Operating Systems & Systems Programming I

Signals and Pipes

Keith Bugeja / Joshua Ellul / Alessio Magro / Kevin Vella

May 9, 2018

Department of Computer Science
Faculty of ICT, University of Malta

Signals

Signal definition

What are Unix **signals**?

Signal definition

What are Unix **signals**?

A Unix signal is a software interrupt:

Signal definition

What are Unix **signals**?

A Unix signal is a software interrupt:

- provides a way of handling asynchronous events (e.g. user trying to stop a program, kernel reporting some violation or error);

Signal definition

What are Unix **signals**?

A Unix signal is a software interrupt:

- provides a way of handling asynchronous events (e.g. user trying to stop a program, kernel reporting some violation or error);
- signals date since early Unix (model not reliable); shortcomings in original model:
  - suffered from lost signals
  - difficult to process or turn off signals during critical regions

Signal definition

What are Unix **signals**?

A Unix signal is a software interrupt:

- provides a way of handling asynchronous events (e.g. user trying to stop a program, kernel reporting some violation or error);
- signals date since early Unix (model not reliable); shortcomings in original model:
  - suffered from lost signals
  - difficult to process or turn off signals during critical regions
- reliable signals standardised by POSIX.1

Every Unix signal has a name

- begins with `SIG`
  - e.g. `SIGABRT`, when a process calls `abort` function, or `SIGALRM`, when the timer set by the `alarm` function goes off
- defined by positive integer constants in `<signal.h>`
  - called signal numbers
- no signal has a number 0
  - signal number 0 is a special case
  - POSIX.1 calls this the null signal

There are numerous conditions that can generate a signal:

Signal generation

There are numerous conditions that can generate a signal:

- Terminal-generated
  - Ctrl+C causes `SIGINT`, Ctrl+Z raises `SIGTSTP`, etc.

Signal generation

There are numerous conditions that can generate a signal:

- Terminal-generated
    - Ctrl+C causes `SIGINT`, Ctrl+Z raises `SIGTSTP`, etc.
- Hardware exceptions
    - division by 0, invalid memory reference, etc.
    - usually detected by hardware, which notifies kernel
    - kernel generates appropriate signal for the process running at the time the condition occurred:
        - e.g. `SIGSEGV` is generated for a process that executes an invalid memory reference

Signal generation

There are numerous conditions that can generate a signal:

- Terminal-generated
  - Ctrl+C causes `SIGINT`, Ctrl+Z raises `SIGTSTP`, etc.
- Hardware exceptions
  - division by 0, invalid memory reference, etc.
  - usually detected by hardware, which notifies kernel
  - kernel generates appropriate signal for the process running at the time the condition occurred:
    - e.g. `SIGSEGV` is generated for a process that executes an invalid memory reference
- `kill` function allows a process to send any signal to another process or process group, provided the process being sent the signal is owned by the sender or the sender is the superuser
  - `kill` command is an interface to the `kill` function

Signal generation

There are numerous conditions that can generate a signal:

- Terminal-generated
  - Ctrl+C causes `SIGINT`, Ctrl+Z raises `SIGTSTP`, etc.
- Hardware exceptions
  - division by 0, invalid memory reference, etc.
  - usually detected by hardware, which notifies kernel
  - kernel generates appropriate signal for the process running at the time the condition occurred:
    - e.g. `SIGSEGV` is generated for a process that executes an invalid memory reference
- `kill` function allows a process to send any signal to another process or process group, provided the process being sent the signal is owned by the sender or the sender is the superuser
  - `kill` command is an interface to the `kill` function
- software conditions can also generate signals to notify processes of various events
  - e.g. `SIGPIPE` (process writes to a pipe that has no reader) or `SIGALRM` (alarm clock set by process expires)

Signals

Signals are classic examples of asynchronous events

- they occur at what appear to be random times to the process
- process can’t simply test a variable to see whether a signal has occurred
  - has to tell kernel what to do if signal occurs
  - this is called the *disposition*, or *action*, associated with the signal

Note

A list of system-supported signals can be generated using `kill -l`.

Overview

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

Signal dispositions fall into either of three categories:

Signal dispositions fall into either of three categories:

1. **Ignore** the signal: works for most signals except for `SIGKILL` and `SIGSTOP`
    - provide kernel and superuser with a surefire way of killing or stopping any process
    - ignoring some signals (e.g. division by 0) leads to undefined behaviour

Signal dispositions

Signal dispositions fall into either of three categories:

1. **Ignore** the signal: works for most signals except for `SIGKILL` and `SIGSTOP`
    - provide kernel and superuser with a surefire way of killing or stopping any process
    - ignoring some signals (e.g. division by 0) leads to undefined behaviour
2. **Catch** the signal: tell the kernel to call a function of ours whenever the signal occurs
    - we can do whatever we want to handle the condition
        - e.g. if `SIGCHLD` is caught, it means that a child process has terminated; signal catching function can then call `waitpid` to fetch id and termination status
        - e.g. if process uses temporary files, handle `SIGTERM` to clean up

Signal dispositions

Signal dispositions fall into either of three categories:

1. **Ignore** the signal: works for most signals except for `SIGKILL` and `SIGSTOP`
   - provide kernel and superuser with a surefire way of killing or stopping any process
   - ignoring some signals (e.g. division by 0) leads to undefined behaviour
2. **Catch** the signal: tell the kernel to call a function of ours whenever the signal occurs
   - we can do whatever we want to handle the condition
     - e.g. if `SIGCHLD` is caught, it means that a child process has terminated; signal catching function can then call `waitpid` to fetch id and termination status
     - e.g. if process uses temporary files, handle `SIGTERM` to clean up
3. Let the **default action** apply: every signal has a default action (for most signals, this is to terminate the process)
   - actions are terminate, terminate+core, ignore, continue/ignore and stop process

