – Threads and Signals



- [Kerrisk10]
  - Chapter 20 – Signals: Fundamental Concepts
  - Chapter 21 – Signals: Signal Handlers

INTRODUCTION TO ASSIGNMENT 07 – “SIGNALS AND PIPES”

Thank you! Questions?

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.
—Rudyard Kipling