

Introduction to Linux processes

Process identifier

Every process has a unique identifier PID or Process Identifier. The PID is a non negative integer.

Although a PID is unique, UNIX reuses the numbers of terminated processes.

PID can be used by concurrent processes for creating unique objects, or temporary filenames.

For example :

```
sprintf(filename, "file-%d" getpid());
```

creates a different process-dependent filename

Some identifiers are reserved

- PID=0.
 - The `swapper` , which is responsible for memory management and process scheduling
- PID=1.
 - `init` which is a daemon
 - invoked at the end of the bootstrap
 - Becomes the parent of each orphan process, i.e., child processes whose parent process already terminated

```
#include <unistd.h>
pid_t getpid(); // Process ID
pid_t getppid(); // Parent Process ID

uid_t getuid(); // User ID
gid_t getgid(); // Group ID
```

`getpid` returns the identifier of the calling process.

`getppid` returns the identifier of the parent process.

There is no system call to obtain the PID of a child.

Associated effective

To the UID and to the GID there exist associated Effective-UID and Effective-GID. A process with a UID/GID can change its identity assuming a different EUID/EGID

Example:

The command `passwd` allows to change the password of a user; this requires root permissions; as a consequence, the process `passwd` with the UID of the user assumes the EUID of the root to perform the operation

Process creation

Windows and UNIX use different procedures.

With Windows API a process is created by means of the system call `CreateProcess` :

- In practice it executes a new process specifying the executable
- The new process is distinct from the caller

In UNIX a new process is generated by means of the system call `fork` :

- It clones the current process

fork()

System call `fork()` creates a new child process, the child is a copy of the parent excluding the PID returned by `fork` .

- The parent process receives the child PID
 - A process may have more than one child that can identify on the basis of its PID

- The child process receives the value 0
 - It can identify its parent by means of the system call `getppid`

`fork` is issued one in the parent process, but returns in two different processes, and returns different values to the parent, and to the child.

```
#include <unistd.h>
pid_t fork (void);
```

returns :

- with out error:
 - Child PID in the parent process
 - Zero in the child process
- With error:
 - -1 (usually occurs because a limit on the number of allowed process has been reached)



Code is equal

Value of the data at the time of the fork is equal.

PCB is different, Process Control Block (a data structure which holds all information about a process).

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(){
    printf("P PID: %d PPID: %d\n", getpid(), getppid());
    fork();
    printf("F PID: %d PPID: %d\n", getpid(), getppid());
    return 0;
}
```

```
→ OPERATING SYSTEMS ./a.out
P PID: 3010085 PPID: 3004901
F PID: 3010085 PPID: 3004901
F PID: 3010086 PPID: 3010085
```

Before the fork i have P, after the fork i have two processes the og process is the first F, and the child process is teh second F. Fork returns two different values, and prints two times, since it makes copy of the C code and runs it with the child values. If you place a sleep:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(){
    printf("P PID: %d PPID: %d\n", getpid(), getppid());
    int p=fork();
    if(p==0)sleep(10); //this sleep is only for child
    printf("F PID: %d PPID: %d\n", getpid(), getppid());
    return 0;
}
```

```
→ OPERATING SYSTEMS ./a.out
P PID: 1961395 PPID: 3004901
```

```
F PID: 1961395 PPID: 3004901
→ OPERATING SYSTEMS F PID: 1961396 PPID: 1870
```

The terminal takes you out of the C program, since the parent doesn't have sleep active, but when the fork goes into the child, it sees sleep for 10 sec and then prints.

So we see the terminal exit, and then we get a print from the "hidden" child C program.

So since the parent terminated before the child, its child got adopted by daemon (systemd) which is father of all orphans.

Resources

The child process is a new entry in the Process table, but the resource can be:

- completely shared among parent and children (same address space)
- Partially shared (address spaces partially overlapped)
- Non shared (separate address spaces)

In UNIX/Linux parent and child share

- The source code (C)
- The open file descriptors (File Description Table)
 - In particular, stdin, stdout, and stderr
 - Concurrent I/O operation implies producing interlaced I/O
- User ID (UID), Group ID (GID), etc.
- The root and the working directory
- System resources and their utilization limits
- Signal Table
- Etc.

In UNIX/Linux parent and child have different

- Return fork value
- PID
 - The parent keeps its PID
 - The child gets a new PID
- Data, heap and stack space
 - The initial value of the variables is inherited, but the spaces are completely separated
 - copy-on-write technique is used by modern OSs
 - New memory is allocated only when one of the processes changes the content of a variable

Process termination

Five standard methods for process termination

- `return` from `main()`
- `exit` system call
- `_exit` or `_Exit`
 - Synonyms defined in ISO C or POSIX
 - Similar effects of `exit`, but different management of stdio flushing etc.
- `return` from `main()` of the last process thread
- `pthread_exit` from the last process thread

Three non-normal method for process termination

- Call of the function `abort`
 - Generates the signal `SIGABORT`, this is a sub-case of the next because a signal is generated
- If a termination signal, or a signal not caught is received
- If the last thread of a process is cancelled

System call `wait ()` and `waitpid ()`

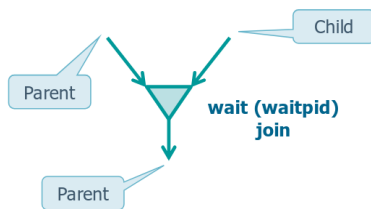
When a process terminates (normally or not)

The kernel sends a signal (SIGCHLD) to its parent

- For the parent this is an asynchronous event
- The parent process may
 - Manage the child termination (and/or the signal)
 - Asynchronously
 - Synchronously
 - Ignore the event (default)

