Operating Systems: Processes & Scheduling

Bachelor's Special Edition

Fabrice BOISSIER <fabrice.boissier@epita.fr>

2022-01-28

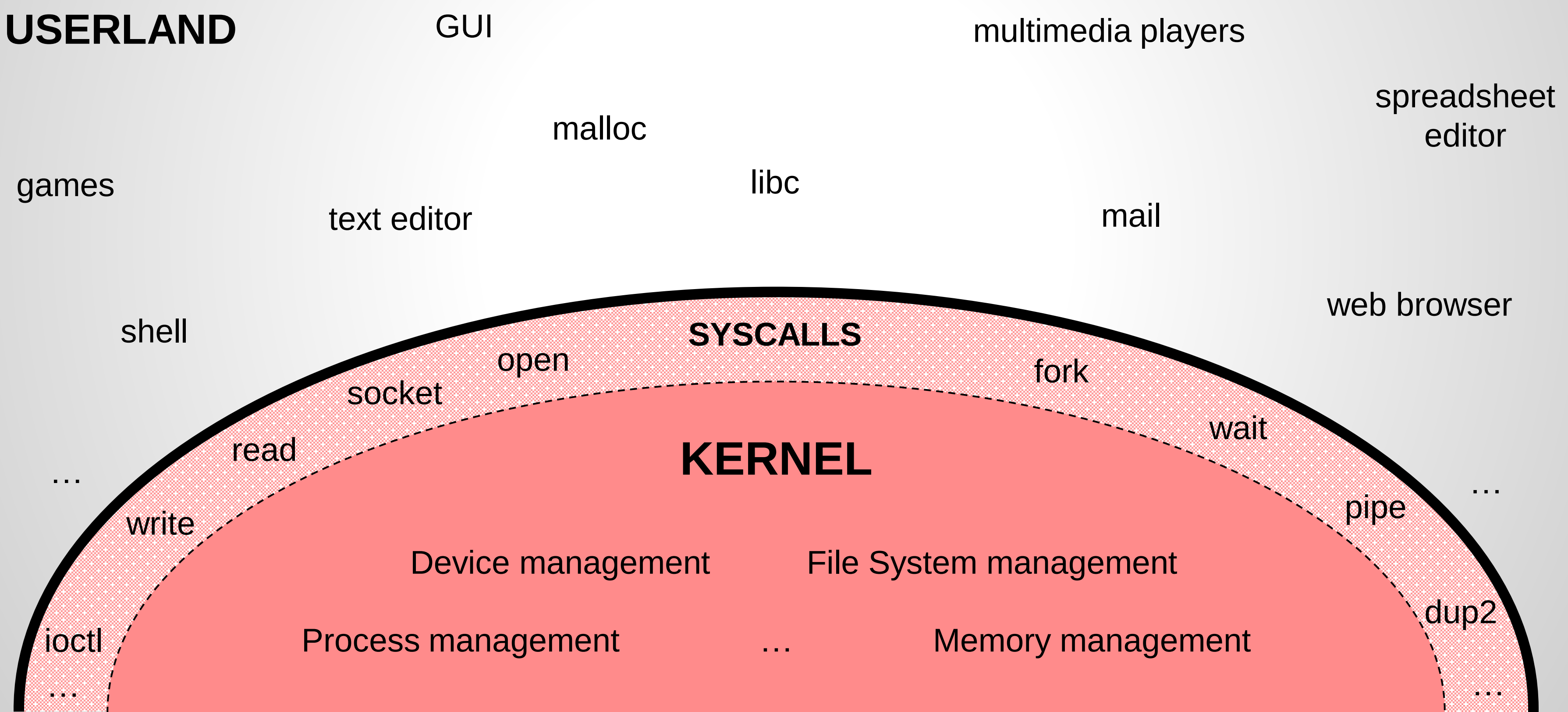
The “OS API”

- Programs respect a specific file format
  - How to store the code, hardcoded values, …
  - Lot of formats: ELF, MACH-O, PE, …

- The **kernel** exposes ***syscalls*** (mostly)
- **Libraries** expose ***functions***

The “OS API”

In order to ask for services to the kernel, a userland code uses the « syscalls »



USERLAND

GUI

multimedia players

spreadsheet editor

malloc

libc

games

text editor

mail

web browser

shell

SYSCALLS

open

socket

fork

wait

read

…

…

write

pipe

dup2

ioctl

…

…

KERNEL

Device management

File System management

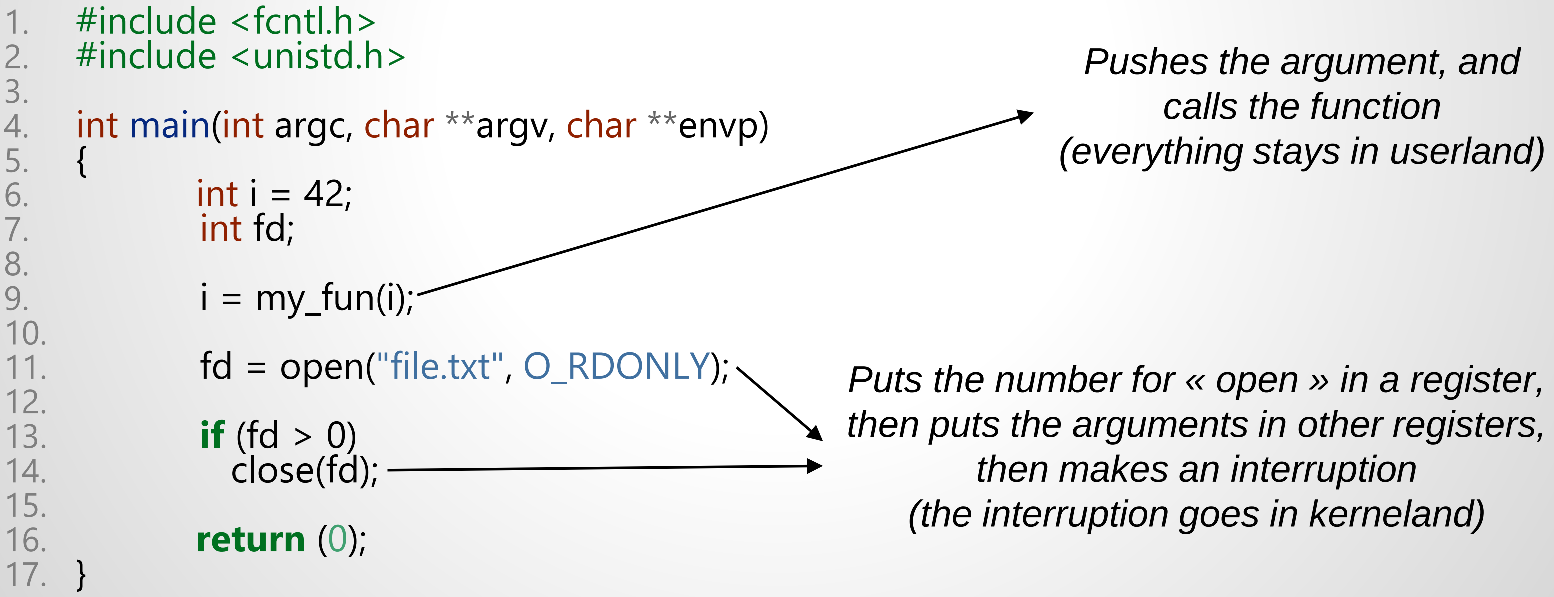
Process management

…

Memory management

The “OS API”: Syscalls

- Syscalls are not exactly « functions » during execution
  - In assembly, it’s not a « *call* » instruction…
  - It’s an interruption (« *int* » instruction) with a precise number



```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv, char **envp)
{
        int i = 42;
        int fd;

        i = my_fun(i);

        fd = open("file.txt", O_RDONLY);

        if (fd > 0)
           close(fd);

        return (0);
}
```

Pushes the argument, and calls the function (everything stays in userland)

Puts the number for « open » in a register, then puts the arguments in other registers, then makes an interruption (the interruption goes in kerneland)

API / ABI

- API: Application Programming Interface
  - Defines useful functions to call for developers
  - Used while “coding”

  - How to use a library written by someone else or query a server?

- ABI: Application Binary Interface
  - Defines how to make a program working in the low level part (assembly, …)
  - *Architecture-dependant (CPU specifications are required)*
  - Used by compilers, OS, eventually libraries

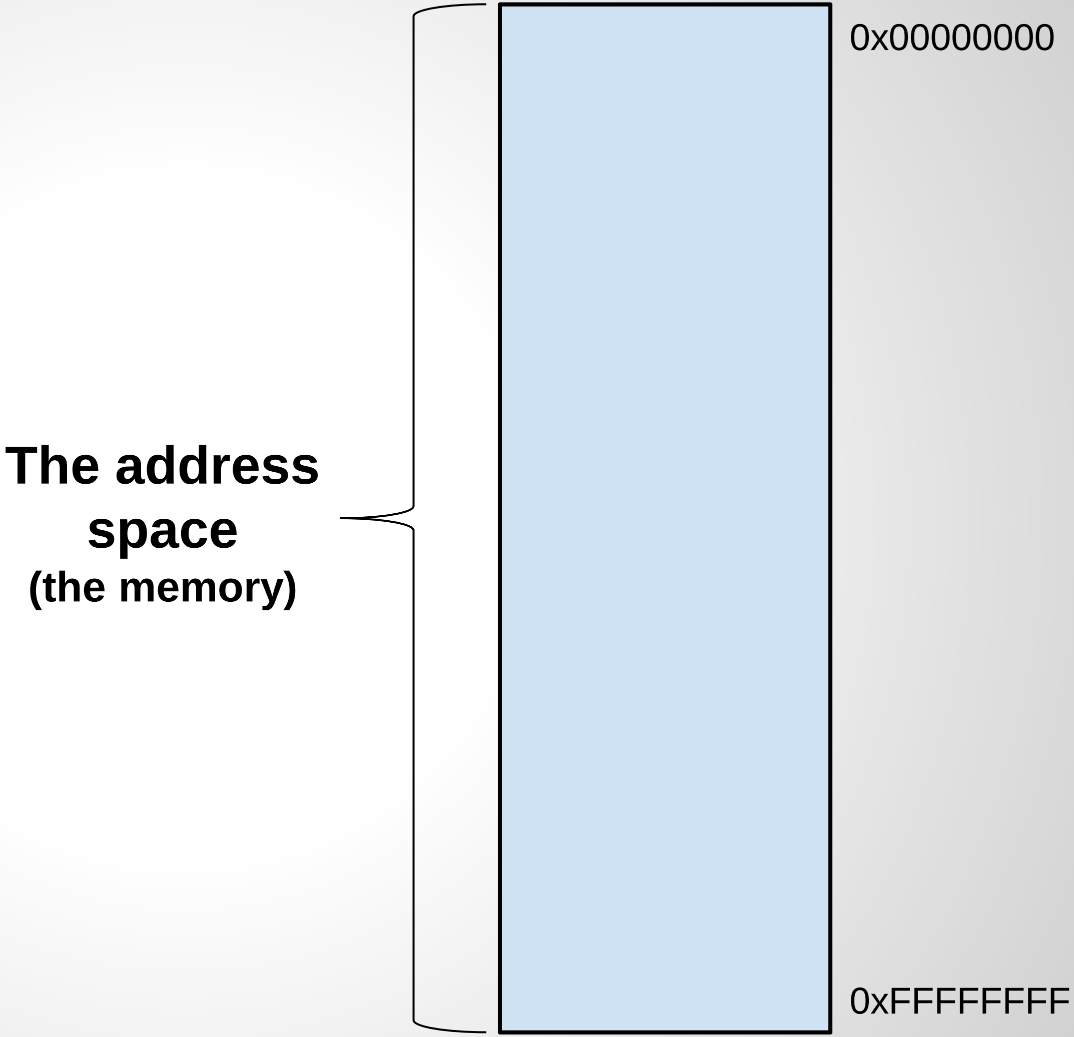
  - How a binary can use the operating system and run?

What is a process?

- Program: static object that contain code
  - The file
- Process: program in execution
  - Something in userland memory
- Context: address space, registers, and other infos
  - Data in kernel

Address Space

```c
char *myStr;
int i = 0;
int     main(void)
{
const char *var = "Test";
int a = 1337;

i = addition(21, 42);
myStr = malloc(32 * sizeof (char));
return (0);
}
```



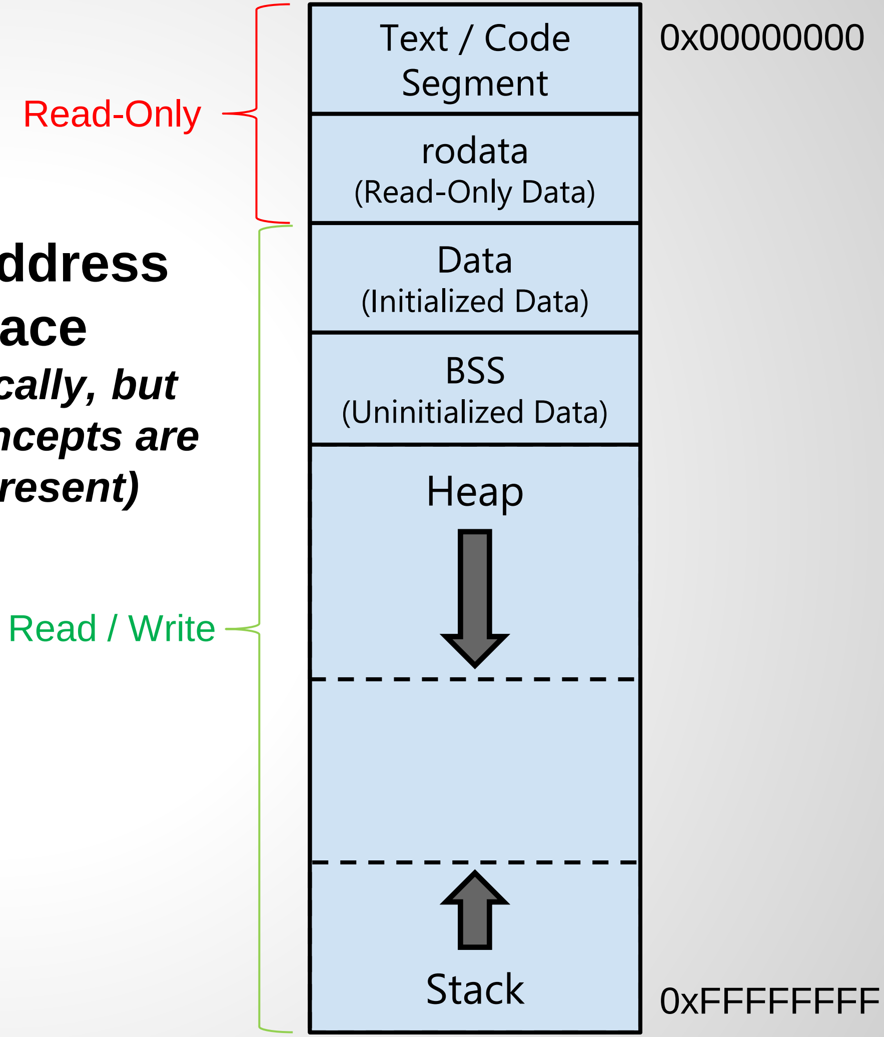
The address space

(the memory)

```c
char *myStr;
int i = 0;
int       main(void)
{
const char *var = "Test";
int a = 1337;

i = addition(21, 42);
myStr = malloc(32 * sizeof (char));
return (0);
}
```

The address space

(historically, but main concepts are still present)



Read-Only

Read / Write

Text / Code Segment

rodata (Read-Only Data)

Data (Initialized Data)

BSS (Uninitialized Data)

