Processes in UNIX

Emmanuel Fleury
B1-201
fleury@cs.aau.dk

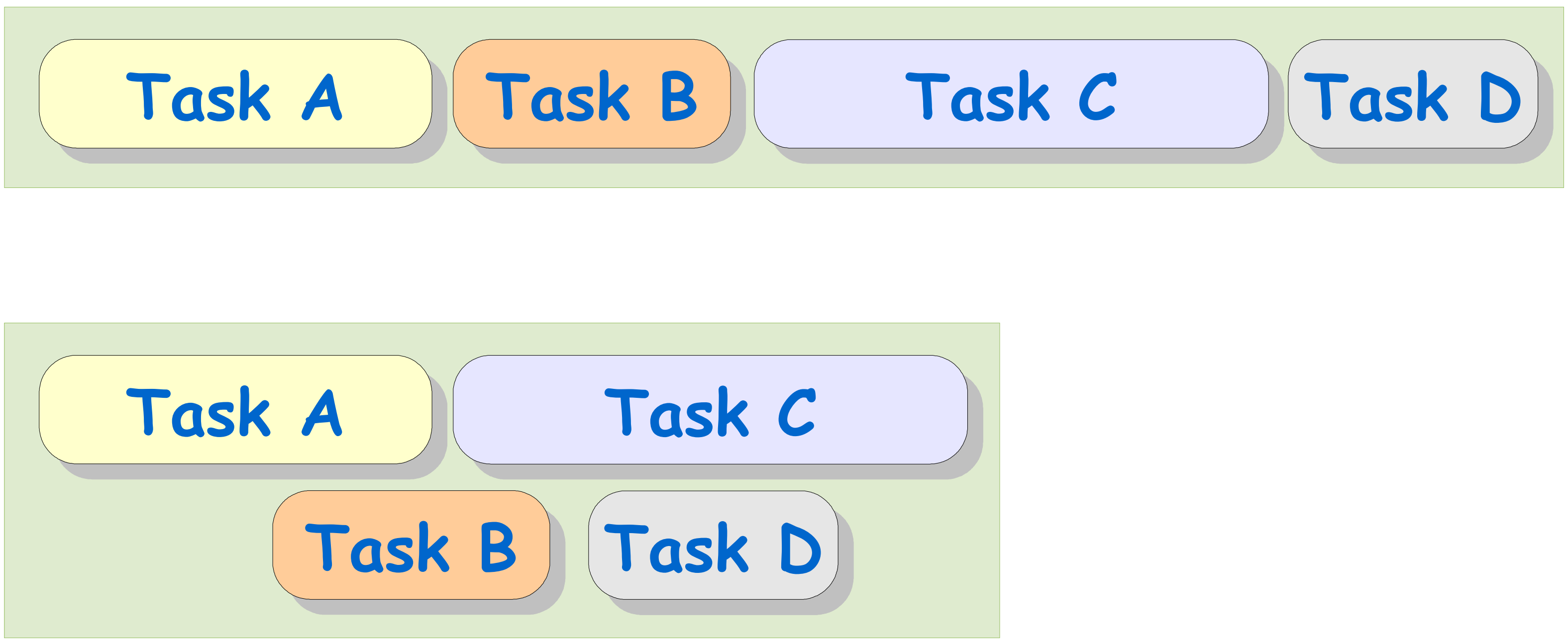
Overview

- Motivations
- Problems in Concurrency
- A Process Life
- Interlude
- Process Management
- Process Scheduling

Motivations

Basic Idea

Do several things at once !



Efficiency, Responsiveness, Scalability, ...

Concurrent People...

A natural thing ?

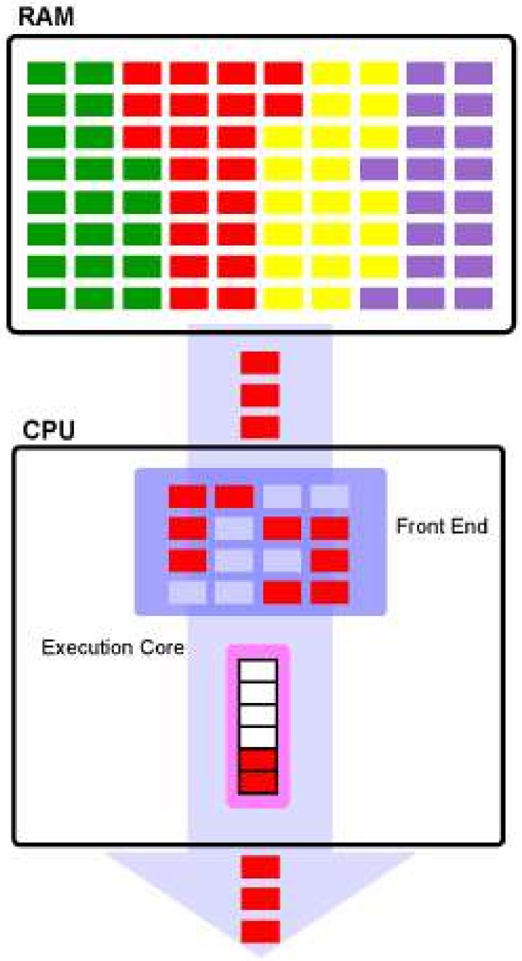
... and Concurrent Machines...

More and more parallelism !!!

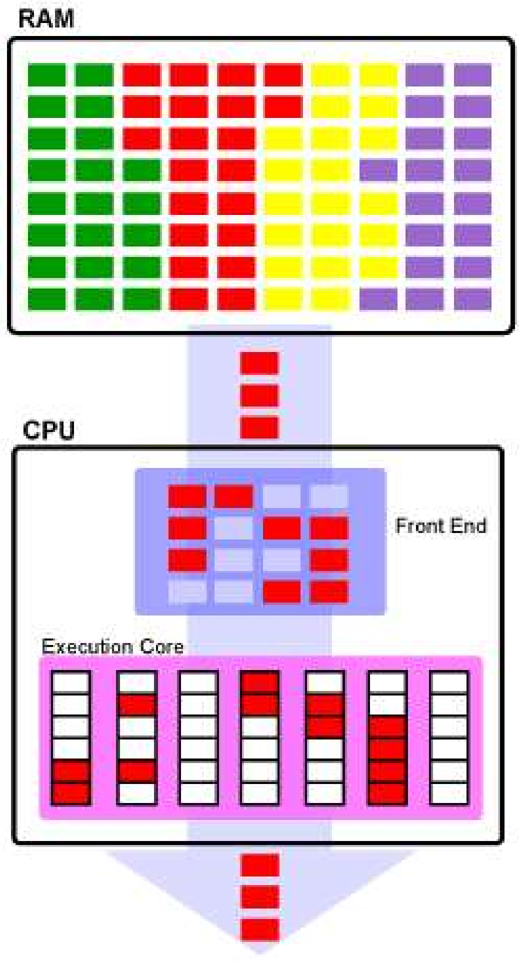
- [1970s] Symmetric Multi-Processing (SMP)
- [1980s] Super-scalar Processors
- [1990s] Hyper-threading Processors
- [2000s] Cell Processors
- ... and other will follow ....

Super-scalar Processors

- Implement Parallelism on a single chip
- Dispatch the tasks among several processing units
- Efficiency strongly depends on the dispatcher
- Difficult to compute the WCET (Worst Case Execution Time)