A parent process can manage child termination

- Asynchronously: using a signal handler for signal SIGCHLD
 - This approach will be introduced in the section devoted to signals
- Synchronously: by means of system calls
 - wait
 - waitpid



wait()

```
#include <sys/wait.h>

pid_t wait (int *statLoc);
```

statLoc : is an integer pointer, if not NULL collects the exit value of the child. The status information are implementation dependent.

return :

- The PID of a terminated child on success
- -1 on error

A call of the system call wait by means of a process:

- Returns an error if the calling process doesn't have children, this is because a process without children is not supposed to do a wait.
- Blocks the calling process if all its children are running (none are already terminated)
 - wait will return as soon as one of its children terminates
- Returns to the process (immediately) the termination status of a child, if at least one of the children has ended (and it is waiting for his termination status to be recovered)
 - When a process ends and the parent does not do a wait, its termination status remains pending
 - Some resources associated with the process remain blocked

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

int main(){
    printf("P PID: %d PPID: %d\n", getpid(), getppid());
    int p=fork();
    if(p==0)sleep(10); //this sleep is only for child
    printf("F PID: %d PPID: %d\n", getpid(), getppid());
    wait(&p);
```

```
    return 0;
}
```

```
→ OPERATING SYSTEMS ./a.out
P PID: 2856215 PPID: 2855711
F PID: 2856215 PPID: 2855711
//here it waited for 10 seconds
F PID: 2856216 PPID: 2856215
```

Without the wait the sleep is done after exiting parent:

```
→ OPERATING SYSTEMS ./a.out
P PID: 1961395 PPID: 3004901
F PID: 1961395 PPID: 3004901
→ OPERATING SYSTEMS F PID: 1961396 PPID: 1870
```

`WIFEXITED(statLoc)` is true if `wait` terminates correctly. In this case `WEXITSTATUS(statLoc)` catches the 8 LSBs of the parameter passed to a `exit` (`_exit` or `_Exit`).

Zombie processes

A child process terminated, whose parent is running, but has not executed `wait` is in the zombie state:

- The data segment of the process remains in the process table because the parent could need the child exit status
- The child entry is removed only when the parent executes `wait`
- If the parent process terminates (without executing `wait`, and the child is still running, the latter is inherited by the `init` process (PID=1). The child does not become a zombie because the system knows that no one is waiting for its exit status.

Orphan processes

If the parent terminates before executing `wait`, the child process:

- Becomes an orphan
- The orphan processes, in order not to remain in this state, are inherited by the `init` process (the one with PID=1) or by a user custom `init` process
- Orphan processes and processes inherited by `init` will no longer become zombie processes

waitpid ()

If a parent needs to `wait` a specific child it is better to use `waitpid`, which

- suspends execution of the calling process until a child, specified by `pid` argument, has changed state
- `waitpid()` has a non-blocking form (not default)