Signal function

The simplest interface to the signal features of Unix is the `signal` function:

```c
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);

// or

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

Signal function

The simplest interface to the signal features of Unix is the `signal` function:

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

// or

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

If `signal` succeeds, it returns the previous function disposition

- on error, it returns `SIG_ERR`

Signal and ISO C

The `signal` function is defined by the ISO C standard:

- assumes no multiple/concurrent processes, process groups, terminal I/O, etc.
- vague enough to be almost useless for modern operating systems
- semantics differ among implementations
  - POSIX standard `sigaction` function addresses most of the shortcomings of `signal`

Signal

`signal` lets us register handlers for specific signals:

```
void my_handler(int signo);
```

- When a signal is received, for which a signal handler is registered, execution is temporarily suspended until the handler returns.
- The `func` (or `sighandler_t`) argument may be `SIG_IGN`, `SIG_DFL` or a function with a prototype like `my_handler` above.

Note

`SIGKILL` and `SIGSTOP` are important signals that the operating system uses to terminate or stop processes; they should not be trapped - many systems, in fact, do not allow them to be trapped

Waiting for signals

A process may wait for a signal, if it so wishes:

- POSIX defines the `pause` system call that puts a process to sleep until a signal is received
- the signal is either handled or the process terminated

```
#include <unistd.h>

int pause(void);
```

A process may wait for a signal, if it so wishes:

- POSIX defines the `pause` system call that puts a process to sleep until a signal is received
- the signal is either handled or the process terminated

```c
#include <unistd.h>

int pause(void);
```

The `pause` function only returns if a signal is received:

- on Linux, `pause` puts the process in interruptible sleep state;
- calls `schedule` to invoke the Linux process scheduler and find another process to run

Let's look at our first example:

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
// signal handler
static void sig_usr(int signo) {
  switch (signo) {
    case SIGUSR1:
      printf("Received SIGUSR1!\n");
      break;
    case SIGUSR2:
      printf("Received SIGUSR2!\n");
      break;
    default:
      printf("Received signal %d\n", signo);
  }
}
// register handlers for SIGUSR1 and SIGUSR2
int main(int argc, char *argv[]) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR1!\n");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR2!\n");

  for (;;) // spin forever
    pause(); // sleep until signal is received
}
```

We build and run the example as a background process, sending `SIGUSR1`, `SIGUSR2` and `SIGKILL`, which terminates the program:

```
$ gcc -o signal ex_signal.c
$ ./signal &
[1] 12130
$ kill -SIGUSR1 12130
Received SIGUSR1!
$ kill -SIGUSR2 12130
Received SIGUSR2!
$ kill -SIGKILL 12130
$ ps
   PID TTY          TIME CMD
  1372 pts/3    00:00:00 bash
 12158 pts/3    00:00:00 ps
[1]+  Killed                  ./signal
```

Overview

We build and run the example as a background process, sending `SIGUSR1`, `SIGUSR2` and `SIGKILL`, which terminates the program:

```
$ gcc -o signal ex_signal.c
$ ./signal &
[1] 12130
$ kill -SIGUSR1 12130
Received SIGUSR1!
$ kill -SIGUSR2 12130
Received SIGUSR2!
$ kill -SIGKILL 12130
$ ps
   PID TTY          TIME CMD
  1372 pts/3    00:00:00 bash
 12158 pts/3    00:00:00 ps
[1]+  Killed                  ./signal
```

Note

Try sending the process signals other than the above to highlight their default dispositions.

What happens to signals when a process calls `fork` or `exec`?

Signal inheritance

What happens to signals when a process calls `fork` or `exec`?

- on `fork` the child process inherits the signal actions of its parent
    - child copies registered actions (ignore, default, handle)
    - pending signals are not inherited (sent to specific pid!)

Signal inheritance

What happens to signals when a process calls `fork` or `exec`?

- on `fork` the child process inherits the signal actions of its parent
  - child copies registered actions (ignore, default, handle)
  - pending signals are not inherited (sent to specific pid!)
- on `exec` all signals are set to their default actions
  - any signal caught by the process before `exec` is reset to the default action after `exec`
  - all other signals remain the same

Signal inheritance

Signal Behaviour	Across `fork`	Across `exec`
ignored	inherited	inherited
default	inherited	inherited
handled	inherited	not inherited
pending signals	not inherited	inherited

Table 1: Signal behaviour across `fork` and `exec`.

Signal Behaviour	Across `fork`	Across `exec`
ignored	inherited	inherited
default	inherited	inherited
handled	inherited	not inherited
pending signals	not inherited	inherited

Table 1: Signal behaviour across `fork` and `exec`.

Note

When a process (or a shell) executes a background process, the newly executed process should ignore the interrupt and quit characters (`SIGINT` and `SIGQUIT` set to `SIG_IGN`).

