Operating systems

Interprocess communication (IPC)
Part 3 of 3: Signal, PIPE and FIFO

Enrico Fraccaroli

enrico.fraccaroli@univr.it

Table of Contents

Signals

Signals

Fundamental concepts

Fundamental concepts (1/2)

A *signal* is a notification to a process that an event has occurred. They interrupt the normal flow of execution of a program; in most cases, it is not possible to predict exactly when a signal will arrive.

A signal is said to be *generated* by some event. Once generated, a signal is later *delivered* to a process. Between the time it is generated and the time it is delivered, a signal is said to be *pending*.

Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or *immediately* if the process is already running.

Fundamental concepts (2/2)

Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal:

- The process is terminated (*killed*).
- The process is suspended (*stopped*).
- The process is *resumed* after previously being stopped.
- The signal is ignored. It is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
- The process executes a *signal handler*, namely a function written by the programmer that performs appropriate tasks in response to the delivery of a signal.

Signals

Signal types

Signal Types and Default Actions (1/3)

Signals to terminate a process:

- **SIGTERM** is delivered to safely terminate a process. A well-designed application should have a handler for SIGTERM that causes the application to exit gracefully.
- **SIGINT** terminates a process (“interrupt process"). It is sent when the user type Contr-C character.
- **SIGQUIT** terminates a process and causes it to produce a core dump, which can then be used for debugging.
- **SIGKILL** terminates a process (always!). It can’t be blocked, ignored, or caught by a handler.

Signals to stop and resume a process:

- **SIGSTOP** stops a process (always!). It can’t be blocked, ignored, or caught by a handler.
- **SIGCONT** resumes a previously stopped process.

Signal Types and Default Actions (2/3)

Other import signals:

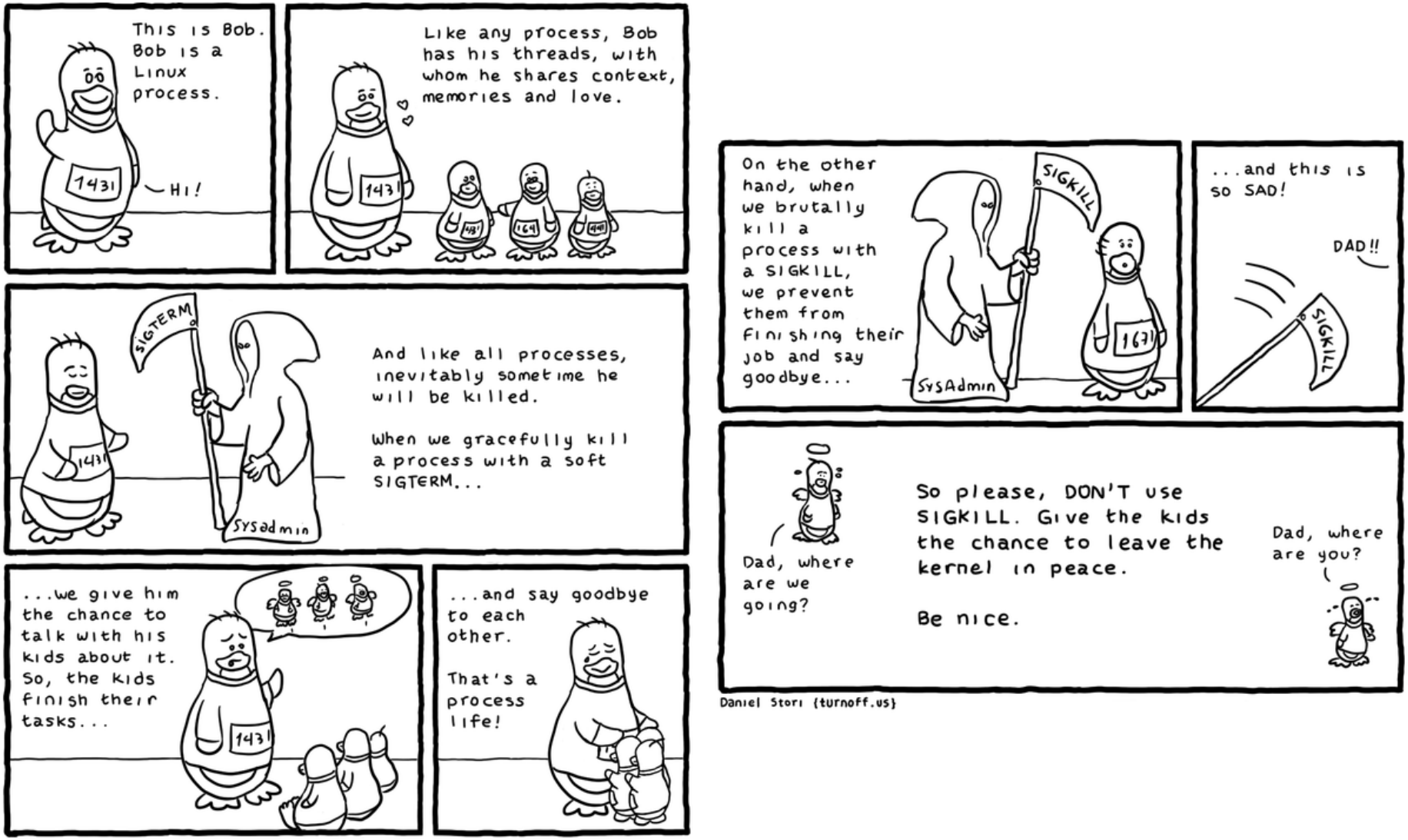
- **SIGPIPE** is generated when a process tries to write to a PIPE, a FIFO for which there is no corresponding reader process (see chapter PIPE/FIFO).
- **SIGALRM** is delivered to a process upon the expiration of a real-time timer set by a call to *alarm* (see next slides).
- **SIGUSR1** and **SIGUSR2** are available for programmer-defined purposes. The kernel never generates these signals for a process.