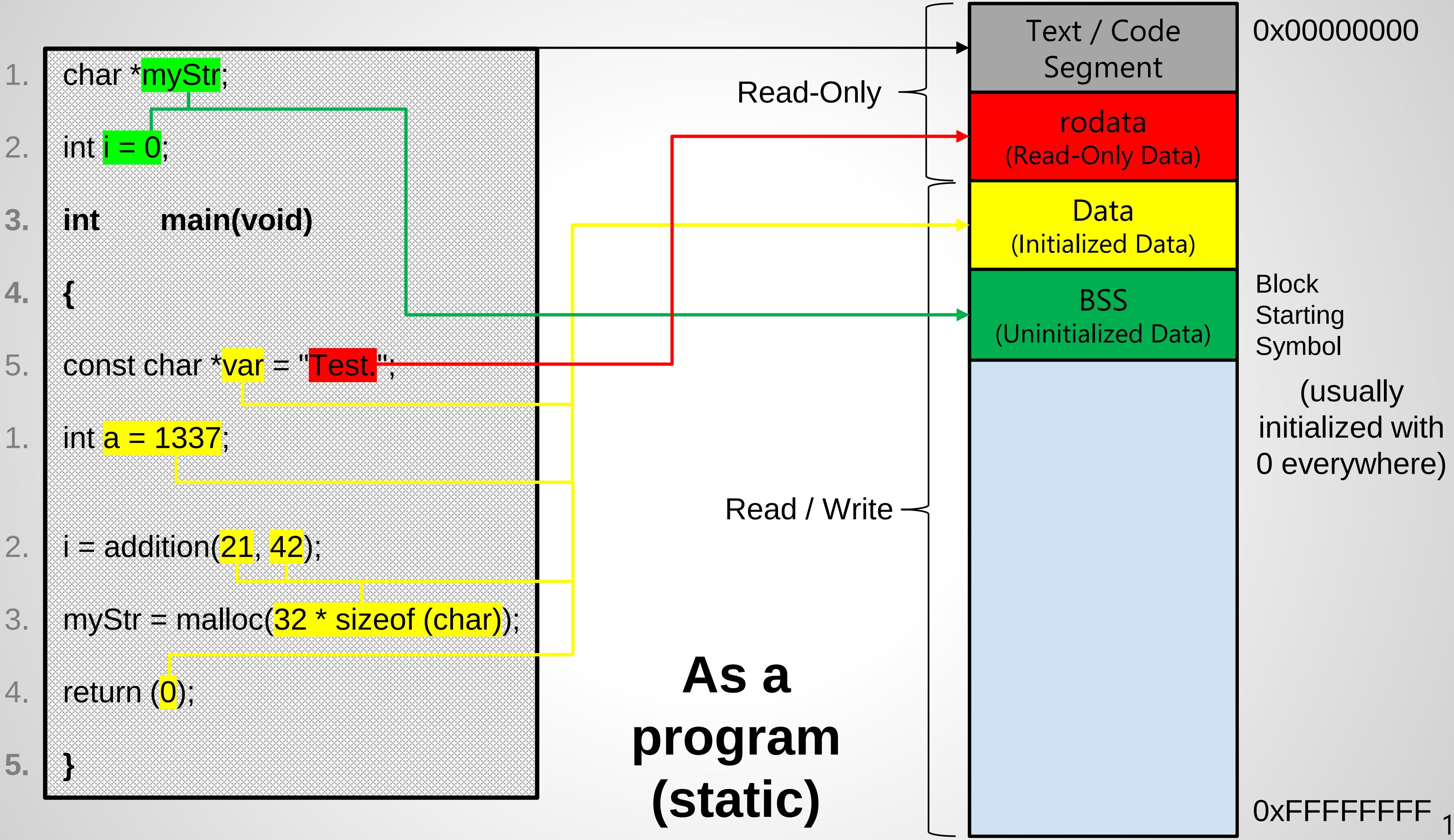
Heap

Stack

0x00000000

0xFFFFFFFF

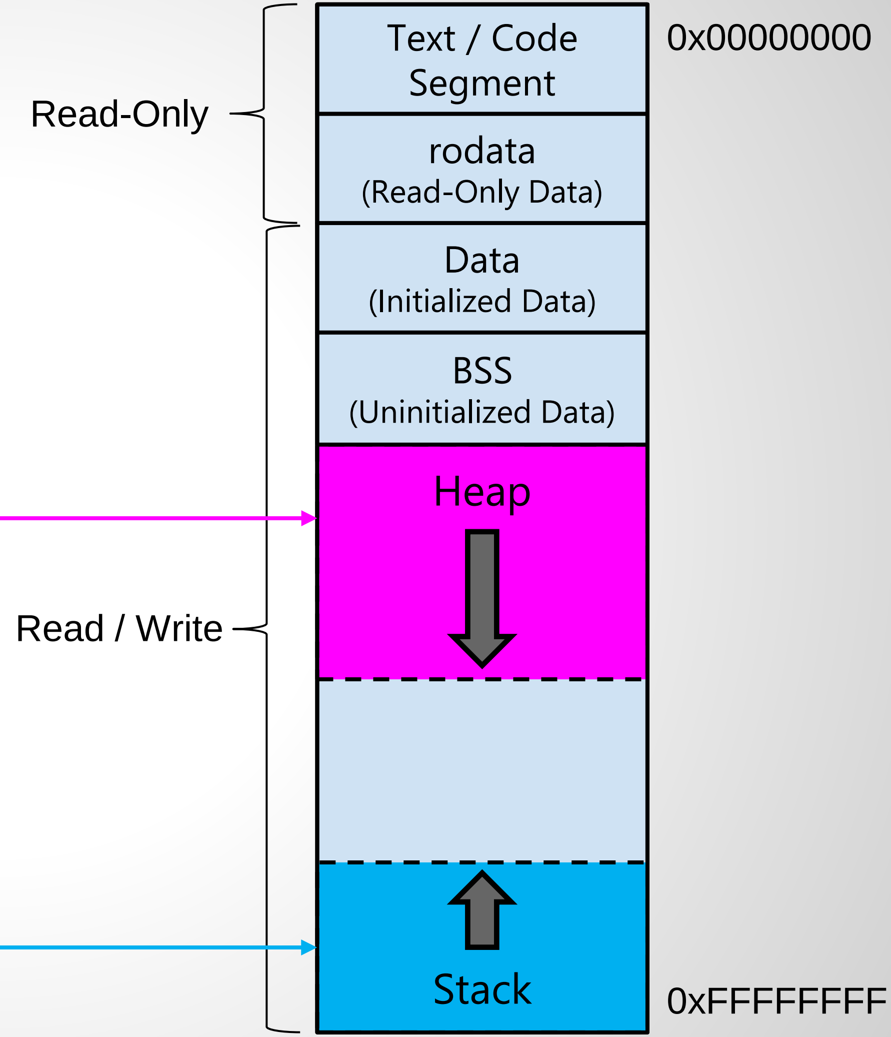
As a program (static)



```
char *myStr;

int i = 0;

int     main(void)

{

const char *var = "Test.";

int a = 1337;


i = addition(21, 42);

mystr = malloc(32 * sizeof (char));

return (0);

}
```

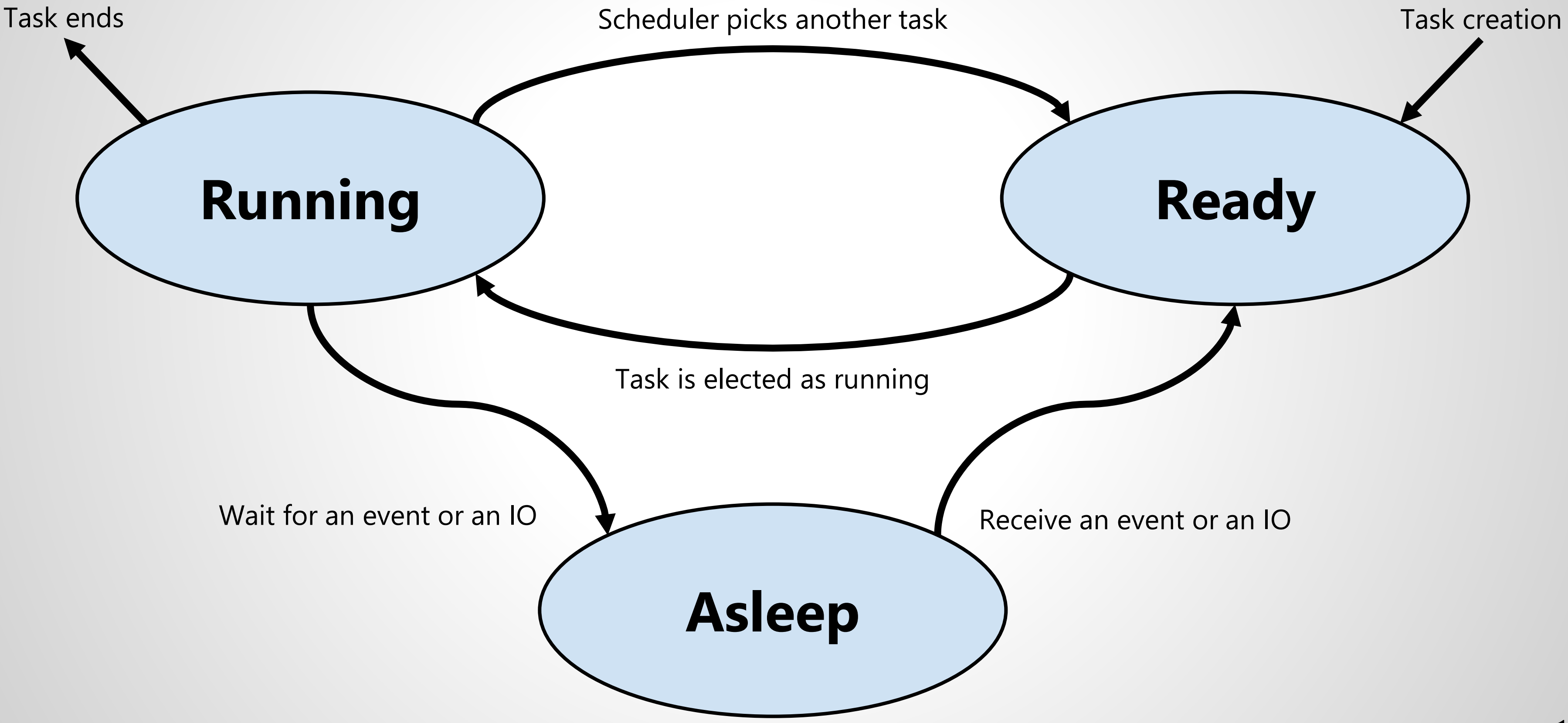
As a process (running)

```
char *myStr;

int i = 0;

int     main(void)

{

const char *var = "Test";

int a = 1337;


i = addition(21, 42);

myStr = malloc(32 * sizeof (char));

return (0);

}
```



Process Creation

Task states


Task ends
Scheduler picks another task
Task creation
Running
Ready
Task is elected as running
Wait for an event or an IO
Receive an event or an IO
Asleep

Process Control Block

- Contains all of the useful informations for a task
    - State of the process
    - Identification & Group of the process
    - Address space informations (pages allocated)
    - Context (Signals, IPC, CPU registers, ...)
    - ...
- *Informations allow to save/load a process in memory*
- *Keep its context intact in case of sleeping*
- *Check permission*

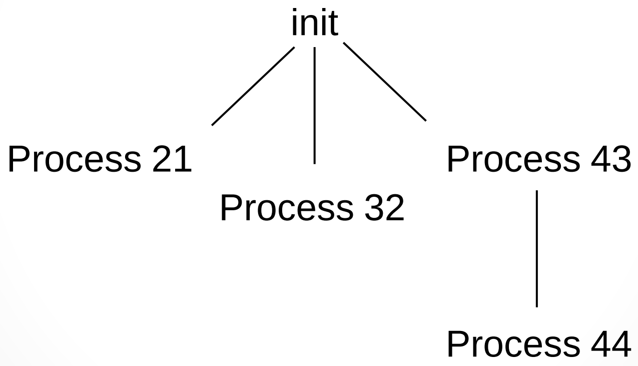
- *struct task_struct* in Linux, *PEB* on Windows

Process Control Block

- State (RUNNING, READY, ASLEEP)
- Stack (state of all local variables)
- Scheduling attributes (which scheduler to use)
- Memory mapping (state of heap variable)
- Pid : process ID
- PPid : parent process ID
- Gid : group ID
- Tgid : thread group ID
- Registers (in *struct thread_info*)
- Uid : user ID
- Signals state
- ...

Process hierarchy

- UNIX/Linux: processes live in a tree
- Multiple groups (signals, resource groups, ...)



- Windows: less obvious, but still some kind of tree

Process Creation

- UNIX-likes: duplication of the current process

- *(In some other OSes, you ask the kernel to create another process and fill it with values you give in parameters… like the memory image you wish to put)*

Process Creation

- pid_t fork(void)

  // pid_t is just an integer…

- *// Linux only*

  ```
  long clone(unsigned long flags,
                     void *child_stack,
                     void *ptid,
                     void *ctid,
                     struct pt_regs *regs)
  ```

Process Creation

fork(2)

- Creates a child process
    - New PID, PPID = parent’s PID
- Duplicates address space
    - [copy on write]
- File descriptors are inherited
    - Required for IPC
- Signals configuration is kept
    - But signals are not transfered to child
- Counters, timers, locks, … are forgotten

Process Creation

fork(2)

- Return values:
  - 0 = Child [the syscall succeeded!]
  - [1 -> PID_MAX] = Parent [the syscall succeeded!]
  - -1 = Error [the syscall failed ☹]

Process Creation

- int execve(const char *filename,
                        char *const argv[],
                        char *const envp[])

- // See exec* family
  // execvp, ...

Process Creation

execve(2)

- Executes the program pointed to by *filename*
  - Uses *argv* array as the arguments given
  - Uses *envp* array as the environment variables

- Replaces the full address space with the one given by the new program

- Therefore, it never returns any value…
  - …except -1 in case of an error

Process Creation

Copy on Write

- Usually, after a fork(2), there is an exec*(2)
  - Not immediately, but there are no modification in memory before exec
- Why copy the full address space during fork, if it will be deleted in the next instruction?
- Do not copy immediately:
  - Keep the original pages in reading mode
  - Wait for any write in memory before copy
  - Or wait for an exec*(2) for rewriting all of the address space

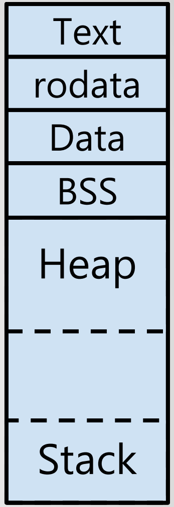
```
#include <err.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv, char **envp)
{
	char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NULL };
	int status;
	pid_t pid_w, pid_f = fork();

	switch (pid_f) {
		case -1:
			err(1, "unable to fork");
		case 0:
			execve(prog_argv[0], prog_argv, envp);
			err(1, "unable to execve %s", prog_argv[0]);
		default:
			pid_w = waitpid(pid_f, &status, 0);
	}
	return 0;
}
```

POV: code

Process 4
main() [L9]

```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NULL };
10.         int status;
11.         pid_t pid_w, pid_f = fork();
12.
13.         switch (pid_f) {
14.                 case -1:
15.                         err(1, "unable to fork");
16.                 case 0:
17.                         execve(prog_argv[0], prog_argv, envp);
18.                         err(1, "unable to execve %s", prog_argv[0]);
19.                 default:
20.                          pid_w = waitpid(pid_f, &status, 0);
21.         }
22.         return 0;
23. }
```

Process 4



Text

rodata

Data

BSS

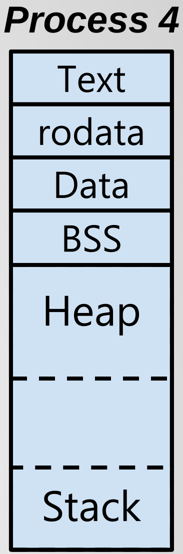
Heap

Stack

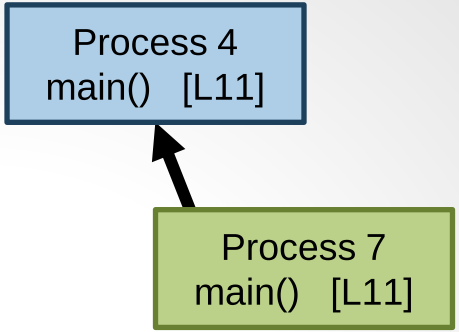
POV: code

Process 4
main()   [L11]

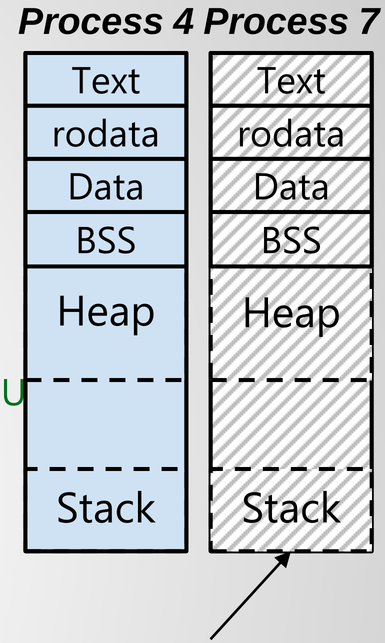
```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NULL };
10.          int status;
11.  ⇨       pid_t pid_w, pid_f = fork();
12.
13.          switch (pid_f) {
14.                  case -1:
15.                          err(1, "unable to fork");
16.                  case 0:
17.                          execve(prog_argv[0], prog_argv, envp);
18.                          err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.                           pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```



POV: code

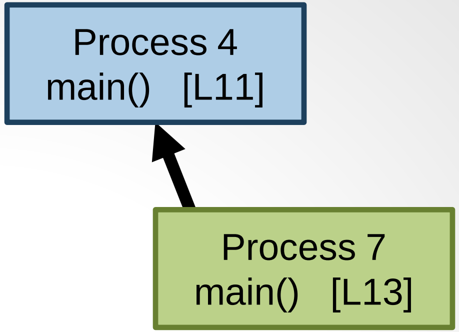