Signal names

Mapping signal numbers to signal name strings

- it is convenient to be able to convert a signal number to a string representation of its name
- three methods to achieve this
    - `sys_siglist` - a statically defined string map
    - `psignal` - function originating from BSD Unix
    - `strsignal` - non-standard GNU extension function

sys_siglist

Mapping signal numbers to signal name strings (`sys_siglist`)

- retrieve the string from a statically defined list
- strings are indexed by signal number

```
extern const char * const sys_siglist[];
```

psignal

Mapping signal numbers to signal name strings (`psignal`)

- use the BSD-defined `psignal` interface, which is supported by Linux:

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

psignal

Mapping signal numbers to signal name strings (`psignal`)

- use the BSD-defined `psignal` interface, which is supported by Linux:

```c
#include <signal.h>

void psignal(int signo, const char *msg);
```

Function is similar in structure to `perror`

- prints to `stderr` the string supplied in `msg`, followed by a colon and the signal name given by `signo`

strsignal

Mapping signal numbers to signal name strings (`strsignal`)

- use GNU `strsignal` function
- non-standard by supported by Linux and many non-Linux systems

```c
#define _GNU_SOURCE
#include <string.h>

char *strsignal(int signo);
```

strsignal

Mapping signal numbers to signal name strings (`strsignal`)

- use GNU `strsignal` function
- non-standard by supported by Linux and many non-Linux systems

```c
#define _GNU_SOURCE
#include <string.h>

char *strsignal(int signo);
```

A call to `strsignal` returns a pointer to a description of the signal given by `signo`.

Example : signal number to string

Example handling `SIGUSR1` and `SIGUSR2`; the handler outputs a string representation of the signal number using all three methods:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

extern const char * const sys_siglist[];

static void sig_usr(int signo) {
  psignal(signo, "Using psignal");
  printf("Using strsignal: %s\n", strsignal(signo));
  printf("Using sys_siglist: %s\n", sys_siglist[signo]);
}

int main(int argc, char *argv[]) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR1!\n");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR2!\n");

  for (;;)
    pause();
}
```

As can be observed by running the program and sending signals to it, the three functions return identical descriptions for the same signal:

```
$ gcc -o sigstring ex_sigstring.c
$ ./sigstring &
[1] 87414
$ kill -SIGUSR1 87414
Using psignal: User defined signal 1
Using strsignal: User defined signal 1
Using sys_siglist: User defined signal 1
$ kill -SIGUSR2 87414
Using psignal: User defined signal 2
Using strsignal: User defined signal 2
Using sys_siglist: User defined signal 2
$ kill -SIGKILL 87414
[1]+  Killed                  ./sigstring
```

Sending signals

Using the `kill` utility we can send signals to processes; but programmatically?

- the `kill` system call sends a signal from one process to another
- it is the basis of the `kill` utility

Sending signals

Using the `kill` utility we can send signals to processes; but programmatically?

- the `kill` system call sends a signal from one process to another
- it is the basis of the `kill` utility

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

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

Using the `kill` utility we can send signals to processes; but programmatically?

- the `kill` system call sends a signal from one process to another
- it is the basis of the `kill` utility

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

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

If `kill` succeeds, it sends the signal `signo` to the process(es) identified by `pid` and returns 0

- a call is considered successful as long as a single signal is sent
- on failure (no signals sent) it returns -1 and sets `errno` appropriately

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

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

In a call to `kill`, `pid` is interpreted as follows:

- `pid > 0` - `signo` sent to the process identified by `pid`
- `pid = 0` - `signo` sent to every process in the same group of the sender
- `pid = -1` - `signo` sent to every process the sender has permission to send a signal to, except itself and `init`
- `pid < -1` - `signo` sent to the process group `abs(pid)`

A process, should it so desire, can send a signal to itself.

- In the general sense, this is accomplished through the `raise` interface

```c
#include <signal.h>

int raise(int signo);
```

A call to `raise` returns 0 on success

- on failure, it returns a non-zero value; it does **not** set `errno`

A process, should it so desire, can send a signal to itself.

- In the general sense, this is accomplished through the `raise` interface

```c
#include <signal.h>

int raise(int signo);
```

A call to `raise` returns 0 on success

- on failure, it returns a non-zero value; it does **not** set `errno`

Note

`raise(signo)` is equivalent to `kill(getpid(), signo)`.

Example : raise signal

In this example, we force the program to sleep for 3 seconds before raising SIGALRM:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

extern const char * const sys_siglist[];

static void sig_alrm(int signo) {
  printf("Received: %s\n", sys_siglist[signo]);
}

int main(int argc, char *argv[]) {
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGALRM!\n");

  sleep(3);
  raise(SIGALRM);

  printf("SIGALRM expired\n");
}
```

The example shows how the final message only appears after SIGALRM has been handled:

```
$ gcc -o raise ex_raise.c
$ ./raise
Received: Alarm clock
SIGALRM expired
$
```

The same functionality of the previous example may be replicated using the `alarm` function together with `pause`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>

extern const char * const sys_siglist[];

static void sig_alrm(int signo) {
  printf("Received: %s\n", sys_siglist[signo]);
}

int main(int argc, char *argv[]) {
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGALRM!\n");

  alarm(3);
  pause();

  printf("SIGALRM expired!\n");
}
```

`alarm` and `pause`

Note that the `alarm` function is asynchronous in nature; removing the following `pause` lets the program terminate before the `alarm` signal triggers and can be handled.

- Remember: the `pause` function puts the process to sleep until a signal is received.

Example: a simple `kill` utility

The next example is a simplistic implementation of the `kill` utility:

- specify signal number and multiple process identifiers on command line
- parse the input and send signal number to each process

Example: a simple `kill` utility

The next example is a simplistic implementation of the `kill` utility:

- specify signal number and multiple process identifiers on command line
- parse the input and send signal number to each process

The usage is:

```
$ skill signo pid1 [pid2 ... pidN]
```

Example: simple kill

Part I : Includes, externally defined variables and input validation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <signal.h>
#include <ctype.h>

// sys_siglist to convert signal number to signal name string
extern const char * const sys_siglist[];

// input validation : check if string is a positive integer
int is_number(char *s) {
  for (; *s; ++s)
    if (!isdigit(*s))
      return false;

  return true;
}
...
```