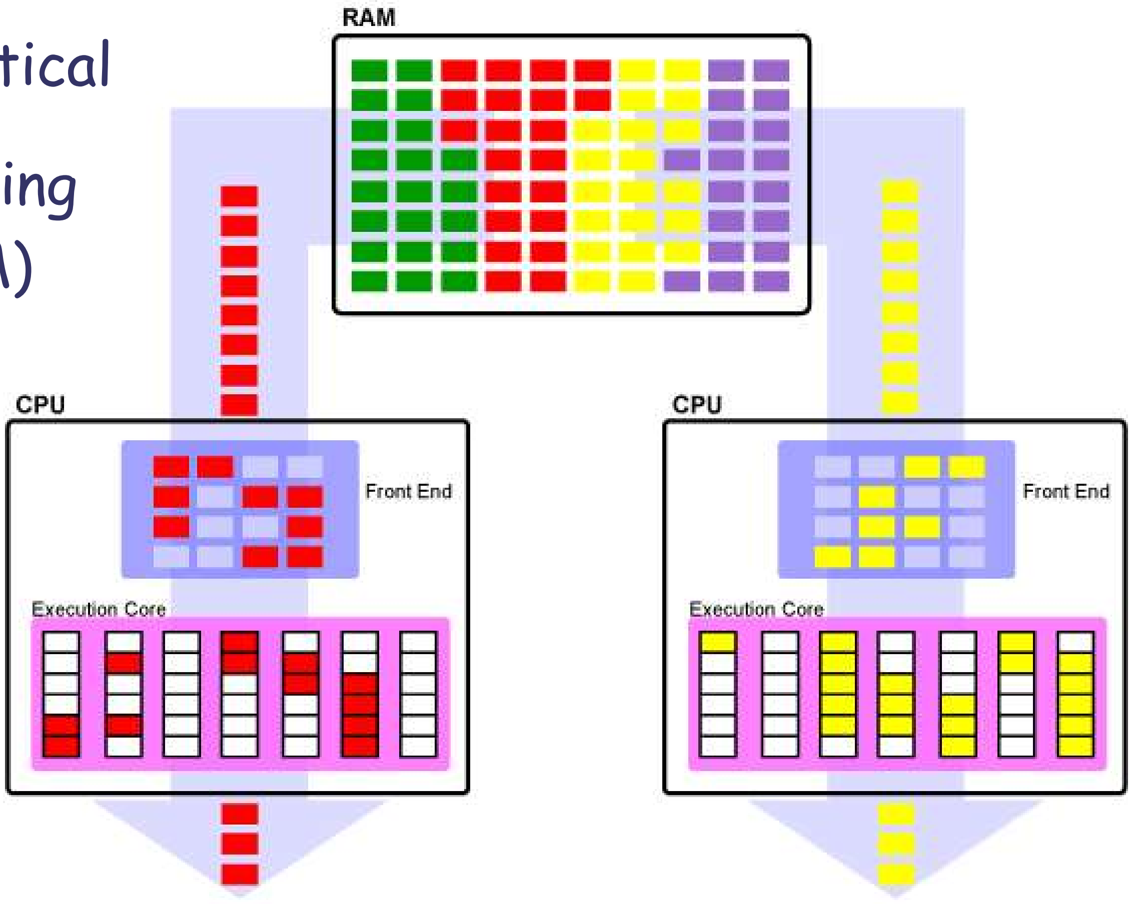
Scalar Processor



Super-Scalar Processor

Symmetric Multi-Processing

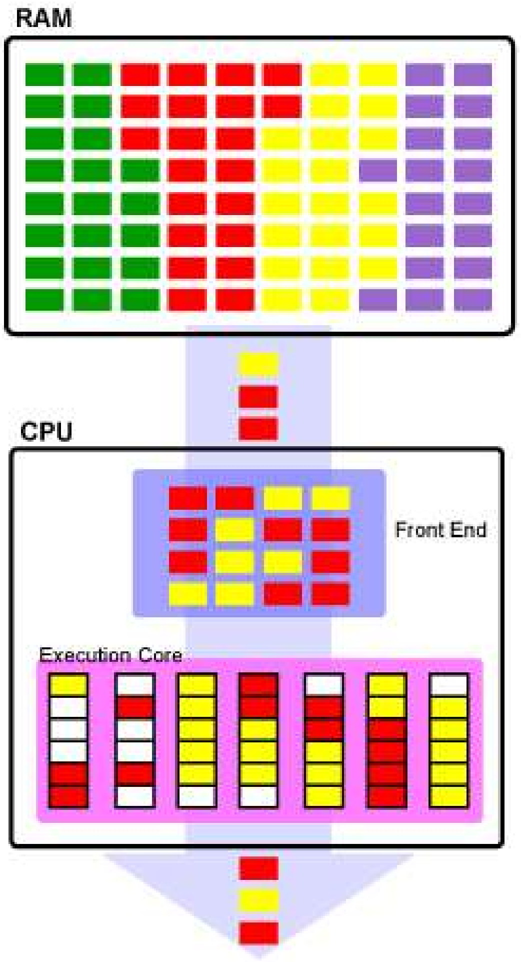
- Multi-processors machines
- All processors are identical
- All processors are sharing the same memory (RAM)
- Multi-processing is for real !!
- Now a days a lot of desktops use this technology



Symmetric Multi-processing

Hyper-Threading

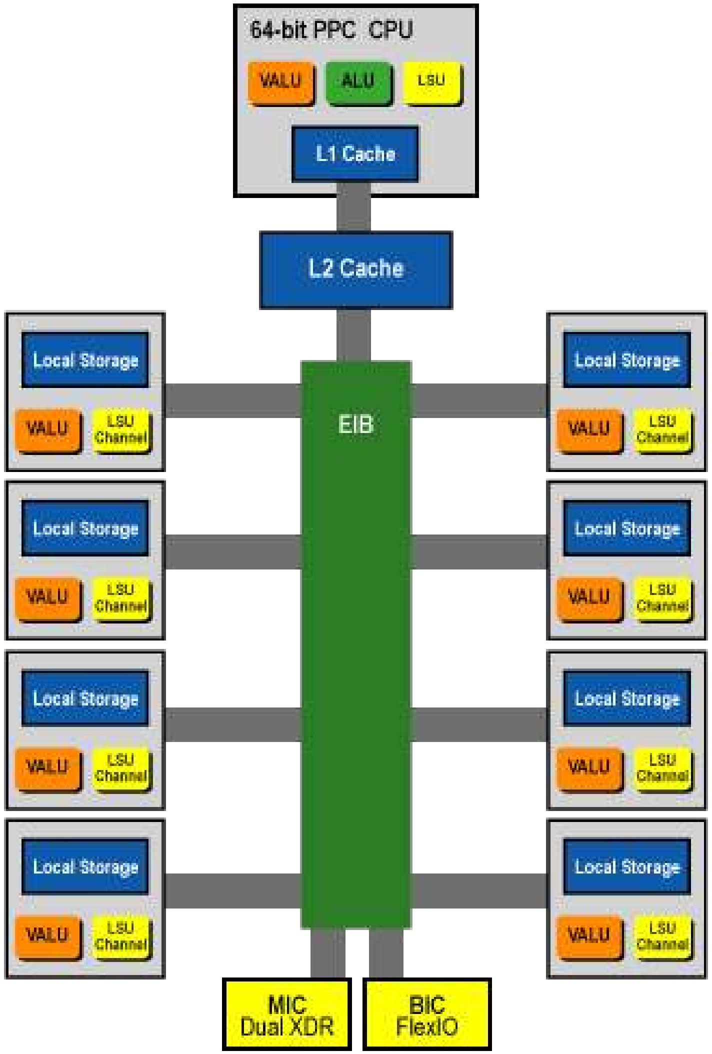
- Emulate two processors in one through the dispatcher
- Improve efficiency and reactivity to multi-processed and/or multi-threaded programs
- Performance improvements of 15-30%
- Tend to be more and more common



Hyper-Threading

Cell Processors

- Developed by IBM in cooperation with Toshiba and Sony (Playstation 3).
- Highly Parallel Architecture
- Designed for streaming (audio, video)
- Main Components:
  - 1 Processing Element (PE)
  - 1 Element Interconnection Bus (EIB)
  - 8 Synergistic Processing Units (SPU)



The CELL Architecture

... in a Concurrent World !

- Applications can spread over networks (Internet)
- Scalability go often through the use of others machines
- Multi-processed and/or Multi-threaded applications can better be adapted for networks

Conclusion ?

Think concurrent !

- The users are more and more familiar with multi-tasking
- The hardware tend to handle more and more efficiently concurrent programs
- The networks are growing in bandwidth and power
- The operating systems as well (schedulers)

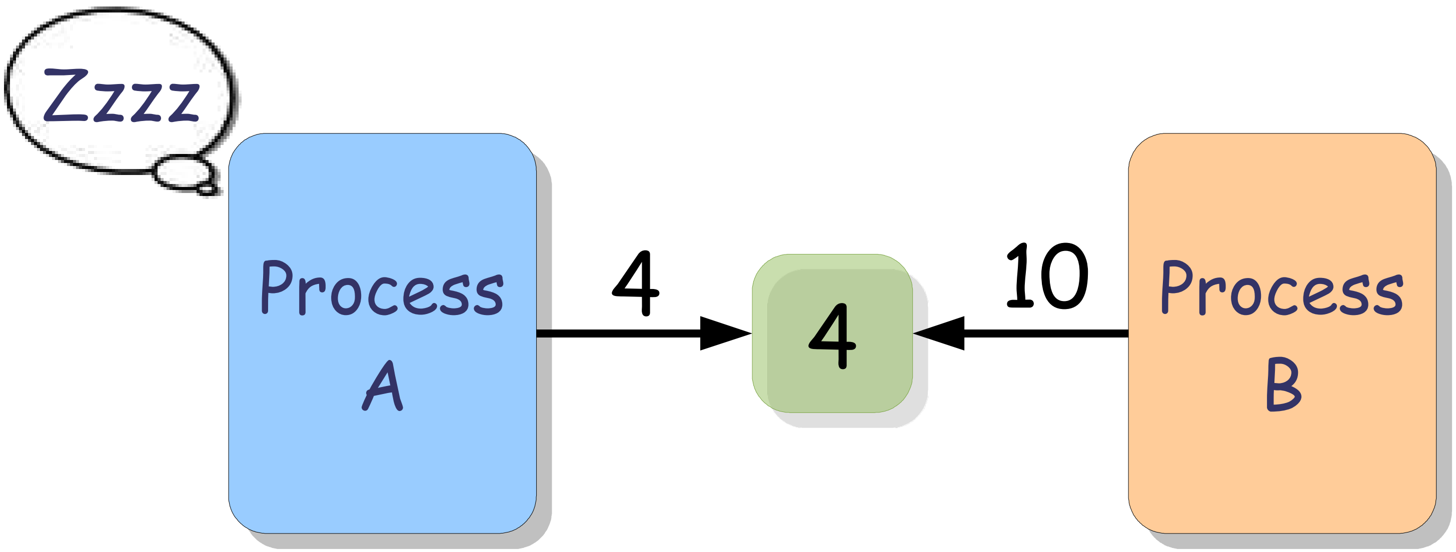
Common Problems in Concurrent Programming

Concurrency is good but ...

