Operating Systems: Processes & Scheduling

Fabrice BOISSIER <fabrice.boissier@epita.fr>

2021-10-19

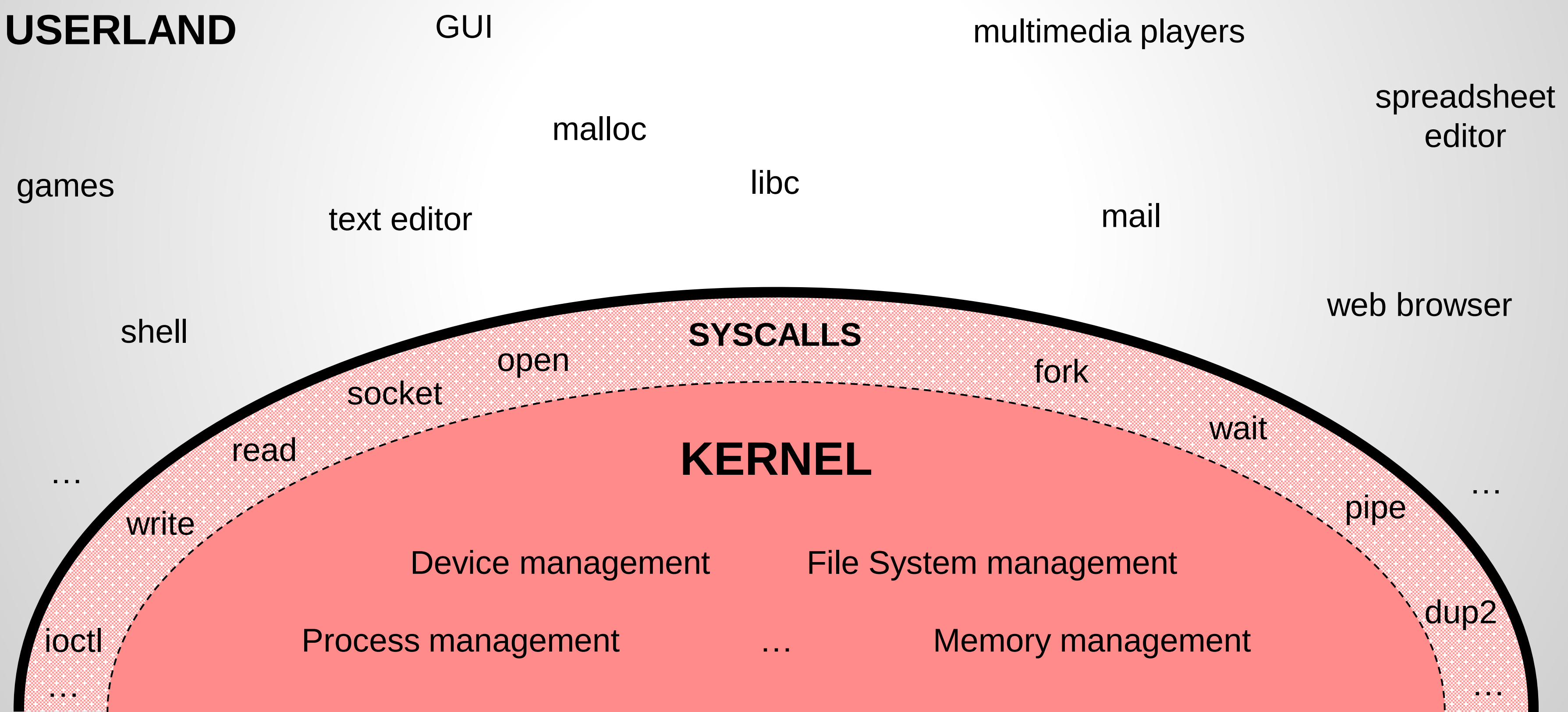
The “OS API”

- 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 »



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?

ABI

- Calling convention
  - Defines how to call a function in machine language (and assembly)
  - Where and in which order puts arguments of a function? Registers? Stack? 1st argument in the lower/upper end of the stack?
- Alignments of data types
  - What is the length in bits of an integer? Of a character?
- Object file format
  - How to describe and load binary code into memory?
- Syscalls
  - How to use a syscall? (push arguments on stack? Or in a register?)
- ...

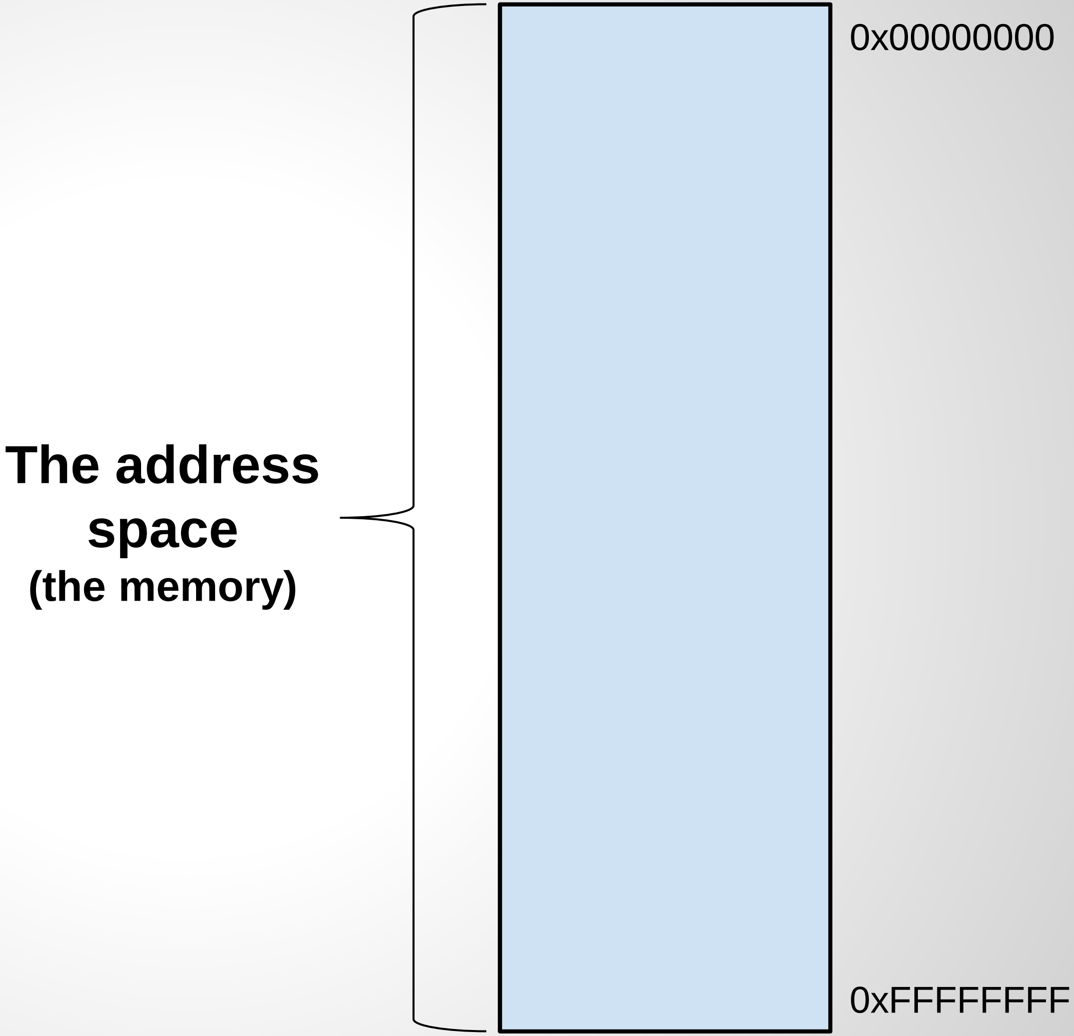
(example https://www.uclibc.org/docs/psABI-x86_64.pdf)