Example: simple kill

Part II : Validate program arguments

```c
int main(int argc, char *argv[])
{
  if (argc < 3) {
    fprintf(stderr, "Usage: skill signo pid-1 <pid-2 ... pid-n>\n");
    exit(EXIT_FAILURE);
  }

  if (!is_number(argv[1])) {
    fprintf(stderr, "Syntax error : signo expected integer\n");
    exit(EXIT_FAILURE);
  }

  // ASCII string to integer
  int signo = atoi(argv[1]);
  ...
```

Example: simple kill

Part III : Iterate through process identifiers, signalling each one

```c
  ...
  // Iterate through all pids
  for (int n = 2; argv[n] != NULL; n++) {
    if (is_number(argv[n])) {
      pid_t pid = atoi(argv[n]);

      printf("Signal [%s -> %d :: %d]\n", sys_siglist[signo],
 ↪    pid, kill(pid, signo));
    } else printf("Ignoring pid [%s]\n", argv[n]);
  }

  exit(EXIT_SUCCESS);
}
```

Caveats of signal handling

- on a number of systems, when a user-defined signal handler executes, the disposition is reset and has to be set again
- most slow system calls on System V implementations trap signals by returning an error value of -1 and setting `errno` to `EINTR`
- system calls may be interrupted by a signal; special care must be taken as to the actions carried out inside the handler
  - only reentrant functions should be used in the signal handler; even so, `errno` variable might still be corrupted

When the kernel raises a signal, a process can be executing code anywhere

- process in the middle of an important operation that cannot be interrupted
- or handling another signal
- or even executing a system call (e.g. `malloc` or `write`)!

Reentrancy in signal handling

When the kernel raises a signal, a process can be executing code anywhere

- process in the middle of an important operation that cannot be interrupted
- or handling another signal
- or even executing a system call (e.g. `malloc` or `write`)!

The signal handler cannot tell where the process is executing when a signal triggers

- signal handlers must practise caution when modifying global data, for instance
- signal handlers must **not** invoke any non-reentrant functions!

Reentrancy in signal handling

By non-reentrant functions, we mean functions that:

- use static or global data structures
- call malloc or free
- are part of the standard I/O library (e.g. printf is not guaranteed to produce expected results)

Reentrancy in signal handling

By non-reentrant functions, we mean functions that:

- use static or global data structures
- call malloc or free
- are part of the standard I/O library (e.g. printf is not guaranteed to produce expected results)

There is only one `errno` variable per thread, and function calls in the signal handler might modify its value.

Reentrancy in signal handling

By non-reentrant functions, we mean functions that:

- use static or global data structures
- call malloc or free
- are part of the standard I/O library (e.g. printf is not guaranteed to produce expected results)

There is only one `errno` variable per thread, and function calls in the signal handler might modify its value.

Note

To conserve the state of the system, signal handlers should only use reentrant system calls (SUS provides a list of functions guaranteed to be reentrant).

Signal sets

Signal sets

A signal set is a data type to represent multiple signals

- POSIX.1 designates `sigset_t` to contain a signal set and provides interfaces for its manipulation:

```c
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);

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

All functions but `sigismember` return 0 on success and -1 on error

- `sigismember` returns 1 if true, 0 if false and -1 on error

Signal sets

```c
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);

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

- `sigemptyset` initialises set so that all signals are excluded
- `sigfillset` initialises set so that all signals are included
- once a signal set is initialised, signals may be added or deleted using `sigaddset` and `sigdelset` respectively
- `sigismember` tests whether a given signal `signo` is enabled in the signal set `set`

Signal sets

Linux also provides a number of non-standard functions:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);
int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);
int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

- `sigisempty` returns 1 if `set` is empty, 0 otherwise
- `sigorset` places the union (the binary OR) of the signal sets left and right in `dest`
- `sigandset` places the intersection (the binary AND) of the signal sets left and right in `dest`
- return 0 on success and -1 on error, setting `errno` appropriately

Note

Although useful, these functions should be avoided if POSIX compliance is desired.

Signal handlers run asynchronously, at any time

- issues raised by non-reentrant functions
- cannot be called within a signal handler

Signal handlers run asynchronously, at any time

- issues raised by non-reentrant functions
- cannot be called within a signal handler

In some cases, however:

- program may need to share data between signal handler and other regions (access shared or global data)
- portions of program might need to execute without interruptions

Blocking signals

Enter critical sections…

- protected by temporarily suspending the *delivery* of signals
- signals become *blocked* and are not handled until they are *unblocked*

Blocking signals

Enter critical sections…

- protected by temporarily suspending the *delivery* of signals
- signals become *blocked* and are not handled until they are *unblocked*

A process may block any number of signals

- the set of signals blocked by a process is called its *signal mask*
- signal masks are stored in the data type `sigset_t`

Blocking signals

POSIX defines a function for managing process signal masks, which is implemented in Linux:

```
#include <signal.h>

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

Blocking signals

POSIX defines a function for managing process signal masks, which is implemented in Linux:

```c
#include <signal.h>

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

Function behaviour hinges on the value of the argument `how`:

- `SIG_SETMASK` - signal mask for the invoking process is changed to `set`
- `SIG_BLOCK` - signals in `set` are added to the process signal mask, which is changed to the union (binary OR'd) of the current mask and `set`.
- `SIG_UNBLOCK` - signals in `set` are removed from the process signal mask: the signal is changed to the intersection (binary AND'd) of the current mask and the negation (binary NOT'd) of `set`. It is illegal to unblock a signal that is not blocked.

```c
#include <signal.h>

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

If `oldset` is not `NULL` the function places the previous signal set in `oldset`

- if `set` is `NULL`, `how` is ignored: used to retrieve the current signal mask

```c
#include <signal.h>

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

On success, the call returns 0; on failure it returns -1 and sets `errno` appropriately.