- **Atomicity**
  - Non-interference problems (race conditions)
- **Synchronization**
  - Rendez-vous problems (deadlocks, livelocks)
- **Mutual Exclusion**
  - Critical sections problems (starvation)

Atomicity

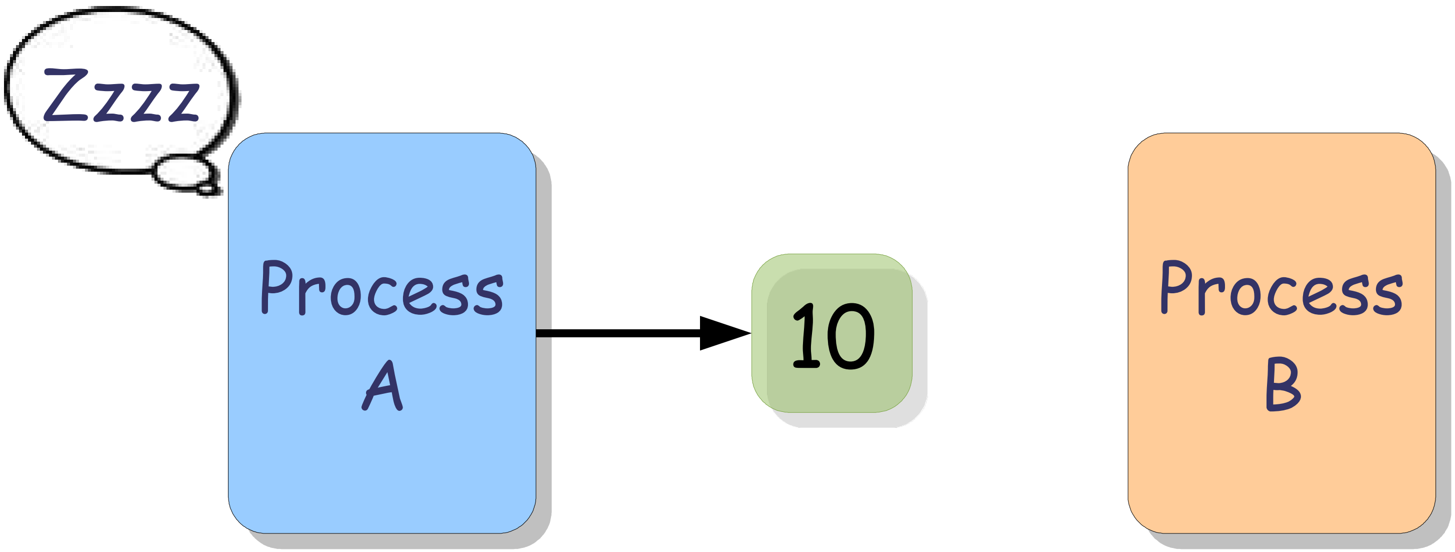
An operation is said **Atomic** if the result can be observed before its termination



Race Condition

Atomicity

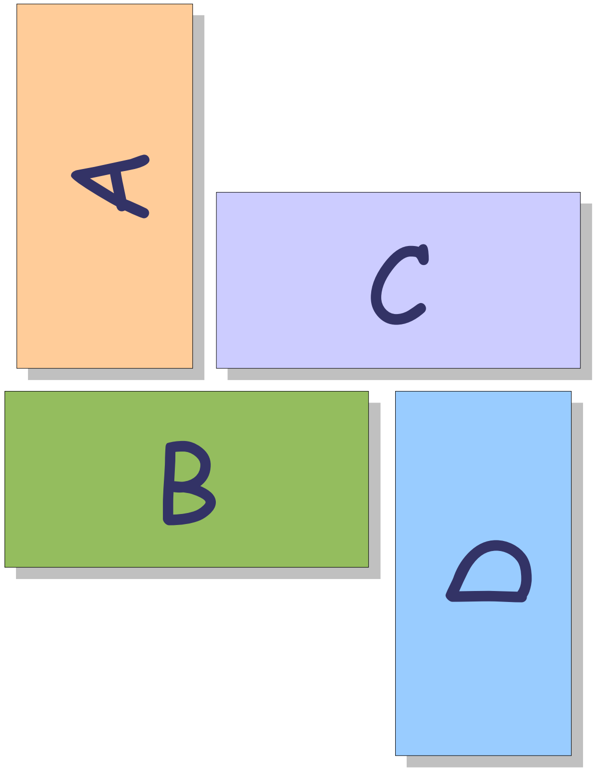
An operation is said **Atomic** if the result can be observed before its termination