```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         pid_t pid_w, pid_f = fork();
12.
13.         switch (pid_f) {
14.                 case -1:
15.                         err(1, "unable to fork");
16.                 case 0:
17.                         execve(prog_argv[0], prog_argv, envp);
18.                         err(1, "unable to execve %s", prog_argv[0]);
19.                 default:
20.                          pid_w = waitpid(pid_f, &status, 0);
21.         }
22.         return 0;
23. }
```

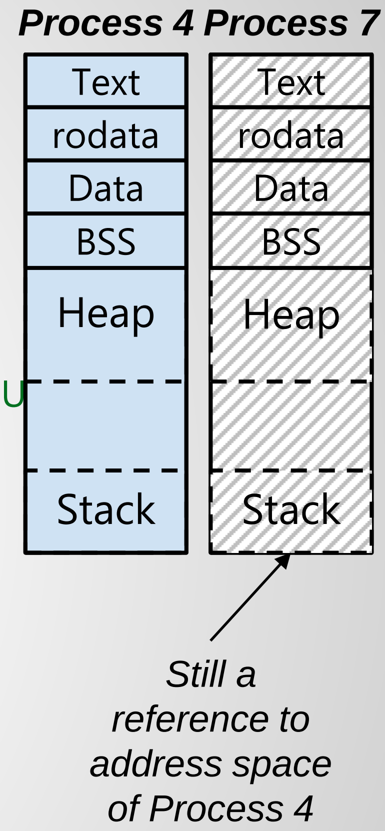


Still a reference to address space of Process 4

POV: code

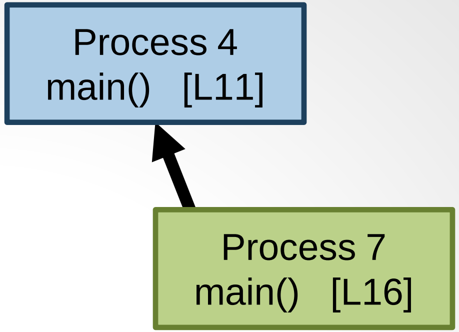


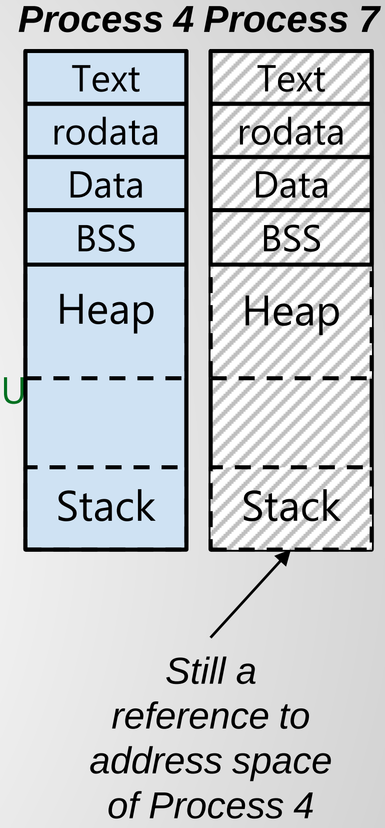
```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.          int status;
11.          pid_t pid_w, pid_f = fork();
12.
13.          switch (pid_f) {
14.                  case -1:
15.                          err(1, "unable to fork");
16.                  case 0:
17.                          execve(prog_argv[0], prog_argv, envp);
18.                          err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.                           pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```



POV: code

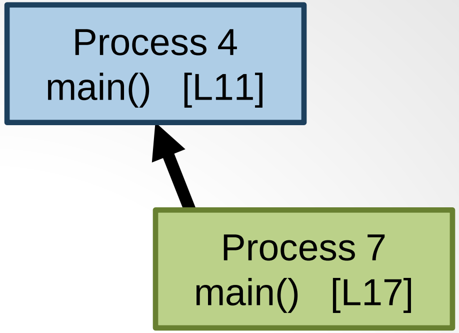
```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.          int status;
11.          pid_t pid_w, pid_f = fork();
12.
13.          switch (pid_f) {
14.                  case -1:
15.                              err(1, "unable to fork");
16.                  case 0:
17.                              execve(prog_argv[0], prog_argv, envp);
18.                              err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.                               pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```

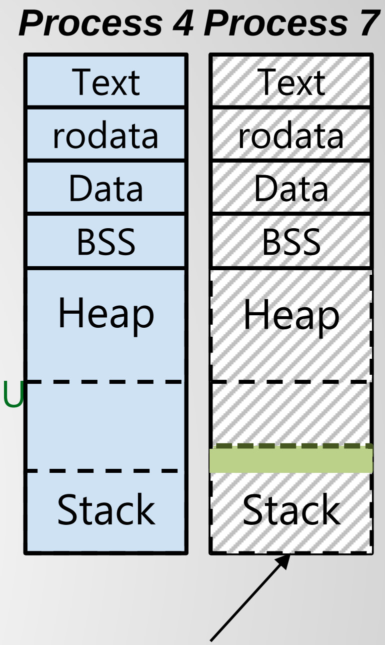




POV: code

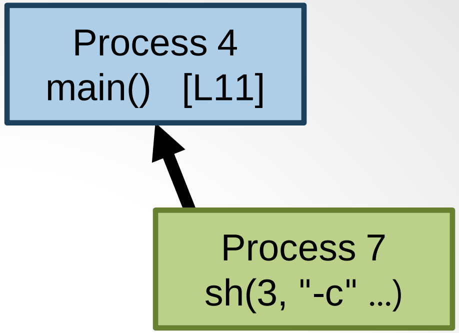
```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.          int status;
11.          pid_t pid_w, pid_f = fork();
12.
13.          switch (pid_f) {
14.                  case -1:
15.                          err(1, "unable to fork");
16.                  case 0:
17.                          execve(prog_argv[0], prog_argv, envp);
18.                          err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.                           pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```



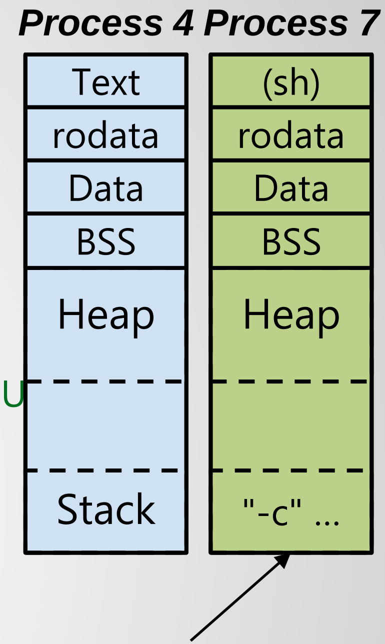


The stack evolved partially with the arguments

POV: code



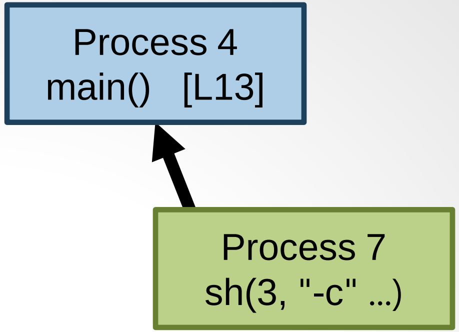
```
1.    #include <err.h>
2.    #include <stddef.h>
3.    #include <sys/types.h>
4.    #include <sys/wait.h>
5.    #include <unistd.h>
6.
7.    int main(int argc, char **argv, char **envp)
8.    {
9.            char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.           int status;
11.           pid_t pid_w, pid_f = fork();
12.
13.           switch (pid_f) {
14.                   case -1:
15.                           err(1, "unable to fork");
16.                   case 0:
17.                           execve(prog_argv[0], prog_argv, envp);
18.                           err(1, "unable to execve %s", prog_argv[0]);
19.                   default:
20.                            pid_w = waitpid(pid_f, &status, 0);
21.           }
22.           return 0;
23.   }
```

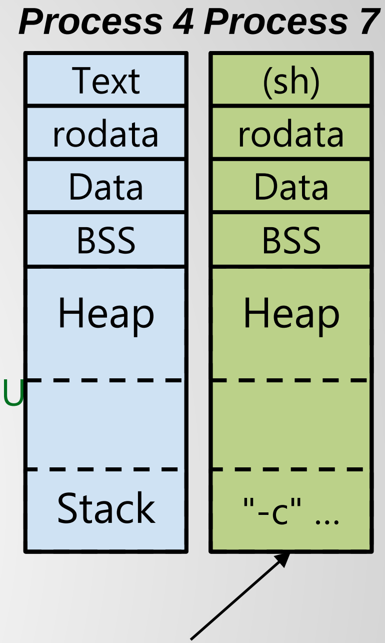


New address space containing « /bin/sh » code and the parameters

POV: code

```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.          int status;
11.          pid_t pid_w, pid_f = fork();
12.
13.  ⇨       switch (pid_f) {
14.                  case -1:
15.                          err(1, "unable to fork");
16.                  case 0:
17.                          execve(prog_argv[0], prog_argv, envp);
18.                          err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.                           pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```

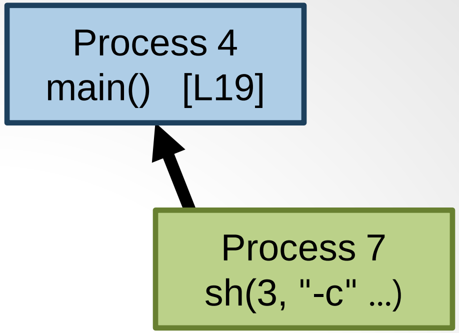


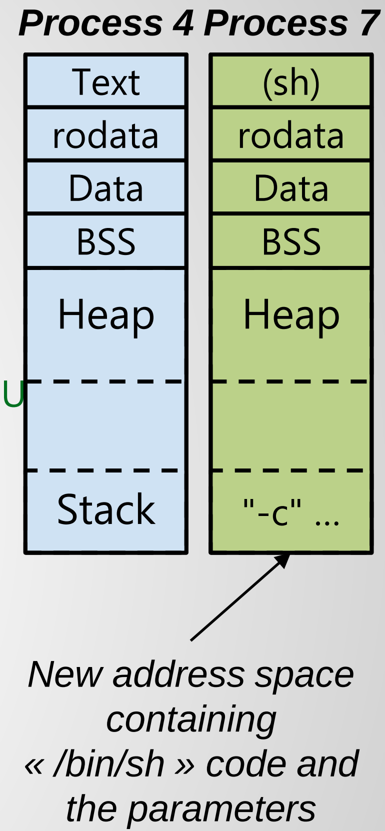


New address space containing « /bin/sh » code and the parameters

POV: code

```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         pid_t pid_w, pid_f = fork();
12.
13.         switch (pid_f) {
14.                 case -1:
15.                         err(1, "unable to fork");
16.                 case 0:
17.                         execve(prog_argv[0], prog_argv, envp);
18.                         err(1, "unable to execve %s", prog_argv[0]);
19.          ⇨      default:
20.                          pid_w = waitpid(pid_f, &status, 0);
21.         }
22.         return 0;
23. }
```

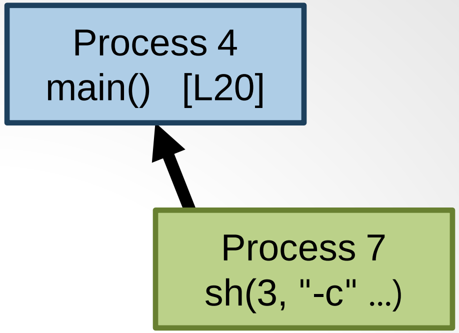




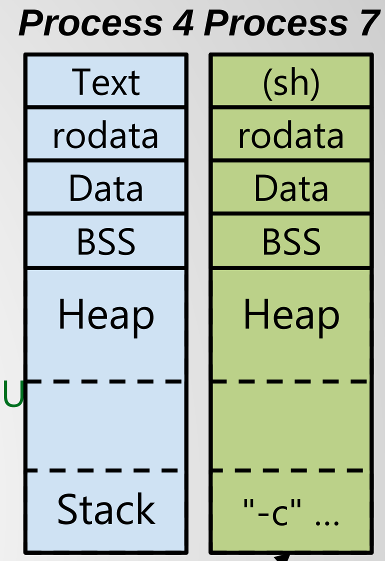
New address space containing « /bin/sh » code and the parameters

POV: code

```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         pid_t pid_w, pid_f = fork();
12.
13.         switch (pid_f) {
14.                 case -1:
15.                         err(1, "unable to fork");
16.                 case 0:
17.                         execve(prog_argv[0], prog_argv, envp);
18.                         err(1, "unable to execve %s", prog_argv[0]);
19.                 default:
20.                          pid_w = waitpid(pid_f, &status, 0);
21.         }
22.         return 0;
23. }
```

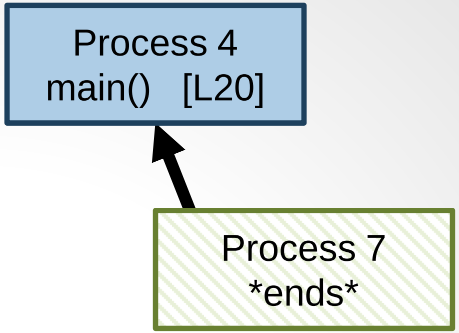


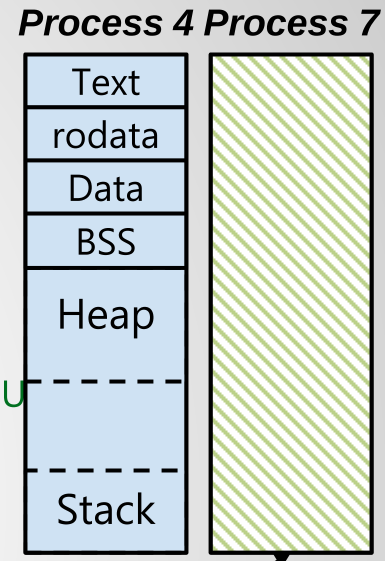
Process 4 is waiting for his child (pid == 7) to die, or at least, get its remains



New address space containing « /bin/sh » code and the parameters

POV: code





```
1.    #include <err.h>
2.    #include <stddef.h>
3.    #include <sys/types.h>
4.    #include <sys/wait.h>
5.    #include <unistd.h>
6.
7.    int main(int argc, char **argv, char **envp)
8.    {
9.            char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.           int status;
11.           pid_t pid_w, pid_f = fork();
12.
13.           switch (pid_f) {
14.                   case -1:
15.                           err(1, "unable to fork");
16.                   case 0:
17.                           execve(prog_argv[0], prog_argv, envp);
18.                           err(1, "unable to execve %s", prog_argv[0]);
19.                   default:
20.                           pid_w = waitpid(pid_f, &status, 0);
21.           }
22.           return 0;
23.   }
```

Process 4 is waiting for his child (pid == 7) to die, or at least, get its remains

Process 7 ends, its address space is released, but its PCB is still there, letting its father get the information

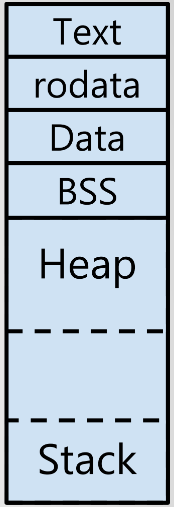
POV: code

Process 4
main()   [L20]

```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NULL };
10.          int status;
11.          pid_t pid_w, pid_f = fork();
12.
13.          switch (pid_f) {
14.                  case -1:
15.                          err(1, "unable to fork");
16.                  case 0:
17.                          execve(prog_argv[0], prog_argv, envp);
18.                          err(1, "unable to execve %s", prog_argv[0]);
19.                  default:
20.              ⇒           pid_w = waitpid(pid_f, &status, 0);
21.          }
22.          return 0;
23.  }
```

Process 4 gets the information about how its child ended

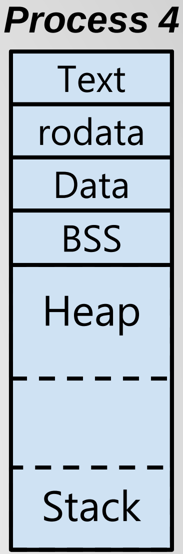
Process 4



POV: code

Process 4
main() [L22]

