# Advanced Robot Programming Notes

Davide Tonelli

November 8, 2023

## Contents

# 1 Posix

**Posix guarantees portability at compile level**

# 2 To install

```
sudo apt install libncurses-dev
```

# 3 Interprocess communication

- Messages
- blackboards
- publish/subscribe
- events

In the old unix terminology the element that connected two different programs was called pipe. In posix the pipe is a FIFO queue. There are different kinds of pipes.

# 4 Ex 2

It's importante that every device is virtualized by a device driver.

## 4.1 MAP

The map is usually a blackboard and many other components use the map. Many components contribute to put elements on the map and they want to read updated data from the map.

## 4.2 Odometry

Knowing the position and the displacement you know the next position.

# 5 Shell

28/09

# 6 The kernel

On the upper level of the kernel we have the users and root. Usually **kernel** is called root + all the application that root can run. Under the syscall layer we have the innersize of the kernel. The idea of having this model is called **monolitic kernel** because is difficult to modify what's inside while the sistem is runnig. Windows for example is layered with several part that can substitued and changed. Other type of kernels are called **microkernels**, meaning that are divided in different small parts.

## 6.1 Inside the kernel

Inside the kernel we have interrupts routines that can be hardware and then there are traps meaning that a process can run an event. When your process asks a function as posix level (system calls) is the way in which the process asks for services to the operating system. A syscall is an interrupt. For comunicating to the core part you rise an interrupt(event). An other part inside the kernel is the one of virtual memory. We can adress more memory that the one that is install. The memory that we adresss is adressed by virtual adresses. We then got the **switcher** and then the **scheduler** who decides what processes need to be replaced in the execution of the CPU. These are very small and mostly written in assembly.

You could change these programs but you are very limited during the execution of the os. There are device drivers and can be installed in the system without changing the code in general.

There is also the **clock** that is hardware and sends a tick. Every time in ticks it sends an interrupt. If the clock stops than all the machine stops.

## 7 Processes

- In traditional Unix systems, each process executes a single sequence of instructions in its own address space: a single program counter specifies the next instruction to be executed
- More modern systems can run multiple threads in the same space (often called *lightweight process*)
- Each process is independent of the others, and can interact with them only through the kernel, by appropriate system call
- The code of a process does not necessarily have to reside all in main memory (various memory management techniques ...)

Each process looks like the only one that is using the machine. Each program has the complete conrol of the memory and the virtual memory. If you have 10 processes every one think that is alone in the system. if it's aware of other processes it can comunicate with the others. Thread in this model are part of the process and all threads share all resources. They share all the memory of the process. We will not share memory with threads.

## 8 Time sharing

It's the opposite of real time becaus eit does not allow to garantee that some process can output before a given deadline. Linux is not real-time in this sense.

Unix is a time-sharing system: a process runs for a *quantum* of time (or until it is suspended by a pending event, eg a terminated I / O)

Real time is achieved with completely different kernels of linux that use a completely different scheduler than "normal" linux.

## 9 Hardware protection

All the architectures suppose that only some parts of the memory can be accessed only by root and the same goes for some functions.

- The kernel runs in kernel mode:
- You can run any machine instruction and access any of memory area
- User processes run in user mode:
- They can not perform certain instructions "dangerous" and can not access the space of kernel addresses and other processes

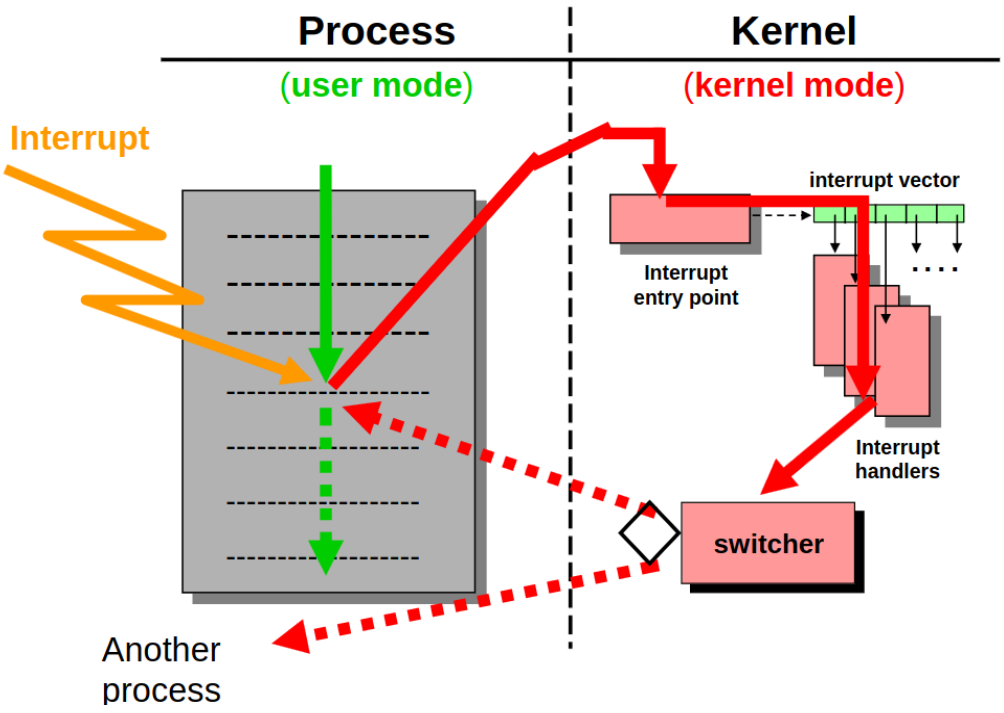
## 10 Process

A process is a foundamental unit in a system. Each process has a PID.

## 11 Daemon

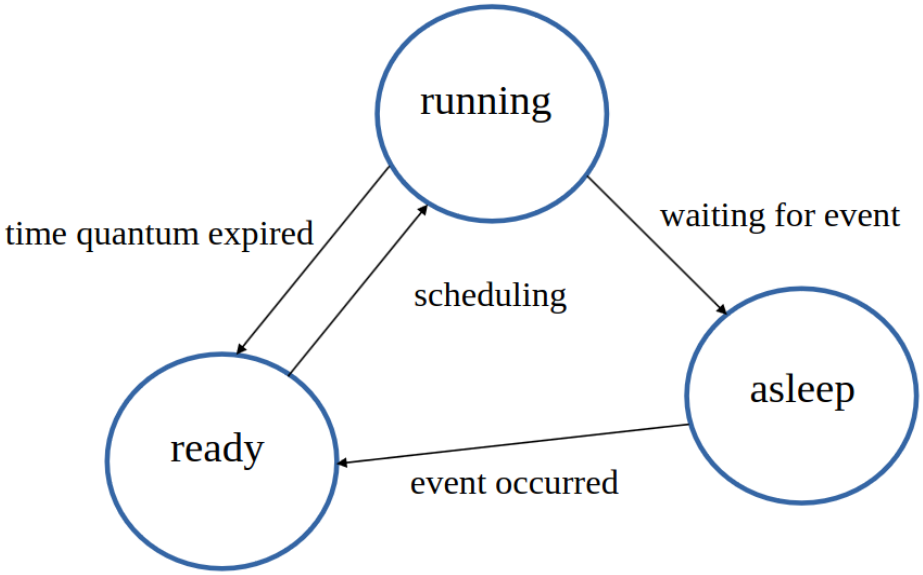
It's a service run by the kernel that's run continuously. Daemons are processes owned by the kernel.