Race Condition

Synchronization

Two processes are **synchronized** when they can exchange some informations

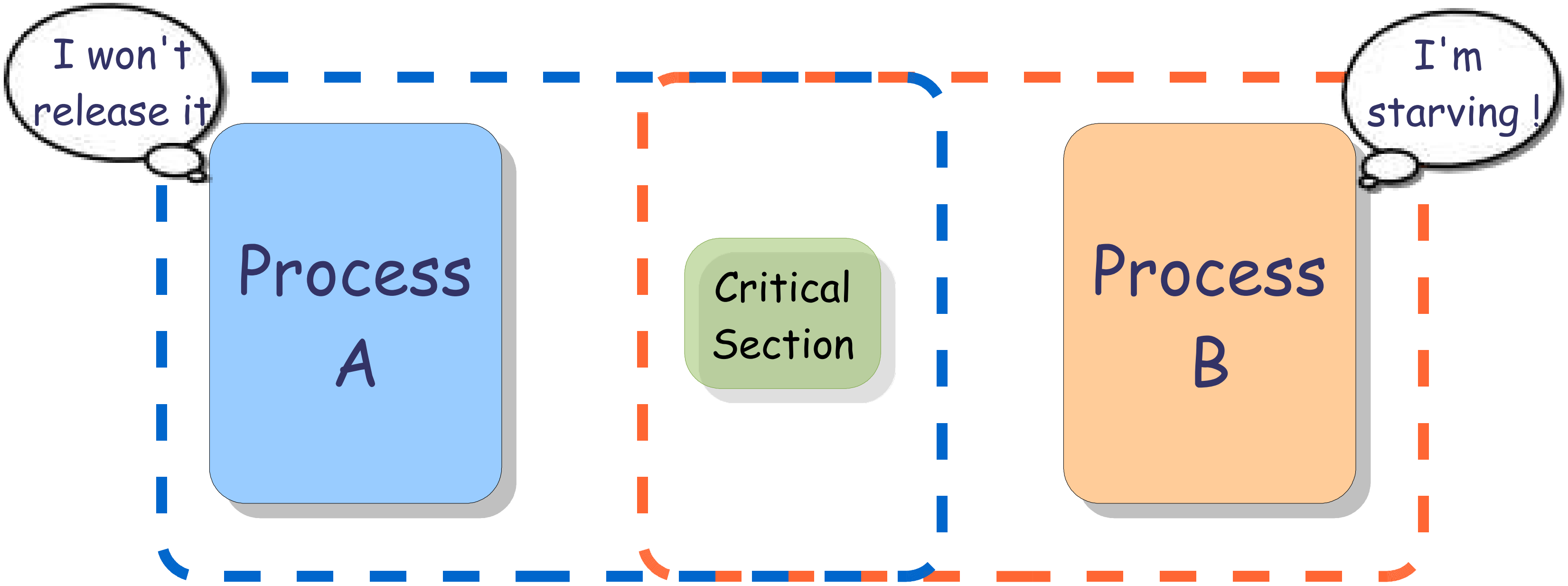


Deadlock

Livelock

Mutual Exclusion

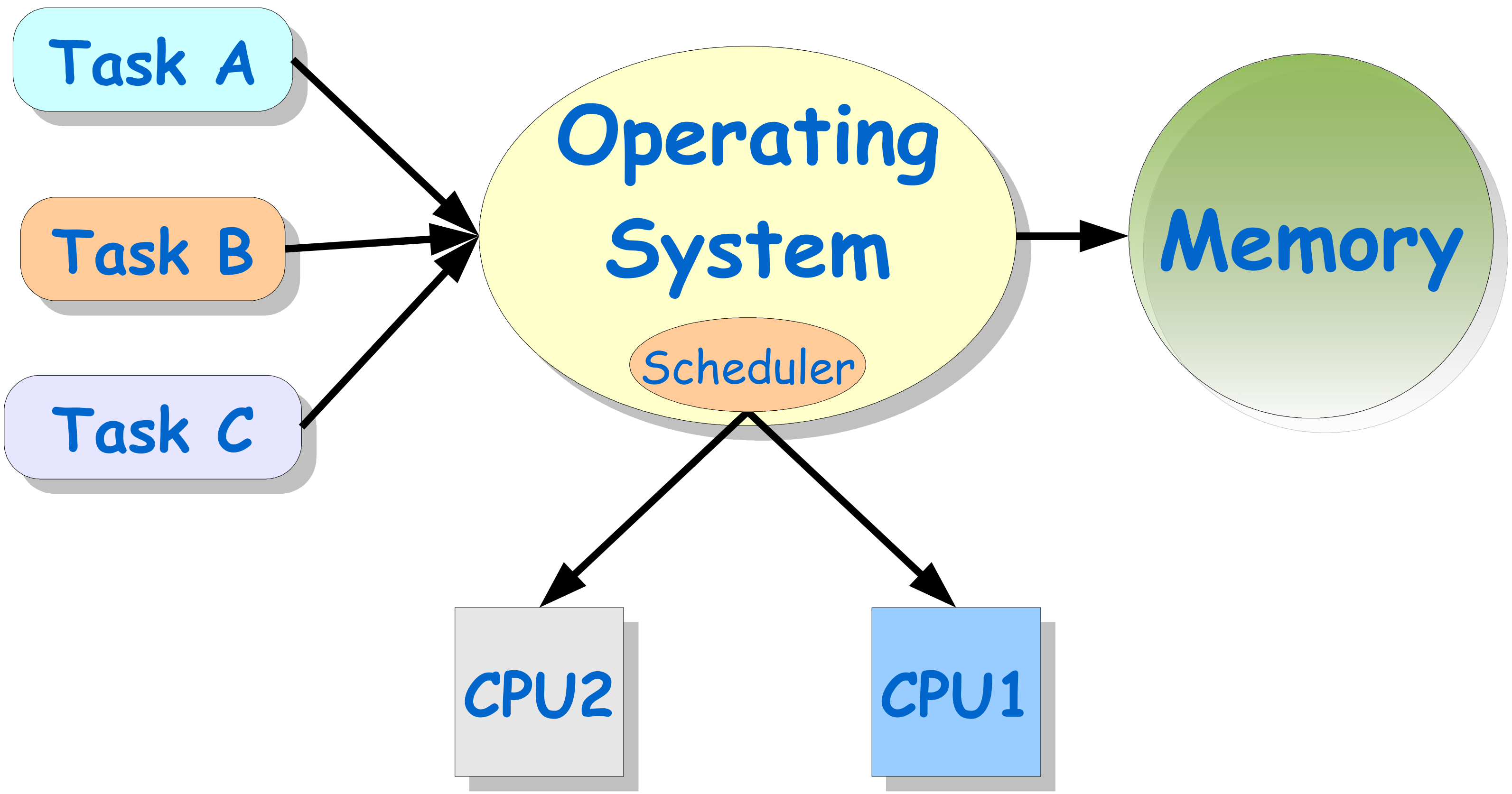
Two processes are **synchronized** when they can exchange some informations



Starvation

A Process Life

The Role of an OS



Brings Abstraction from the Hardware !

The Role of the OS

The programmer can assume:

1. Unlimited Resources
   (CPU, Memory, ...)

2. Each task is protected from the others
   (Execution, Memory, ...)

3. Access to the Resources is "fair"
   (No starvation induced by the Scheduler)

Why Processes ?

- **More Tasks than Processors**

  The Scheduler needs to split tasks into smaller units that can be executed on the processor(s) one after one.

- **Making it easy for the Programmer**

  The Scheduler make believe to each task that it is the only one on the machine (errors in a task won't interfere with others).

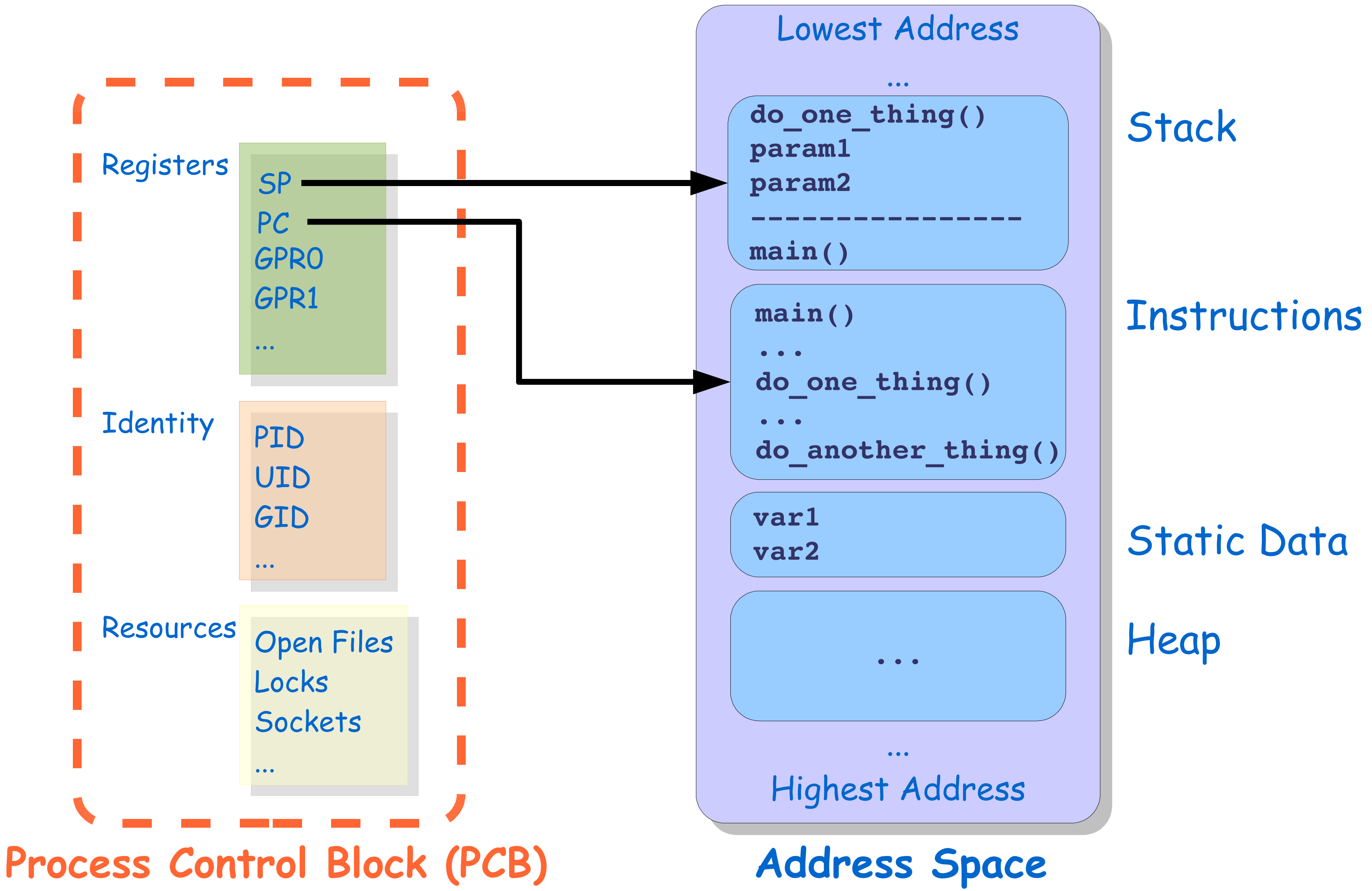
- **Prioritizing Programs**

  The Scheduler provides some control from user-space on how often is executed one task and, more generally, what resources does it takes.

What is a Process ?

- One of the two main abstraction of Unix
  (the other one is "everything is a file")
- A Process is the biggest processing unit that can be scheduled
  (the smallest are the threads)
- A Process always spawn from another one
  (except the process `init`)