```
#include <sys/wait.h>
pid_t waitpid ( pid_t pid, int *statLoc, int options);
```

`pid` allows waiting for:

- Any child (`waitpid==wait`) (`pid = -1`)
- The child whose `PID==pid` (`pid > 0`)
- Any child whose `GID` is equal to that of the calling process (`pid = 0`)
- Any child whose `GID==abs(pid)` (`pid < -1`)

options :

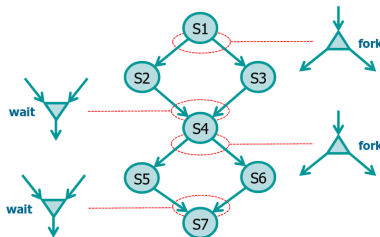
Uno delle preoccupazioni che ho e' di deluderti, e per qualche motivo invece di spronarmi a fare di piu, ultimamente mi mette solo in crisi.

Pero ora mi sento a disagio nel scriverti ste cose perche son sicuro che stai giudicando pure questo.

Non mi va bene che non mi parli dei tuoi problemi e le tue preoccupazioni, se te non inizi a farlo, smetto io.

Option	Description
0	Default behavior — block until a child terminates
WNOHANG	Don't block if no child has exited yet — return 0 immediately
WUNTRACED	Report if a child has stopped (but not terminated) due to a signal (e.g. SIGSTOP)
WCONTINUED	Report if a previously stopped child has continued due to SIGCONT
WNOWAIT (POSIX.1-2001)	Keep the child's state info — don't remove it from the process table (can wait for it again)

❖ Implement this Precedence Graph (PG) by means of the system calls **fork** and **wait**



Theoretical Aspects

Definitions algorithm and program

Algorithm: a logical procedure that in a finite number of steps solves a problem

Program: formal expression of an algorithm by means of a programming language

Process

A sequence of operations performed by a program in execution on a given set of input data.

- Dynamic entity
- Program in execution (running)
 - Text area (executable code)
 - Data area (glob. var)
 - Stack (func params and local var)
 - Heap (dynamic variable allocated during execution)
 - Registers (Program counter, stack pointer, etc.)

PCB

Process Control Block

The kernel stores for each process a set of data

- the process state
- copy of the CPU registers
 - Their number and type is hardware-dependent
- The program counter (Address of the next instruction to be executed)
- Data useful for CPU scheduling
 - Priority, pointers to queues
- Data useful for memory management
 - Base register, limit register, segment and paging registers
- Various administration data
 - CPU usage, limits, etc.
- I/O status information

- I/O device list, open files, etc.

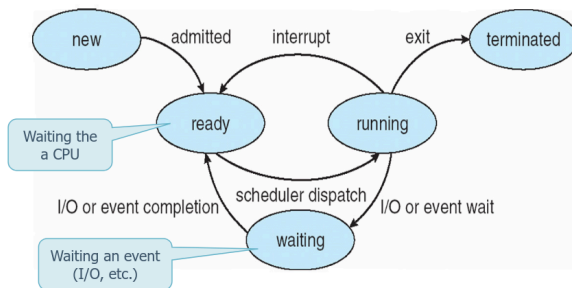
pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

Process state

- New: process is created and submitted to the OS
- Running: a CPU is allocated to the process (in execution)
- Ready: logically ready to run, waiting that a CPU is available
- Waiting: for an event or for resources
- Terminated: releases the resource it is using

State diagram

The possible state evolution of a process is described by a state diagram.

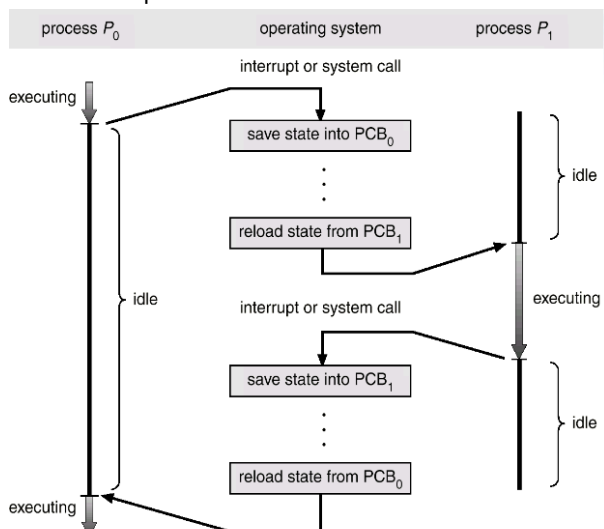


Context switching

When a CPU is assigned to another process, the kernel:

- saves the state of the running process
- Loads the state previously saved for the new process

The time devoted to the context switching is overhead, i.e., time not directly useful for any process, this amount of time is hardware-dependent



Process scheduling

Multiprogramming aims at maximizing the CPU usage by processes

Processes can be classified as:

- I/O-bound
 - Spend more time for I/O than for computation
 - Require short CPU service times
- CPU-bound
 - Spend more time for computation than for I/O
 - Require long CPU service times

CPU scheduler

The context Switching operations are controlled by the scheduler of the CPU.

The goal of the scheduler is to:

- Maximize the CPU usage (by means of the processes)
- Attempting to satisfy various system-level requests timely (hardware and software interrupts)

Ultimately, the scheduler is in charge of:

- Determine when the current process should finish its execution
- Select the next process to run from the available processes

Advanced Control exec

Use of fork

System call `fork` created a new process duplicating the calling process.

There are two main applications of this mechanism:

- Parent and child execute different code sections
 - example: a network server duplicates itself at each client request, and the child serves the request while the parent waits for a new client request
- Parent and child execute different code:
 - Uses the family of `exec` system calls.

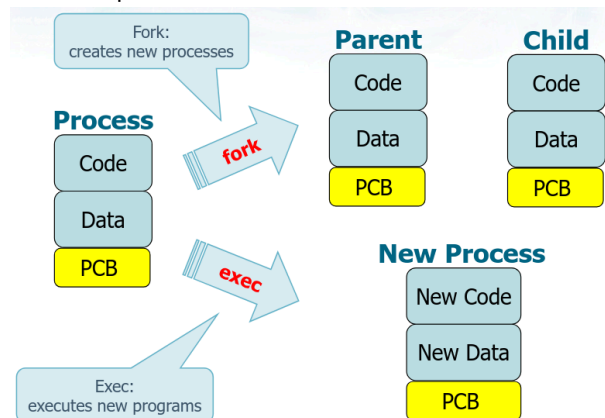
exec

System call `exec` substitutes the process code with the executable code of another program. The new program begins its execution as usual (from `main`)

In particular it:

- Does not create a new process
- Substitutes the calling process image (i.e., its code, its data, the stack and the heap) with the image of another program.
- The process PID does not change
 - `fork` -> duplicates an existent process
 - `exec` -> executes a new program

Address space



exec versions

6 versions of exec system call

- `execl`, `execvp`, `execle`
- `execv`, `execvp`, `execve`

Type	Action	
l (list)	Arguments are a list of strings	The first variable is the name, then n arguments follow, each with its own declaration, so a list is not passed.
v (vector)	Arguments is a vector of strings arguments (char **)	arg[0] name, arg[1...n-1] argument, arg[n]=NULL
p (path)	The executable filename is looked for in the directories listed in the environment variable PATH	
e (environment)	The last argument is an environment vector envp[] which defines a set of new associations strings name=value	Only here environment variables are not inherited from the parent process. They are explicitly specified, in a vector of pointers to string characters. The strings specify the values of the desired environment variable. (variable=value).

```
#include <unistd.h>
int execl (char *path, char *arg0, ..., (char *)0);
int execvp (char *name, char *arg0, ..., (char *)0);
int execle(char *path, char *arg0,..., (char *)0,
char *envp[]);
int execv (char *path, char *arg[]);
int execvp (char *name, char *arg[]);
int execve (char *path, char *arg[], char *envp[]);
```

return :

- none on success
- -1 on error

path :

Pathname of the executable file. It can be the name, or the path.

- In the exec has 'p', its better to use only name (its inherited from the environment variable PATH)
- In the non -'p', the path is unknown if not specified

l version

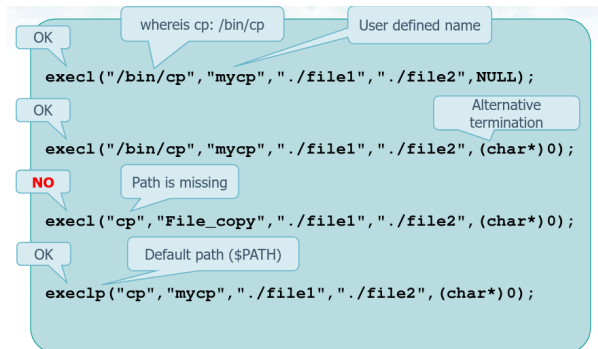
```
execl("/bin/ls", "ls", "-l", "-a", NULL); // here -l and -a are arguments
```

v version