```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NULL };
10.         int status;
11.         pid_t pid_w, pid_f = fork();
12.
13.         switch (pid_f) {
14.                 case -1:
15.                         err(1, "unable to fork");
16.                 case 0:
17.                         execve(prog_argv[0], prog_argv, envp);
18.                         err(1, "unable to execve %s", prog_argv[0]);
19.                 default:
20.                          pid_w = waitpid(pid_f, &status, 0);
21.         }
22.         return 0;
23. }
```



POV: code

Process 4 ended correctly (return (0);),

its address space is freed,

and its father will get the information

Process state: Zombie

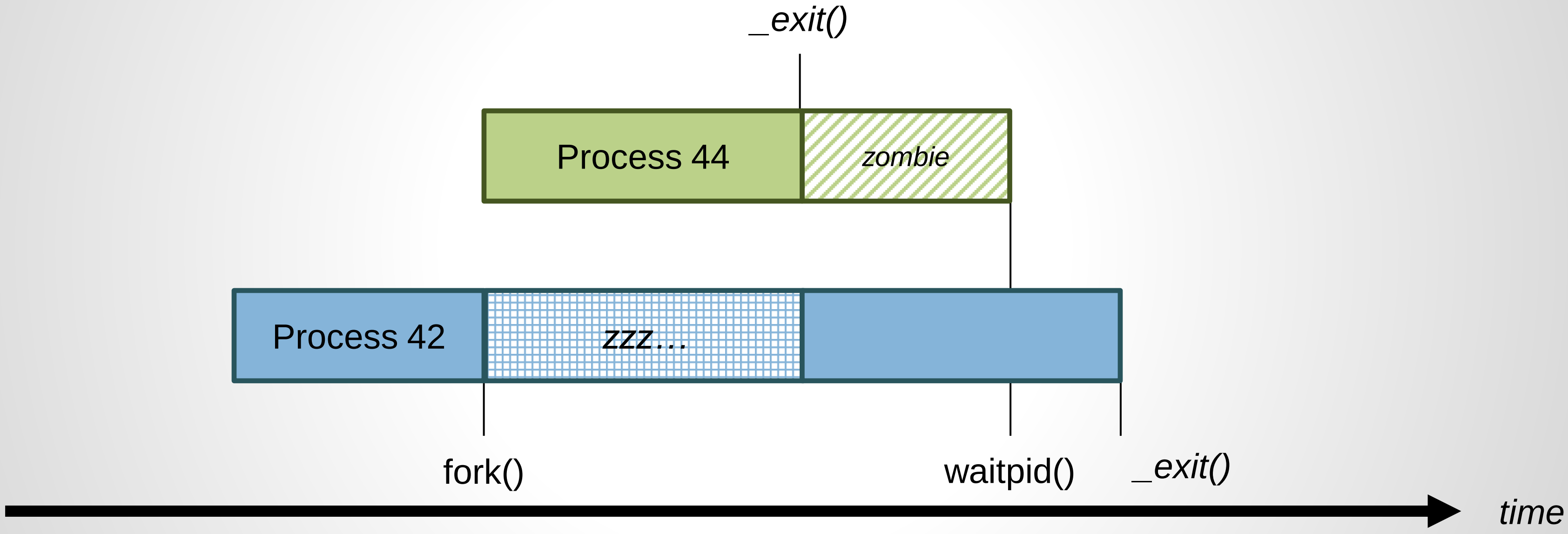
- When child finishes « *too quickly* »
  - Before the father reaches a wait(2) or waitpid(2) syscall
  - Before the father’s end
  - If the father didn’t ask to mask the SIGCHLD

- PCB structure is still allocated
  - Contains the return code
  - To let the father knows why its child ended
  - PID is still reserved until it is fully released with the PCB

- Can be seen in « ps » as ‘Z’ (for ‘zombie’)

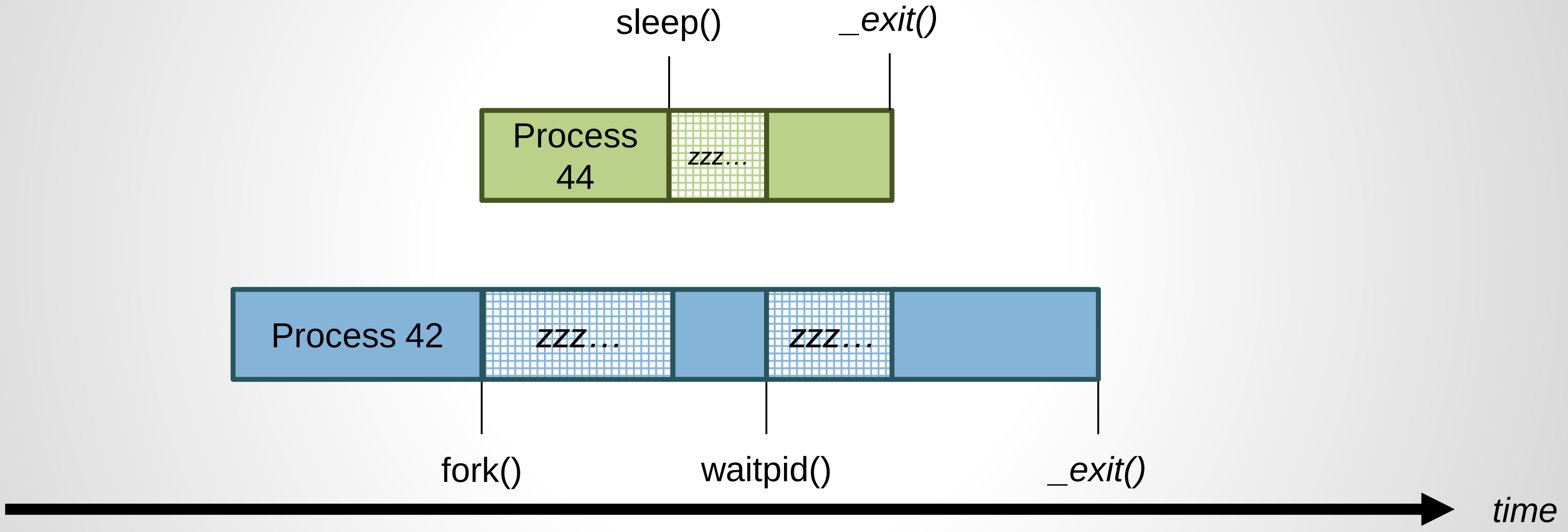
POV: kernel/scheduler

Case 1: child ends before the wait/waitpid



POV: kernel/scheduler

Case 2: child ends after the wait/waitpid



Process ending

-
  ```
  pid_t wait(int *status)
  ```

-
  ```
  pid_t waitpid(pid_t wpid,
                int *status,
                int options)
  ```

Process ending

wait(2)

- Waits for a child to end, and fills *status* with informations
  - Blocking syscall
  - Returns the PID of the process managed
  - Informations returned:

    - Return value (on 1 Byte/8 bits/256 values)
    - Signal that eventually terminated the child
    - ...

- If a child is in zombie state, wait(2) directly returns its return value

Process ending

wait(2)

- Macros to test *status*:

`WIFEXITED`(*status*): test if normal exit

`WIFSIGNALED`(*status*): test if abnormal temination by signal

`WIFSTOPPED`(*status*): test if child process is stopped

`WIFCONTINUED`(*status*): test if child process re-run after stop

Process ending

wait(2)

- Macros to get informations:

`if (WIFEXITED(`*status*`))`

`WEXITSTATUS(`*status*`)`: get return value (exit(2) parameter)

`if (WIFSIGNALED(`*status*`))`

`WTERMSIG(`*status*`)`: get the signal number

`if (WIFSTOPPED(`*status*`))`

`WSTOPSIG(`*status*`)`: get the signal number

Process ending

wait(2)

BEWARE: signal numbers are NOT standard

You MUST check <signal.h> on each OS in order to get the translation signal number/name

Process ending

wait(2)

- Sole error case: the process has no child process

Process ending

waitpid(2)   pid_t waitpid(pid_t wpid, int *status, int options)

- 4 cases based on *wpid*:

1. *wpid* > 0   waits the process with PID == *wpid*
2. *wpid* == -1   waits for any child (like wait(2))
3. *wpid* == 0   waits for any child with same PGID
4. *wpid* < -1   waits for any child whose PGID == |*wpid*|

Process ending

waitpid(2)   pid_t waitpid(pid_t wpid, int *status, int options)

- 3 *options*:

1. `WNOHANG`: syscall will not block if the child still runs
   return value becomes 0
2. `WUNTRACED`: report process stopped by a signal
3. `WCONTINUED`: report process that awoken from stop

Process ending

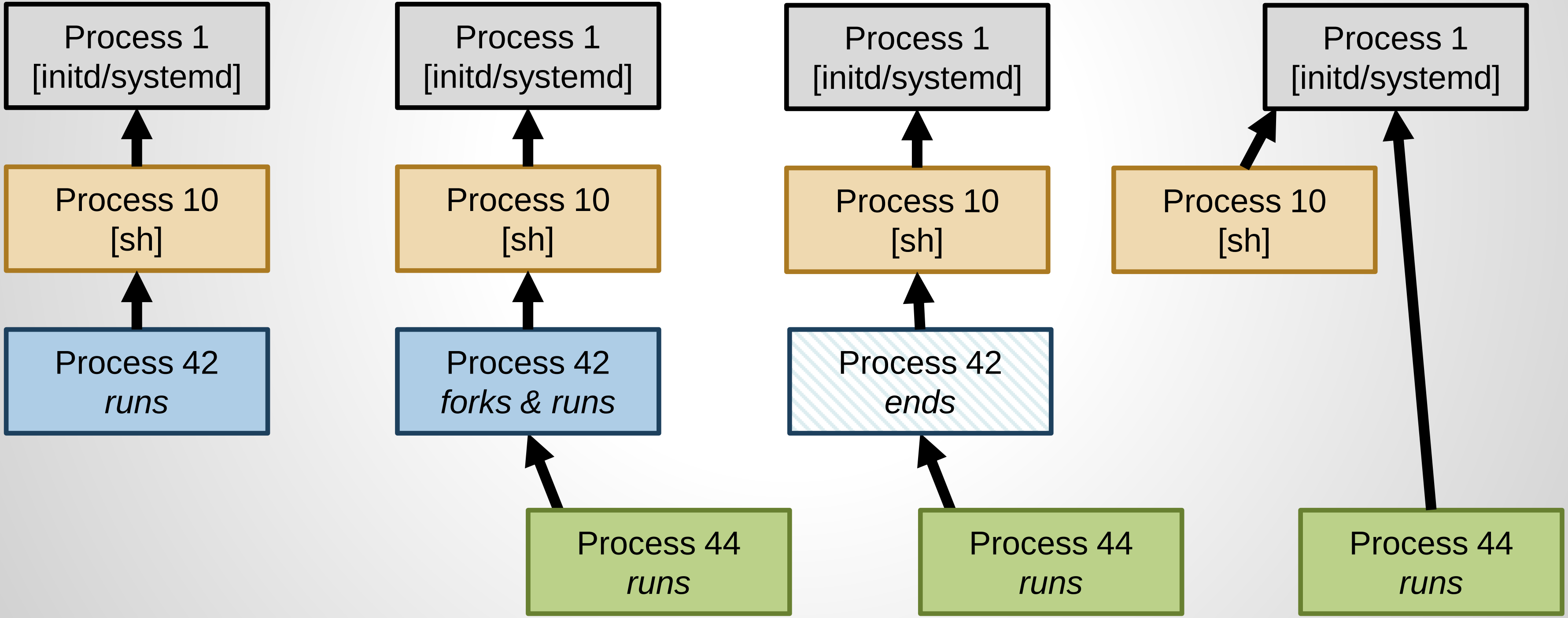
waitpid(2)

- Error cases:

- The given PID (*wpid)* does not exist, or is not a child
- The given PGID (*wpid*) does not exist, or is not a child

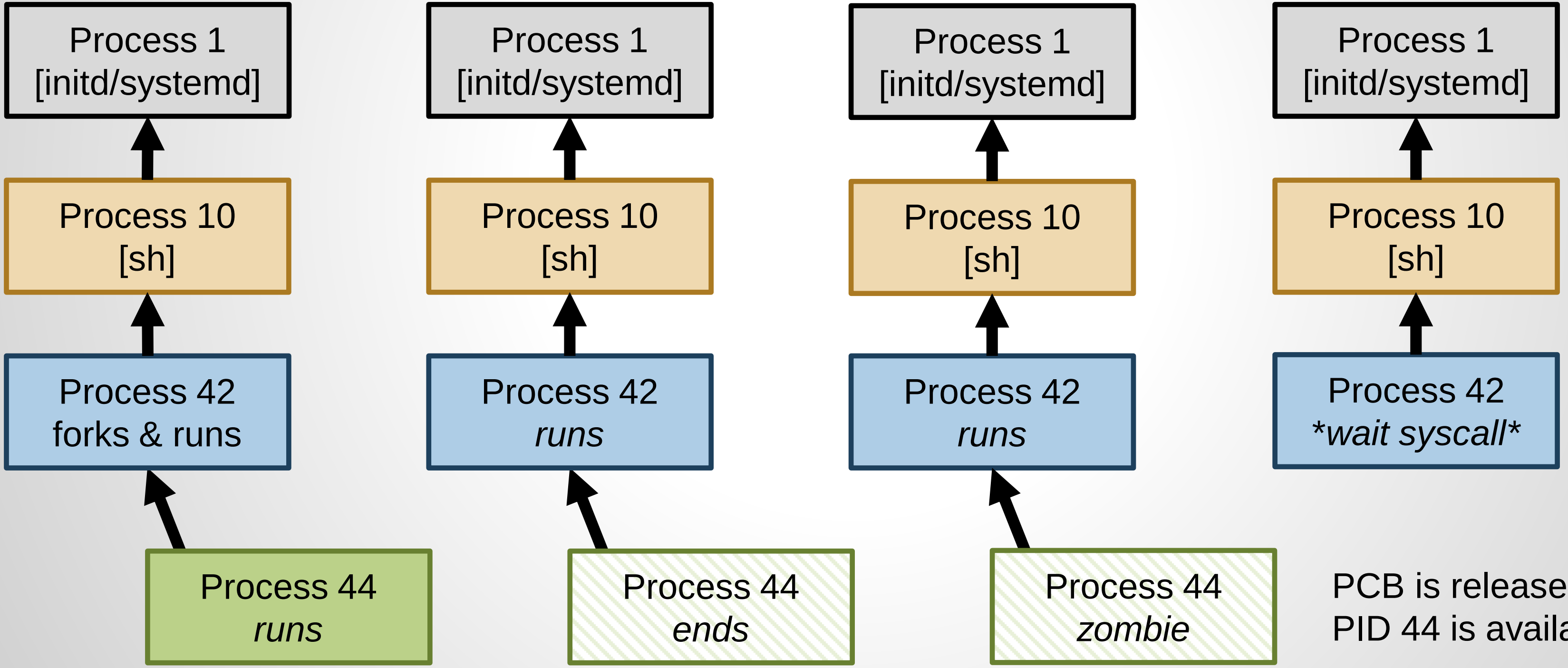
Process ending: father ends first

- If the father ends first, child is linked to PID = 1



Process ending: child ends first

- If the child ends first, it is still partially in memory



PCB is released and
PID 44 is available

Process ending: death, daemons (and witchcraft?)

- DAEMON: Disk And Execution MONitoring

- Processes running in background/attached to initd
  - Perfect for server-side programs (in client/server model)

- Easy creation with double fork(2) and waitpid(2)
  1. Fork => create 1st child
  2. Re-fork in 1st child => create 2nd child
  3. Exit the 1st child/Waitpid for the 1st child
  4. 2nd child becomes a daemon

Process state

- When a process ends, the Kernel sends a SIGCHLD signal to the father
  - The father can therefore use a method to automatically manages the death of each of his childs
- If the father asked the Kernel to ignore SIGCHLD… …the Kernel will automatically delete the child
  - No zombie

Process Manipulation

- Signals

- Process receive signals
  - From another process
  - From the kernel

- Multiple signals
  - ~36 signals

- Very similar to interrupts managed by the CPU

Process Manipulation

- Signals

- Signals can be caught (signal handler) or ignored (mask)
    - The developper declares to the system which signals to ignore or catch
    - Ignored: the system does nothing if the signal arrives
    - Caught: the developper writes a specific code to execute

- Asynchronous
    - **Management of a signal can be interrupted by another signal**
    - *When managing a signal you should [must...] mask other signals*

Process Manipulation

SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGTRAP	SIGABRT	…	SIGUSR2
0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	…	0 / 1

- Flag for each kind (binary value « stored »)
  - If the same signal n° is received multiple times, and not handled…
  - …only one will be caught and managed

- SIGUSR1 and SIGUSR2 are user defined
  - No default behavior
  - The developer writes what the program should do

Process Manipulation

- Signal handler
  - A custom procedure can replace the default one used by the OS
  - Beware: some functions or syscalls are forbidden in the signal handler
    Particularly malloc(3) or printf(3)… well, any non-reentrant function is forbidden… requires the *async-signal safe* property (read *signal-safety(7)*)

- 2[~3] signals are impossible to caught or to ignore:

  - SIGSTOP process is stopped
    (it's waiting for a SIGCONT to run again)

  - SIGKILL process is killed
  - *[SIGCONT process is awakened from a SIGSTOP] ← SPECIFIC CONTEXT*

Process Manipulation

- *fork* keeps the signal handlers in the child

- *execve* erases the handlers...
- ...but ignored signals are kept ignored
  - Handlers are in the address space: if it is erased, they are erased...
  - But ignored signals are kept in the PCB which is intact

Process Manipulation