The complete list of available signals in Linux can be retrieved with the bash command “man 7 signal”.

Signal Types and Default Actions (3/3)

name	number	can be caught?	default action
SIGTERM	15	yes	terminates a process
SIGINT	2	yes	terminates a process
SIGQUIT	3	yes	dumps + terms a process
SIGKILL	9	no	terminates a process
SIGSTOP	17	no	stops a process
SIGCONT	19	yes	resumes a stopped process
SIGPIPE	13	yes	terminates a process
SIGALRM	14	yes	terminates a process
SIGUSR1	30	yes	terminates a process
SIGUSR2	31	yes	terminates a process

Column "number" reports the signal number for x86 and arm architecture. A signal may have a different number in other architectures



Signals

Signal handler

Signal handler

A signal handler (also called a signal catcher) is a function that is called when a specified signal is delivered to a process. It has always the following general form:

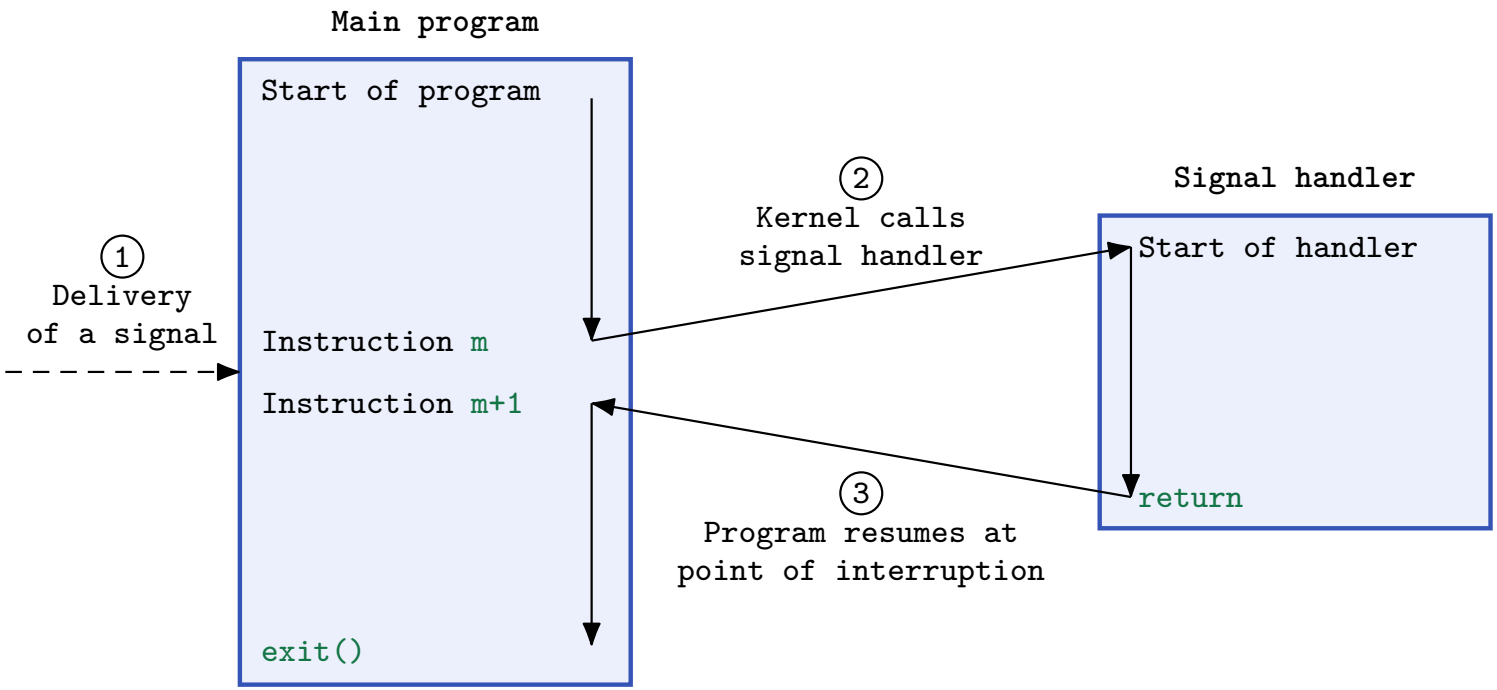
```
void sigHandler(int sig) {
    /* Code for the handler */
}
```