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

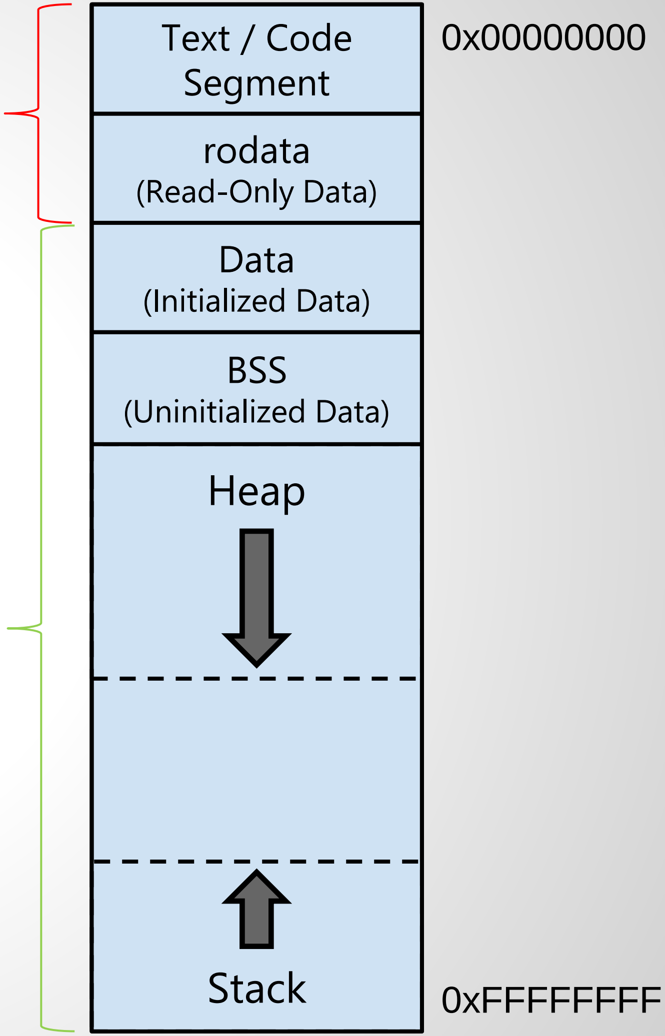
```c
char *myStr;
int i = 0;
int       main(void)
{
const char *var = "Test";
int a = 1337;

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



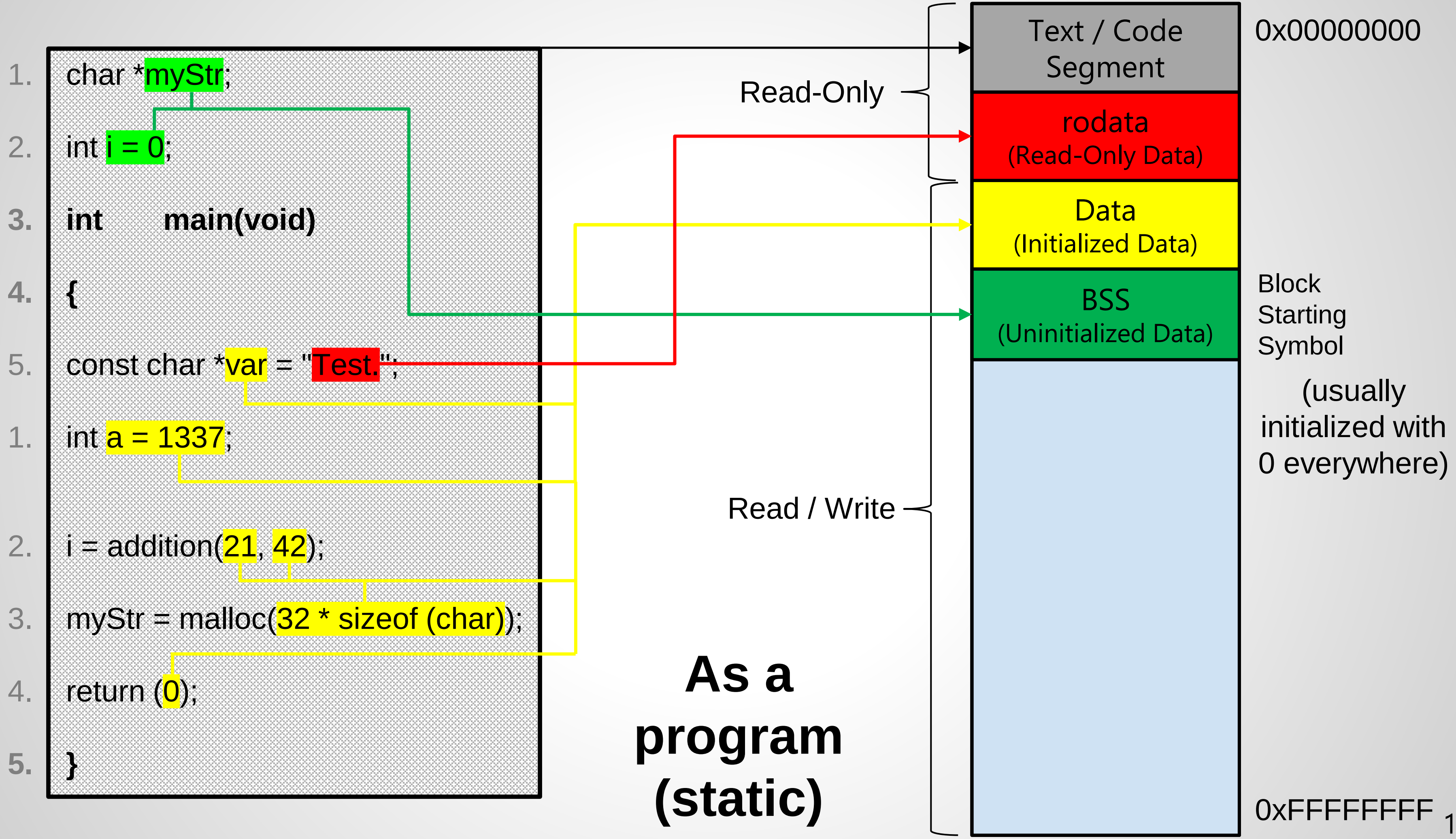
```c
char *myStr;
int i = 0;
int     main(void)
{
const char *var = "Test";
int a = 1337;

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

The address space

(historically, but main concepts are still present)