```
char *args[] = {"ls", "-l", "-a", NULL};
execv("/bin/ls", args);
```

e version

```
char *env[] = {
"USER=unknown",
"PATH=/tmp",
NULL
};
execle (path, arg0, ..., argn, 0, env);
execve (path, argv, env);
```



Considerations

Note that during the exec

- all open file descriptors are maintained (including stdin, stdout, stderr)
- This allow the process to inherit possible redirections previously set (e.g., by shell)

Many kernels

- Implement only system call execve
- The other versions are macros that use this system call

UNIX shell skeleton

Command run in foreground

<command> :

```
while (TRUE) {
    write_prompt();
    read_command (command, parameters);
    if (fork() == 0){
        /* Child: Execute command */
        execve (command, parameters);
    }
    else{
        /* Parent: Wait child */
        wait (&status);
    }
}
```

<command> &:

```
while (TRUE) {
    write_prompt();
    read_command (command, parameters);
    if (fork() == 0){
```

```

        /* Child: Execute command */
        execve (command, parameters);
    }
    /* else */
    /* Parent: DOES NOT wait */
    /* wait (&status); */
}

```

system()

It can be useful to execute a shell command from a process, this is the purpose of `system()`

```

#include <stdlib.h>
int system (const char *string);

```

System call `system()` :

forks a shell, which executes the string command, while the parent process waits the termination of the shell command returns:

- -1 if fork or waitpid fail
- 127 if the exec fails
- The exit value of the shell that executed the command

```

int system (const char *cmd) {
    pid_t pid;
    int status;
    if (cmd == NULL){
        return(1);
    }
    if ( (pid = fork()) < 0) { //Error in fork
        status = -1;
    }
    else if (pid == 0) {
        execl("/bin/sh", "sh", "-c", cmd, (char *) 0);
        _exit(127);
    }
    else {
        while (waitpid (pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1;
                break;
            }
    }
    return(status);
}

```

Signals

A signal is a software interrupt, which is an asynchronous notification sent by the kernel or by another process to a process to notify it of an event that occurred.

They permit asynchronous events to communicate.

Interrupts

Interruption of the current execution due to the occurrence of an extraordinary event:

- A hardware device that sends a service request to the CPU
- A software process that requires the execution of a particular operation

Common signals

Examples of common signals

- Termination of a child
 - `SIGCHLD` sent to the parent; default action = ignore the signal
- Press on the terminal `Ctrl-C`
 - `SIGINT` sent to the running process (in foreground); default action = terminate the process
- Invalid memory access
 - `SIGTSTP` sent by the kernel to the process;
 - default action = suspend the execution
- The system call `alarm(t)`
 - `SIGALRM` sent after `t` seconds; default action = terminate the process
- Press on the terminal `Ctrl-Z`
 - `SIGTSTP` sent to the running process (in foreground); default action = suspend the execution
- Press on the terminal `Ctrl-\`
 - `SIGQUIT` sent to the running process (in foreground); default action = terminate the process and dump core

Eccezione	Exception Handler	Segnale
Divide error	<code>divide_err()</code>	<code>SIGFPE</code>
Debug	<code>debug()</code>	<code>SIGTRAP</code>
Breakpoint	<code>int3()</code>	<code>SIGTRAP</code>
Overflow	<code>overflow()</code>	<code>SIGSEGV</code>
Bounds check	<code>bounds()</code>	<code>SIGSEGV</code>
Invalid opcode	<code>invalid_op()</code>	<code>SIGILL</code>
Segment not present	<code>segment_not_present()</code>	<code>SIGBUS</code>
Stack segment fault	<code>stack_segment()</code>	<code>SIGBUS</code>
General protection	<code>general_protection()</code>	<code>SIGSEGV</code>
Page fault	<code>page_fault()</code>	<code>SIGSEGV</code>
Interval reserved	none	None
Floating point error	<code>coprocessor_err()</code>	<code>SIGFPE</code>

Characteristics

Standardized by the POSIX standard, they are now stable and relatively reliable

- Each signal has a name
- Names start with `SIG...`
- The file `signal.h` defines signal names
- Unix FreeBSD, Mac OS X and Linux support 31 signals
- Solaris (oracle) supports 38 signals

Main signals

You can display the complete list of signals using the shell command