Process Internal



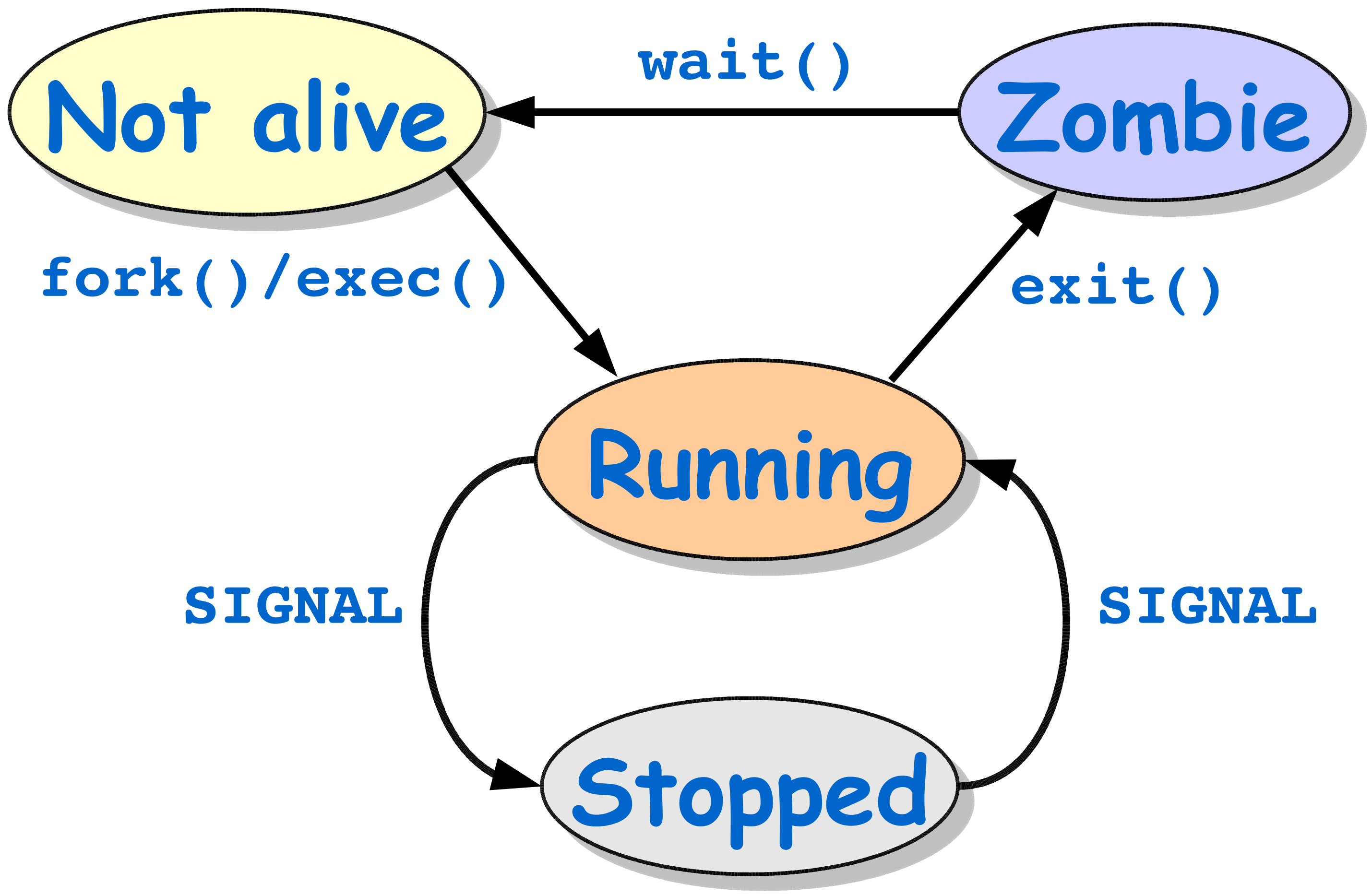
Process Control Block (PCB)

- PID (Process ID): Task's unique process ID, which periodically wraps, though never restarting at zero.
- PPID (Parent Process ID): Process ID of a task's parent.
- UID (User ID): Effective user ID of the task's owner.
- USER (User Name): Effective user name of the task's owner.
- GROUP (Group Name): Effective group name of the task's owner.
- PR (Priority): Priority of the task.
- NI (Nice value): Nice value of the task. A negative nice value means higher priority, whereas a positive nice value means lower priority.

The command "top"

```
top - 00:36:43 up 16:49,  5 users,  load average: 0.91, 0.60, 0.32
Tasks:  76 total,   1 running,  75 sleeping,   0 stopped,   0 zombie
Cpu(s):  6% us,  1.0% sy,  0.0% ni, 92% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    507576k total,   495948k used,    11628k free,    20016k buffers
Swap:   497972k total,     4184k used,   493788k free,   212796k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4299 root       5 -10  202m  69m 6884 S  2.7 14.0  15:55.13 XFree86
 5363 fleury    15   0 93896  42m  10m S  1.7  8.5   8:20.37 rhythmbox
 4671 fleury    15   0 30752  14m 7780 S  1.3  2.9   0:56.37 terminal
 4665 fleury    16   0 11556 7476 5900 S  0.7  1.5   0:46.25 metacity
    1 root      16   0  1500  516  456 S  0.0  0.1   0:00.47 init
    2 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    3 root       5 -10     0    0    0 S  0.0  0.0   0:00.73 events/0
    4 root       5 -10     0    0    0 S  0.0  0.0   0:00.00 khelper
   16 root      15 -10     0    0    0 S  0.0  0.0   0:00.00 kacpid
  107 root       5 -10     0    0    0 S  0.0  0.0   0:00.10 kblockd/0
  120 root      15   0     0    0    0 S  0.0  0.0   0:00.00 khubd
  197 root      15   0     0    0    0 S  0.0  0.0   0:00.02 pdflush
  198 root      15   0     0    0    0 S  0.0  0.0   0:00.03 pdflush
  200 root      15 -10     0    0    0 S  0.0  0.0   0:00.00 aio/0
```

Process States

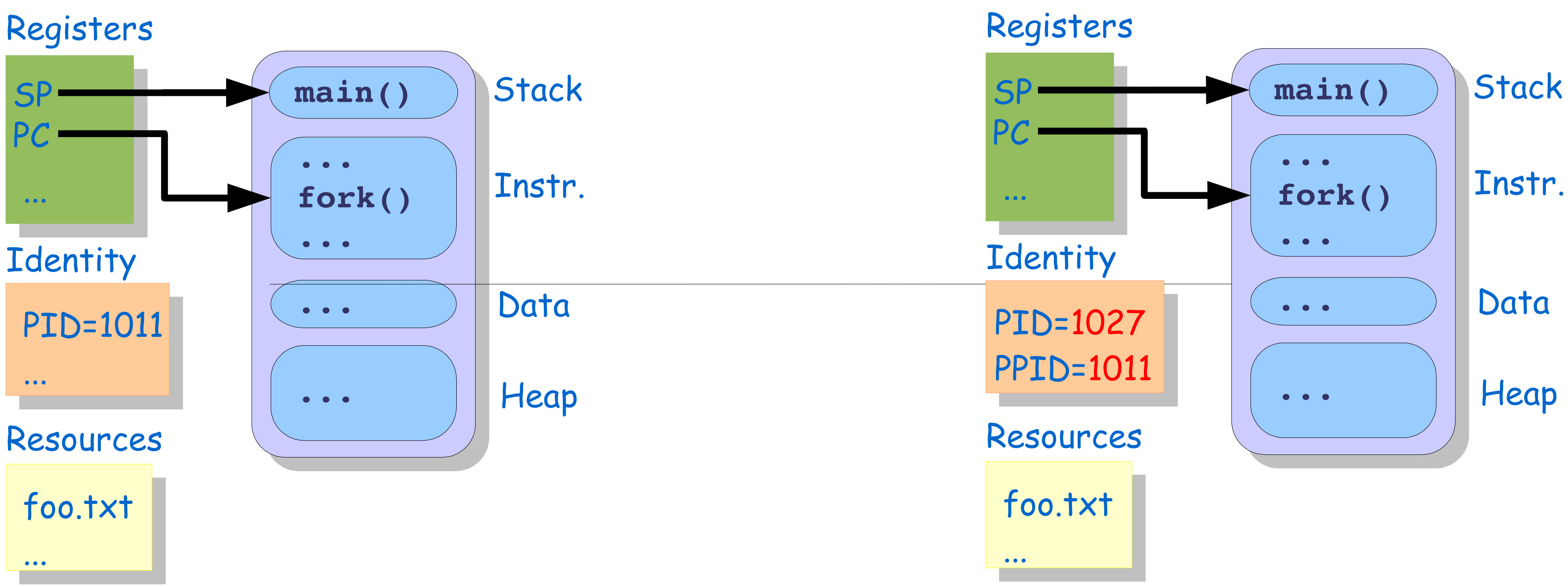


Creation of a Process

- System Call fork():
  - Creates (nearly) identical copy of process
  - Return value different for child/parent

- System Call exec():
  - Over-write with new process memory
  - Return value is 0 for success and 1 for failure

Creation of a Process (fork)



Return Value:

- In Parent Process: "**Child Process ID**" or "**-1**" (on failure)
- In Child Process: "**0**" (always)