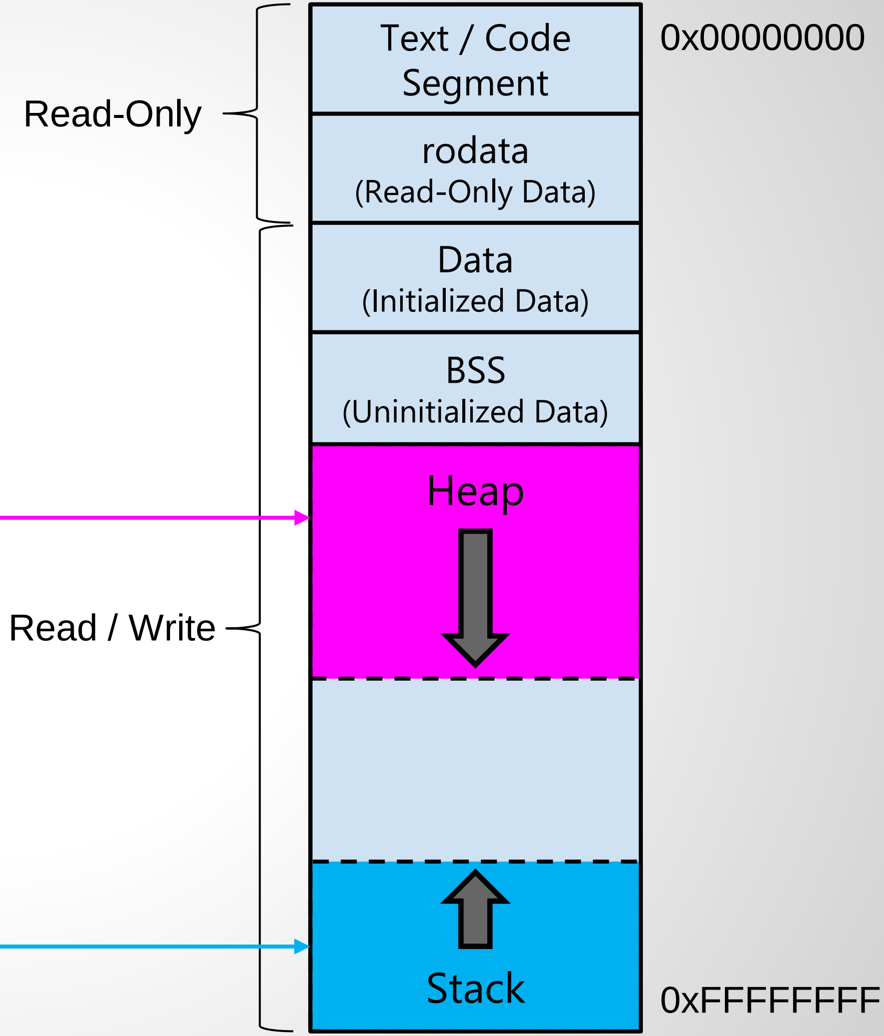


As a program (static)



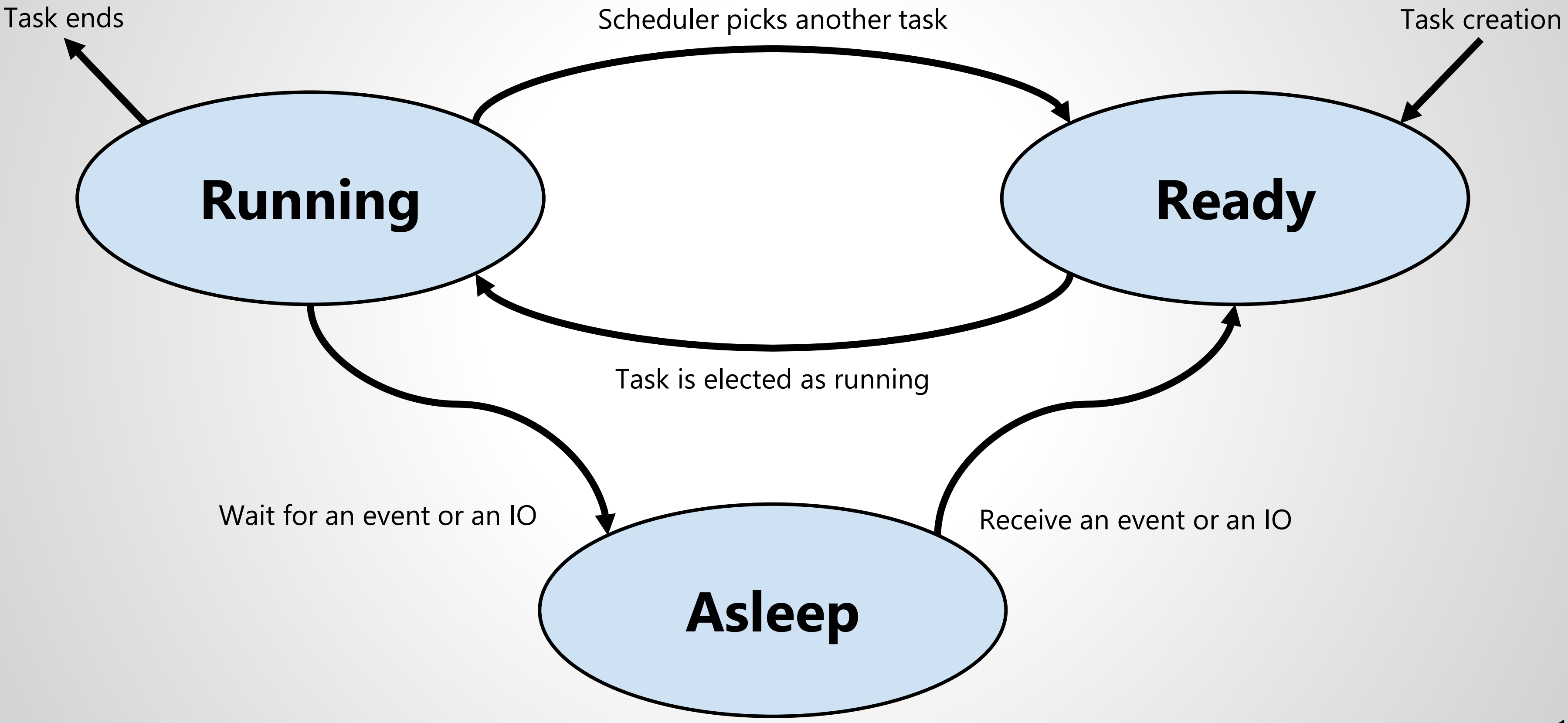
As a process (running)

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

i = addition(21, 42);
mySt = 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*

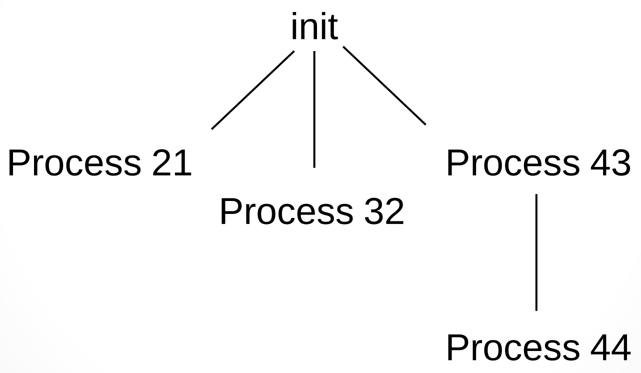
- *struct task_struct* in Linux, *PEB* on Windows