This function returns nothing (void) and takes one integer argument (*sig*). When the signal handler is invoked by the kernel, *sig* is set to the signal number delivered to the process.

Typically, *sig* is used to determine which signal caused the handler to be invoked when a same handler catches different types of signals.

Signal handler (execution)

Invocation of a signal handler may interrupt the main program flow at any time. The kernel calls the signal handler, and when the handler returns, execution of the program resumes at the point where the handler interrupted it.



Changing signal dispositions (1/2)

The *signal* system call changes the default signal-handler for a defined signal in a process.

```
#include <signal.h>

typedef void (*sighandler_t)(int);
// Returns previous signal disposition on success, or SIG_ERR on error
sighandler_t signal(int signum, sighandler_t handler);
```

signum identifies the signal whose disposition we wish to change in the process. *handler* can be one of the following:

- the address of a user-defined signal handler.
- the constant SIG_DFL, which resets the default disposition of the process for the signal *signum*.
- the constant SIG_IGN, which sets the process to ignore the delivery of the signal *signum*.

Changing signal dispositions (2/2)

```c
void sigHandler(int sig) {
    printf("The signal %s was caught!\n",
        (sig == SIGINT)? "Ctrl-C" : "signal User-1");
}
int main (int argc, char *argv[]) {
    // setting sigHandler to be executed for SIGINT or SIGUSR1
    if (signal(SIGINT,  sigHandler) == SIG_ERR ||
        signal(SIGUSR1, sigHandler) == SIG_ERR) {
        errExit("change signal handler failed");
    }
    // Do something else here. During this time, if SIGINT/SIGUSR1
    // is delivered, sigHandler will be used to handle the signal.
    // Reset the default process disposition for SIGINT and SIGUSR1
    if (signal(SIGINT, SIG_DFL) == SIG_ERR ||
        signal(SIGUSR1, SIG_DFL) == SIG_ERR) {
        errExit("reset signal handler failed");
    }
    return 0;
}
```

Signal handler (important notes)

What you should keep in mind when you use signal handlers:

- SIGKILL and SIGSTOP cannot be caught.
- A signal is an asynchronous event. We cannot predict when it arrives.
- When a signal handler is invoked, the signal that caused its invocation is automatically blocked. It is unblocked when the signal handler returns to the normal execution flow of the program.
- If a blocked signal is generated several times, when unblocked, it is delivered to the process only once!
- The execution of a signal handler can be interrupted by the delivery of an unblocked signal.
- The signal dispositions are inherited between process *parent* and process *child*.