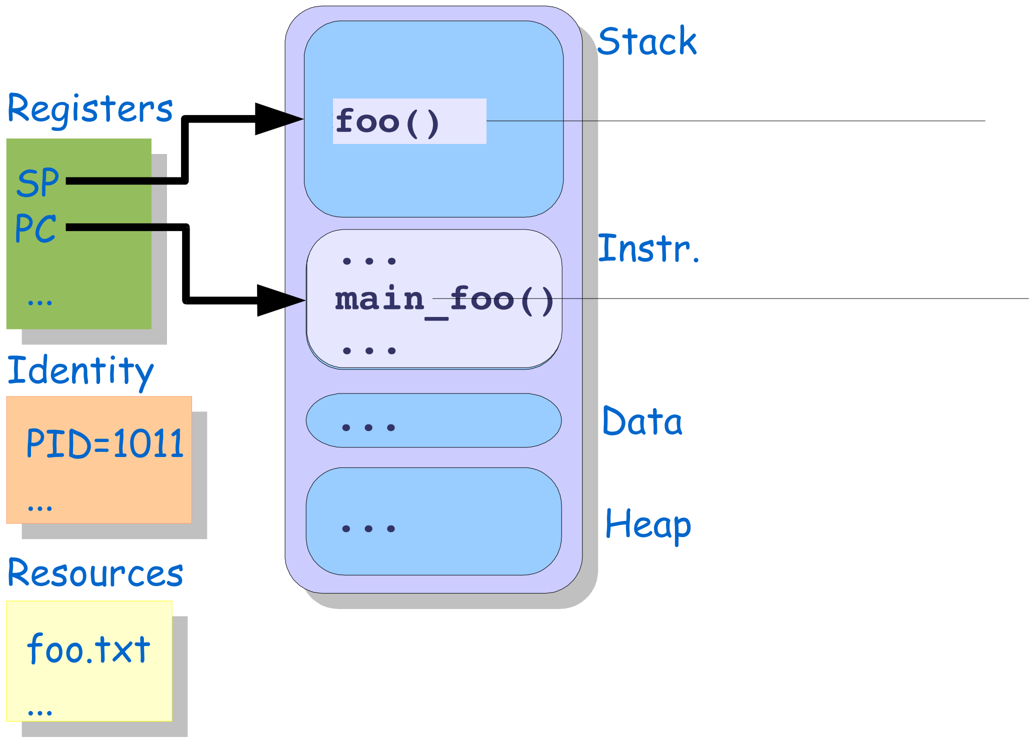
fork()

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

int main() {
  pid_t pid;

  switch(pid = fork())
    {
    case -1: /* Failure */
      perror("forking");
      exit(1);

    case 0: /* Child code */
      printf("Child is running\n");
      exit(0);

    default: /* Parent Code */
      printf("Parent is running\n");
      exit(0);
    }
}
```

```
[fleury@hermes]$ ./forking
Child is running
Parent is running
[fleury@hermes]$
```

Note: The Linux kernel run **always** the child first.

Creation of a Process (exec)



Return Value:

- "**-1**" (on failure) and "**errno**" is set to the error number
- Does <u>**not return**</u> on success

execve()

```
[fleury@hermes]$ ./executing
total 36
-rwxr-xr-x  1 fleury fleury 12K Mar 25 17:54 executing
-rw-r--r--  1 fleury fleury 286 Mar 25 17:54 executing.c
-rwxr-xr-x  1 fleury fleury 13K Mar 25 18:10 forking
-rw-------  1 fleury fleury 371 Mar 25 18:10 forking.c
[fleury@hermes]$
```

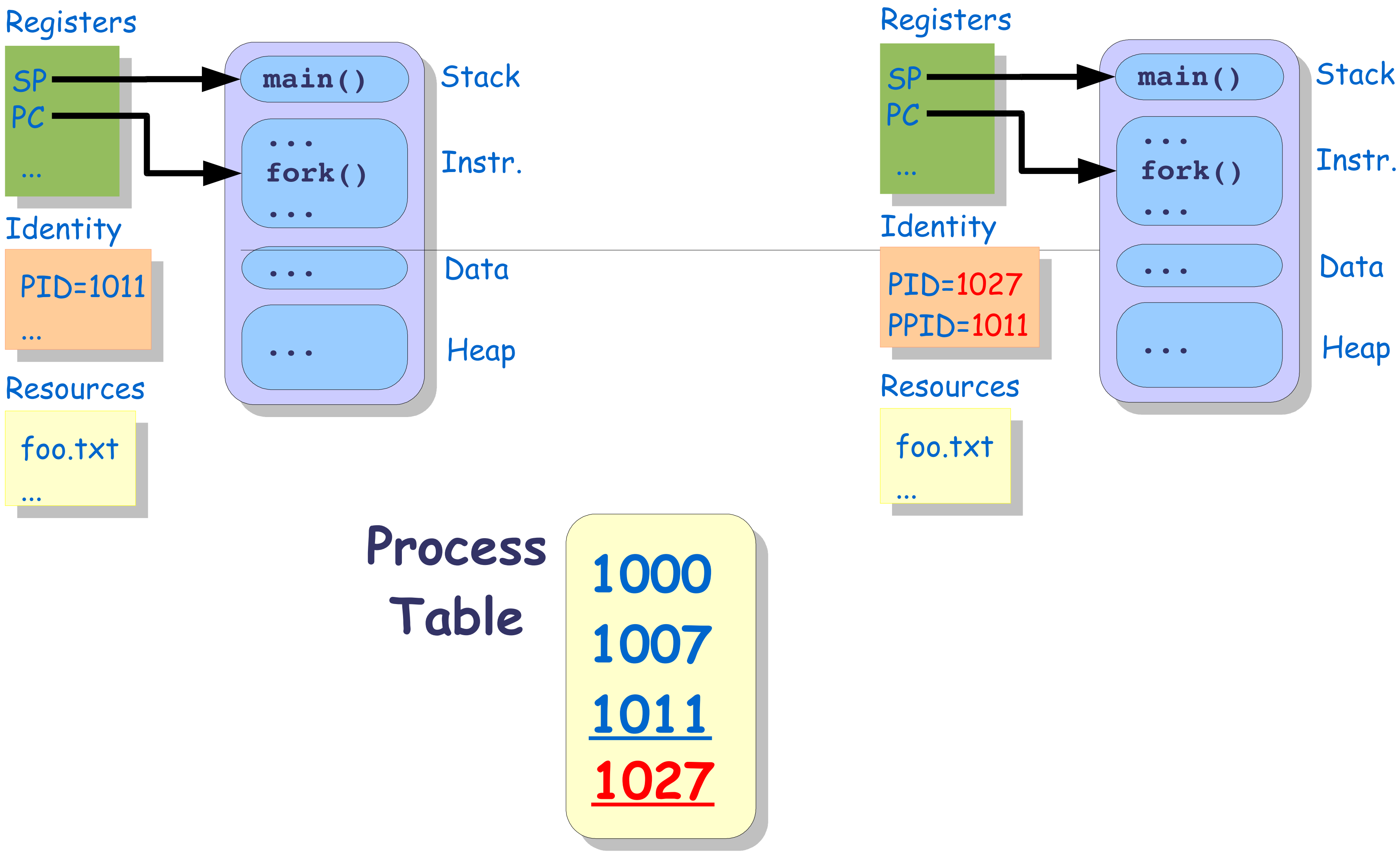
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
  char *cmd[] = {"ls", "-lh", (char *)NULL};
  char *env[] = {"HOME=/usr/home", "LOGNAME=home", (char *)0};

  if (execve("/bin/ls", cmd, env)) {
    perror("foo");
    exit(1); /* Failure */
  }

  printf("I'm still alive !\n");
  exit(0);
}
```

Note: This code is never executed.