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

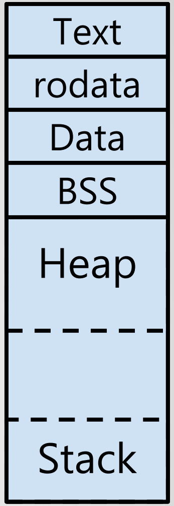
```c
#include <err.h>
#include <stddef.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

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

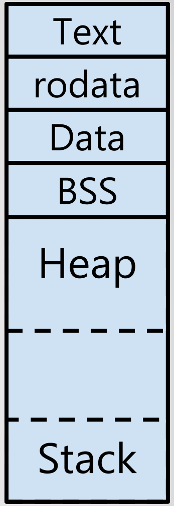


POV: code

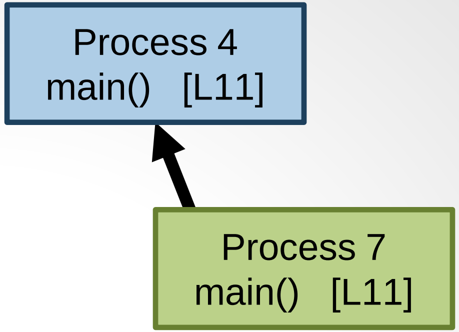
Process 4
main() [L11]

```
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. }
```

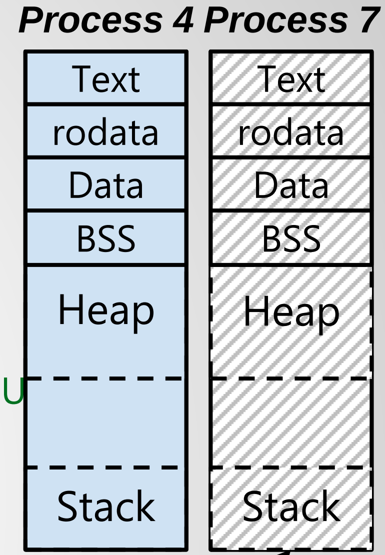
Process 4



POV: code

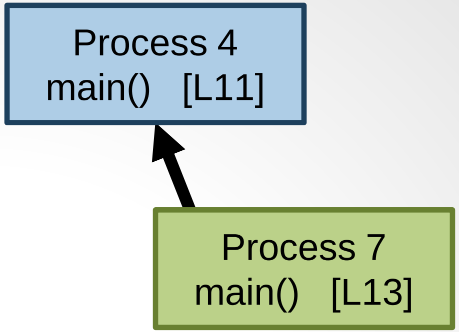



```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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. }
```

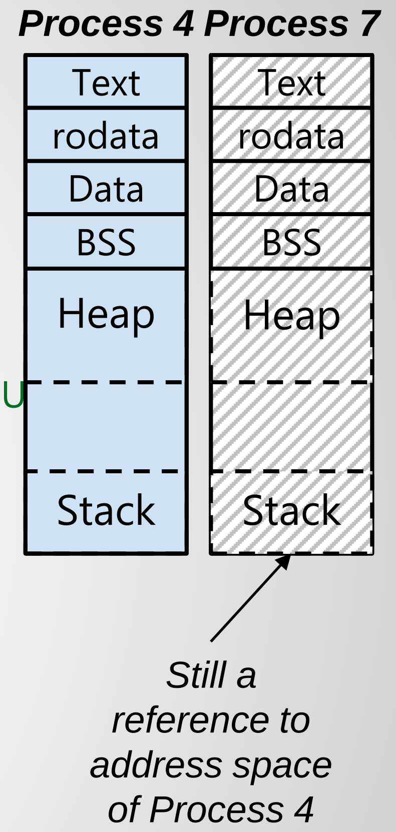


Still a reference to address space of Process 4

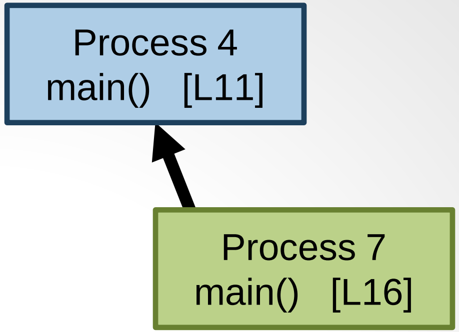
POV: code



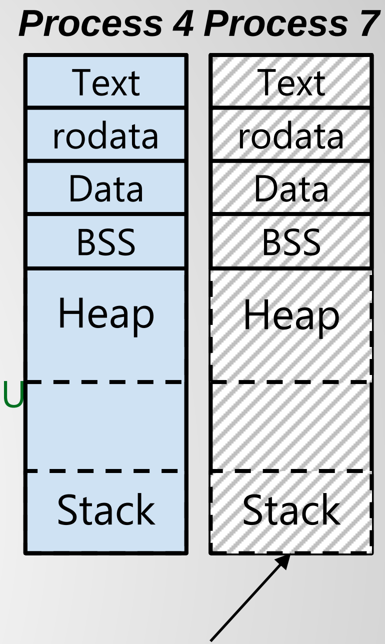

```c
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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



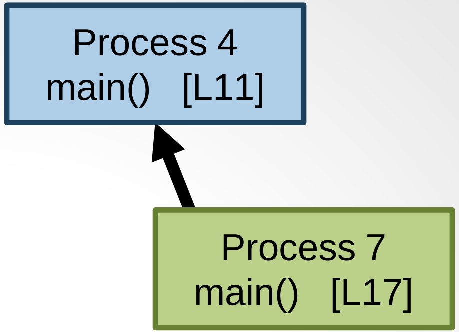

```c
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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. }
```

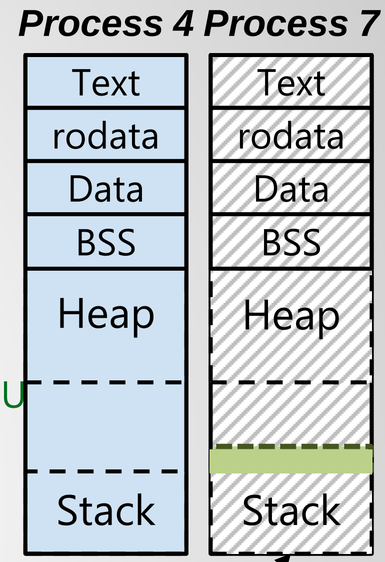


Still a reference to address space of Process 4

POV: code

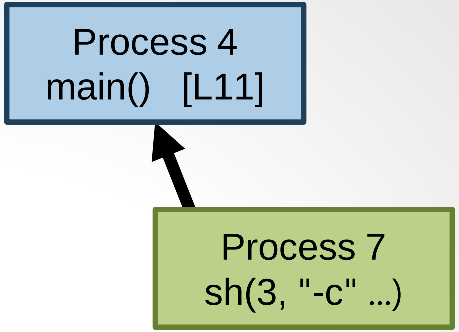
```
1.   #include <err.h>
2.   #include <stddef.h>
3.   #include <sys/types.h>
4.   #include <sys/wait.h>
5.   #include <unistd.h>
6.
7.   int main(int argc, char **argv, char **envp)
8.   {
9.           char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.          int status;
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.  }
```