```
kill -l
```

Name	Description
<code>SIGABRT</code>	Process abort, generated by system call <code>abort</code>
<code>SIGALRM</code>	Alarm clock, generated by system call <code>alarm</code>
<code>SIGFPE</code>	Floating-point exception
<code>SIGILL</code>	Illegal instruction
<code>SIGKILL</code>	Kill (non-maskable)

Name	Description
SIGPIPE	Write on a pipe with no reader
SIGSEGV	Invalid memory segment access
SIGCHLD	Child process stopped or exited
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Notes:

- Default action = terminate the process
- SIGUSR1 and SIGUSR2 are available for use in user applications

Signal management

Signal management goes through three phases:

1. Signal generation

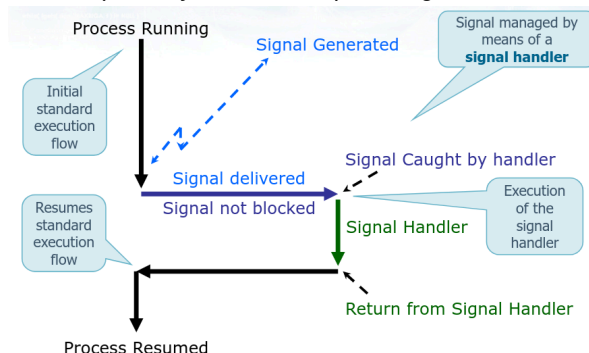
- When the kernel or a source process causes an event that generate a signal

2. Signal delivery

- A not yet delivered signal remains pending
- At signal delivery a process executes the actions related to that signal
- The lifetime of a signal is from its generation to its delivery
- There is no signal queue; the kernel sets a flag in the process table

3. Reaction to a signal

- To properly react to the asynchronous arrival of a given type of signal, a process must inform the kernel about the action that it will perform when it will receive a signal of that type
- A process may
 - Accept the default behavior (be terminated)
 - Declare to the kernel that it wants to ignore the signals of that type
 - Declare to the kernel that it wants to catch and manage the signals of that type by means of a signal handler function (similarly to the interrupt management)



Signal management can be carried out with the following system calls:

- `signal`
 - Instantiates a signal handler
- `kill (and raise)`
 - Sends a signal
- `pause`
 - Suspends a process, waiting the arrive of a signal
- `alarm`
 - Sends a `SIGALARM` signal, after a preset time
- `sleep`
 - Suspends the process for a specified amount of time (waits for signal `SIGALRM`)

Notice:

- The terms `signal` and `kill` are relatively inappropriate, `signal` does not send a signal

signal()

Allow to instantiate a signal handler

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

`sig` : specifies the signal to be managed (`SIGALRM` , `SIGUSR1` , etc.)

`func` : the function use to manage it, i.e. the signal handler (you give a pointer to that function, it takes `int` and returns `int`)

return:

- on success, the previous value of the signal handler (pointer to the previous signal handler pointer)
- on error: `SIG_ERR` , `errno` is set to indicate the cause

Reaction to a signal

`signal` system call allows setting three different reactions to the delivery of a signal:

1. Accept the default behavior

- `signal (SIGname, SIG_DFL)`
- Where `SIG_DFL` is defined in `signal.h`
 - `#define SIG_DFL ((void (*)(int)) 0)`
- Every signal has its own default behavior, defined by the system
- Most of the default reactions is `process termination`

2. Ignore signal delivery:

- `signal (SIGname, SIG_IGN)`
- Where `SIG_IGN` is defined in `signal.h`
 - `#define SIG_IGN ((void (*)(int)) 1)`
- Applicable to the majority of signals
 - Ignoring a signal often leads to an indefinite behavior
- Some signals cannot be ignored
 - `SIGKILL` and `SIGSTOP` cannot be ignored because the kernel and the superuser would maintain the possibility to control all processes
 - Ignoring an illegal memory access, signaled by `SIGSEGV` , would produce an undefined process behavior

3. Catch the signal

- `signal (SIGname, signalHandlerFunction)`
- where
 - `SIGname` indicates the signal type
 - `signalHandlerFunction` is the user defined signal handler function
- The signal handler
 - Can take action considered correct for the management of the signal
 - Is executed asynchronously when the signal is received
 - When it returns, the process continues with the next instruction, as it happens for interrupts

example:

```
void manager (int sig) {
    if (sig==SIGUSR1)
        printf ("Received SIGUSR1\n");
    else if (sig==SIGUSR2)
        printf ("Received SIGUSR2\n");
    else printf ("Received %d\n", sig);
    return;
}
...
int main () {
    ...
```

```
signal (SIGUSR1, manager);
signal (SIGUSR2, manager);
...
}
```

kill()

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

send signal `sig` to a process or to a group of processes `pid`
return:

- 0 on success
- -1 on error

Notice:

- If `sig=0` a NULL signal is sent (i.e., no signal is sent). This is often used to check if a process exists: if the kill returns -1 the process does not exist.

To send a signal to a process you must have the rights. A user process can send signals only to processes having the same UID . The superuser can send signal to any process

pid Value	Signal Sent To
> 0	The process with PID equal to <code>pid</code>
== 0	All processes with GID equal to the sender's GID (if it has the rights)
< 0	All processes with GID equal to the absolute value of <code>pid</code> (if it has the rights)
== -1	All processes (if it has the rights)

Notice:

- All processes excludes a set of system processes.

raise()

The raise system call allows a process to send a signal to itself

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

`raise (sig)` is equivalent to `kill (getpid(), sig)`

pause()

Suspends the calling process until a signal is received

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

Returns after the completion of the signal handler

- In this case the function returns -1

alarm()

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

return :

- The number of seconds remaining until the delivery of a previously scheduled alarm
- 0 if there was no previously scheduled alarm

warning:

- The signal is generated by the kernel
 - It is possible that the process get the CPU control after some time, depending on the scheduler decisions
- There is only one time counter for each process, and system calls sleep and alarm uses the same kernel timer

example of implementation of sleep using alarm and pause:

```
#include <signal.h>
#include <unistd.h>
static void sig_alm(int signo) {return;}
unsigned int sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alm) == SIG_ERR){
        return (nsecs);
    }
    alarm (nsecs);
    pause ();
    return (alarm(0)); //Returns 0, or the remaining time before the delivery if pause returns because
    another signal has been received
}
```

Notice:

- The signal handler must be instanced before setting the alarm

example of implementation of alarm using fork, signal, kill, and pause:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void myAlarm (int sig) {
    if (sig==SIGALRM)
        printf ("Alarm on ...\n");
    return;
}

int main (void) {
    pid_t pid;
    (void) signal (SIGALRM, myAlarm);
    pid = fork();
    switch (pid) {
        case -1: /* error */
            printf ("fork failed");
            exit (1);
        case 0: /* child */
            sleep(5);
            kill (getppid(), SIGALRM); //The child waits and sends SIGALRM
            exit(0);
    }
    /* parent */
    pause (); //The parent pauses, and continues only when it receives the SIGALRM sent by the child
    exit (0);
}
```

Signal limitations

- Signals do not convey any information
- The memory of the pending signals is limited

- Max one signal pending (sent but not delivered) per type. Forthcoming signals of the same type are lost
- Signals can be ignored
- Signals require functions that must be reentrant
- Produce race conditions