Waiting for a signal (1/2)

Calling *pause* suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).

```c
#include <unistd.h>
// Always return -1 with errno set to EINTR
int pause();
```

The *sleep* function suspends execution of the calling process for the number of seconds specified in the seconds argument or until a signal is caught (thus interrupting the call).

```c
#include <unistd.h>
// Returns 0 on normal completion, or number of
//  unslept seconds if prematurely terminated
unsigned int sleep(unsigned int seconds);
```

Waiting for a signal (2/2)

Waiting the interrupt signal (Ctrl-C), which must occur within 30 seconds

```c
void sigHandler(int sig) { printf("Well done!\n"); }

int main (int argc, char *argv[]) {
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("change signal handler failed");

    int time = 30;
    printf("We can wait for %d seconds!\n", time);
    time = sleep(time); // the process is suspended for max. 30sec.
    printf("%s!\n", (time==0)? "out of time", "just in time");
}
```

Signals

Sending a signal

Sending a signal (kill) (1/4)

The system call *kill* let a process send a signal to another process.

```c
#include <signal.h>

// Returns 0 on success, or -1 on error
int kill(pid_t pid, int sig);
```

The *pid* argument identifies one or more processes to which the signal specified by *sig* is to be sent.

- ($pid > 0$): the signal is sent to the process having PID equals to *pid*.
- ($pid = 0$): the signal is sent to every process in the same process group as the calling process, including the calling process itself.
- ($pid < 0$): the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- ($pid =$ -1): the signal is sent to every process for which the calling process has permission to send a signal, except init and the process itself.

Sending a signal (kill) (2/4)

Sending a SIGKILL signal to a child process

```
int main (int argc, char *argv[]) {
    pid_t child = fork();
    switch(child) {
    case -1:
        errExit("fork");
    case 0:   /* Child process */
        while(1); // <- child is stuck here!
    default:  /* Parent process */
        sleep(10);                // wait 10 seconds
        kill(child, SIGKILL); // kill the child process
    }
    return 0;
}
```

Sending a signal (alarm) (3/4)

The *alarm* system call arranges for a SIGALRM signal to be delivered to the calling process after a fixed delay.

```
#include <signal.h>

// Always succeeds, returning number of seconds remaining on
// any previously set timer, or 0 if no timer previously was set
unsigned int alarm(unsigned int seconds);
```

- The *seconds* argument specifies the number of seconds in the future when the timer is to expire. At that time, a SIGALRM signal is delivered to the calling process.
- Setting a timer with *alarm* overrides any previously set timer.

Sending a signal (alarm) (4/4)

Setting a timer with the *alarm* system call.

```c
void sigHandler(int sig) { printf("Out of time!\n"); _exit(0); }

int main (int argc, char *argv[]) {
    if (signal(SIGALRM, sigHandler) == SIG_ERR)
        errExit("change signal handler failed");

    int time = 30;
    printf("We have %d seconds to complete the job!\n", time);
    alarm(time); // setting a timer

    /* Do something else here. */

    time = alarm(0); // disabling timer
    printf("%s seconds before timer expirations!\n", time);
    return 0;
}
```

Signals

Setting and blocking a signal

Signal set (1/2)

The *sigset_t* data type represents a signal set. The functions *sigemptyset* and *sigfillset* must be used to initialize a signal set, before using it in any other way.

```
#include <signal.h>

typedef unsigned long sigset_t;

// Both return 0 on success, or -1 on error.
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

sigemptyset initializes a signal set to contain no signal.
sigfillset initializes a set to contain all signals.

Signal set (2/2)

After initialization, individual signals can be added to a set using *sigaddset* and removed using *sigdelset*.

```c
#include <signal.h>

// Both return 0 on success, or -1 on error
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

For both *sigaddset* and *sigdelset*, the *sig* argument is a signal number.

The *sigismember* function is used to test for membership of a set.

```c
#include <signal.h>

// Returns 1 if sig is a member of set, otherwise 0
int sigismember(const sigset_t *set, int sig);
```

Blocking signal delivery (1/3)

For each process, the kernel maintains a signal mask, namely a set of signals whose delivery to the process is currently blocked. If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.