## 12 Interrupts




The process in user mode is interrupted and execution is transferred in kernel mode. In kernel mode it jumps in specific routines. After the execution of an interrupt handler the execution does not return to the user program but the execution is given to the scheduler. The switcher decides if the original process can go on or another process can have the CPU. The scheduling of the system is simply driven by interrupts. Every 10 ms there is an interrupt for example and the scheduler has to decide. When you ask the system for a service you raise an internal interrupt.

## 13 States of processes




During execution a process can be put back in ready queue when:

- external interrupt
- internal interrupt
- —

## 14 priority

Each process has a priority level:

… -2 -1 0 1 2 …

← higher priority ——————

- The priorities are allocated dynamically
- Processes running on behalf of the kernel (their UID is root) have higher priority than user
- Processes of equal priority are scheduled in turn ("round-robin")

The priority of a process has a *static component (base)* and a *dynamic one* which depends on the CPU time received:

- Every clock tick increments the counter of CPU usage of the running process
- Every second (for example), the priorities of all processes are recalculated according to the formula:

  **new priority = base + CPU usage / 2**
- The base is usually 0, but can be increased with the `nice` command, which worsens the priority
- Thus, the ready processes that have not had a lot of CPU are "rewarded", while the running process can be "penalized"

## 15 Different spaces

Every operating system has two different levels:

- user
- kernel

Every cpu also has these two different modes. We also have services that we can call through system calls and our processes can sometimes also become root. The lower part of the system is the inner core of the kernel and you cannot modify it.

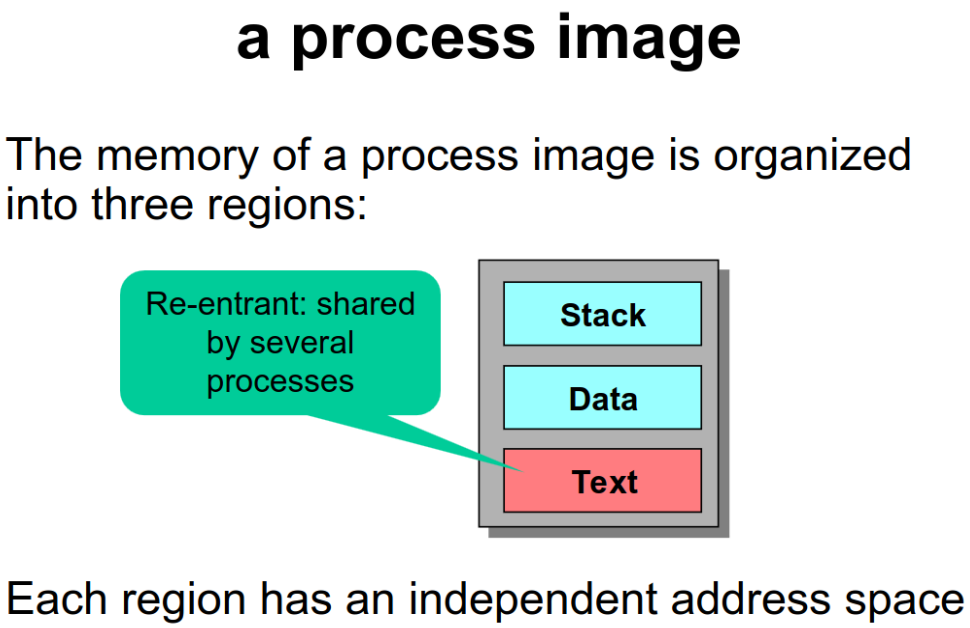
## 16 Kernel

Every other part can be modified. The inner code is written in a monolitic way and you cannot change it. Inside the kernel we have interrupt routines that are called directly by external or internal interrupts. The scheduling method of the processes is called by interrupts. If no interrupt comes the machine will stay doing nothing. The main interrupt is the clock interrupt. All system calls are interfaces to functions inside the kernel.

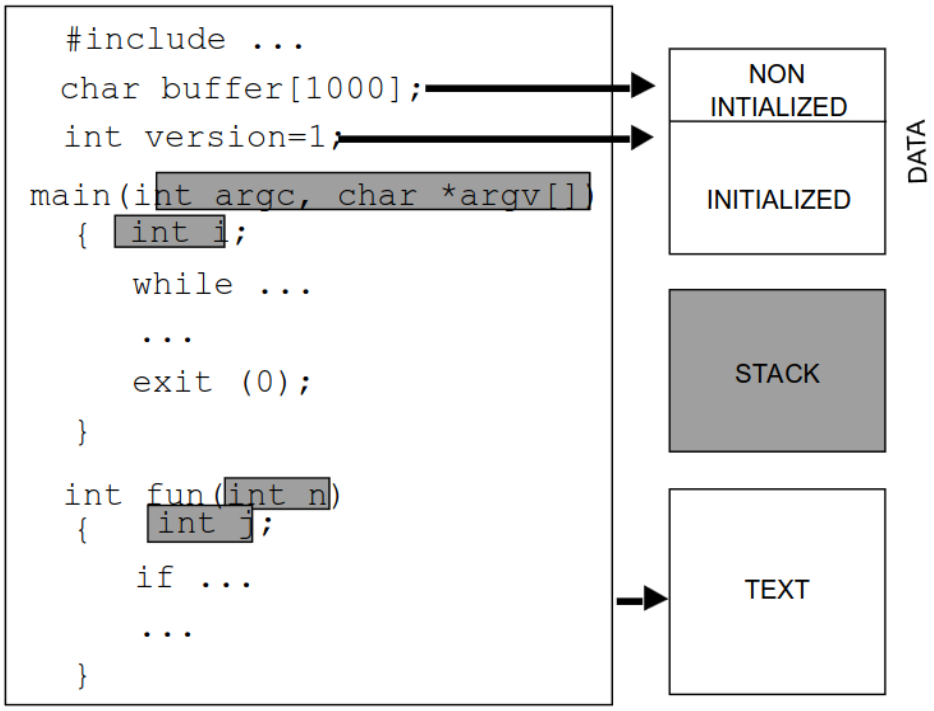
### 16.1 Graphic model

In modern systems filesystems can also be managed in user space. In user space we also get the graphical manager which is a server in user space. We can have a kernel without graphics.