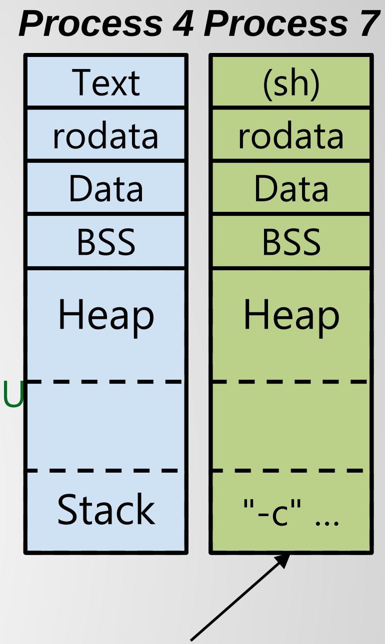


The stack evolved partially with the arguments

POV: code



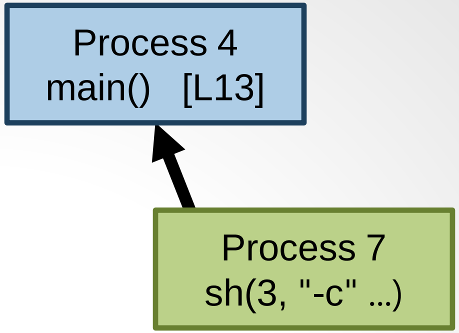

```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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. }
```

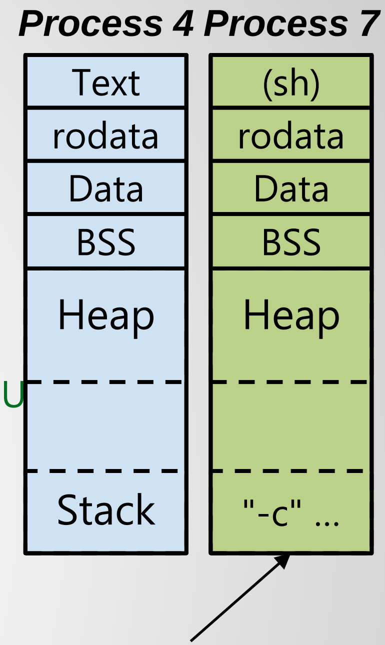


New address space containing « /bin/sh » code and the parameters

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. }
```

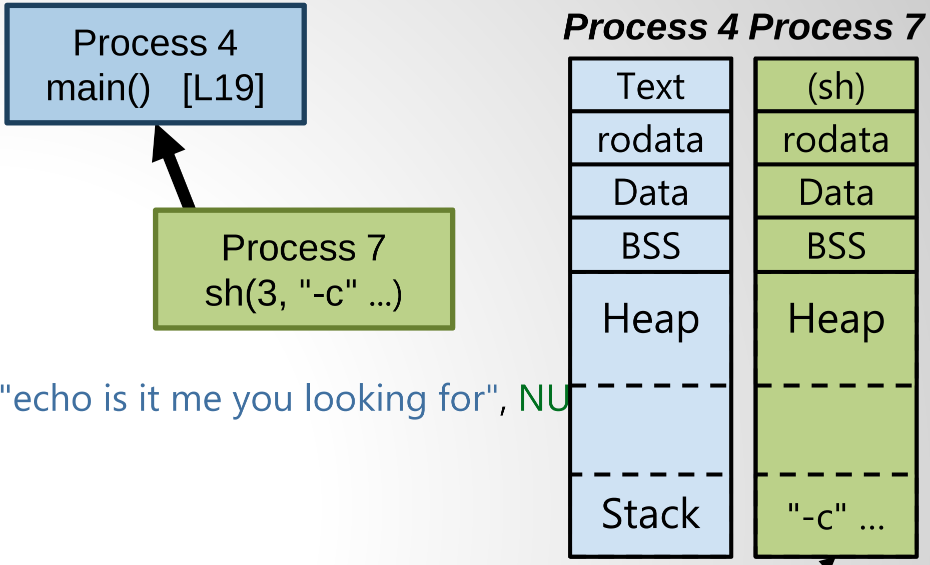




New address space containing « /bin/sh » code and the parameters

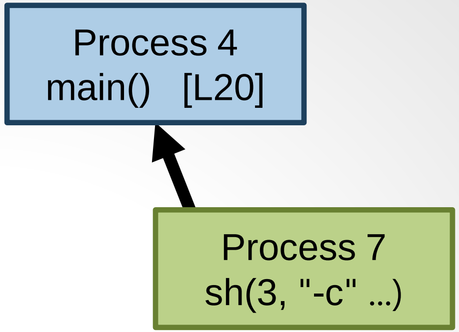
POV: code

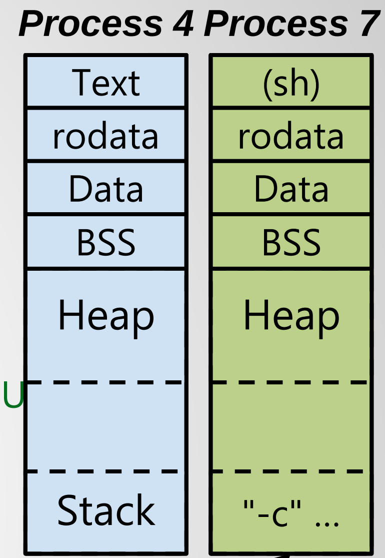
```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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



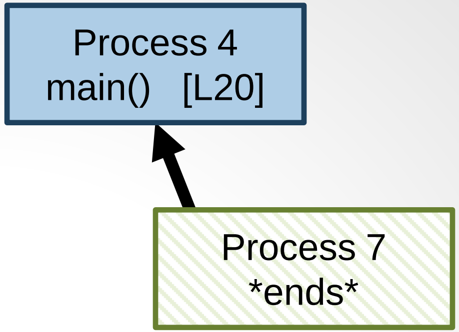



```
1.  #include <err.h>
2.  #include <stddef.h>
3.  #include <sys/types.h>
4.  #include <sys/wait.h>
5.  #include <unistd.h>
6.
7.  int main(int argc, char **argv, char **envp)
8.  {
9.          char *prog_argv[] = { "/bin/sh", "-c", "echo is it me you looking for", NU
10.         int status;
11.         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

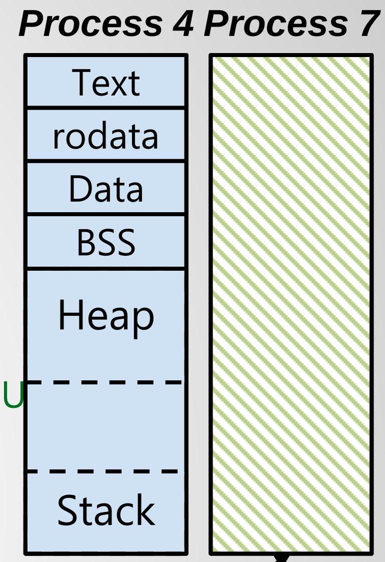
New address space containing « /bin/sh » code and the parameters

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

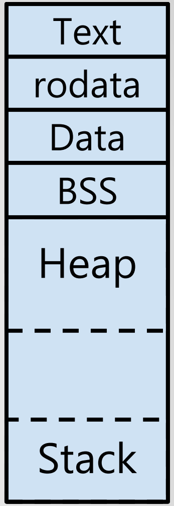
POV: code

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

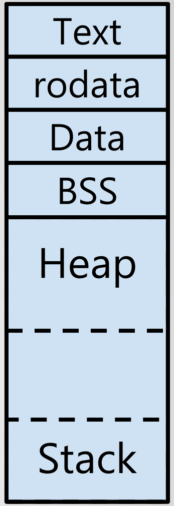


POV: code

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. }
```