The *sigprocmask* system call can be used at any time to explicitly add signals to, and remove signals from, the signal mask.

```
#include <signal.h>

// Returns 0 on success, or -1 on error
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Blocking signal delivery (2/3)

The *how* argument determines the changes that *sigprocmask* makes to the signal mask:

- SIG_BLOCK The set of blocked signals is the union of the current set and the *set* argument.
- SIG_UNBLOCK The signals in *set* argument are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- SIG_SETMASK The set of blocked signals is set to the argument *set*.

In each case, if the *oldset* argument is not NULL, it points to a *sigset_t* buffer that is used to return the previous signal mask.
If we want to retrieve the signal mask without changing it, then we can specify NULL for the *set* argument, in which case the *how* argument is ignored.

Blocking signal delivery (3/3)

Blocking all signals but SIGTERM.

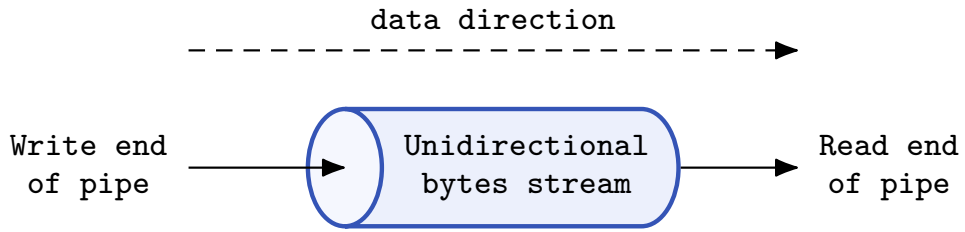
```
int main (int argc, char *argv[]) {
    sigset_t mySet, prevSet;
    // initialize mySet to contain all signals
    sigfillset(&mySet);
    // remove SIGTERM from mySet
    sigdelset(&mySet, SIGTERM);
    // blocking all signals but SIGTERM
    sigprocmask(SIG_SETMASK, &mySet, &prevSet);

    /* Code that shouldn't be interrupted by signals but SIGTERM */

    // reset the signal mask of the process
    sigprocmask(SIG_SETMASK, &prevSet, NULL);
    // if SIGTERM is pending, only at this point it is
    // delivered to the process
    return 0;
}
```

PIPEs

PIPEs

Fundamental concepts

Fundamental concepts (1/2)



A *PIPE* is a byte stream (technically speaking, it is a buffer in kernel memory), which allows processes to exchange bytes.
A *PIPE* has the following properties:

- it is unidirectional. Data travels only in one direction through a PIPE. One end of the PIPE is used for writing, the other one for reading;
- data passes through the PIPE sequentially. Bytes are read from a PIPE in exactly the order they were written;
- no concept of messages, or message boundaries. The process reading from a PIPE can read blocks of data of any size, regardless of the size of blocks written by the writing process.

Fundamental concepts (2/2)

- Attempts to read from an empty PIPE blocks the reader until, either at least one byte has been written to the PIPE, or a no-terminating signal occurs (errno EINTR).
- If the write-end of a PIPE is closed, then a process reading from the PIPE will see end-of-file once it has read all remaining data in the PIPE.
- A write is blocked until, either sufficient space is available to complete the operation atomically[1], or a no-terminating signal occurs (errno EINTR).
- Writes of data blocks larger than PIPE_BUF[2] bytes may be broken into segments of arbitrary size (which may be smaller than PIPE_BUF bytes).

[1] On Linux, pipe capacity is 65536 bytes

[2] On Linux, PIPE_BUF has the value 4096 bytes

PIPEs

Creating and using PIPEs

Creating and using PIPEs (1/3)

The *pipe* system call creates a new PIPE.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error
int pipe(int filedes[2]);
```

A successful call to *pipe* returns two open file descriptors in the array *filedes*.

- filedes[0] stores the *read-end* of the PIPE.
- filedes[1] stores the *write-end* of the PIPE.

As with any file descriptor, we can use the *read* and *write* system calls to perform I/O on the PIPE.

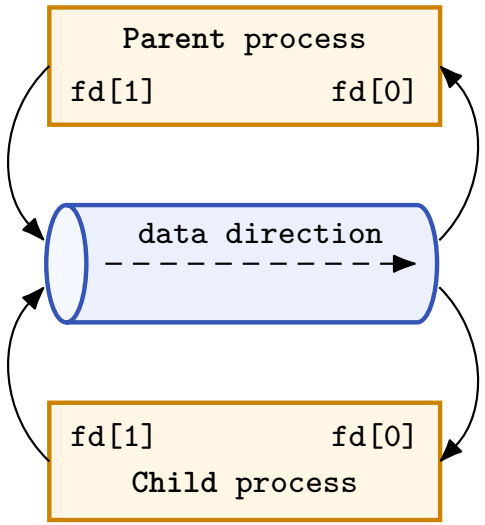
Normally, we use a PIPE to allow communication among related processes. To connect two processes using a PIPE, we follow the *pipe* call with a call to *fork*.