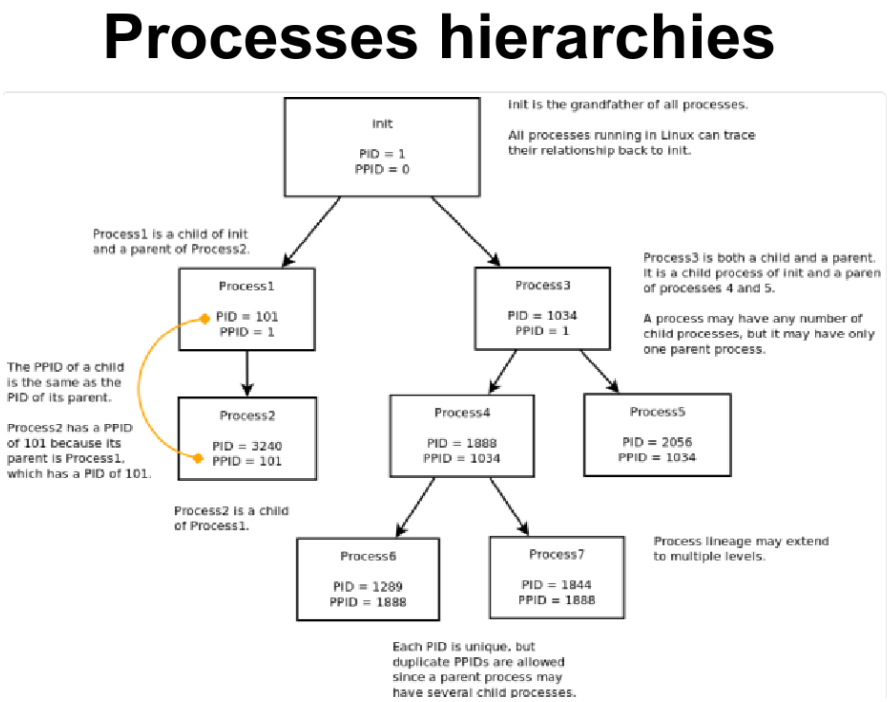
## 17 Process image




Every language is based on this model. Text is pure text and there is non data inside. This is because can happen that different processes share the same code. Data is permanent data used by the program. Stack is the stack where temporary data inside the process is stored.


#include ...
char buffer[1000];
int version=1;
main(int argc, char *argv[])
{ int i;
while ...
...
exit (0);
}
int fun(int n)
{ int j;
if ...
...
}
NON INTIALIZED
INITIALIZED
DATA
STACK
TEXT

## 18 Process hierarchies



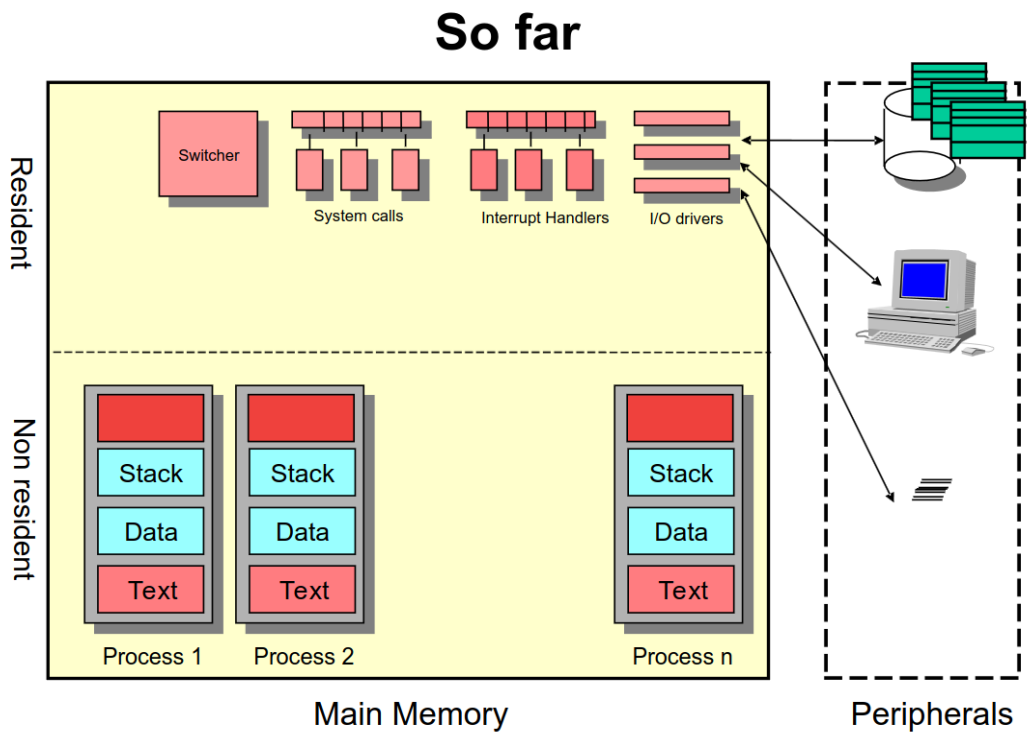

When the computer is started the bootloader is started which is the first process. The first process is called **init**. Processes are identified by a number and init is identified by 1 as pid owned by root. All these processes are in user space even if owned by root.

- PPId is very important because identifies the parent id.

Conventionally root is orphan and the ppid is 0. Any other processes has a PPID. The parent is a process that calls a syscall to generate a child. The core kernel implements the generation of new processes. It's impossible to have two processes with the same number. The numbers are sequentially generated and the system sometimes need to be restarted to reset the numbers. During the starting phase of the system you cannot dialog to the system and then at the end the dialog will appear. The graphical server is just one of the processes.

## 19 Structure

Resident means that that part is always present




## 20 Data structures for process management

### Process Table

- It is in the kernel, and is resident
- It contains an entry for each process, and is sized statically at the time of system configuration
- for each process contains information that allow the scheduling, and which must always be resident

### U-Area (user area)

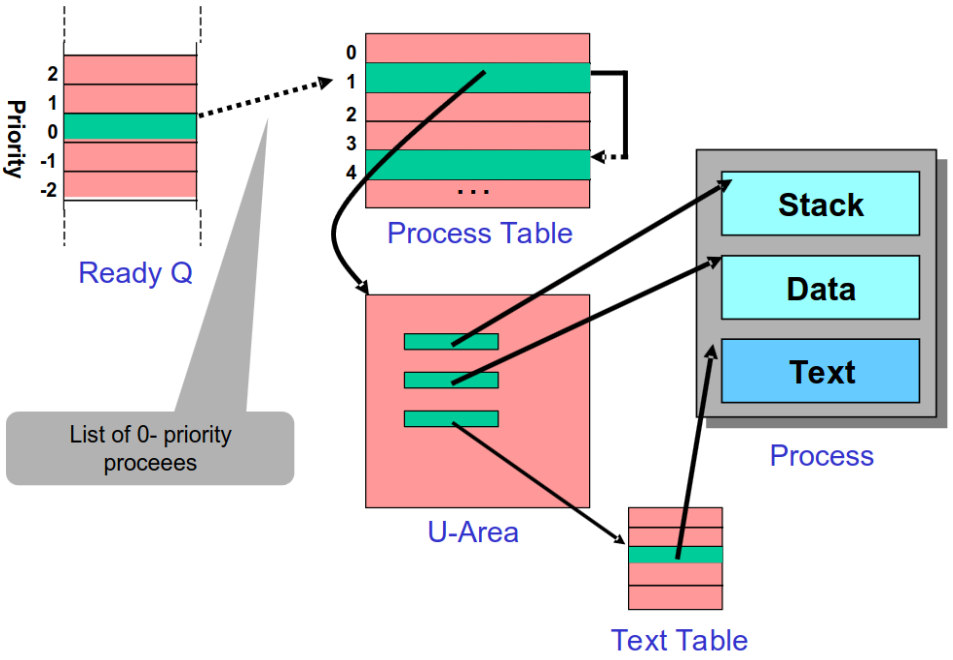
- It is in the kernel, but not necessarely resident in memory
- It contains the information necessary to the kernel for the management of the process (open files, signal masks, computing times and similar), but it is not necessarely always resident in memory