-
  ```
  int   kill(pid_t pid,                          // (2) Send signal
             int sig);
  ```

-
  ```
  sighandler_t   signal(int signum,             // (3) Manages signal
                        sighandler_t handler);
  ```

-
  ```
  int   sigaction(int signum,                    // (2) Manages signal
                  const struct sigaction *act,
                  struct sigaction *oldact);
  ```

Process Manipulation

kill(2)

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

- Sends the signal « sig » to the process number « pid »
- The program might not have enough rights to send this signal to the targeted process…
  - Anyone cannot stop others’ processes, or even initd/systemd…

Process Manipulation

sigaction(2)

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

- Puts a handler to the signal *signum*, or ignore the signal
    - If *act* is NULL: ignore the signal
    - If *act* is not NULL: put the associated handler to manage it

- *oldact* is written by sigaction to give you the last handler

Scheduling

When to schedule ?

- Blocked/Sleeping process
- Terminated (or killed) process
- New process spawn
- Blocked/Sleeping process becomes ready

Multiprogramming

[old concept that is obvious nowadays]

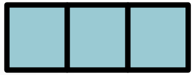
- Multiple programs are expected to run on the same system
- A blocking I/O allows another program to run

- It was opposed to the case were only 1 program could be run at a time in the whole address space
  - If an I/O was in a wait state (waiting for an input): nothing else was running or happening…

Example: Tasks to execute

Task 1

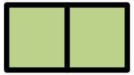
3 cycles to execute with one I/O
(ask the user for an input)



Task 1 is sent first

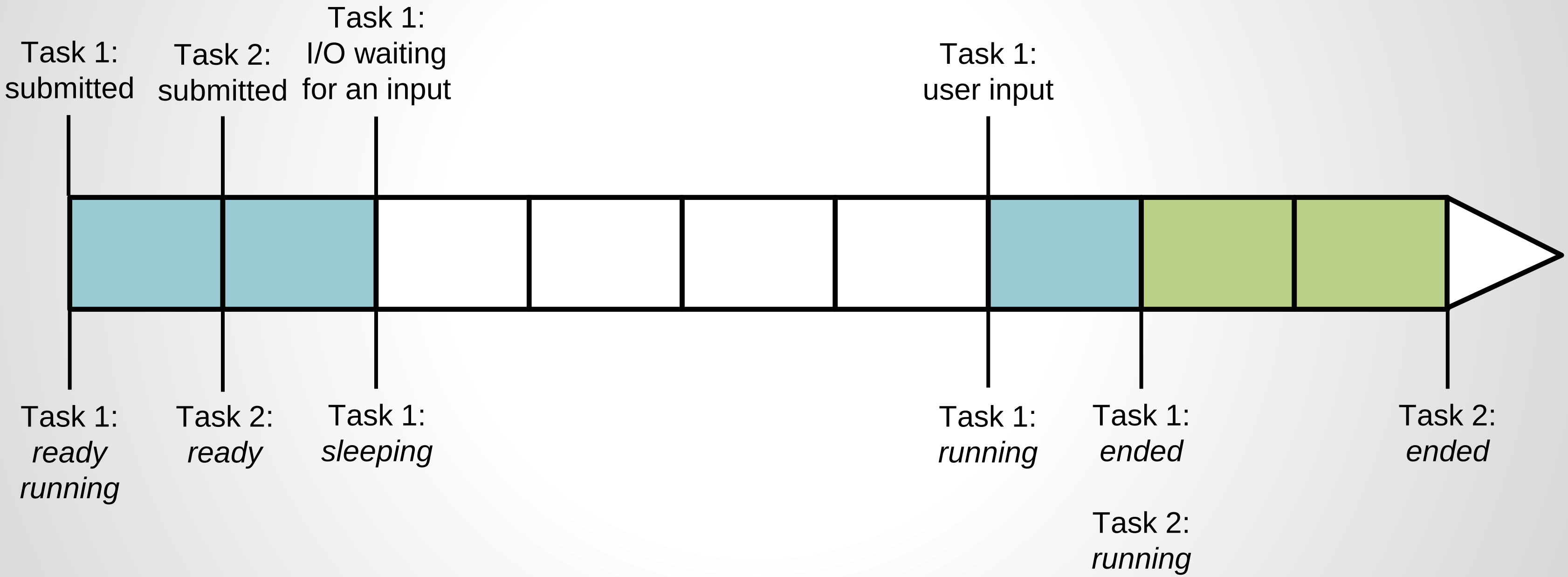
Task 2

2 cycles to execute without any I/O

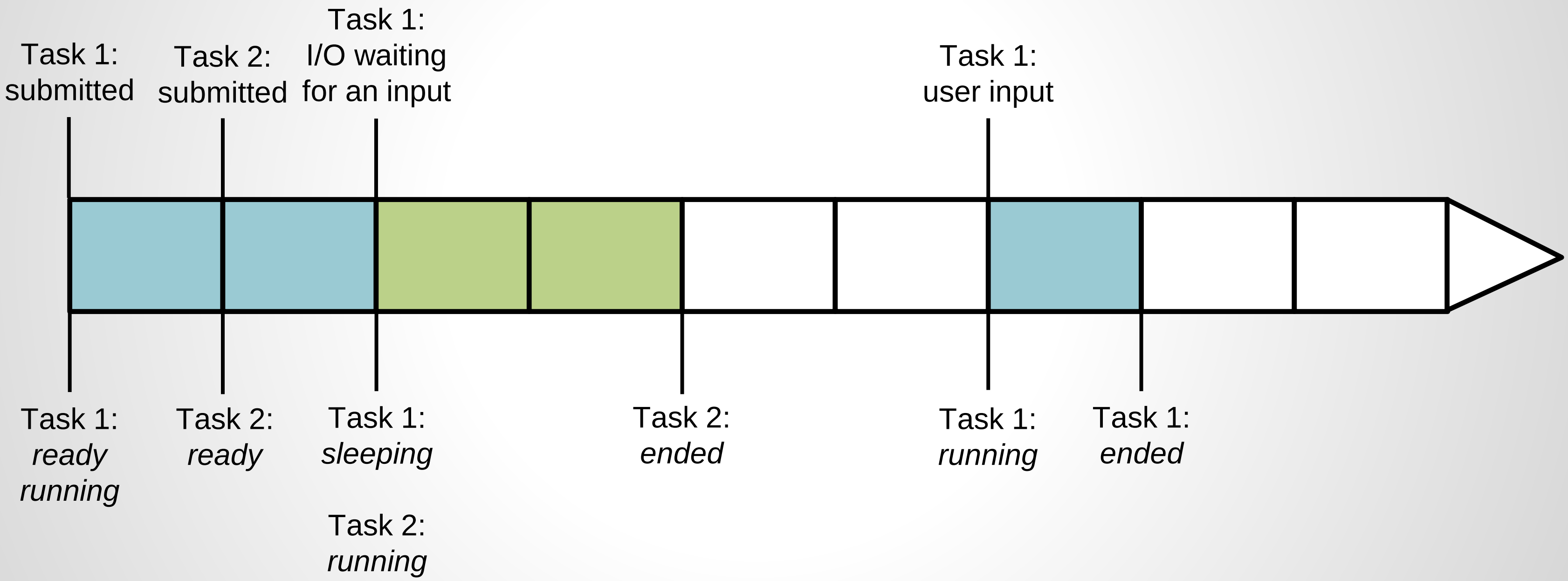


Task 2 is sent few time after Task 1

Example: A very simple (and old) scheduling

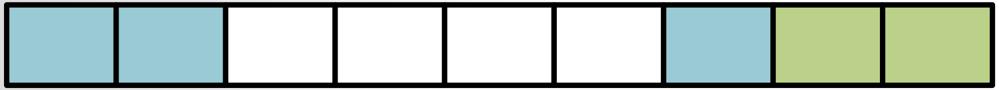


Example: Multiprogramming



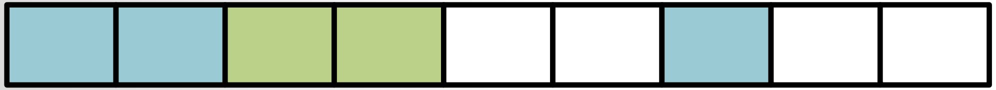
Example: Discussions

Case 1: Very simple (and old) scheduling



- 9 cycles were required to make all the tasks
- It depends of the device or user in case of an I/O
- No optimization at « all »/Time is lost !

Case 2: Multiprogramming



- 7 cycles were required to make all the tasks.
- It still depends of the device or user in case of an I/O
- Minimal optimization: in case of a blocking I/O, time is used

Multitasking

- Multiple programs share the resources of the system…
- …especially the CPU

- Multiple methods
  - Cooperative multitasking: each task decides when to release the CPU
  - Preemptive multitasking: the OS decides when to release the CPU

Types of schedulers

- **Cooperative:**
  Only blocked/sleeping or terminated processes

- **Preemptive:**
  All types of events
  (Requires a hardware support)

Scheduling criterias

- Different criteria to consider when trying to select the "best" scheduling algorithm
  - CPU utilization
  - Throughput
  - Turnaround time
  - Waiting time
  - Response time

Cooperative Multitasking

- The programmer must write instructions that release the CPU for another task (yield instructions, blocking I/O, syscalls, ...)

- If the program is buggy:
  the system might crash or stay in an infinite loop

Preemptive Multitasking

- The operating system is giving a *quantum of time* (or *time slice*) to each process
  - Processes are stopped by the operating system automatically…
  - …or during some specific instructions (blocking I/O, syscalls, …)

- If the program is buggy:
  it will spoil only its own time

Example: Tasks to execute

Task 1

Long task without any I/O

Task 1 is sent first

Task 2

Medium task with one I/O :

A « read » syscall is made on a file from a disk device

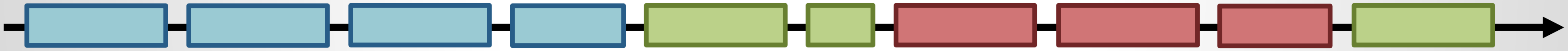
Task 2 is sent few time after Task 1

Task 3

Medium task without any I/O

Task 3 is sent few time after Task 2

Example: Cooperative Multitasking



Task 1:
ready
running

Task 2:
ready

Task 3:
ready

Task 1:
ended

Task 2:
running

Task 2:
« read »
syscall

Task 2:
sleeping

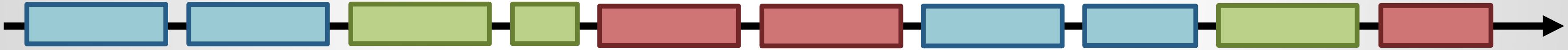
Task 3:
running

« read »
data are
ready

Task 3:
ended

Task 2:
ended

Example: Preemptive Multitasking



Task 1:
ready
running

Task 2:
ready

Task 1:
sleeping

Task 3:
ready

Task 2:
running

Task 2:
« read »
syscall

Task 2:
sleeping

Task 3:
running

« read »
data are
ready

Task 3:
sleeping

Task 1:
running

Task 1:
ended

Task 2:
running

Task 2:
ended

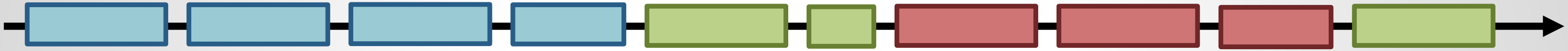
Task 3:
running

Task 3:
ended

Quantums of time per process: 2

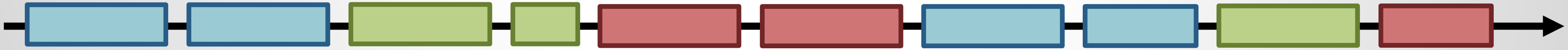
Example: Discussions

Case 1: Cooperative Multitasking



- Task 1 takes all the CPU first… Same for task 3 later (not very cooperative ☹)
- Depends on how the developper wrote his program
- OS has no control as long as there are no syscall (beware of infinite loop)
- User has nearly no control on the scheduling of tasks (just the choice on the launch)

Case 2: Preemptive mutlitasking (2 quantums of time maximum per process)



- All the applications were able to run regularly (time was nearly equally shared)
- Depends on the scheduler algorithm and parameters
- The OS has the control on the running tasks…
- …Therefore, the user can ask the OS to stop a buggy process

Time Sharing: why

- The small amount of times given to each process is enough to make the machine feels responsive for a human user
    - Web browser, music player, PDF reader running together smoothly

- All the tasks will run
    - No starvation of time for any process

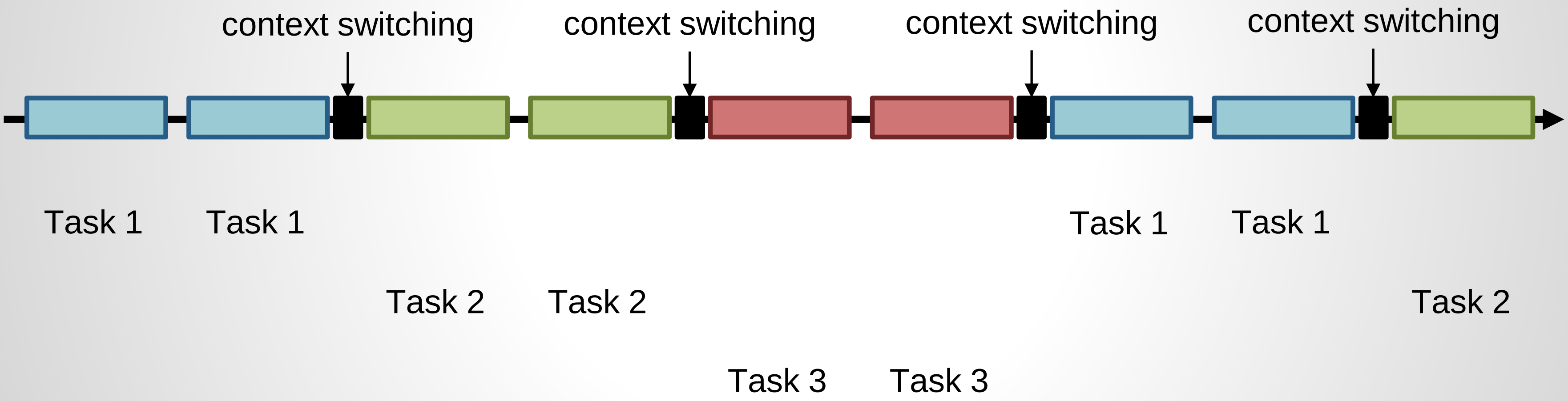
Time Sharing: how

- A specific clock makes regular interruptions
  - The OS takes back the control and eventually choose another process to run
  - The quantum of time chosen is the maximum value a process can use before the OS takes back the control

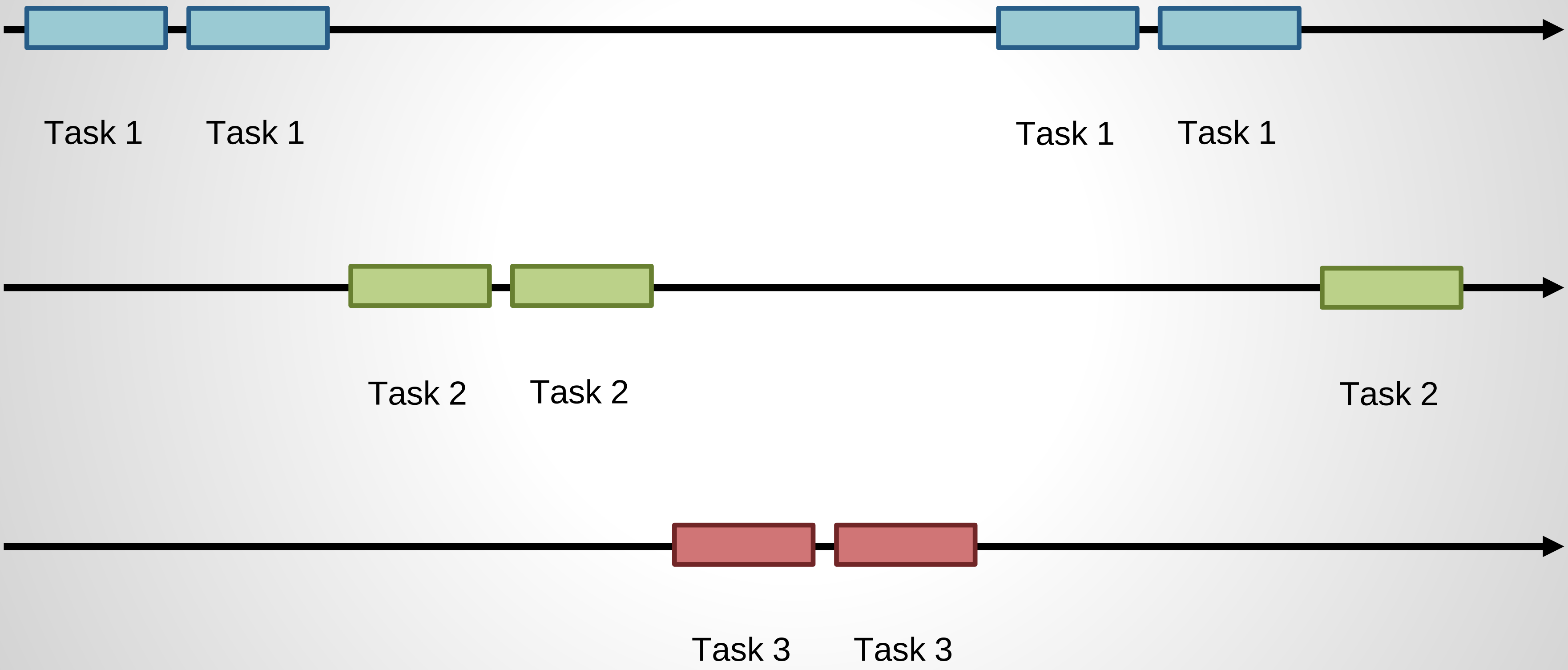
- When changing of process a *context switching* happens
  - It’s a pretty heavy operation: all of the registers, pages used, … (the context of the process) must be saved/restored

- The more active processes there are:
  - The more it will be long for a task to take back the control
  - The more contexts switching will happen… and will lose time

Time Sharing: OS POV



Time Sharing: Process POV



Time Sharing

- Simplified example with 3 tasks using *Round Robin* algorithm and 2 quantums of time

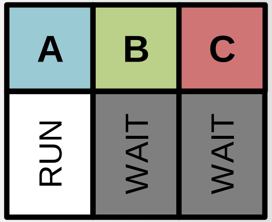
- Each process is waiting for the 2 others to use their times

- What if there are more than 3 processes?...
  - Longer time to wait for each process to get the CPU

Scheduling

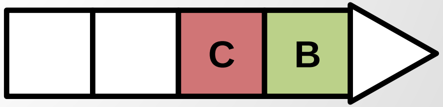
- Process table (list all processes)
  - ○ https://en.wikipedia.org/wiki/Process_(computing)
- Queue with ready processes
- Queues with blocked processes

process table



A	B	C
RUN	WAIT	WAIT

ready queue



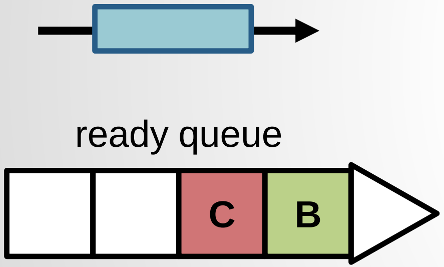
Simplified example of context switching

- Each task can use its quantum (or time slice) for running

- At the end of the quantum:
  1. An interruption stops the running process A
  2. The process A is put in a « waiting » state in the OS
  3. The OS saves the context of the process A
  4. The OS puts the process A at the tail of the ready queue
  5. The OS takes the process at the head of the ready queue (process B)
  6. The OS loads the context of the process B
  7. The process B is put in « running » state

Example of scheduler: Round Robin

- The process A is running

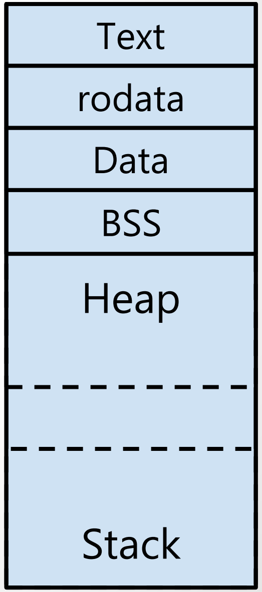


ready queue

process table

A	B	C
RUN	WAIT	WAIT