Creating and using PIPEs (2/3)

```c
int fd[2];
// checking if PIPE successed
if (pipe(fd) == -1)
    errExit("PIPE");
// Create a child process
switch(fork()) {
    case -1:
        errExit("fork");
    case 0:  // Child
        //...child reads from PIPE
        // (next slide)
        break;
    default: // Parent
        //...parent writes to PIPE
        // (next slide)
        break;
}
```

1. *pipe(...)* creates a new PIPE.
   fd[0] is the read-end of the PIPE.
   fd[1] is the write-end of the PIPE.

2. *fork()* creates a child process, which inherits the file descriptor table of the parent process.



Creating and using PIPEs (3/3)

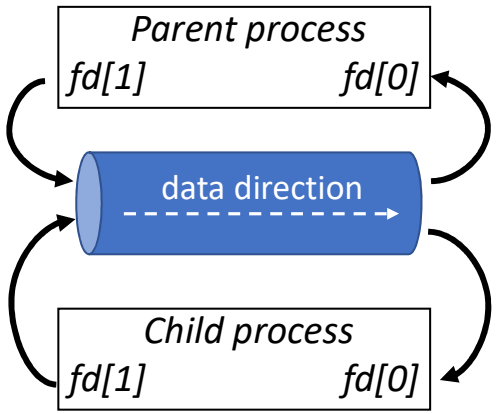
case 0: // child reads from PIPE

```
char buf[SIZE];
ssize_t nBys;

// close unused write-end
if (close(fd[1]) == -1)
    errExit("close - child");
// reading from the PIPE
nBys = read(fd[0], buf, SIZE);
// 0: end-of-file, -1: failure
if (nBys > 0) {
    buf[nBys] = '\0';
    printf("%s\n", buf);
}
// close read-end of PIPE
if (close(fd[0]) == -1)
    errExit("close - child");
```

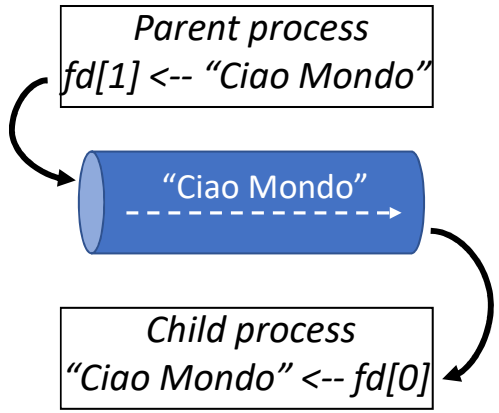
default: // parent writes to PIPE

```
char buf[] = "Ciao Mondo\n";
ssize_t nBys;

// close unused read-end
if (close(fd[0]) == -1)
    errExit("close - parent");
// write to the PIPE
nBys = write(fd[1], buf, strlen(buf));
// checkig if write successed
if (nBys != strlen(buf)) {
    errExit("write - parent");
}

// close write-end of PIPE
if (close(fd[1]) == -1)
    errExit("close - child");
```

Good and bad practice



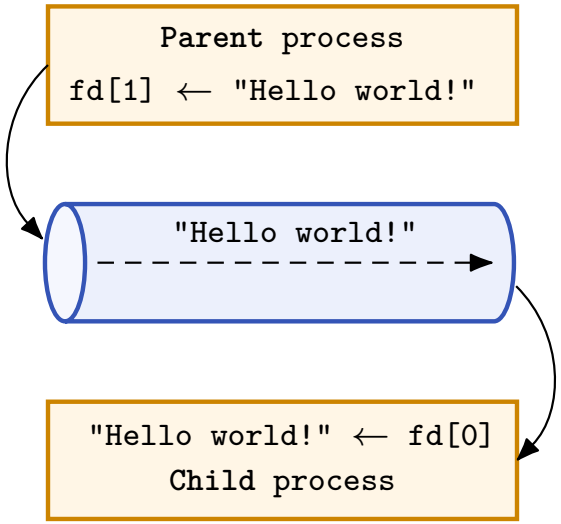
After fork()



After closing unused descriptors

Why should we close the unused PIPE file descriptor?
What problem may we have?

Good and bad practice



Why should we close the unused PIPE file descriptor?
What problem may we have?