### Ready Queue

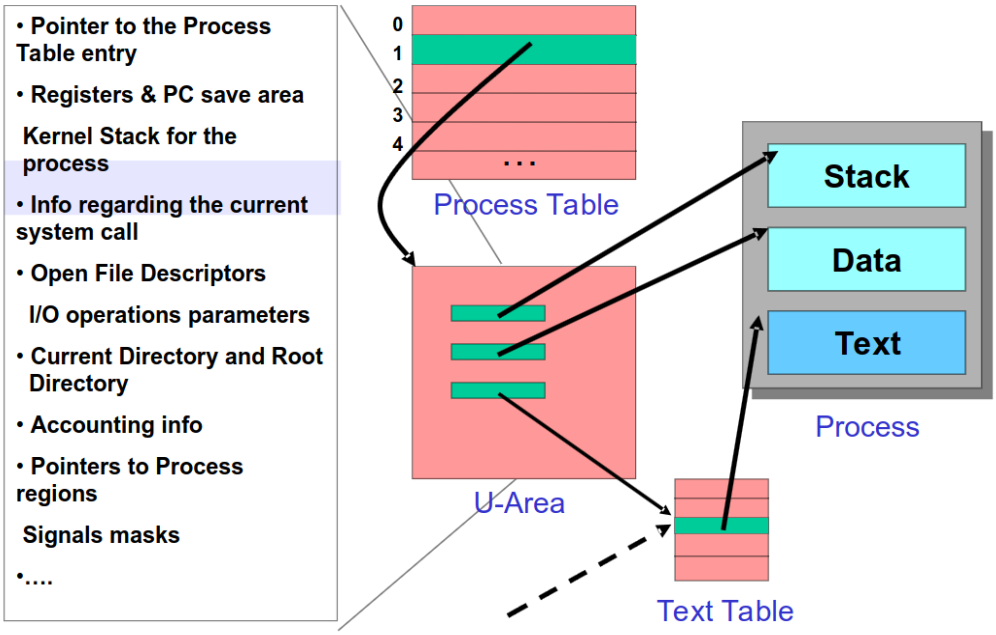
- Lists of ready processes (one for each priority level)

- Process table contains for every process the info about the process itself and the scheduling information.
- The user area is partially in the inner kernel and partially in the user space because is

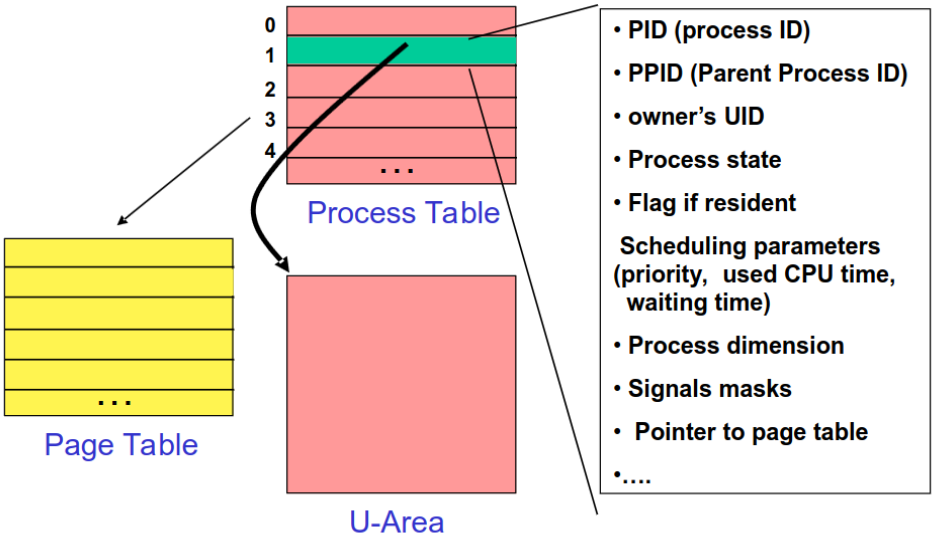
the interface between data used by the process and system calls. For example when you say open a file descriptor you are using the user area.




We are using lots of pointers at this level, and the ready queue is an ordered list that points inside the process table. The process table point to the U-Area. Text may have text tables because this could be shared between different processes. The text may not be embedded in the process.

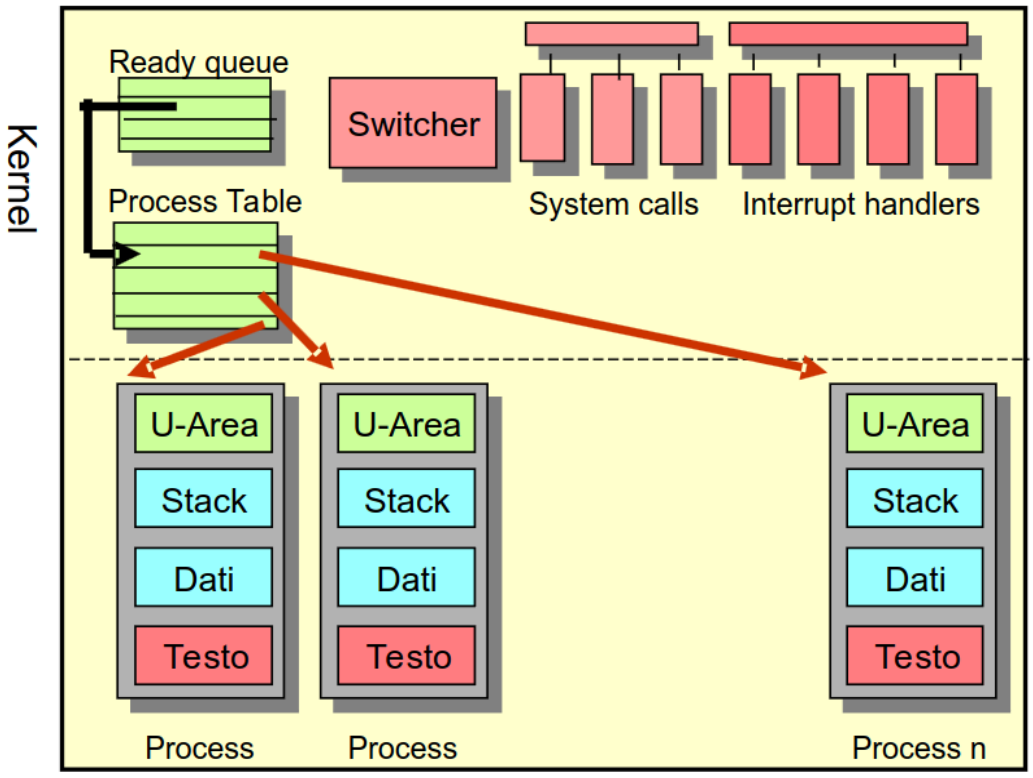


Registers are used to save informations needed when running a process in a static part of the kernel.