```
MOV   R2, R1
ADD   R2, 42
MUL   R2, 3
MOV   R1, R3
ADD   R1, R2
MUL   R1, 2
CMP   R1, R2
JGT   BIGGER
JMP   LOWER
```



Text

rodata

Data

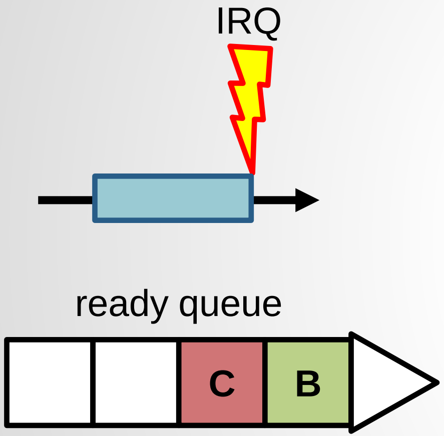
BSS

Heap

Stack

Example of scheduler: Round Robin

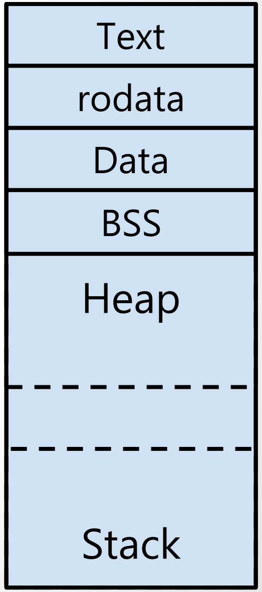
1. An interruption stops the running process A



process table

A	B	C
RUN	WAIT	WAIT

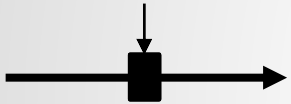
```
MOV   R2, R1
ADD   R2, 42
MUL   R2, 3
MOV   R1, R3
ADD   R1, R2
MUL   R1, 2
CMP   R1, R2
JGT   BIGGER
JMP   LOWER
```



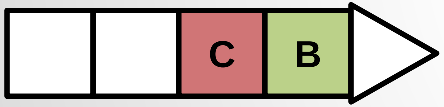
Example of scheduler: Round Robin

2. The process A is put in a « waiting » state in the OS

context switching



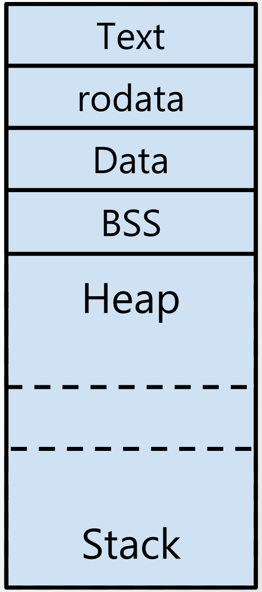
ready queue



process table

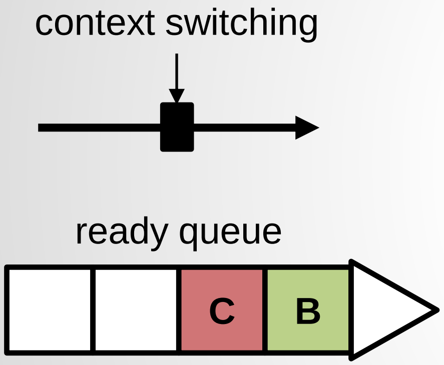
A	B	C
WAIT	WAIT	WAIT

```
MOV   R2, R1
ADD   R2, 42
MUL   R2, 3
MOV   R1, R3
ADD   R1, R2
MUL   R1, 2
CMP   R1, R2
JGT   BIGGER
JMP   LOWER
```



Example of scheduler: Round Robin

3. The OS saves the context of the process A


context switching

ready queue

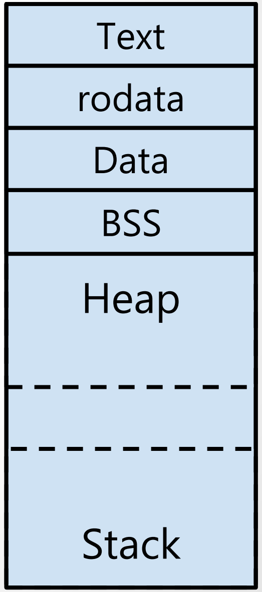
C B

process table

A	B	C
WAIT	WAIT	WAIT

Context of Process A:
- stopped at instruction 5
- registers had values:

  R1 = XXX
  R2 = YYY
  R3 = ZZZ
  …
- …


Text

rodata

Data

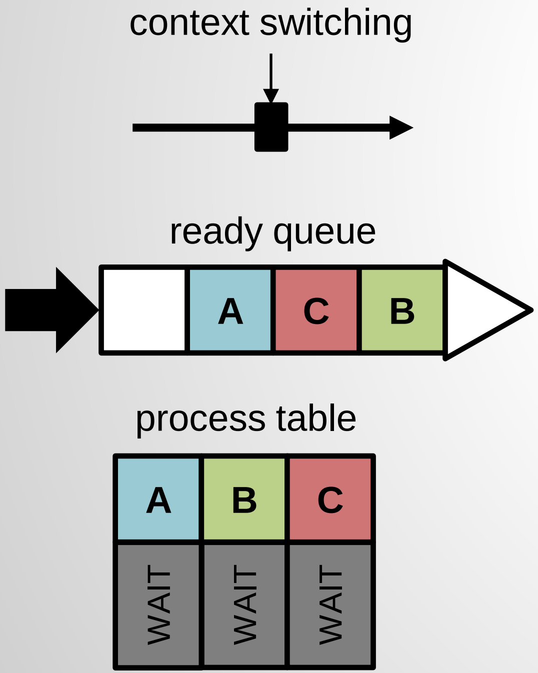
BSS

Heap

Stack

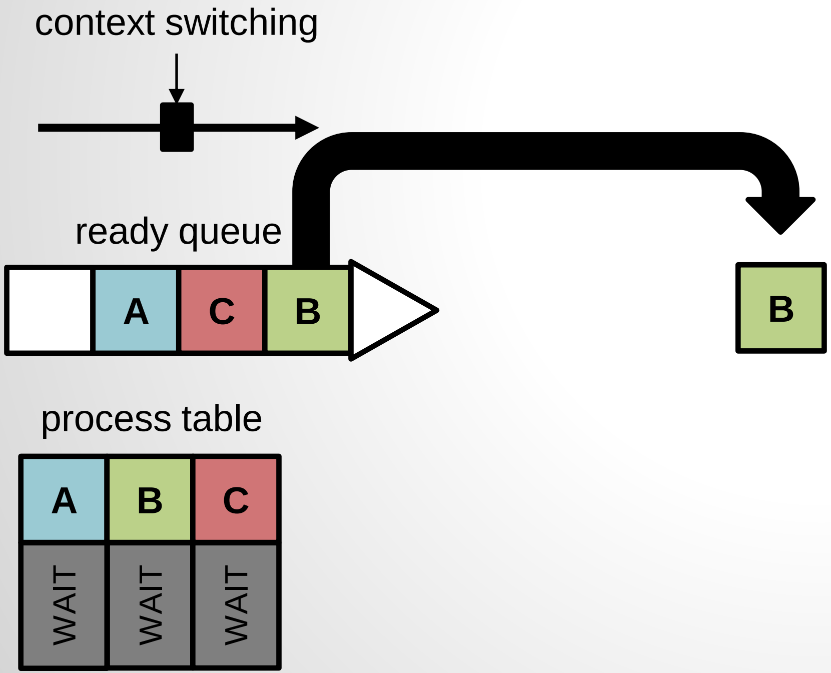
Example of scheduler: Round Robin

4. The OS puts the process A at the tail of the ready queue



Example of scheduler: Round Robin

5. The OS takes the process at the head of the ready queue



context switching

ready queue

	A	C	B

B

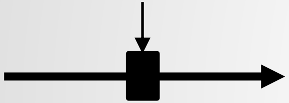
process table

A	B	C
WAIT	WAIT	WAIT

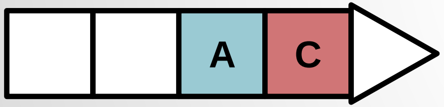
Example of scheduler: Round Robin

6. The OS loads the context of the process B

context switching



ready queue

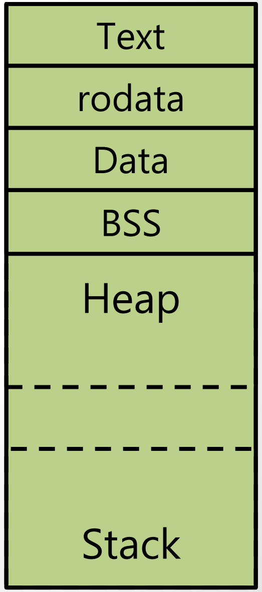


process table

A	B	C
WAIT	WAIT	WAIT

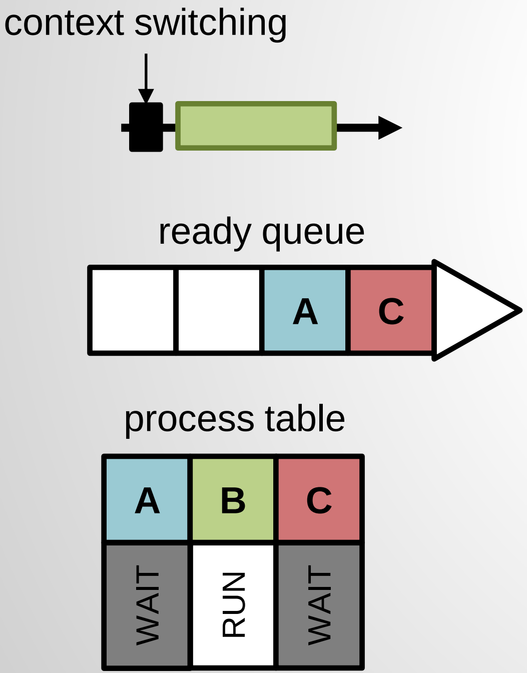
Context of Process B:

- stopped at instruction 2
- registers had values:
  R1 = 111
  R2 = 222
  R3 = 333
  …
- …



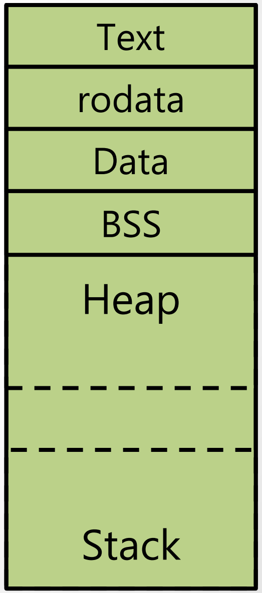
Example of scheduler: Round Robin

8. The process B is put in « running » state



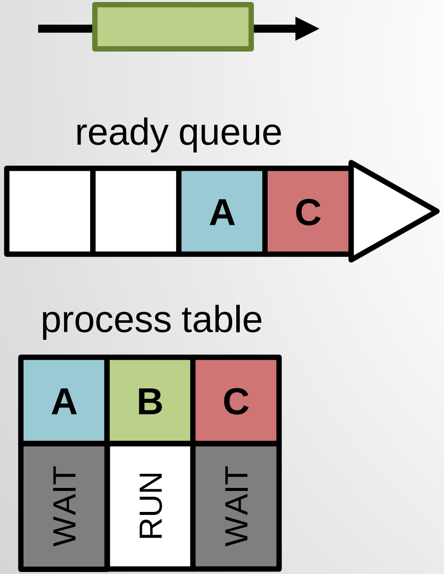
A	B	C
WAIT	RUN	WAIT

```
MUL    R4, 66
SUB    R1, 10
DIV    R1, 2
CMP    R1, R4
JNE    DIFFER
MUL    R1, 2
CMP    R1, R4
JNE    CATCH
CLR    R1
```



Example of scheduler: Round Robin

- The process B is running

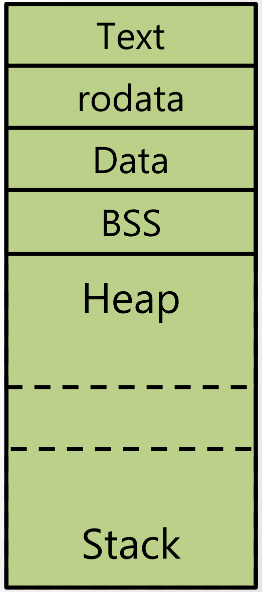


ready queue

process table

A	B	C
WAIT	RUN	WAIT