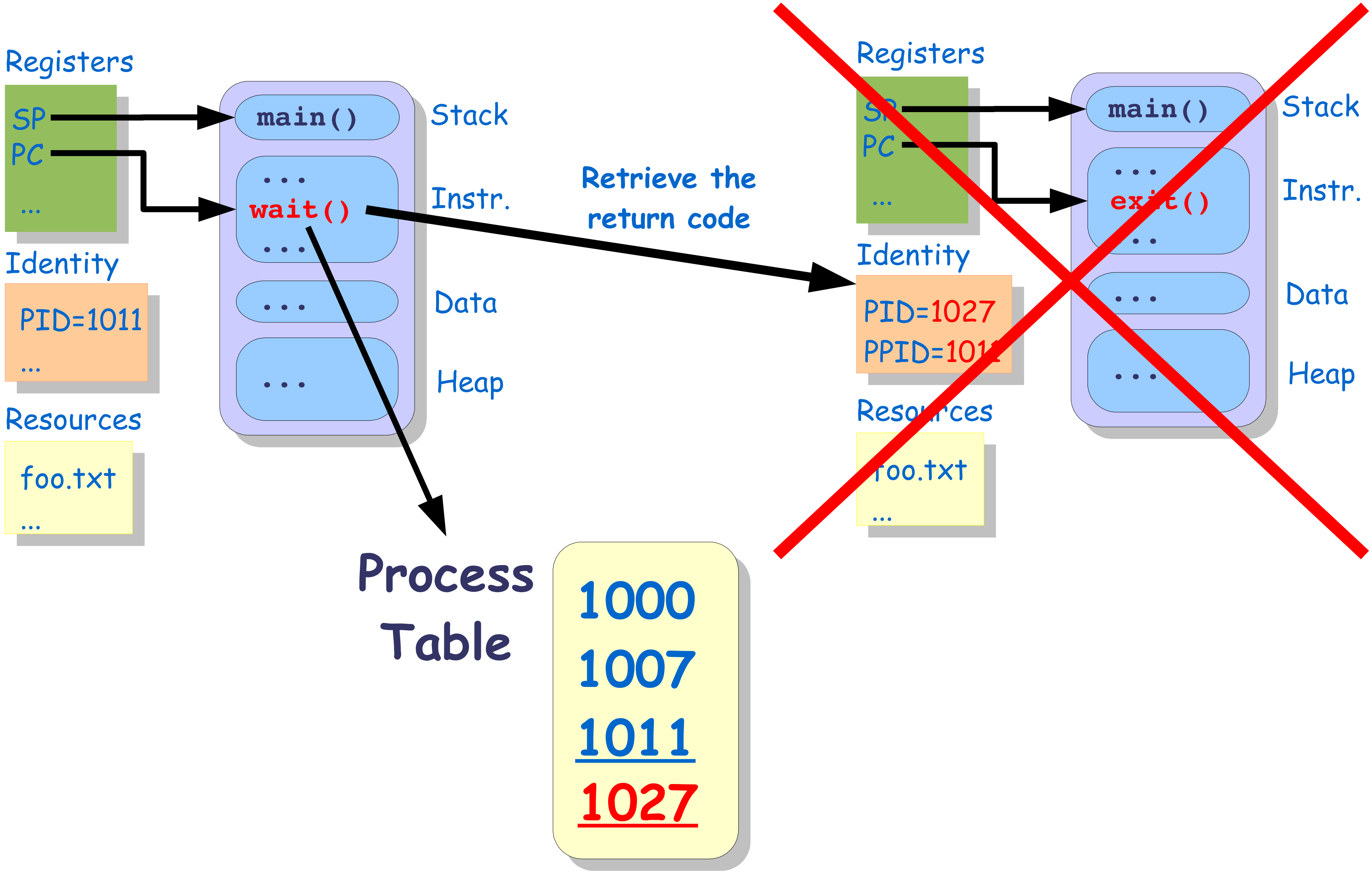
The exec*() Familly

- `execve()`: Original system call, all other exec*() functions are just front-end to it.
- Other exec*() functions are: `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`.
- What's the difference ?
  - `l/v` = Arguments are given as a "list" or a "vector"
  - `p/e` = Environment is used as such (p) or given as an argument (e).
  - Examples:
    - `execlp("li", "li", "-al", 0);`
    - `execl("/usr/bin/sh", "sh", "-c", "li -l *.c", 0);`

Termination of a Process (exit)



Termination of a Process (exit)



wait()

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

int main() {
  pid_t pid;

  switch(pid = fork())
    {
    case -1: /* Failure */
      perror("waiting");
      exit(1);

    case 0: /* Child code */
      printf("Child is running\n");
      exit(0);

    default: /* Parent Code */
      printf("Parent is running\n");
      while(1);
      exit(0);
    }
}
```

```
[fleury@hermes]$ ./waiting
Child is running
Parent is running

[3]+  Stopped                 ./waiting
[fleury@hermes]$ ps a | grep waiting
PID   TTY      STAT   TIME CMD
24859 pts/3    T      0:01 ./waiting
24860 pts/3    Z      0:00 [waiting] <defunct>
```

- **D** Uninterruptible sleep (usually IO)
- **R** Running or runnable (on run queue)
- **S** Interruptible sleep (waiting for an event to complete)
- **T** Stopped, either by a job control signal or because it is being traced.
- **X** dead (should never be seen)
- **Z** Defunct ("zombie") process, terminated but not reaped by its parent.

wait()

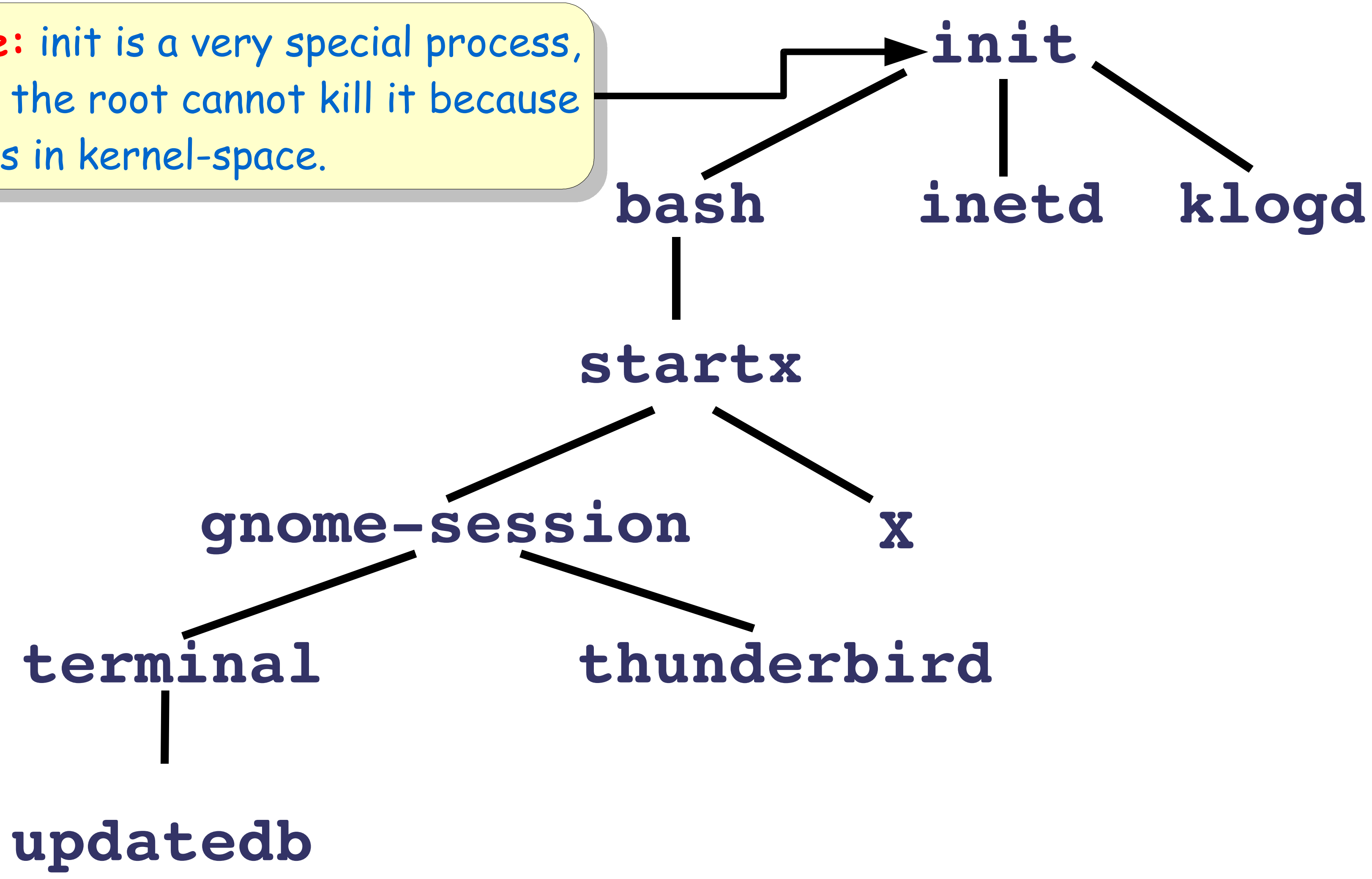
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  pid_t pid;

  switch(pid = fork())
    {
    case -1: /* Failure */
      perror("waiting");
      exit(1);
    case 0: /* Child code */
      printf("Child is running\n");
      exit(0);
    default: /* Parent Code */
      printf("Parent is running\n");
      while (pid != wait(&status));
      printf("The Child %i has returned the value %i\n", pid, status/256);
      exit(0);
    }
}
```

```
[fleury@hermes]$ ./waiting
Child is running
Parent is running
The Child 25543 has returned the value 0
[fleury@hermes]$
```

Note: We are waiting for a precise child (pid) but we have only one, this could be avoided.

Process Tree

Note: init is a very special process, even the root cannot kill it because it lies in kernel-space.



Interlude:
Processes in (nut)Shell

Basic Commands

- bg (^Z or &):
  Send the current job in the background
- fg:
  Run the background job in the foreground
- jobs:
  List all the jobs present on the shell
- kill (^C):
  Terminate a job
- wait:
  Wait for the termination of a job

Demonstration of bg, fg, jobs, kill

- **bg:** Put several jobs in the background
- **jobs:** List them all
- **fg:** Select one and run it in foreground
- **kill:** Send termination signals to some of the background jobs

wait (Shell)

```
#!/bin/sh

# Job 1
ls -a &
# PID of Job 1
p1=$!
# Job 2
ls -al &
# Display status of Job 1
wait $p1
echo Job 1 exited with status $?
# Display status of Job 2
wait $!
echo Job 2 exited with status $?
```

```
[fleury@hermes]$ ./test.sh
.   ..   test.sh
Job 1 exited with status 0
total 16
drwxr-xr-x    2 fleury fleury 4096 Mar 25 11:06 .
drwxr-xr-x   74 fleury fleury 4096 Mar 25 11:05 ..
-rwxr-xr-x    1 fleury fleury  152 Mar 25 11:06 test.sh
Job 2 exited with status 0
[fleury@hermes]$ ./test.sh
total 16
drwxr-xr-x    2 fleury fleury 4096 Mar 25 11:06 .
drwxr-xr-x   74 fleury fleury 4096 Mar 25 11:05 ..
-rwxr-xr-x    1 fleury fleury  152 Mar 25 11:06 test.sh
.   ..   test.sh
Job 1 exited with status 0
Job 2 exited with status 0
[fleury@hermes]$
```

Note: The order of execution might change.