This is in a higher level and there are different status. This are changed during the life of a process. This is not related to the computational state.

This is the structure so far



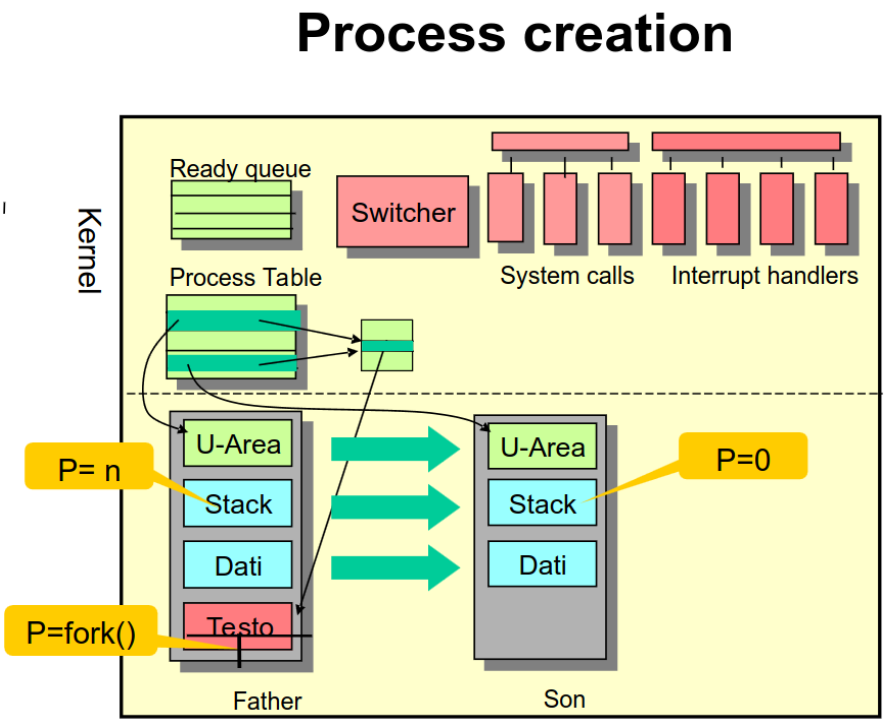

## 21 Memory management

There are several different architectures to work with paging. There are two different processes that manage memory:

- Pagedaemon: if finds the memory too busy then frees some pages by writing those pages to the filesystem. It can be a special filesystem called swap managed directly by inner kernel or also in user space filesystem.
- Swapper: activates real time if some page are missing. So if we access a page that is not in memory then the switcher activates and the process is activated again.

## 22 Unix processes

exit changes the process table from running to exited and must be erased by the system after all the controls are terminated from the kernel. From exit the process return to the father a value. Fork is very optimized even at an hardware level.




In the previous image we see how a process is duplicated.

# 23 Functions

## 23.1 Fork

- Allocates an entry in the Process Table for a new process
- Assigns a unique PID to the new process and initializes the Process Table entry fields
- Creates a copy of the parent process image (the text is not duplicated but increments a reference count)
- Increases appropriate open file counters
- Initializes the counters of accounting in the u-area of the new process
- Places the new process in ready state
- Returns the PID of the child to the father, 0 to the son, or -1 on error

- File counter is the number of processes that are opening the file. If a process is forked also the child will have the same file opened.
- The child will also immediately be ready
- All system calls return a negative integer number in case of error.

### 23.2 Exec

Substitutes the new process with a new one.

*exec (pathname, argomenti)*

- Replaces the image of the caller with the executable file *pathname*, and makes it running by passing *arguments*

```
p = fork();
if p < 0     { /* fork failed */ }
else if p=0     { /*son's code */ } exec (...)
        else     { /* parent's code*/ } ......
```

Exec is linked to the regular filesystem.

### 23.3 Exit

In Unix process terminate through the system call

```
void  exit(int status);
```

which:

- terminates the calling process;
- makes available to the parent process the value of *status*, putting it in its process table entry
- flushes and closes all open files

(The father can read *status* through the wait primitive)

This allows the father to have some control on what the childs have done. When a process terminates remains in memory until the father receives the return status so that you can be share to have all the info of all terminated processes.

### 23.4 Wait

**pid_t wait (int *status)**

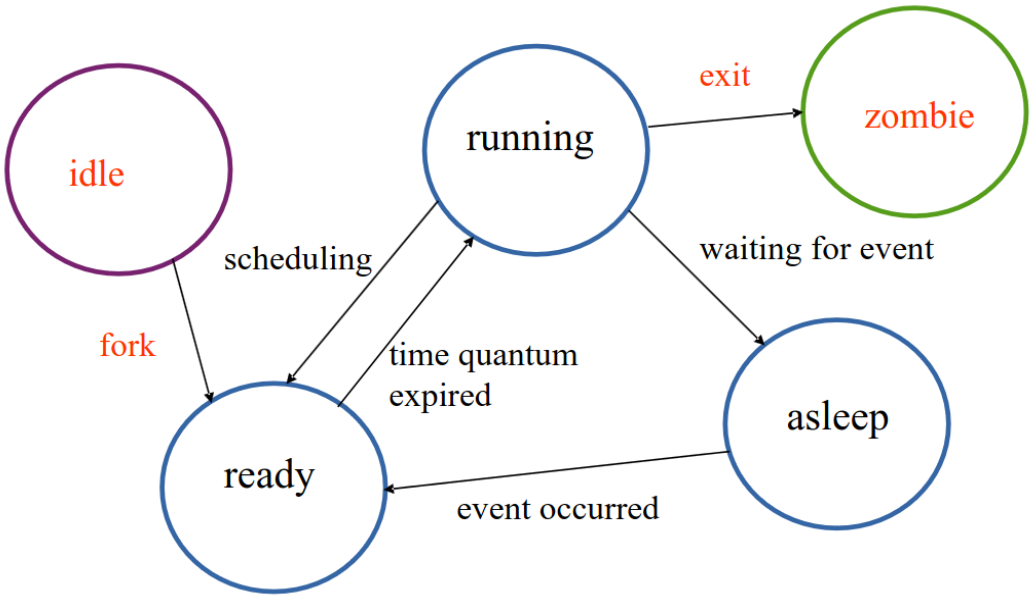
- It suspends the calling process until one of its children terminates
- returns the PID of the terminated child (in case they are more than one) or a -1 if there are no children
- assigns status the exit status of the the child
- **waitpid (…)** does the same for a specific child