```
MUL    R4, 66
SUB    R1, 10
DIV    R1, 2
CMP    R1, R4
JNE    DIFFER
MUL    R1, 2
CMP    R1, R4
JNE    CATCH
CLR    R1
```



Example 1: Round Robin scheduler

- Each task gets N quantums of time on each turn
(or less if a blocking I/O is made)

- Each task in the queue will be run when it will be its turn
(queue = FIFO = First In, First Out)

- -> Each task is « sure » to be executed

Example 1: Round Robin scheduler

Quantums of time per process: 2

Task 1

Long task without any I/O

Task 1 is sent first

Task 2

Medium task with one I/O :

A « read » syscall is made on a file from a disk device

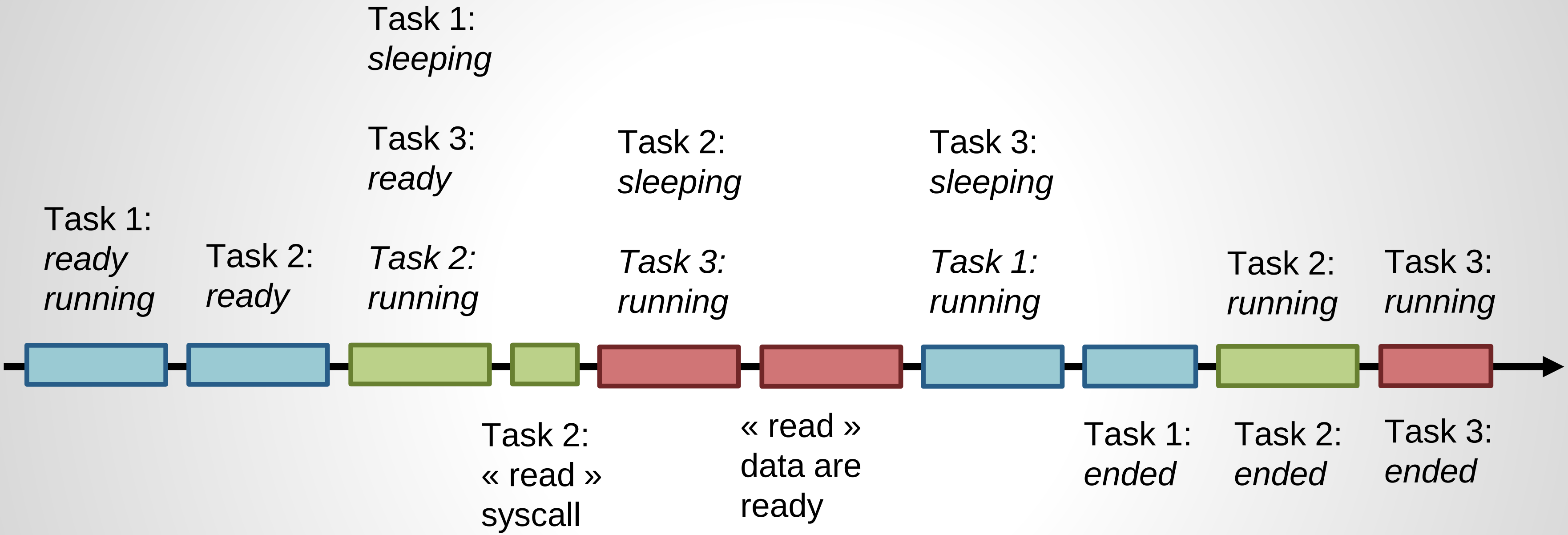
Task 2 is sent few time after Task 1

Task 3

Medium task without any I/O

Task 3 is sent few time after Task 2

Example 1: Round Robin scheduler



Task 1: *ready* *running*

Task 2: *ready*

Task 1: *sleeping*

Task 3: *ready*

Task 2: running

Task 2: « read » syscall

Task 2: *sleeping*

Task 3: running

« read » data are ready

Task 3: *sleeping*

Task 1: running

Task 1: *ended*

Task 2: *running*

Task 2: *ended*

Task 3: *running*

Task 3: *ended*

Quantums of time per process: 2

Example 2: Round Robin + priority

- Same requirements as the Round Robin
  - Each task gets N quantums of time on each turn (or less if a blocking I/O is made)
  - Each task in the queue will be run when it will be its turn

- Priorities are added!
  - Priority on GUI tasks (graphical client)
  - Priority on background tasks (servers)
  - Manual priorities
  - ...

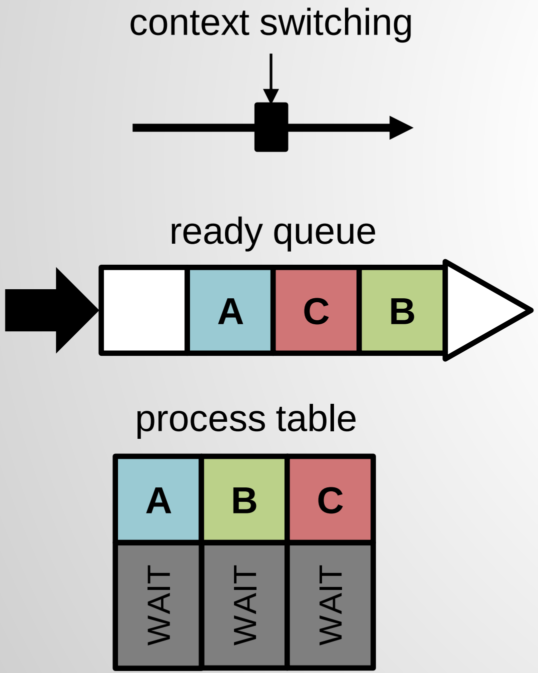
Example 2: Round Robin + priority

- The task with the higher priority will be run first
  - The 1st in the queue if they have the same priority

- During each context switching, priorities in the ready queue are updated
  - +1 for all tasks
  - +5 for prefered tasks (background/foreground/any criterion)

Example of scheduler: Round Robin + priority

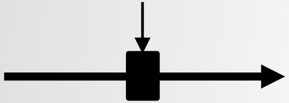
4. The OS puts the process A at the tail of the ready queue



Example of scheduler: Round Robin + priority

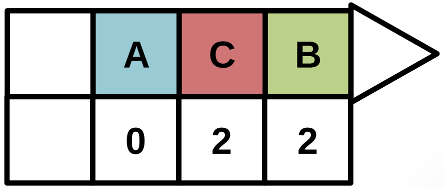
- The OS puts the process A at the tail of the ready queue

context switching



and updates the priorities

ready queue



	A	C	B
	0	2	2

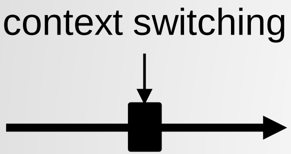
process table

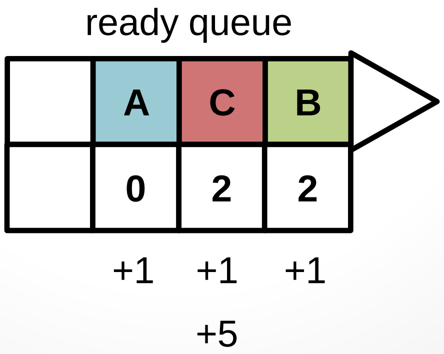
A	B	C
WAIT	WAIT	WAIT

Example of scheduler: Round Robin + priority

- The OS puts the process A at the tail of the ready queue

and updates the priorities




ready queue

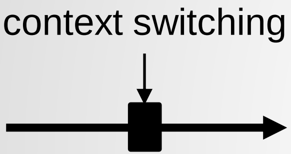
	A	C	B
	0	2	2
	+1	+1	+1
		+5	

process table

A	B	C
WAIT	WAIT	WAIT

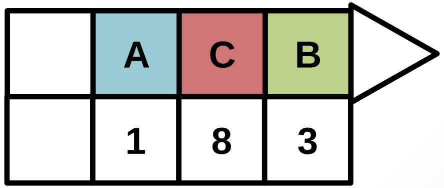
Example of scheduler: Round Robin + priority

- The OS puts the process A at the tail of the ready queue



and updates the priorities

ready queue



	A	C	B
	1	8	3

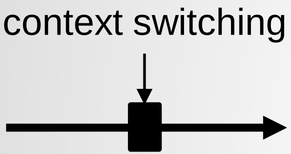
process table

A	B	C
WAIT	WAIT	WAIT

Example of scheduler: Round Robin + priority

- The OS puts the process A at the tail of the ready queue

and updates the priorities



ready queue

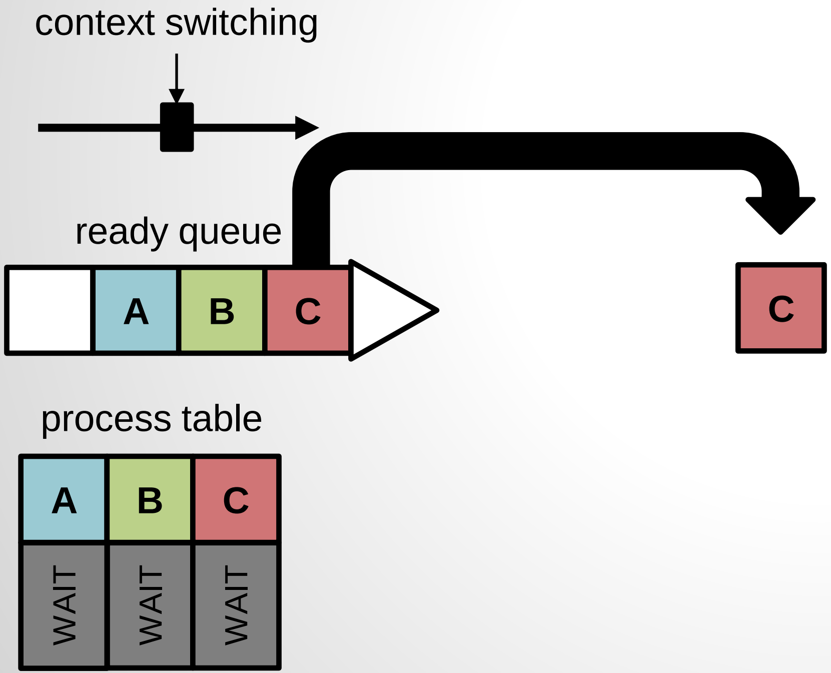
	A	B	C
	1	3	8

process table

A	B	C
WAIT	WAIT	WAIT

Example of scheduler: Round Robin + priority

5. The OS takes the process with the higher priority at the head of the ready queue



Example 2: Round Robin + priority

- Way better if a priority criterion is required!

- Graphical applications are more responsives
  - But they might process slower

- Servers processes more
  - Some might be I/O prioritized…
  - Others might be calculus oriented…

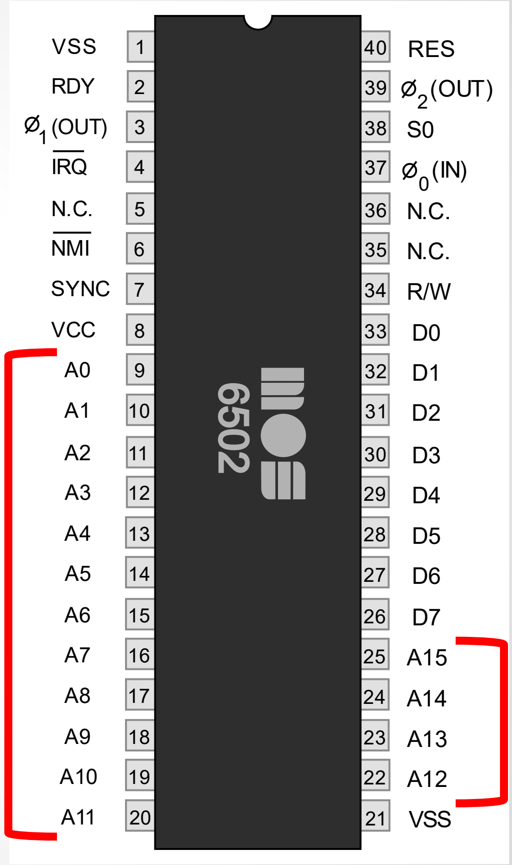
ps(1) & kill(1)

- Check the various status of a process in the man
- Question:
  Why a process in the « Z » state cannot disappear with a SIGKILL?