Process 4



POV: code

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

its address space is freed,

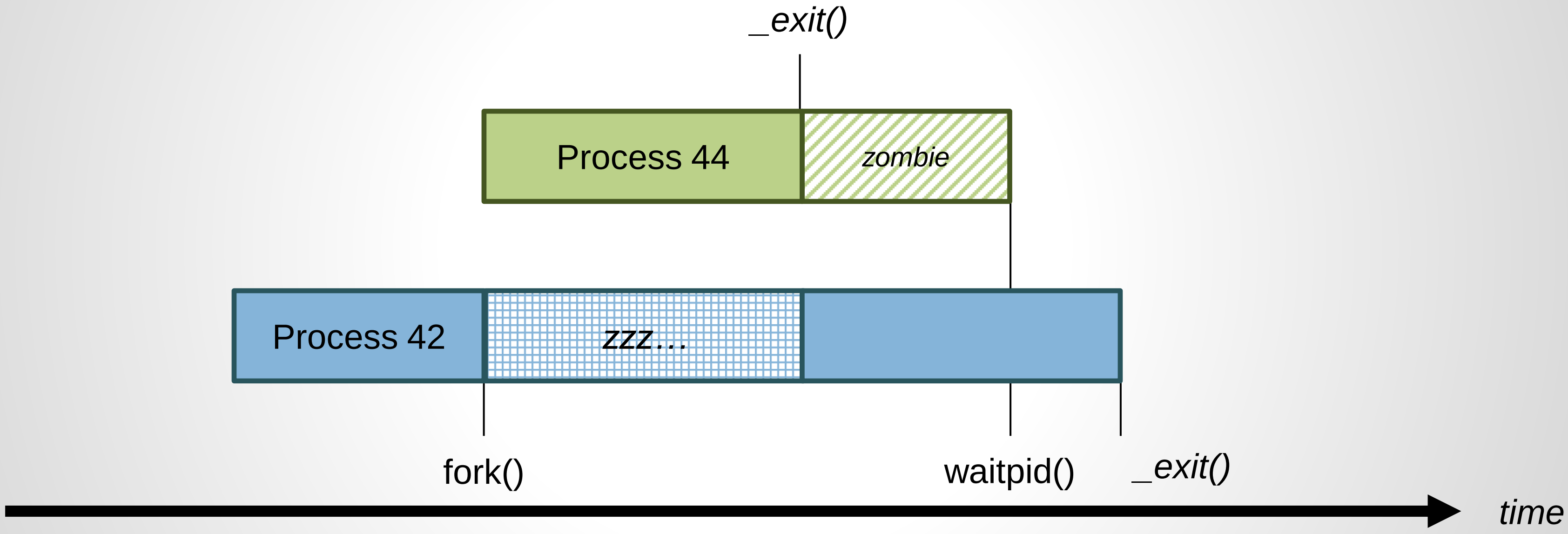
and its father will get the information

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’)

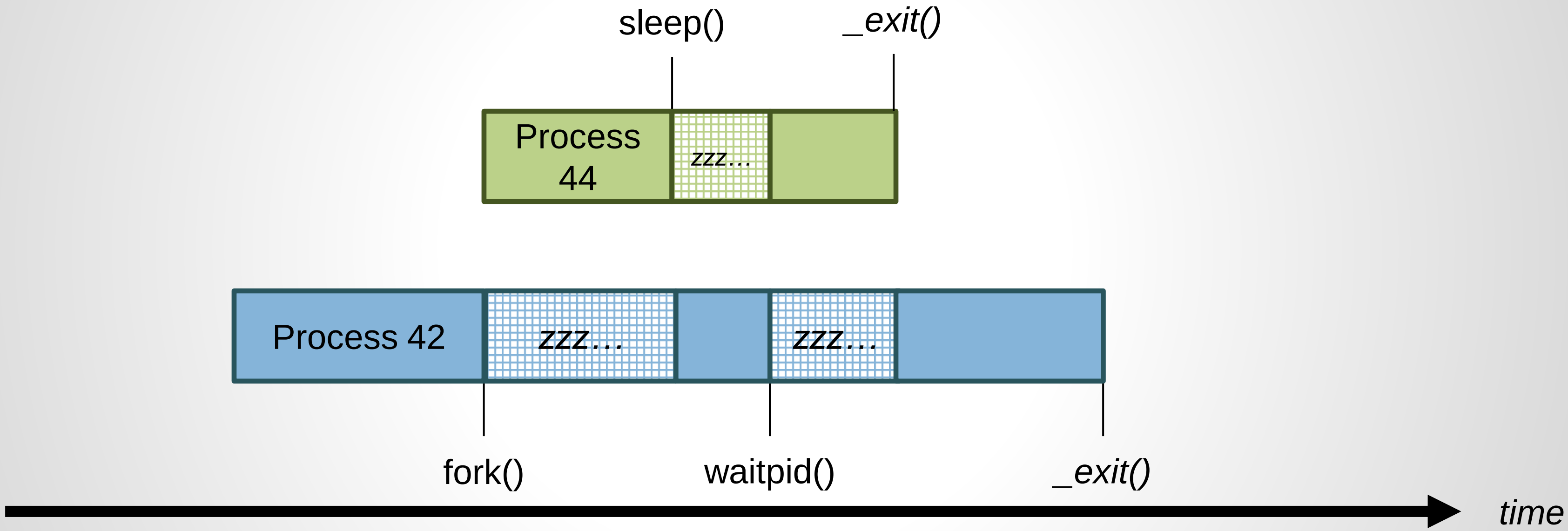
POV: kernel/scheduler

Case 1: child ends before the wait/waitpid



POV: kernel/scheduler

Case 2: child ends after the wait/waitpid



About the two last slides: SMP

- The previous slide was an example in the case of a single processor containing only one core

- In the case of SMP, processes could have run in parallel
  - But there would probably be a zombie

- *SMP: Symmetric multiprocessing*
  or Shared-memory multiprocessing
  - *Mutiple processes or threads can be run in parallel in the same OS*