# 24 Process states

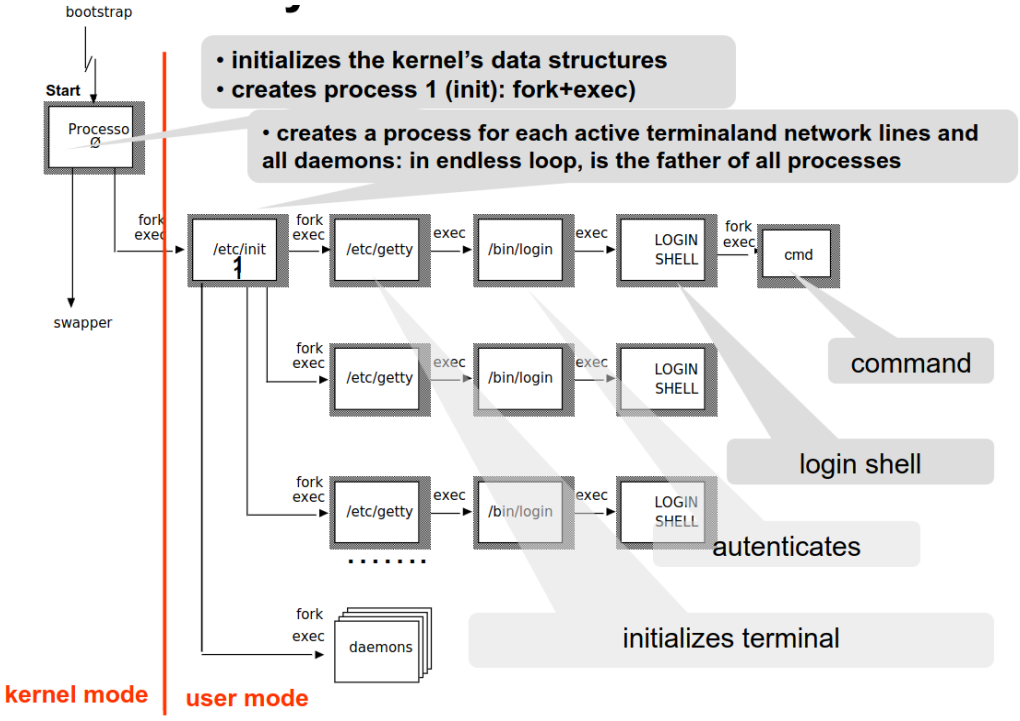
## 24.1 Zombies and orphans

- a terminated process goes into a *zombie state*, and it is finally removed after the father received his state of termination through a wait
- a zombie process occupies a minimum set
- of resources (process table entry)
- an *orphaned* process (that is, whose father
- terminated) is "adopted" by the init process,
- therefore every process always has a father

Normally in robotics programs all programs run forever. If a son is orphaned because the father dies becomes son of the init process.



# 25 System initialization




# 26 NOTE

He will upload rules to follow and on those rules we will be evauluated

# 27 System Calls

All system call return a value and usually negative numbers mean error.

## 27.1 Errno.h

If we check the errno in veery system calls it specifies the error message.

## 27.2 Perror

Converts the code in errno into a string literal. Perror should be used everywhere an error could occur. Try catch cannot be done in concurrentt applications.

### 27.3 man

- Section Description
- 1 General commands
- 2 System calls
- 3 Library functions, covering in particular the C standard library
- 4 Special files (usually devices, those found in /dev) and drivers
- 5 File formats and conventions
- 6 Games and screensavers
- 7 Miscellanea
- 8 System administration commands and daemons

```
man [number] [function_name]
```

```
apropos [function]
```

Checks inside the manual for a string and searches for all the pages that are usefull for that.

### 27.4 File primitives

- Creating files, directories, special files
- Open / Close files
- File access
- File and record locking
- Creating and destroying a link
- Reading file attributes
- Changing file attributes
- Changing the current directory
- Redirection and pipeline

Files are shared and many processes can read and write the same file at the same time. We do not have mutual exclusion. The file is a stream of data. It's an unlimited stream of numbers. File descriptor is the way in witch processes communicate. There are three standard file descriptors that are:

- 0 -> standard input
- 1 -> standard output
- 2 -> standard error

The user area in two different processes that are one forked from the other is the same and this is done to share open file descriptors. The whole hierarchy of the processes has the same file descriptors. For example when executing first he already has three file descriptors opened: 0,1 and 2. Remember that we have to close the file and the related file descriptor is erased. We are in stack memory. The standard structure to work on files is:

```
int fd;
...
fd=open(pathname, ...);

...
read(fd, ...);
...
write(fd,...);
...
close(fd);
```

**Note:**

A file can be opened more than once, so that multiple file descriptors can point to the same file

### 27.4.1 Open

```
int open (const char *pathname, int flag [,…]);
```

- opens (or creates)the file pathname,
  with policy defined by `flag`
- returns a file descriptor fur futher use (or -1 if error)

| | | |
|---|---|---|
| `flag:` | `O_RDONLY` | read-only |
| | `O_WRONLY` | write-only |
| | `O_RDWR` | read and write |
| | `O_CREAT` | create if non existing |
| | ……. | |

Flags are *bitwise* constants that can be *or-composed* ("`|`")
Additional mode specifications (`r, w, x`) can be added for user and group (see `creat`)

Most of the syscalls have variable number of parameters.

### 27.4.2 Creat

```
int creat (const char *path, mode_t mode);
```

- creates a new regular file with `path`, opened for writing
- `mode` specifies permissions; owner is the effective user-id of the process
- an existing file is emptied (owner and mode do not vary)
- returns the file descriptor, or -1 if error
- read manual for details
- a file can be created also by `open`, but:
  `creat` always creates a new empty file
  `open` preserves an existing file

Creat creats a file if it does not exists and deletes it if it already exists and then recreates it.

### 27.4.3 Close

```
int close (int fildes);
```

- closes the file descriptor `fildes`
- returns the operation's exit status (0 or -1)

**Note:**

When a process exits, all its files are closed by an implicit `close`.

### 27.4.4 Read

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- reads in `*buf` a sequence of `nbyte` bytes from current position in file `fildes`
- updates the current position
- returns the number of bytes effectively read, or -1 if error

We see files as a sequence of bytes. It's up to the programmer to decide the type of the datum to use and how many bytes are necessary.

### 27.4.5 Write

```
ssize_t write(int fildes,
  const void *buf,size_t nbyte);
```

- writes `nbyte` bytes into file `fildes` from `*buf` starting from the current position in the file
- updates the current position
- returns the number of bytes effectively written, or -1 If error

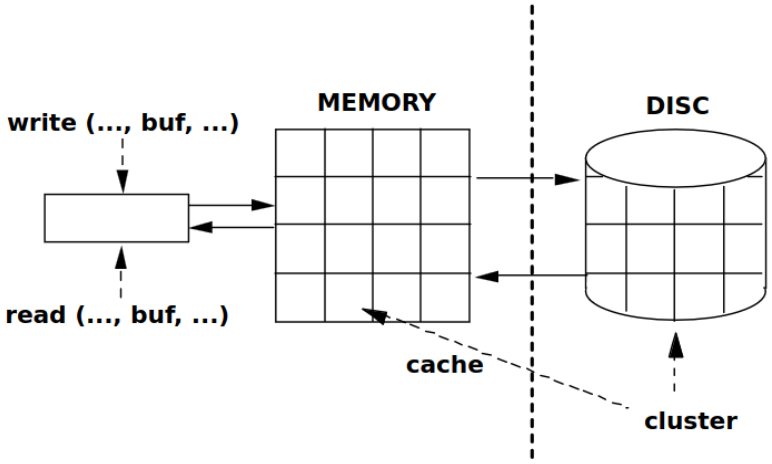
### 27.4.6 Typical example of reading and writing in stard input and outputs

```
#include ...

#define BUFFSIZE 8192

int main(void) {
    int n;
    char buf[BUFFSIZE];
    while( (n=read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            perror("main");
        if (n<0)
            perror("main");
    exit(0);
}
```

### 27.4.7 Kernel caching

The kernel strongly uses caching in the disk of the filesystem. There is a cache inside the operating system and so the writing is not executed immediately after the execution of write.