Some limitations are avoided in POSIX.4

Memory limit

The memory related to "pending" signals is limited

- There is at most one "pending" signal (sent, delivered, but not managed) for each type of signal
 - Subsequent signals (of the same type) are lost
- Signals can be blocked, i.e., you can avoid receiving them

```
...
static void sigUsl1 (int);
static void sigUsl2 (int);
static void sigUsl1 (int signo) {
    if (signo == SIGUSR1){
        printf("Received SIGUSR1\n");
    }
    else{
        printf("Received wrong SIGNAL\n");
    }
    fprintf (stdout, "sigUsl1 sleeping ...\n");
    sleep (5);
    fprintf (stdout, "... sigUsl1 end sleeping.\n");
    return;
}

static void sigUsl2 (int signo) {
    if (signo == SIGUSR2){
        printf("Received SIGUSR2\n");
    }
    else{
        printf("Received wrong SIGNAL\n");
    }
    fprintf (stdout, "sigUsl2 sleeping ...\n");
    sleep (5);
    fprintf (stdout, "... sigUsl2 end sleeping.\n");
    return;
}

int main (void) {
    if (signal(SIGUSR1, sigUsl1) == SIG_ERR) {
        fprintf (stderr, "Signal Handler Error.\n");
        return (1);
    }
    if (signal(SIGUSR2, sigUsl2) == SIG_ERR) {
        fprintf (stderr, "Signal Handler Error.\n");
        return (1);
    }
    while (1) {
        fprintf (stdout, "Before pause.\n");
        pause ();
        fprintf (stdout, "After pause.\n");
    }
    return (0);
}
```

```
> ./pgrm &
[3] 2636
> Before pause.
```

```
> kill -USR1 2636
> Received SIGUSR1
sigUsrc sleeping ...
... sigUsrc end sleeping.
After pause.
Before pause.
> kill -USR2 2636
> Received SIGUSR2
sigUsrc sleeping ...
... sigUsrc end sleeping.
After pause.
Before pause.
```

```
> kill -USR1 2636 ; kill -USR2 2636
> Received SIGUSR2
sigUsrc sleeping ...
... sigUsrc end sleeping.
Received SIGUSR1
sigUsrc sleeping ...
... sigUsrc end sleeping.
After pause.
Before pause.
```

Reentrant functions

A signal has the following behavior:

1. The interruption of the current execution flow
2. The execution of the signal handler
3. The return to the standard execution flow at the end of the signal handler

Consequently

- The kernel knows where a signal handler returns, but
- The signal handler does not know where it was called, i.e., the control flow was interrupted by the signal

Suppose a `malloc` is interrupted, and the signal handler calls another `malloc`

- Function `malloc` manages the list of the free memory regions, which could be corrupted

Suppose that the execution of a function that uses a static variable is interrupted, but is then called by the signal handler

- The static variable could be used to store a new value, i.e., it does not remain the same it was before the signal was delivered

The "Single UNIX Specification" defines the reentrant functions, which can be interrupted without problems

- `read`, `write`, `sleep`, `wait`, etc.

Most of the I/O standard C functions are not reentrant

- `printf`, `scanf`, etc.
- They use static variables or global variables
- They must be used carefully and being aware of possible problems

Race conditions

- The result of more concurrent processes working on common data depends on the execution order of the processes instructions.
- Concurrent programming is subject to race conditions and using signals increases the probability of race conditions.
- Using signals increases the probability of race conditions.

example a:

suppose a process decides to suspend itself for a given number of seconds,.

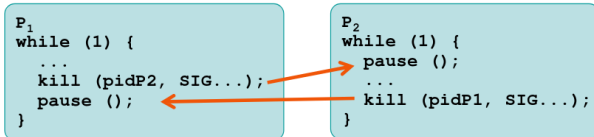
The signal could be delivered before the execution of pause due to a context switching and scheduling decisions (especially in high loaded systems).

example b:

Suppose two processes P_1 and P_2 decide to synchronize by means of signals.

Unfortunately:

- If P_1 (P_2) signal is delivered before P_2 (P_1) executes pause
- Process P_2 (P_1) blocks forever waiting a signal



Despite their defects, signals can provide a rough synchronization mechanism.

Shell commands for process management

Foreground execution

The "standard" shell commands

- Allow executing processes sequentially
- Each process is executed in foreground, i.e., using the control terminal

```
> command1
Output of command1
> command2
Output of command2
```

Background execution

The shell interpret character `&` as an indication to run the command in the background

- The process is executed in concurrency with the shell. It loses the control terminal input
- The shell outputs immediately a new prompt
- It is possible to run several processes in parallel

```
> command1 &
> command2 &
>
Output of command1
Output of command2
```

ps

The command `ps` (process status of active process):

- ☐ Lists active processes and related details
- ☐ Without options (default) prints (in a compact format) the status of the processes with the same user ID of the user from which the command is executed

options:

- ☐ `-a` Lists the processes of all system users
- ☐ `-u` Prints more detailed information (resident size, virtual size, etc.)
- ☐ `-u <user>` Shows only the `<user>` processes
- ☐ `-x` Adds to the list the processes that do not have a control terminal

(e.g., daemon)