Note

`SIGKILL` and `SIGSTOP` cannot be blocked; the function silently ignores attempts to block either signal.

Part I : Signal handler

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

extern const char * const sys_siglist[];

sigset_t set;

void signal_handler(int signo) {
  sigprocmask(SIG_BLOCK, &set, &oldset);
  printf("Signals blocked while handling: %s; sleeping for 10 s...\n", sys_siglist[signo]);
  sleep(10);
  printf("Ready; unblocking signals...\n");
  sigprocmask(SIG_UNBLOCK, &oldset, NULL);
}
...
```

Part II : Signal dispositions

```c
...
int main(int argc, char *argv[]) {
  sigemptyset(&set);
  sigaddset(&set, SIGALRM);
  sigaddset(&set, SIGUSR1);
  sigaddset(&set, SIGUSR2);

  if (signal(SIGUSR1, signal_handler) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR1!\n");
  if (signal(SIGUSR2, signal_handler) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGUSR2!\n");
  if (signal(SIGALRM, signal_handler) == SIG_ERR)
    fprintf(stderr, "Can't catch SIGALRM!\n");

  printf("Calling alarm(5)\n");
  alarm(5);
  printf("Calling raise(SIGUSR1)\n");
  raise(SIGUSR1);
  printf("Calling raise(SIGUSR2)\n");
  raise(SIGUSR2);
}
```

When the kernel raises a blocked signal, it is not delivered

- these signals are termed *pending*

When the kernel raises a blocked signal, it is not delivered

- these signals are termed *pending*

When a pending signal is unblocked, kernel forwards it to the process to handle

POSIX defines a function to retrieve the set of pending signals:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

Retrieving signals

POSIX defines a function to retrieve the set of pending signals:

```c
#include <signal.h>

int sigpending (sigset_t *set);
```

A successful call to `sigpending` places the set of pending signals in `set` and returns 0

- on failure, the call returns -1 and sets `errno` to `EFAULT`, signifying that `set` is an invalid pointer.

A third POSIX-defined function allows a process to temporarily change its signal mask and then wait until a signal is raised that either terminates or is handled by the process:

```
#include <signal.h>

int sigsuspend (const sigset_t *set);
```

If a signal terminates the process, `sigsuspend` does not return

If a signal is raised and handled, `sigsuspend` returns -1 after the handler returns, setting `errno` to `EINTR`

If `set` is an invalid pointer, `errno` is set to `EFAULT`

Waiting for signal sets

`sigsuspend` is used in conjunction with `sigprocmask` to prevent delivery of a signal during the execution of a critical code section

- caller first blocks the signals with `sigprocmask`
- when critical code has completed, called waits for signals by calling `sigsuspend` with the signal mask that was returned by `sigprocmask` (in the `oldset` argument).

Advanced signal management

The signal function examined so far is very basic:

- part of the standard C library
- has to reflect minimal assumptions about the capabilities of operating system
- offers a lowest common denominator to signal management

Advanced signal management

The signal function examined so far is very basic:

- part of the standard C library
- has to reflect minimal assumptions about the capabilities of operating system
- offers a lowest common denominator to signal management

POSIX standardises `sigaction`, providing much greater signal managment capabilities, e.g.:

- block reception of specified signals while handler runs
- retrieve data about system and process state when handler is raised

```c
#include <signal.h>

int sigaction (int signo, const struct sigaction *act, struct sigaction
    *oldact);
```

`sigaction`

`sigaction` changes the behaviour of the signal identified by `signo`

- can be any value except `SIGKILL` and `SIGSTOP`
- behaviour is specified by `act` (if not `NULL`)
- the call stores the previous (or current, if `act` is NULL) behaviour of the signal in `oldact`, if `oldact` is not `NULL`

sigaction

The `sigaction` structure allows for fine-grained control over signals:

```c
struct sigaction {
        void (*sa_handler)(int);   /* signal handler or action */
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;          /* signals to block */
        int sa_flags;              /* flags */
        void (*sa_restorer)(void); /* obsolete and non-POSIX */
};
```

The `sa_handler` field dictates the action to take upon receiving the signal:

- `SIG_DFL` for default action
- `SIG_IGN` instructing the kernel to ignore signal for process
- or a pointer to a signal-handling function with the same prototype as that installed by `signal`:
  ```
  void my_handler (int signo);
  ```

If SA_SIGINFO is set in sa_flags, sa_sigaction, and not sa_handler, dictates the signal-handling function:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

The signal number is received in the first parameter, a **siginfo_t** structure as the second, and a ucontext_t structure as the third.

- **siginfo_t** provides information to the signal handler

Note

On some machine architectures (and possibly other Unix systems), sa_handler and sa_sigaction are in a union, and you should not assign values to both fields.

The `sa_mask` field denotes the set of signals that the system should block while the signal handler is executing

- Enforces proper protection from reneentrancy among multiple signal handlers
- `SIGKILL` and `SIGSTOP` cannot be blocked.

The `sigaction` structure allows for fine-grained control over signals, e.g.:

- `SA_NOCLDSTOP` - if `signo` is `SIGCHLD`, instruct the system not to provide notification when a child process stops or resumes
- `SA_NOCLDWAIT` - if `signo` is `SIGCHLD`, enable automatic child reaping: children are not converted to zombies on termination, and the parent need not (and cannot) call `wait` on them
- `SA_RESETHAND` - enables "one-shot" mode; signal is reset to the default once the signal handler returns
- `SA_ONSTACK` - instructs the system to invoke the given signal handler on an alternative signal stack, as provided by `sigaltstack`.
- `SA_RESTART` - enables BSD-style restarting of system calls that are interrupted by signals