**Note:**
The kernel strongly uses caching to limit physical I/O




Caching is stopped and flushed by the "flush" system calls

### 27.4.8 fsync

- A series of POSIX system calls provide *I/O flushing,* as to say synchronizing a file's in-core state with storage device
- Different functions are available (most popular: sync, fflush, fsync and others).
- The simplest one is

```
int fsync ( int fd );
```

Where `fd` is a *file descriptor* referring to the I/O stream we want to flush, emptying any cache inside the operating system. It is typically used when writing.

If you want to immediately write in the disk then fsync is the answer even if is less efficient

### 27.4.9 Unlink

```
int unlink (const char *path);
```

- each file can have several active links
- the first link is setup by **creat**
- deletes the hard link `path` and, if it is the last link, deletes (deallocates) the file
- returns `0` or `-1`)

**Note:**

- if the file is in use, only its directory entry is deallocated; the file will be deallocated only when all its file descriptors have been closed.
- an executable can make a `link` to itself to prevent someone to cancel its file during execution

### 27.4.10 Locking and unlocking files

- locking / unlocking in POSIX is complex
- several system calls are available

  `fcntl(), flock() lockf()`
- The simplest system call is:

```
int flock(int fd, int operation);
```

### 27.4.11 File descriptor vs streams

- C language has a *standard library* for file management based on the type `FILE`
- `FILE` is a type of a "file pointer" (or **stream**) variable

  ```
  FILE * fp;
  ```

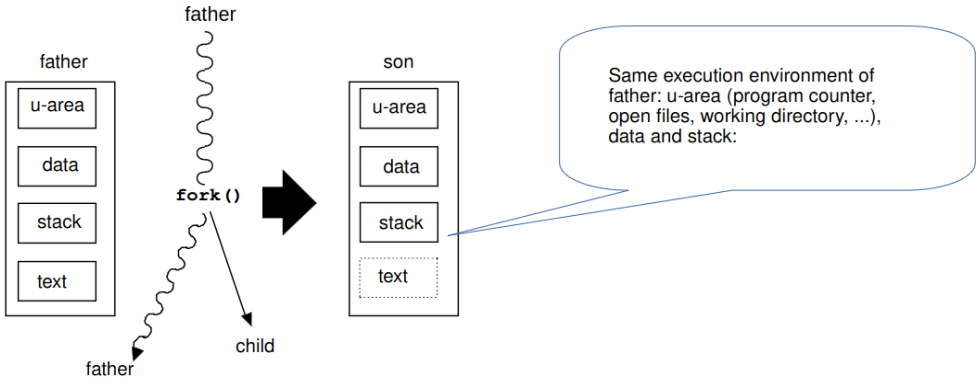
  which **is not** a file descriptor.
- A stream can be opened, closed, read and written by specific C standard library functions (they all are prefixed by "`f`"):

  ```
  fopen(), fclose(), fread(), fwrite(),
  fscanf(), fprintf()
  ```

The pointer is not a file descriptor but a stream. They are very similar and is also possible to have the pointer from the descriptor.

### 27.4.12 Fork

- A generic POSIX process is created by a *parent* process by the **fork** system call




Text in general is not duplicated because there there can be multiple processes.

```
pid_t fork (void);
```

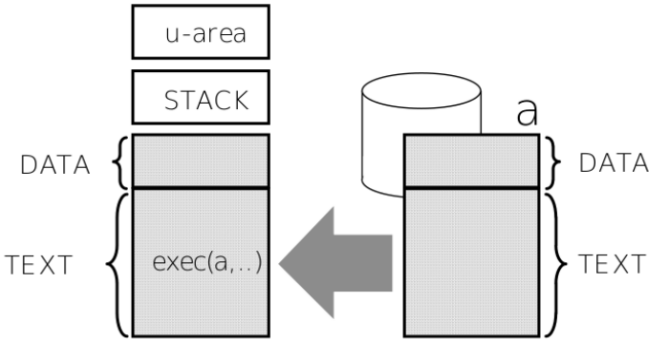
- creates a *child* process "identical" to the parent, and returns:

| | |
|---|---|
| to the child: | `0` |
| to the parent: | PID of the child |
| in case of error: | `-1` |

### 27.4.13 Exec

```
pid_t exec (pathname, arguments)
```

- substitutes the caller's image with the executable file `pathname`, and executes it transferring `arguments`




U-area and stack are the same while the rest is replaced

```
int execl ( const char *pathname, const char *arg0,... );      … list
int execv ( const char *pathname, char *const argv[] );        … vector
int execle ( const char *pathname, const char *arg0, ...,
             char *const envp[] );                             … environment
int execve ( const char *pathname, char *const argv[],
             char *const envp[] );
int execlp ( const char *filename, const char *arg0, ... );    … path
int execvp ( const char *filename, char *const argv[] );
```

Environment is very important. Environment are a series of strings in POSIX. String with values and these values can be used or ignored completely by your executable. These environment are a series of global values that every process can use.

### 27.4.14 Sleep

- in several situations a process may require rescheduling for a certain time amount
- the **sleep** family of system calls is available at this purpose
- a sleep action moves the process to the *waiting state* for a specified time interval; after that, the process goes to *ready state*
- the family is composed by three system calls with different time scales: **sleep** (seconds), **usleep** (microseconds) and **nanosleep** (nanoseconds)
- very short intervals may use *busy waiting* instead of rescheduling (no *switcher* action)

Nanosleep will use busy waiting if you select a very low amount of nanoseconds.

### 27.5 Pipes

- **Unnamed pipes:**
  - Producer / consumer model using kernel-internal FIFO channels
- **Named pipes:**
  - As above, using FIFO channels in the file system

For the first two homeworks we have used named pipes. The name of the pipe was the name of the file. In FIFOs what you read is consumed and what you write is pushed. FIFO is very efficient. Consuming is blocked in case there is no data in the FIFO. In case the FIFO is full the producer is blocked.

### 27.6 Select

This function selects the pipe that has data. This allows non blocking structures. In Posix all messages are blocking.

```
fd1 = open (pipe 1), fd2 =  open (pipe2);
while (TRUE) {
   select fd1 or fd2 or both
   switch (data available?) {
   case (fd1 has data && fd2 has data):
      choose randomly fd, read (fd, data), break
   case (fd1 has data):
      fd = fd1, read (fd, data), break
   case (fd2 has data):
      fd = fd2, read (fd, data), break
   default: suspend execution for dt seconds
   }
}
```

You suspend for some seconds to avoid doing a busy waiting. The select may also interrupt sleeping as soon as data arrives. Select is the only way to get messages from more producers. Non-determinism is related to non blocking and fairness, meaning that we don't want to privilege some of the processes. Most of the time the privilege should be decided with a coin toss. Another concept is the busy or not busy waiting meaning that we can wait before running the select again or stay asleep until some other signal awakes it. This costs something in the implementation of the kernel.