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

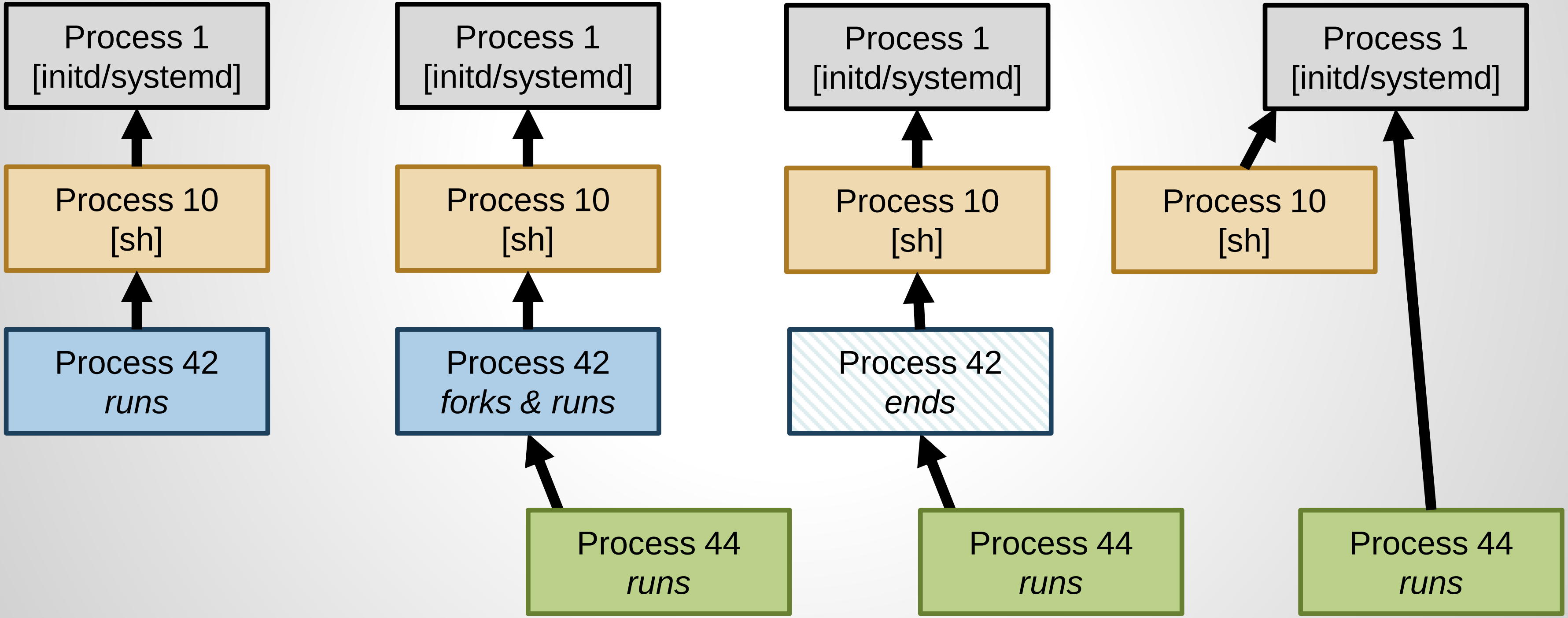
waitpid(2)

- 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

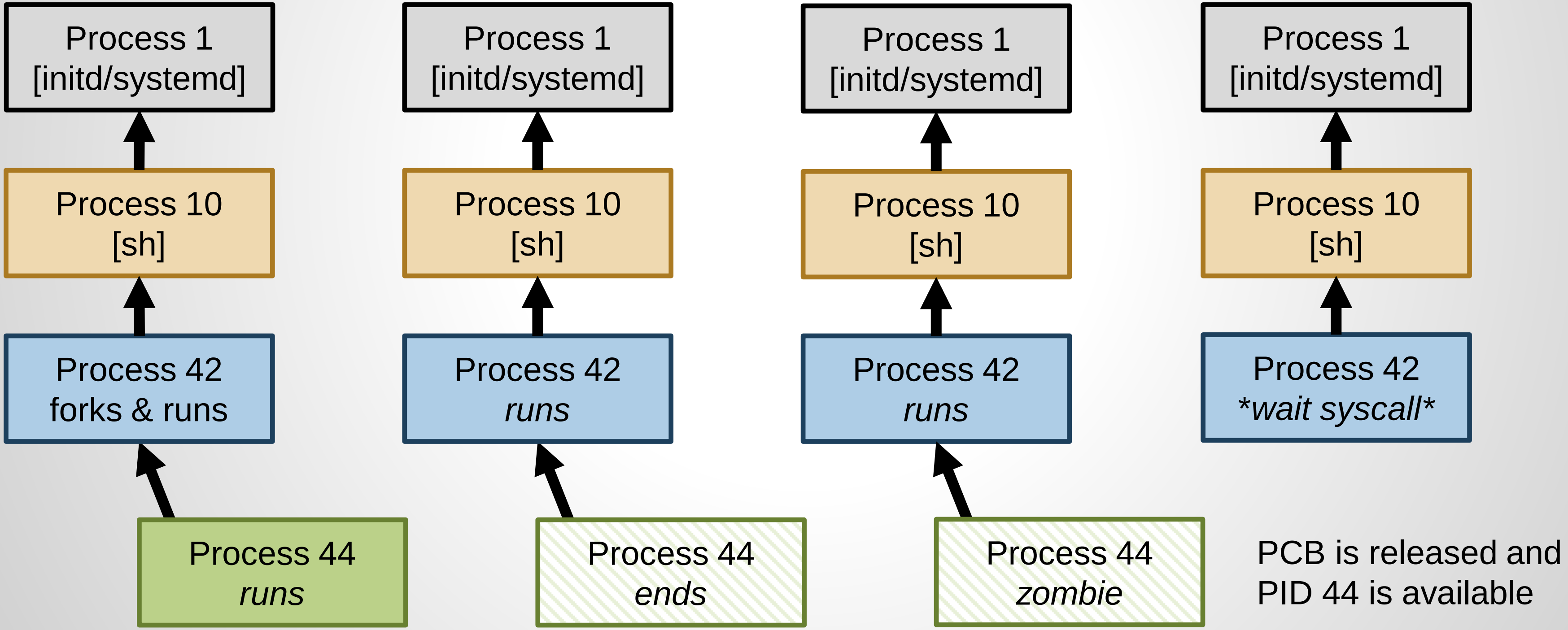
Process ending: father ends first

- 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



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, int sig); // (2) Send signal

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

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

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

sighandler_t   signal(int signum,
                      sighandler_t handler);

int   sigaction(int signum,
                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]

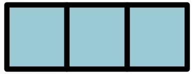
- Multiple programs are expected to run on the same system
- A blocking I/O allows another program to run

- 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

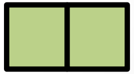
3 cycles to execute with one I/O
(ask the user for an input)