```c
#include <signal.h>

int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);
```

`sigaction` returns 0 on success; on failure it returns -1 and sets `errno` to one of the following:

- `EFAULT` `act` or `oldact` is an invalid pointer
- `EINVAL` `signo` is an invalid signal, `SIGKILL` or `SIGSTOP`

Signal handlers registered with `SA_SIGINFO` are passed a **`siginfo_t`** parameters

- contains a field named `si_value`
- an optional payload passed from the signal generator to the signal receiver

The `sigqueue` function, defined by POSIX, allows a process to send a signal with this payload:

```
#include <signal.h>

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

`sigqueue` works similarly to `kill`: on success, the signal identified by signo is queued to the process and the function returns 0

- signal payload is given by value through the union:

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

On failure, the call returns -1 and sets `errno` to the respective value

Pipes

Inter-process communication so far

Inter-process communication (IPC) allows different processes to communicate among themselves.

Inter-process communication so far

Inter-process communication (IPC) allows different processes to communicate among themselves.

So far, process have been able to communicate using:

- parent-child inheritance through `fork`
- passing arguments in `exec` calls
- shared memory segments
- signals and signal payloads

Inter-process communication so far

Inter-process communication (IPC) allows different processes to communicate among themselves.

So far, process have been able to communicate using:

- parent-child inheritance through `fork`
- passing arguments in `exec` calls
- shared memory segments
- signals and signal payloads

Pipes are the oldest and most widely implemented form of IPC

Pipes

Pipes are the oldest and most widely implemented form of IPC; they have two limitations:

- data can flow in one direction (half-duplex)
- pipes can be used between processes that have a common ancestor (e.g., parent and child after `fork`)

Note

Some systems provide full-duplex pipes, but for portability one should never assume this is the case.

Pipes

Despite these limitations, pipes are still the most commonly used form of IPC

- shell processing of commands in a pipeline (e.g. ls -la | grep user | sort)
    - shell creates a number of child processes
    - links the standard output of one process to the standard input of the next through a pipe

Pipes

A pipe is created by calling the `pipe` function:

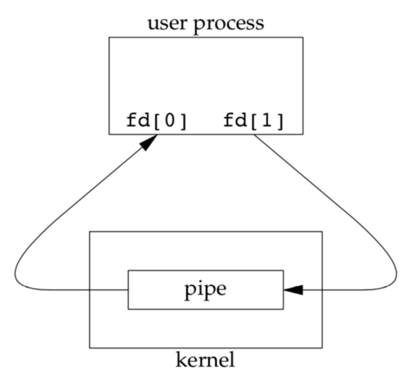
```
#include <unistd.h>

int pipe(int fd[2]);
```

On success the function returns 0

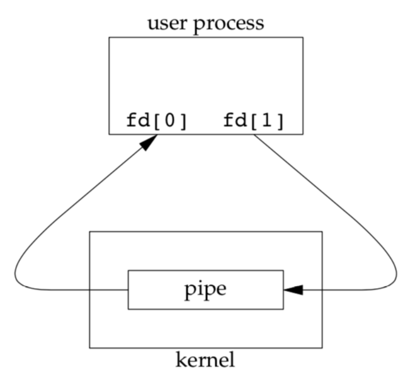
- two file descriptors are returned through the `fd` argument: `fd[0]` is open for reading and `fd[1]` is open for writing;
- `fd[0]` and `fd[1]` are thus the two ends of the pipe: the output of `fd[1]` is the input for `fd[0]`.

On error, the function returns -1

The result of a call to `pipe` can be visualised as follows:



The result of a call to `pipe` can be visualised as follows:

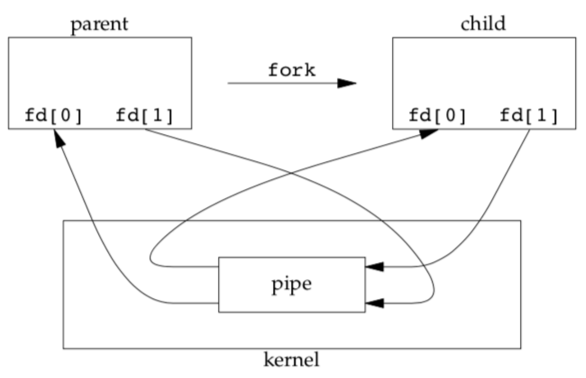


Note

A `pipe` in a single process is next to useless.