Creating and using PIPEs - Wrong! (1/2)

case 0: // child reads from PIPE

```
// close unused write-end
//if (close(fd[1]) == -1)
//  errExit("close - child");

char buf[SIZE];
ssize_t nBys;
// reading from the PIPE
nBys = read(fd[0], buf, SIZE);
// 0: end-of-file, -1: failure
if (nBys > 0)
    printf("%s\n", buf);

// close read-end of PIPE
if (close(fd[0]) == -1)
    errExit("close - child");
```

default: // parent writes to PIPE

```
// close unused read-end
if (close(fd[0]) == -1)
    errExit("close - parent");

// ...nothing to send

// close write-end of PIPE
if (close(fd[1]) == -1)
    errExit("close - child");
```

Why is this program wrong?
Advice: the reading process is waiting for data...

Creating and using PIPEs - Wrong! (2/2)

case 0: // child reads from PIPE

```
// close unused write-end
if (close(fd[1]) == -1)
    errExit("close - child");

// ...nothing to read

// close read-end of PIPE
if (close(fd[0]) == -1)
    errExit("close - child");
```

Why is this program wrong?
Advice: Whoam the writing process is sending data to?
(SIGPIPE, errno EPIPE)

default: // parent writes to PIPE

```
// close unused read-end
//if (close(fd[0]) == -1)
//  errExit("close - parent");

char buf[] = "Ciao Mondo\n";
size_t len = strlen(buf);
// write to the PIPE
nBys = write(fd[1], buf, len);
// checkig if write successed
if (nBys != len)
    errExit("write - parent");

// close write-end of PIPE
if (close(fd[1]) == -1)
    errExit("close - child");
```

FIFOs (named PIPEs)

FIFOs (named PIPEs)

Fundamental concepts

Fundamental concepts

A *FIFO* is a byte stream (technically speaking, it is a buffer in kernel memory), which allows processes to exchange bytes.
Semantically, a *FIFO* is similar to a *PIPE*.

The principal difference between *PIPEs* and *FIFOs* is that a *FIFO* has a name within the file system, and is opened and deleted in the same way as a regular file. This allows a *FIFO* to be used for communication between unrelated processes.

Just as with PIPEs, a *FIFO* has a write-end and a read-end, and data is read from the FIFO in the same order as it is written.

FIFOs (named PIPEs)

Creating, opening, and using FIFOs

Creating a FIFO

The *mkfifo* system call creates a new *FIFO*.

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error
int mkfifo(const char *pathname, mode_t mode);
```

The *pathname* parameter specifies where the *FIFO* is created. As a normal file, the *mode* parameter specifies the permissions for the *FIFO* (see chapter file system, system call *open*).

Once a *FIFO* has been created, any process can open it.

Opening a FIFO (1/2)

The *open* system call open a *FIFO*.

```c
#include <unistd.h>

// Returns file descriptor on success, or -1 on error.
int open(const char *pathname, int flags);
```

The *pathname* parameter specifies the location of the *FIFO* in the file system. The *flags* argument is a bit mask of one of the following constants that specifies the access mode for the *FIFO*.

Flag	**Description**
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only

Opening a FIFO (2/2)

The only sensible use of a *FIFO* is to have a reading process and a writing process on each end.

By default, opening a *FIFO* for reading (O_RDONLY flag) blocks until another process opens the *FIFO* for writing (O_WRONLY flag). Conversely, opening the *FIFO* for writing blocks until another process opens the FIFO for reading. In other words, opening a *FIFO* synchronizes the reading and writing processes.

If the opposite end of a FIFO is already open (perhaps because a pair of processes have already opened each end of the FIFO), then *open* succeeds immediately.

Creating and using a FIFO

Receiver

```c
char *fname = "/tmp/myfifo";
int res = mkfifo(fname, S_IRUSR|S_IWUSR);
// Opening for reading only
int fd = open(fname, O_RDONLY);

// reading bytes from fifo
char buffer[LEN];
read(fd, buffer, LEN);

// Printing buffer on stdout
printf("%s\n", buffer);

// closing the fifo
close(fd);

// Removing FIFO
unlink(fname);
```

Sender

```c
char *fname = "/tmp/myfifo";

// Opening for wringing only
int fd = open(fname, O_WRONLY);

//reading a str. (no spaces)
char buffer[LEN];
printf("Give me a string: ");
scanf("%s", buffer);

// writing the string on fifo
write(fd, buffer, strlen(buffer));

// closing the fifo
close(fd);
```

Statements checking errors were omitted due to lack of space.