- ☐ `-e` (or `-A`) Lists all processes running in the system
- ☐ `-f` Extended format
- ☐ `r` (not `-r`) Shows only the "running" processes

top

Display and updates information about the system used resources, and the active processes

```
top - 11:45:49 up 18:20, 1 user, load average: 0.40, 0.44, 0.45
Tasks: 386 total, 2 running, 384 sleep, 0 d-sleep, 0 stopped, 0 zombie
%Cpu(s): 1.5 us, 0.6 sy, 0.0 ni, 97.6 id, 0.0 wa, 0.2 hi, 0.1 si, 0.0 st
MiB Mem : 31759.1 total, 18310.2 free, 5772.3 used, 8060.9 buff/cache
MiB Swap: 4096.0 total, 4089.2 free, 6.8 used, 25986.8 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
13583	gandalf	20	0	1906948	290776	223040	S	20.3	0.9	0:00.51	spectacle
1689	gandalf	-2	0	1802448	197696	133564	S	9.5	0.6	7:30.48	kwin_wayland
11502	gandalf	20	0	2625484	292396	162120	S	4.0	0.9	1:07.41	Isolated Web Co
1843	gandalf	20	0	5713772	527192	204116	R	2.8	1.6	3:57.75	plasmashell
2347	gandalf	20	0	11.9g	887980	318840	S	0.8	2.7	12:17.45	firefox
11822	gandalf	20	0	2697796	298376	164216	S	0.8	0.9	0:06.42	Isolated Web Co
1621	gandalf	20	0	6248	4436	2348	S	0.4	0.0	0:03.33	dbus-broker
1758	gandalf	20	0	385612	69576	46376	S	0.4	0.2	0:25.95	Xwayland
1922	gandalf	20	0	176020	29800	25128	S	0.4	0.1	0:26.06	ksystemstats
4189	gandalf	20	0	33.1g	268120	224224	S	0.4	0.8	1:55.28	electron
4919	gandalf	20	0	2855508	437464	180188	S	0.4	1.3	0:20.73	Isolated Web Co
8804	root	20	0	0	0	0	I	0.4	0.0	0:00.15	kworker/6:0-events
9389	gandalf	20	0	6993464	424776	177856	S	0.4	1.3	0:39.29	Isolated Web Co
11875	gandalf	20	0	2732924	289264	167220	S	0.4	0.9	0:03.60	Isolated Web Co
12103	root	0	-20	0	0	0	I	0.4	0.0	0:02.27	kworker/u81:3-i915_flip
1	root	20	0	23520	14612	10196	S	0.0	0.0	0:03.95	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.00	pool_workqueue_release
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-rcu_gp
5	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-sync_wq
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-kvfree_rcu_reclaim
7	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-slab_flushwq
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-netns
10	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H-events_highpri
13	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/R-mm_percpu_wq
15	root	20	0	0	0	0	S	0.0	0.0	0:00.03	ksoftirqd/0

kill

kill allows sending signal from the shell

Format:

- ❑ `kill [-sig] pid`
- ❑ Sends signal `sig` to process with `PID=pid`
- ❑ Option `sig` indicates the signal code
- ❑ `pid` is the process identifier (PID) of the target process

A signal `sig` can be indicated by mean of its name or its number:

- ❑ `SIGKILL` = `KILL` = 9
- ❑ `SIGUSR1` = `USR1` = 10
- ❑ `SIGUSR2` = `USR2` = 12
- ❑ `SIGALRM` = `ALRM` = 14

If you type `kill -l`, you get all options

The default is `SIGTERM`

example of killing 10234:\

```
kill -9 10234
kill -SIGKILL 10234
kill -KILL 10234
```

killall

Shell command `killall` terminates all process with a specified name

```
killall -9 name
```

Useful to terminate all processes generated by the same program avoiding to specify their PIDs

Inter-process communication

Indipendent and cooperating process

Concurrent processes can be:

- independent
- cooperating

And independent process

- Cannot be influenced by other processes
- Cannot influence other processes

A set of cooperating processes

- can cooperate only by sharing data or by exchanging messages
- Both require appropriate synchronization mechanisms

IPC

Information sharing among processes is referred to as InterProcess Communication

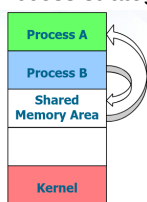
The main communication models are based on:

- Shared memory
- Message exchange

Shared memory

Sharing of a memory area and writing of data in this area

- Normally the kernel does not allow a process to access the memory of another process.
- Processes must agree on the:
 - Access rights
 - Access strategies



The most common methods for shared buffer use a:

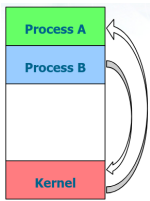
- File
 - Sharing the name or the file pointer or descriptor before `fork / exec`
- Mapped file in memory
 - A file mapped in memory associates a shared memory region to the file

These techniques allow sharing a large amount of data

Message exchange

Communication takes place through the exchange of messages:

- Need to setup a communication channel
- Useful for exchanging limited amounts of data
- Uses system calls
 - which request kernel intervention
 - and introduce overhead



A communication channel can offer direct or indirect communication

- direct
 - Is performed naming the sender or the receiver
 - send (to_process, message)
 - receive (from_process, &message)
- Indirect
 - Performed through a mailbox
 - send (mailboxAddress, message)
 - receive (mailBoxAddress, &message)

A communication channel can be characterized by

- Synchronization
 - Both sending or receiving messages can be
 - Synchronous, i.e., blocking
 - Asynchronous, i.e., non-blocking
- Capacity
 - If the capacity is zero, the channel cannot allow waiting messages (no buffering); the sender blocks waiting for the receiver
 - If the capacity is limited the sender blocks when the queue is full
 - If the capacity is unlimited the sender does not block

UNIX makes available:

- Half-duplex pipes
- FIFOs //not covered
- Full-duplex pipes //not covered
- Named full-duplex pipes //not covered
- Message queues
- Semaphores //for process synchronization
- Sockets //network process communication. Each process is identified by a socket to which it is associated a network address
- STREAMS

Pipes

Pipes are the oldest form of communication in UNIX SO.

They provide a communication channel, which is:

- Direct
- asynchronous
- with limited capacity

They live in memory and they are more efficient than using of files

They exist to create a data stream among processes

- The user interface to a pipe is similar to file access
- A pipe is accessed by means of two descriptors (integers), one for each end of the pipe
- A process (P1) writes to an end of the pipe, another process (P2) reads from the other end

Historically they have been half-duplex

- Data can flow in both directions (from P1 to P2 or from P2 to P1), but not at the same time
- Full-duplex models have been proposed more recently, but they have limited portability

Simplex: Mono-directional

Half-Duplex: One-way, or bidirectional, but alternate (walkie-talkie)

Full-Duplex: Bidirectional (telephone)

A pipe can be used for communication among a parent and its children, or among processes with a common ancestor

- The file descriptor must be common to the two communicating processes and therefore these processes must have a common ancestor

pipe()