Memory Management

Memory Protection

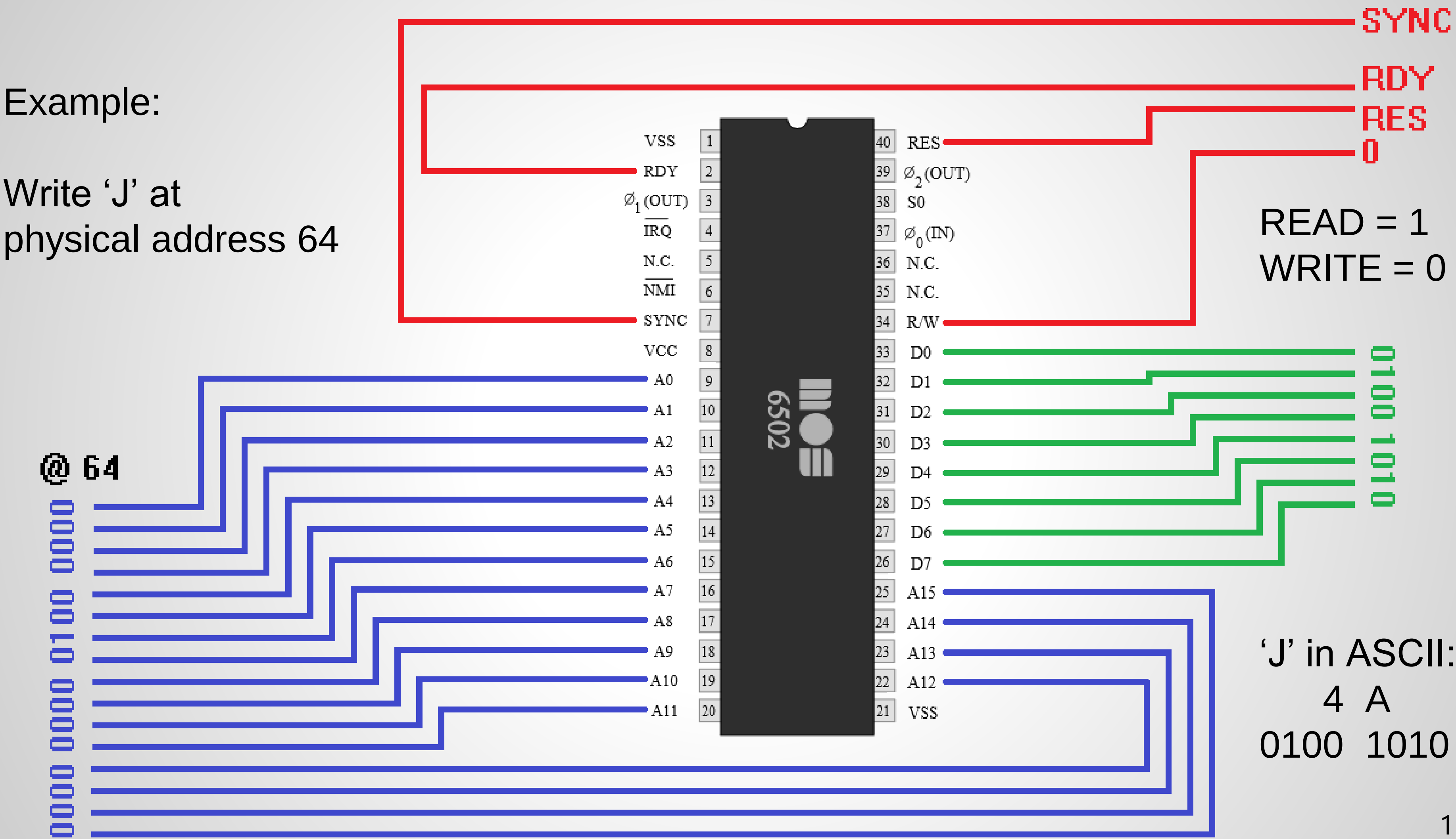
- Physical Addresses
- Chips are accessed by wires at physical addresses
  - *Through the address bus*



(6502 reproduit par Bill Bertram)

Example:

Write ‘J’ at
physical address 64



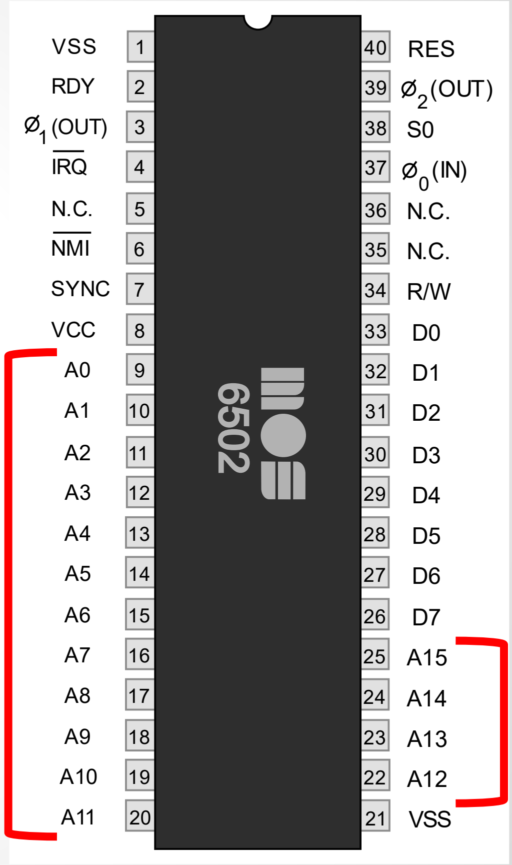
READ = 1
WRITE = 0

‘J’ in ASCII:
4 A
0100 1010

Memory Protection

- Anything is possible!
  - Any process might write anywhere…
  - …even on his own code…
  - …even on the code of other processes
  - *(cf Core War)*

- How to avoid problems?
  - Separation
  - Privileges
  - Abstraction
  - …



(6502 reproduit par Bill Bertram)

Memory Protection

- Requires a mechanism for protection
  - Keep a list of « areas » per processes
  - Keep the rights per processes

  - **Memory Management Unit (MMU)**

- Virtualization
  - Segmentation *(obsolete)*
  - Pagination

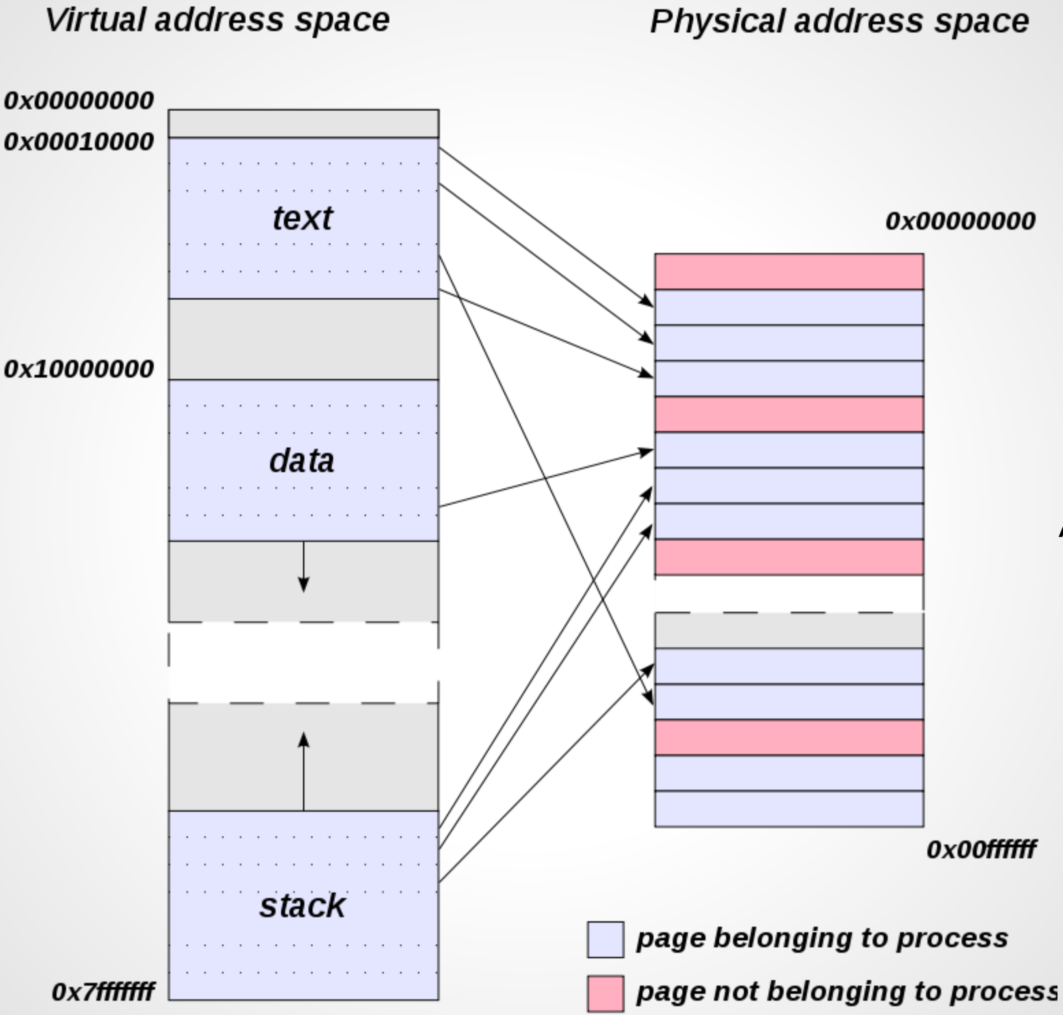
Memory Virtualization

- In the CPU
    - Memory Management Unit (MMU)
    - Page Table/Page Directory: contains memory mappings
    - Page Directory Base Register (PDBR): address to an address space
- In the OS
    - 1 PDBR per task => isolated address space

Linear address

Address in the virtual address space

Userland programs use pointers in the virtual address space only



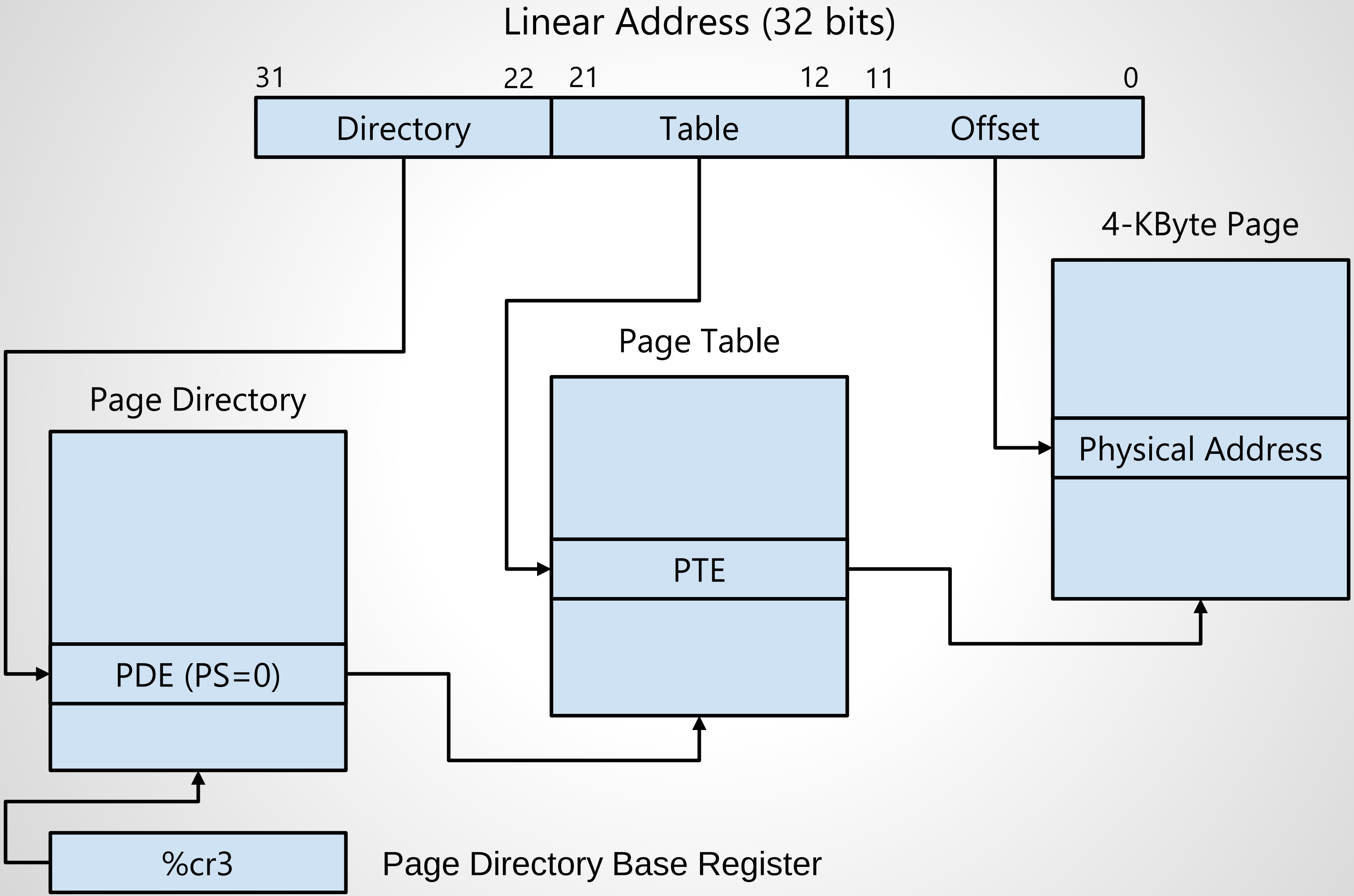
Physical address

Address in the physical address space

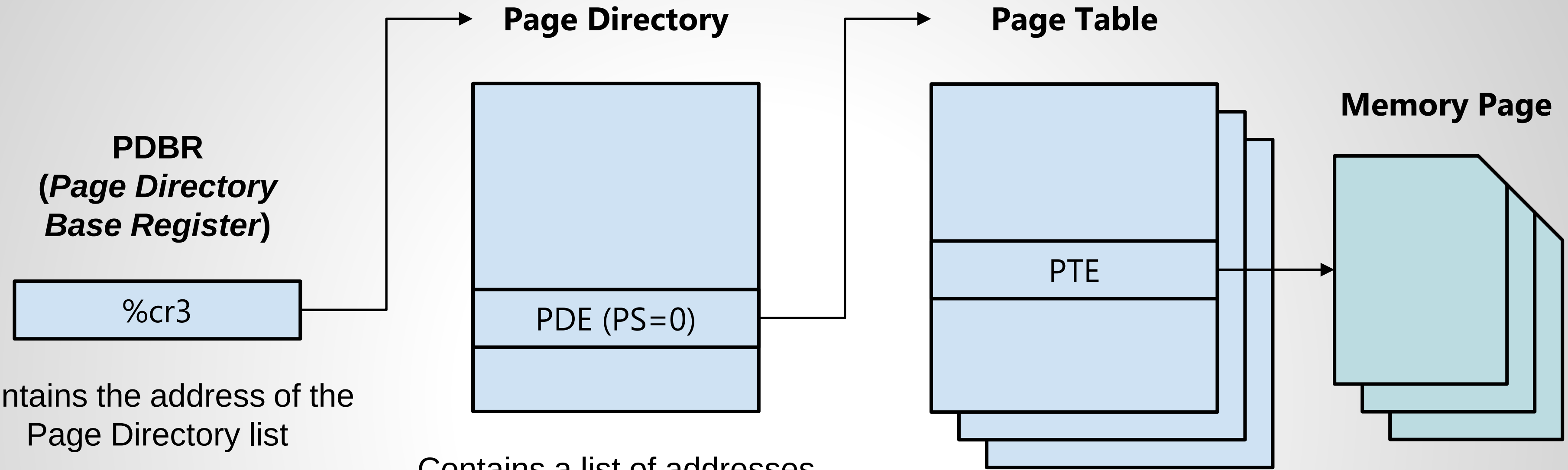
Kernel works with both virtual and physical addresses

(it can indirectly manipulates the physical addresses)

In the case of 32 bits x86



In the case of 32 bits x86



Contains the address of the Page Directory list

(allows to find the list of PDE)

1 PDBR per process

Contains a list of addresses pointing to Page Table arrays

(allows to find the right list of PTE)

Multiple Page Directories exist per process

Contains a list of addresses pointing to Memory Pages

(allows to find the right page we are searching for)

Multiple Page Table exist per Page Directory

Contains multiple addresses for storing data

(usually 4096 addresses per pages)

In the case of 32 bits x86

Example

Virtual address 0xCAFEBABE - 1100 1010 1111 1110 1011 1010 1011 1110

PDE(10b) - 32B 1100 1010 11

PTE(10b) - 3EB 11 1110 1011

Offset(12b) - ABE 1010 1011 1110

cr3 - 0x00104000

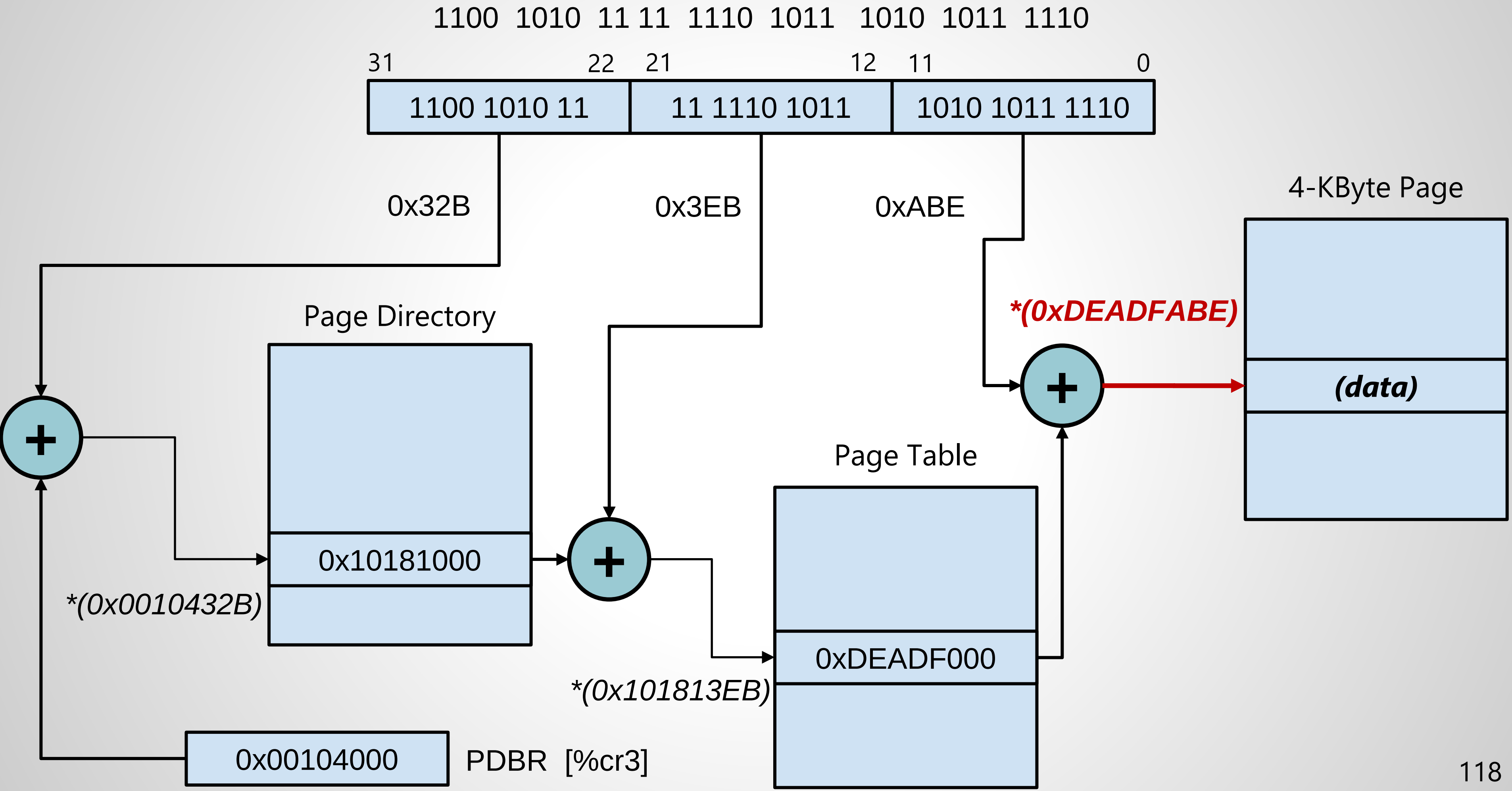
PDE at *(0x00104000 + 0x32B) == *(0x0010432B) -> (0x10181000 | flags)

PTE at *(0x10181000 + 0x3EB) == *(0x101813EB) -> (0xDEADF000 | flags)

Virtual address 0xCAFEBABE -> Physical address 0xDEADFABE

In the case of 32 bits x86

0xCAFEBABE



1100 1010 11 11 1110 1011 1010 1011 1110

31 22 21 12 11 0

1100 1010 11 | 11 1110 1011 | 1010 1011 1110

0x32B

0x3EB

0xABE

4-KByte Page

(data)

***(0xDEADFABE)**

Page Directory

0x10181000

*(0x0010432B)

Page Table

0xDEADF000

*(0x101813EB)

0x00104000

PDBR [%cr3]

Memory Virtualization

- Example in the 32 bits case for intel x86
  - 2 levels of directories and tables (before the page itsel)
  - Variations in the 64 bits case
  - (or even in some others 32 bits cases)
- PDBR (Page Directory Base Register)
  - Might be a PTBR (Page Table Base Register) if no Page Directory
- The same pattern may repeat itself for larger cases
  - 4 levels in the current 64 bits cases
  - Just check which offsets of the linear address are used

Memory Virtualization

- Kernel maps new pages in memory
    - Kernel updates the PCB
    - and all the structures that references the pages (PTE, PDE, ...)

- Kernel manages the context switching
    - Load/Unload %cr3 from PCB
    - Update various informations (R/W/E on each page, ...)

- Kernel manages also optimization mechanisms
    - Translation Lookaside Buffer (TLB), ...