- Another critical point is
    - `suspend execution for dt seconds`
- Suspending is necessary to avoid a continuous looping with multiple system calls: this would lead to an abnormal cpu usage
- Suspending can be achieved by two methods:
  - *Busy waiting*: a time-wasting loop is repeated a certain number of times until the desired delay `dt` is obtained
  - *Kernel suspension*, in which the process is set in the waiting state for a dt interval; the scheduler puts the process in ready state after `dt`. In Posix this is normally obtained using a `sleep(), usleep(), nanosleep()` syscall.
- Busy waiting must be avoided because of its inefficiency.

## 28 Pipes again

Please use unnamed pipes.

## 29 Assigment 1

NB The master will generate all the pipes that will be inherited by the childs
19/10

## 30 Signals

- Signals may be thought as software equivalent of hardware **interrupts**
- The kernel manages a signal vector table *for each process*
- Conversely, for hardware interrupts there is a *single* system-wide table
- POSIX defines several signals (with names defined as integers)
- The **default action** performed to handle a signal is to **terminate** the process
- It is possible to register a custom a **signal handler** for each signal

There are different signals and they are indexed by number. Standard posix signals are up to 30. A process is killed by default for safety reasons when it receives a signal. To receive a signal you should use:

```
sigaction();
```

| NAME | NBER | DEFAULT ACTION | COMMENTS |
|---|---|---|---|
| SIGABRT | 6 | Terminate (core dump) | Process abort signal |
| SIGALRM | 14 | Terminate | Alarm clock |
| SIGFPE | 8 | Terminate (core dump) | Erroneous arithmetic operation |
| SIGHUP | 1 | Terminate | Hangup |
| SIGILL | 4 | Terminate (core dump) | Illegal instruction |
| SIGINT | 2 | Terminate | Terminal interrupt signal |
| SIGKILL | 9 | Terminate | Kill (cannot be caught or ignored) |
| SIGPIPE | 13 | Terminate | Write on a pipe with no one to read it (broken pipe) |
| SIGQUIT | 3 | Terminate (core dump) | Terminal quit signal |
| SIGSEGV | 11 | Terminate (core dump) | Invalid memory reference |
| SIGTERM | 15 | Terminate | Termination signal |
| SIGTRAP | 5 | Terminate (core dump) | Trace/breakpoint trap |
| SIGUSR1 | N/A | Terminate | User-defined signal 1 |
| SIGUSR2 | N/A | Terminate | User-defined signal 2 |

| NAME | NBER | DEFAULT ACTION | COMMENTS |
|---|---|---|---|
| SIGSTOP | 19 | Suspend | Puts the process in wait state (cannot be caught or ignored) |
| SIGCONT | 18 | Resume | Puts the process in ready state (cannot be caught or ignored) |
| | | | |
| | | | Signal numbers may vary with architectures |
| | | | SIGSTOP AND SIGCONT are used for |
| | | | scheduling processes |
| | | | The shell uses them for controlling |
| | | | foreground/background execution |

### 30.1 Final remarks

- Signals *seem* powerful: they are, but with strict logical limitations
- Signals could replace any form of IPC but...
- They must be reserved for **specific, single, urgent actions**, such as errors, interruptions, abortions, stops, restarts, reports, emergencies and the like
- The main limitation is: **signals are not reentrant** (they are in real time kernel)
- "Nested" signal handlers cannot be managed, and signals can interfere with kernel actions, such as *select*
- The only way to make them (enough) safe, i.e. deterministic, is to *serialize* them via masking; it is an action logically similar to mutual exclusion. However, this limits the signals to the specific actions mentioned above.

If you receive a signal you should stop receiving signals until you have finished doing what you have to do. Signals are not queued.

## 31 Implementing a watchdog

The best way to implement a watchdog is to send a signal to the childs and see if they react. If they react then they are alive.

## 32 NOTE

- For select always use sleeping time because is usefull to reduce stress on the system.
- In select you could store ready fds and then scramble them before serving them to not have priority between them. This is non determinism.

## 33 Real time

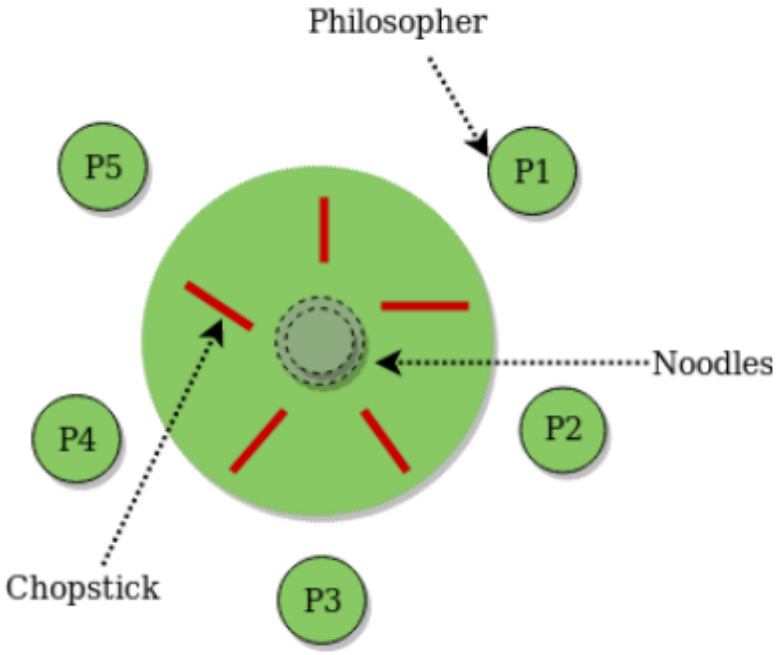
In real time signal handling is very usefull and there are very high amount of signal that can be used by the user.

## 34 Signals

A process that reacts to a signal is alive.

# 35 Theory problems for concurrent programming

## 35.1 Dining philosopher's problem




Dining Philosophers Problem The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

```
process P[i]
while(True){
    THINK;
    PICKUP(CHOPSTICK[i], CHOPSTICK[(i+1)%5]);
    EAT;
    PUTDOWN(CHOPSTICK[i], CHOPSTICK[(i+1)%5]);
}
```

An arrows indicates what process has some resource. The arrow goes from the process to the possessed resource. Another useful graph is the one representing on what processes other processes are waiting. So the arrows go from the process that is waiting to the process that has the resource. If in the final graph there is a cycle then a deadlock is created. In this case the watchdog becomes useful. The watchdog can detect deadlock and also hardware failures. There is also the problem of **starvation**. Processes work as expected but some of them could never be executed, maybe for a lower priority or something like this.

A proposed solution to the deadlock problem in this case is to release the forks if a philosopher is not able to acquire both of them. This could cause starvation. To avoid even this a good solution would be to add random waiting times for the processes.

Another idea could be to impose a maximum number of process to run at the same time. NB there will be questions about this problem