```
# include <unistd.h>
int pipe (int fileDescr[2]);
```

This system call creates a pipe

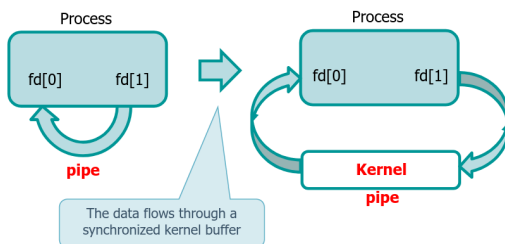
It returns two file descriptors in vector `fileDescr` `fileDescr[0]` : Typically used for reading `fileDescr[1]` : Typically used for writing The input stream written on `fileDescr[1]` corresponds to the output stream read on `fileDescr[0]`

return:

- 0 on success
- -1 or error

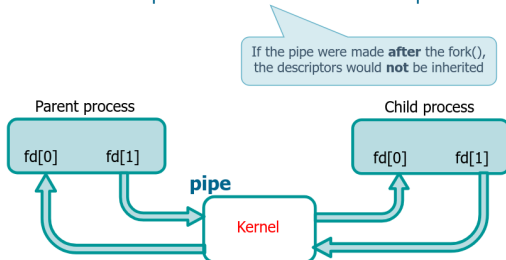
Resources associated to a pipe released when all involved processes close their terminals or they are terminated.

Using a pipe inside a process is possible but not that useful:



A pipe typically allows a parent and a child to communicate. Parent must fork after creating the pipe.

- The child process **inherits** the file descriptors



Pipe I/O

The descriptor of the pipe is an integer number

R/W on pipes do not differ to R/W on files

- Use read and write system calls
- It is possible to have more than one reader and writer on a pipe, but
 - The standard case is to have a single writer and a single reader
 - Data can be interlaced using more than one writer
 - Using more readers, it is undetermined which reader will read the next data from the pipe

System call read

- Blocks the process if the pipe is empty (it is blocking)

- If the pipe contains less bytes than the ones specified as argument of the read, it returns only the bytes available on the pipe
- If all file descriptors referring to the write-end of a pipe have been closed, then an attempt to read from the pipe will see end-of-file (read returns 0)

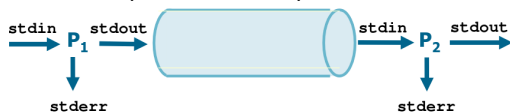
System call write

- Blocks the process if the pipe is full (it is blocking)
- The dimension of the pipe depends on the architecture and implementation
 - Constant `PIPE_BUF` defines the number of bytes that can be written atomically on a pipe
 - Standard value of `PIPE_BUF` is 4096 on Linux
- If all file descriptors referring to the read-end of a pipe have been closed, then a write to the pipe will cause a `SIGPIPE` signal to be generated for the calling process
- If the end of the write operations are not to be verified based on the return value of the read, it is always possible to transfer a "sentinel" (end-of-message marker)

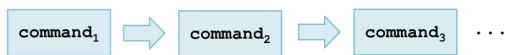
Shell commands for Pipes and redirections

Pipe

Inter-process communication can be performed also by processes executed by shell commands. A shell pipe connects the standard output of a sender process, and the standard input of a receiving process



```
command1 | command2 | command3 ...
```



Examples:

```
ls -la | more
ps | grep main
cat file1.txt file2.txt file3.txt | sort
```

I/O redirection

The term redirection indicates the deviation of the standard channels (`stdin` 0, `stdout` 1, `stderr` 2). So a process (command) read/writes to a non standard source/destination.

null file

`/dev/null` is a special file, writing on it doesn't produce any output, but reading it returns a sequence of zeros. It acts as a sink in the write.

Redirection types

Channel	Action	Operator	Description	Example
Standard Input	Read from file	<code>< file</code>	<code>stdin</code> (descriptor 0) reads input from the file instead of the keyboard.	<code>prgm < file</code>
Standard Output	Overwrite file	<code>> file</code> or <code>1> file</code>	<code>stdout</code> (descriptor 1) is redirected to the file, overwriting any existing content.	<code>wc prgm.c > file_pout.txt</code>
Standard Output	Append to file	<code>>> file</code> or <code>1>> file</code>	<code>stdout</code> (descriptor 1) is redirected to the file, appending to the existing content.	<code>wc prgm.c >> file_pout.txt</code>
Standard Error	Overwrite file	<code>2> file</code>	<code>stderr</code> (descriptor 2) is redirected to the file, overwriting any existing content.	<code>prgm 2> fileErr</code>

Channel	Action	Operator	Description	Example
Standard Error	Append to file	2>> file	stderr (descriptor 2) is redirected to the file, appending to the existing content.	
Both (Output/Error)	Overwrite file	&> file or >& file	stdout and stderr are both redirected to the file, overwriting it. (Shorthand for > file 2>&1)	
Both (Output/Error)	Append to file	&>> file	stdout and stderr are both redirected to the file, appending to it. (Shorthand for >> file 2>&1)	

Multiple redirections

You can redirect standard output and standard error to different files:

- bash shell:

```
`command 1> fileOut 2> fileErr`
```

ONLY u10 is missing, not part of course