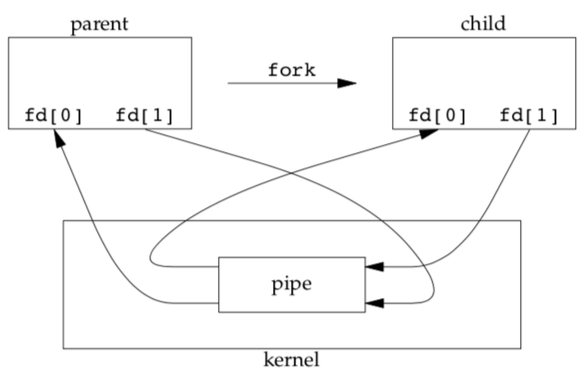
Pipes

Normally, the process that calls `pipe` follows with a call to `fork`, creating an IPC channel from the parent to the child or vice versa:



Pipes

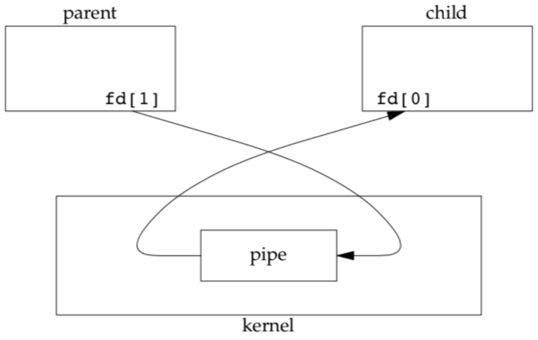
Normally, the process that calls `pipe` follows with a call to `fork`, creating an IPC channel from the parent to the child or vice versa:



What happens after the `fork` depends on the desired data flow direction.

The desired direction of data flow dictates the actions of the processing sharing the pipe:

- for a pipe from the parent to the child, the parent closes the read end of the pipe `fd[0]` and the child closes the write end of the pipe `fd[1]`



Pipes

When one end of a pipe is closed, two rules apply:

Pipes

When one end of a pipe is closed, two rules apply:

1. for a pipe whose write end has been closed, a `read` returning 0 indicates the end of file (all data has been read / no more writers for the pipe);

Pipes

When one end of a pipe is closed, two rules apply:

1. for a pipe whose write end has been closed, a `read` returning 0 indicates the end of file (all data has been read / no more writers for the pipe);
2. for a pipe whose read end has been closed, a `write` raises `SIGPIPE`; if the signal is ignored or caught, write returns -1 with `errno` set to `EPIPE`.

When one end of a pipe is closed, two rules apply:

1. for a pipe whose write end has been closed, a `read` returning 0 indicates the end of file (all data has been read / no more writers for the pipe);
2. for a pipe whose read end has been closed, a `write` raises `SIGPIPE`; if the signal is ignored or caught, write returns -1 with `errno` set to `EPIPE`.

Note

The constant `PIPE_BUF` specifies the kernel's pipe buffer size. Writes of `PIPE_BUF` bytes or less will not be interleaved with writes from other processes to the same pipe.

Pipes Example

Open pipe between parent and child; parent sends string to child who prints it:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  pid_t pid;
  int fd[2];

  if (pipe(fd) < 0) {
    perror("pipe() failed");
    exit(EXIT_FAILURE);
  }

  if ((pid = fork()) < 0) {
    perror("fork() failed");
    exit(EXIT_FAILURE);
  } else if (pid > 0) { // parent
    char str[] = "Hello from your parent!\n";
    close(fd[0]); // close read end
    write(fd[1], str, sizeof(str)); // write end
  } else { // child
    char str[256];
    close(fd[1]); // close write end
    int bytes_read = read(fd[0], str, sizeof(str)); // read end
    write(STDOUT_FILENO, str, bytes_read);
  }
}
```

Pipes Example

The parent process first creates a pipe, then forks;

- the created pipe is inherited by the child
- following the fork, parent closes read end and child closes write end
- parent `write`s a string to pipe write end (pipe has a file descriptor and can be operated upon as if it were a file)
- child `read`s data from pipe read end and prints it out

The parent process first creates a pipe, then forks;

- the created pipe is inherited by the child
- following the fork, parent closes read end and child closes write end
- parent `write`s a string to pipe write end (pipe has a file descriptor and can be operated upon as if it were a file)
- child `read`s data from pipe read end and prints it out

```
$ gcc -o pipe ex_pipe.c
$ ./pipe
Hello from your parent!
$
```

Pipes Example

In the example, `read` and `write` were called directly on the pipe descriptors

- it is more interesting to duplicate the pipe descriptors onto standard input or standard output
- when the child runs some other program, that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

In the example, `read` and `write` were called directly on the pipe descriptors

- it is more interesting to duplicate the pipe descriptors onto standard input or standard output
- when the child runs some other program, that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

We can change the example above to wire the pipe read end to standard input and execute `cat` to output the message.

Pipes Example

The snippet below illustrates how the pipe can be duplicated onto standard input (using `dup2`) before executing `cat`:

```
if (pid > 0) { // after fork
  char str[] = "Hello from your parent!\n";
  close(fd[0]);
  // duplicate pipe write end file descriptor on stdout
  if (dup2(fd[1], STDOUT_FILENO) == 1) {
    perror ("dup2() failed);
    exit (EXIT_FAILURE);
  }
  printf("%s\n", str);
} else {
  char *args[] = {"cat", NULL};
  close(fd[1]);
  // duplicate pipe read end file descriptor on stdin
  if (dup2(fd[0], STDIN_FILENO) == -1) {
    perror ("dup2() failed);
    exit (EXIT_FAILURE);
  }
  execvp(args[0], args); // error checking omitted
}
```

Pipes Example

The snippet below illustrates how the pipe can be duplicated onto standard input (using `dup2`) before executing `cat`:

```
if (pid > 0) { // after fork
  char str[] = "Hello from your parent!\n";
  close(fd[0]);
  // duplicate pipe write end file descriptor on stdout
  if (dup2(fd[1], STDOUT_FILENO) == 1) {
    perror ("dup2() failed);
    exit (EXIT_FAILURE);
  }
  printf("%s\n", str);
} else {
  char *args[] = {"cat", NULL};
  close(fd[1]);
  // duplicate pipe read end file descriptor on stdin
  if (dup2(fd[0], STDIN_FILENO) == -1) {
    perror ("dup2() failed);
    exit (EXIT_FAILURE);
  }
  execvp(args[0], args); // error checking omitted
}
```

Exercise

Change the program to send all input arguments (`argv`) over the pipe for `cat` to print.

Dup

What does the dup function do exactly?

Dup

What does the **dup** function do exactly?

For a start, there are two variants of **dup** which are standardised:

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Dup

What does the **dup** function do exactly?

For a start, there are two variants of **dup** which are standardised:

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup creates a copy of the file descriptor specified by `oldfd`

- uses the lowest-numbered unused file descriptor for copy
- the new and old descriptors may be used interchangeably

Dup

What does the dup function do exactly?

For a start, there are two variants of dup which are standardised:

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup creates a copy of the file descriptor specified by oldfd

- uses the lowest-numbered unused file descriptor for copy
- the new and old descriptors may be used interchangeably

dup2 performs the same task as dup, but instead of using the lowest-numbered unused file descriptor, it uses newfd

- if previously open, newfd is silently closed before being reused

Dup

What does the `dup` function do exactly?

For a start, there are two variants of `dup` which are standardised:

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup` creates a copy of the file descriptor specified by `oldfd`

- uses the lowest-numbered unused file descriptor for copy
- the new and old descriptors may be used interchangeably

`dup2` performs the same task as `dup`, but instead of using the lowest-numbered unused file descriptor, it uses `newfd`

- if previously open, `newfd` is silently closed before being reused

On success, they return the new file descriptor; on error, -1 is returned, and `errno` is set appropriately.

`dup2`, `pipe`, `fork` and `exec` can be used to construct arbitrary command pipelines, similar to those employed by the shell

Command pipelines

`dup2`, `pipe`, `fork` and `exec` can be used to construct arbitrary command pipelines, similar to those employed by the shell

When implementing such a pipeline consider that:

- parent process forks new child for each pipeline stage
- new pipe created for each producer-consumer pair
- write end of pipe wired to the producer (closes read end)
- read end of pipe wired to the consumer (closes write end)
- stages that are both consumers and producers wired to two different pipes (read end with previous stage, write end with next stage)
- parent process closes both ends of each created pipe (after child forks)

Let's look at an example where a parent process (or shell) needs to execute the following command pipeline:

```
$ ps -eaf | more
```

```
$ ps -eaf | more
```

By the guidelines above:

- parent process has to spawn two children (fork twice): `ps -eaf` and `more`
- a single pipe is created, shared by one producer-consumer pair `ps -eaf => more`
- write end of pipe is wired to `ps -eaf`, the producer
- read end of pipe is wired to `more`, the consumer
- parent process closes both ends of pipe after children have been forked
    - failure to do so will leave the pipe open beyond the expected end-of-file, preventing the pipeline processes from terminating

Part I: initialisation and pipe creation

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  pid_t pid_p1, pid_p2;
  int fd[2];

  if (pipe(fd) < 0) {
    perror("pipe() failed");
    exit(EXIT_FAILURE);
  }

  ...
```

Part II: fork first stage and wire pipe write end to standard output

```c
...
if ((pid_p1 = fork()) < 0) {
  perror("fork() failed");
  exit(EXIT_FAILURE);
} else if (pid_p1 == 0) {
  char *args[] = {"ps", "-eaf", NULL};

  close(fd[0]);
  dup2(fd[1], STDOUT_FILENO);

  if (execvp(args[0], args) == -1)
  {
    perror("execvp() failed");
    exit(EXIT_FAILURE);
  }
}
...
```

Command pipeline example

Part III: fork second stage and wire pipe read end to standard input

```
...
if ((pid_p2 = fork()) < 0) {
  perror("fork() failed");
  exit(EXIT_FAILURE);
} else if (pid_p2 == 0) {
  char *args[] = {"more", NULL};

  close(fd[1]);
  dup2(fd[0], STDIN_FILENO);

  if (execvp(args[0], args) == -1)
  {
    perror("execvp() failed");
    exit(EXIT_FAILURE);
  }
}
...
```

Command pipeline example

Part IV: close pipes in parent and wait for last stage of pipeline to terminate

```c
  ...
  // parent process closes both pipe ends
  close(fd[0]);
  close(fd[1]);

  // wait for termination of last pipeline stage
  int status;
  waitpid(pid_p2, &status, 0);

  printf("Pipeline execution complete. \n");
}
```

Compiling and running the pipeline example yields the following output:

```
$ gcc -o pipeline ./ex_pipeline.c
$ ./pipeline
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 May02 ?        00:00:22 /sbin/init auto
root           2       0  0 May02 ?        00:00:00 [kthreadd]
root           4       2  0 May02 ?        00:00:00 [kworker/0:0H]
root           6       2  0 May02 ?        00:00:00 [mm_percpu_wq]
root           7       2  0 May02 ?        00:00:08 [ksoftirqd/0]
root           8       2  0 May02 ?        00:02:00 [rcu_sched]
root           9       2  0 May02 ?        00:00:00 [rcu_bh]
root          10       2  0 May02 ?        00:00:00 [migration/0]
root          11       2  0 May02 ?        00:00:00 [watchdog/0]
root          12       2  0 May02 ?        00:00:00 [cpuhp/0]
root          13       2  0 May02 ?        00:00:00 [cpuhp/1]
root          14       2  0 May02 ?        00:00:00 [watchdog/1]
root          15       2  0 May02 ?        00:00:00 [migration/1]
root          16       2  0 May02 ?        00:00:17 [ksoftirqd/1]
--More--
```

Questions?