Process Management

The get*() Familly

- **`getpid()`**: Get process ID
- **`getppid()`**: Get parent process ID
- **`getuid()`**: Get user ID
- **`geteuid()`**: Get effective user ID (ignoring set ID calls)
- **`getgid()`**: Get group ID
- **`getegid()`**: Get effective group ID (ignoring set ID calls)
- **`getresuid()`**: Get real, effective and saved user ID
- **`getresgid()`**: Get real, effective and saved group ID
- **`getgroups()`**: Get the list of groups to which belong the user

The command "id"

```
[fleury@hermes]$ id
uid=1000(fleury) gid=1000(fleury) groups=29(audio),1000(fleury)
[fleury@hermes]$ cp /bin/sh .
[fleury@hermes]$ chmod +s sh
[fleury@hermes]$ su -
Password:
[root@hermes]$ id
uid=0(root) gid=0(root) groups=0(root)
[root@hermes]$ ls -l ~fleury/sh
-rwsr-sr-x  1 fleury fleury 667180 Mar 26 17:26 /home/fleury/sh
[root@hermes]$ ~fleury/sh
[root@hermes]$ id
uid=0(root) gid=0(root) euid=1000(fleury) egid=1000(fleury) groups=0(root)
[root@hermes]$ exit
[root@hermes]$ id
uid=0(root) gid=0(root) groups=0(root)
[root@hermes]$ exit
[fleury@hermes]$
```

getgroups()

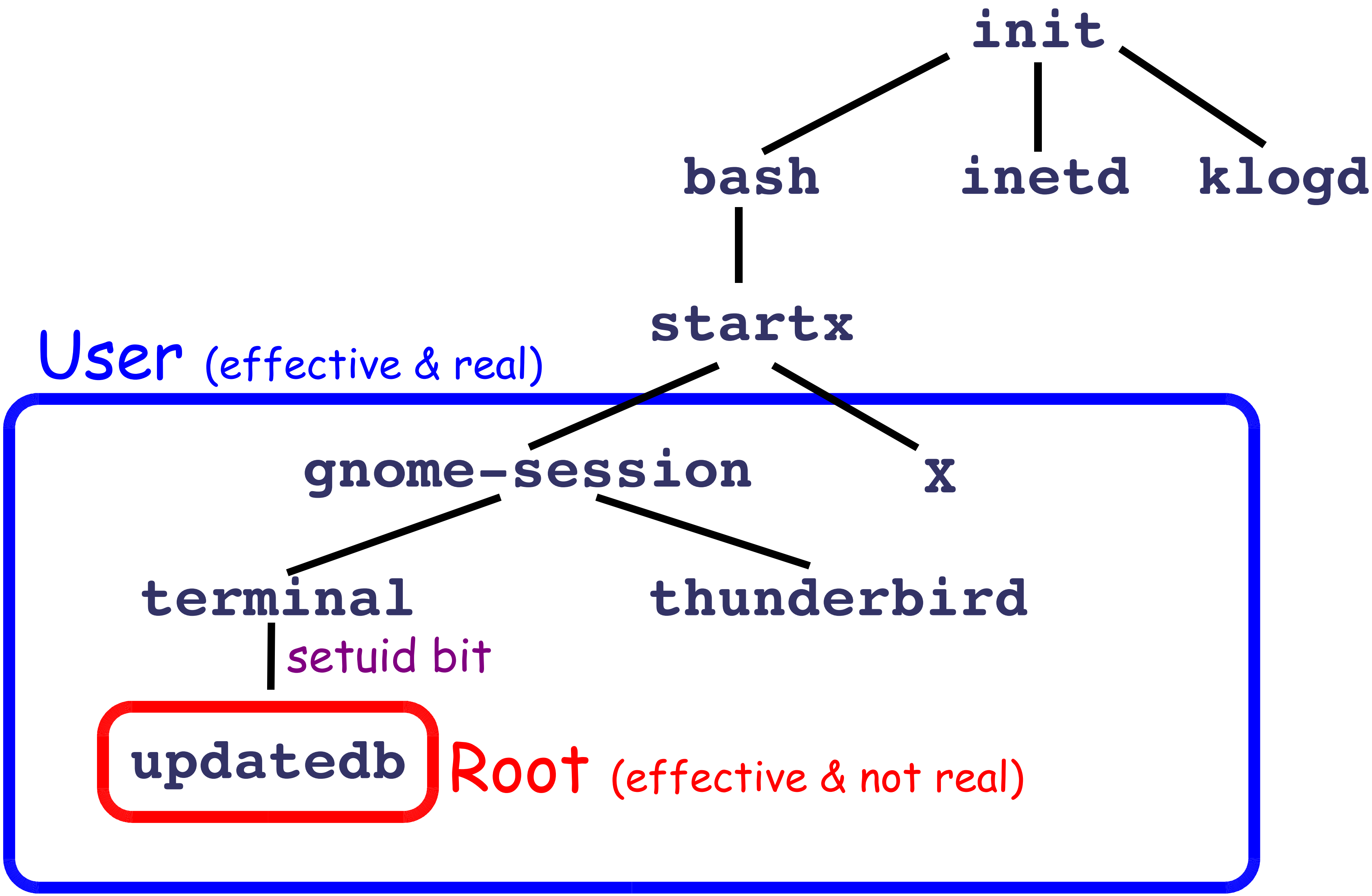
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
  int size, i;
  gid_t *gid_table;

  /* Get the size of the table */
  if ((size = getgroups(0, NULL)) < 0) {
    perror("get_groups");
    exit(1);
  }
  /* Memory allocation of the table */
  if ((gid_table = calloc(size, sizeof(gid_t))) == NULL){
    perror("get_groups");
    exit(1);
  }
  /* Get the group list */
  if (!getgroups(size, gid_table)) {
    perror("get_groups");
    exit(1);
  }
  /* Display the list */
  for (i=0; i<size; i++)
    printf("group[%i] = %u\n", i, gid_table[i]);
  free(gid_table);

  exit(0);
}
```

```
[fleury@hermes]$ id
uid=1000(fleury) gid=1000(fleury) groups=29(audio),1000(fleury)
[fleury@hermes]$ ./get_groups
group[0] = 29
group[1] = 1000
[fleury@hermes]$
```

The set*() Familly

- **`setuid()/setgid()`:**
  Sets the effective user/group ID of the current process
- **`setresuid()/setresgid()`:**
  Sets the real user ID, the effective user ID, and the saved (effective) user ID of the current process.
- **`seteuid()/setegid()`:**
  Sets the effective user/group ID of the current process
- **`setreuid()/setregid()`:**
  Sets real and effective user IDs of the current process.

Process Tree



sleep()

- **`sleep()`:**
  Sleep for the specified number of seconds
- **`usleep()`:**
  Suspend execution for microsecond intervals
- **`nanosleep()`:**
  Pause execution for a specified time

Process Scheduling

Getting & Setting Priority

- **`nice()`**:
  Change the nice value of a process. Return the new priority value or "-1" in case of failure

- **`renice()`**:
  Alter priority of running processes

- **`getpriority()`**:
  Get program scheduling priority

- **`setpriority()`**:
  Set program scheduling priority

nice()

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

int main() {

  printf("process priority is %i\n",
         getpriority(PRIO_PROCESS, 0));
  printf("process group priority is %i\n",
         getpriority(PRIO_PGRP, 0));
  printf("user priority is %i\n",
         getpriority(PRIO_USER, 0));
  printf("=====\n");
  printf("new nice value: %i\n", nice(3));
  printf("new nice value: %i\n", nice(3));
  printf("new nice value: %i\n", nice(-9));
  printf("=====\n");
  printf("process priority is %i\n",
         getpriority(PRIO_PROCESS, 0));
  printf("process group priority is %i\n",
          getpriority(PRIO_PGRP, 0));
  printf("user priority is %i\n",
         getpriority(PRIO_USER, 0));

  exit(0);
}
```

Note: Priority is a value between -20 and 19. Only root can go under 0.

```
[fleury@hermes]$ ./priority
process priority is 0
process group priority is 0
user priority is 0
=====
new nice value: 3
new nice value: 6
new nice value: -1
=====
process priority is 6
process group priority is 6
user priority is 0
[fleury@hermes]$ su -c ./priority
Password:
process priority is 0
process group priority is 0
user priority is -10
=====
new nice value: 3
new nice value: 6
new nice value: -3
=====
process priority is -3
process group priority is -3
user priority is -10
```

See also ...

- mlock() / munlock()
- mlockall() / munlockall()
- sched_get_priority_max() / sched_get_priority_min()
- sched_getaffinity() / sched_setaffinity()
- sched_getparam() / sched_setparam()
- sched_getscheduler() / sched_setscheduler()
- sched_rr_get_interval()
- sched_yield()
- capabilities()
- …

Questions ?

Next Week

- Signals in Unix
- Inter-process communication
  - files
  - pipes
  - named pipes
  - sharing memory chunks
  - FIFO
  - Semaphores