Task 1 is sent first

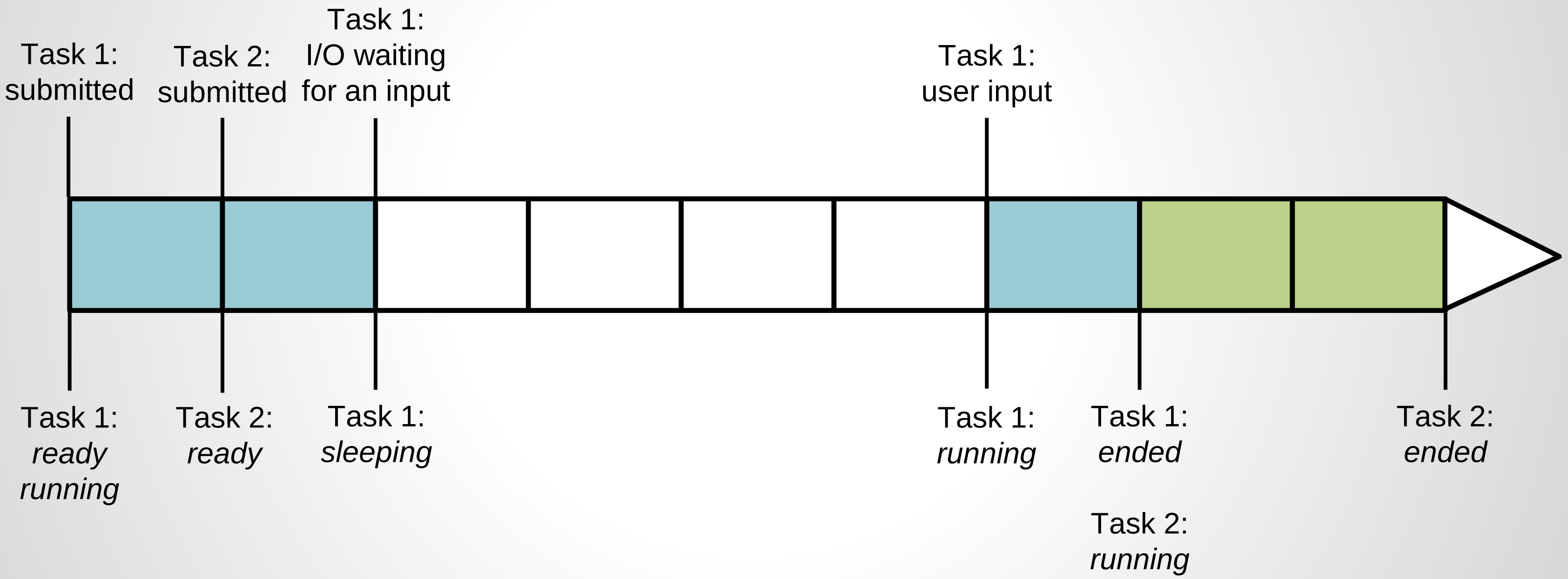
Task 2

2 cycles to execute without any I/O

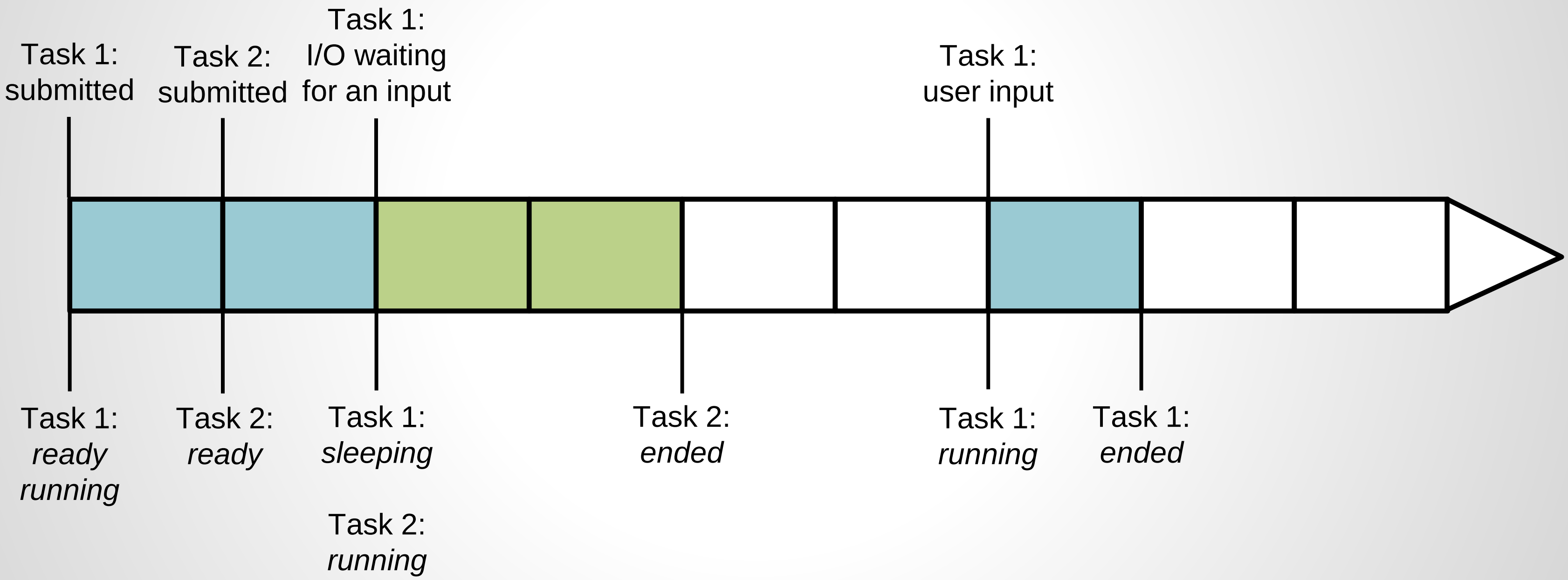


Task 2 is sent few time after Task 1

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

Types of tasks

- CPU bound
  - Computes large amount of data
- Interactive (I/O bound)
  - Response time: delay between submission and resolution of a request
  - Wait time: time passed in ready state
- Real Time (Time bound)
  - Respect of deadlines
  - Predictability

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

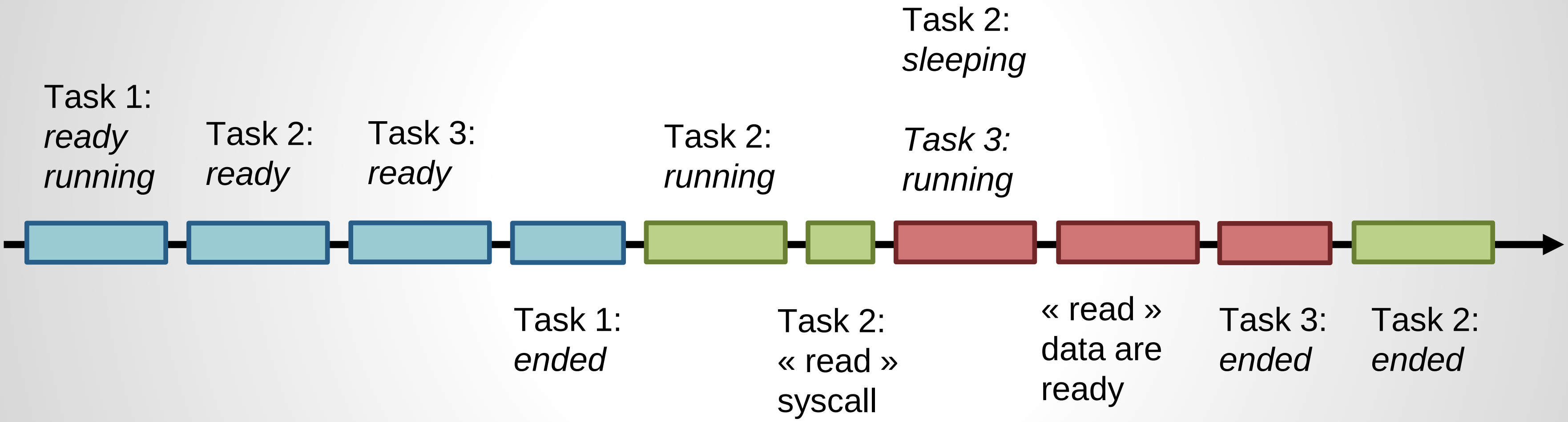
- The operating system is giving a *quantum of time* (or *time slice*) to each process
  - Processes are stopped by the operating system automatically…
  - …or during some specific instructions (blocking I/O, syscalls, …)

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

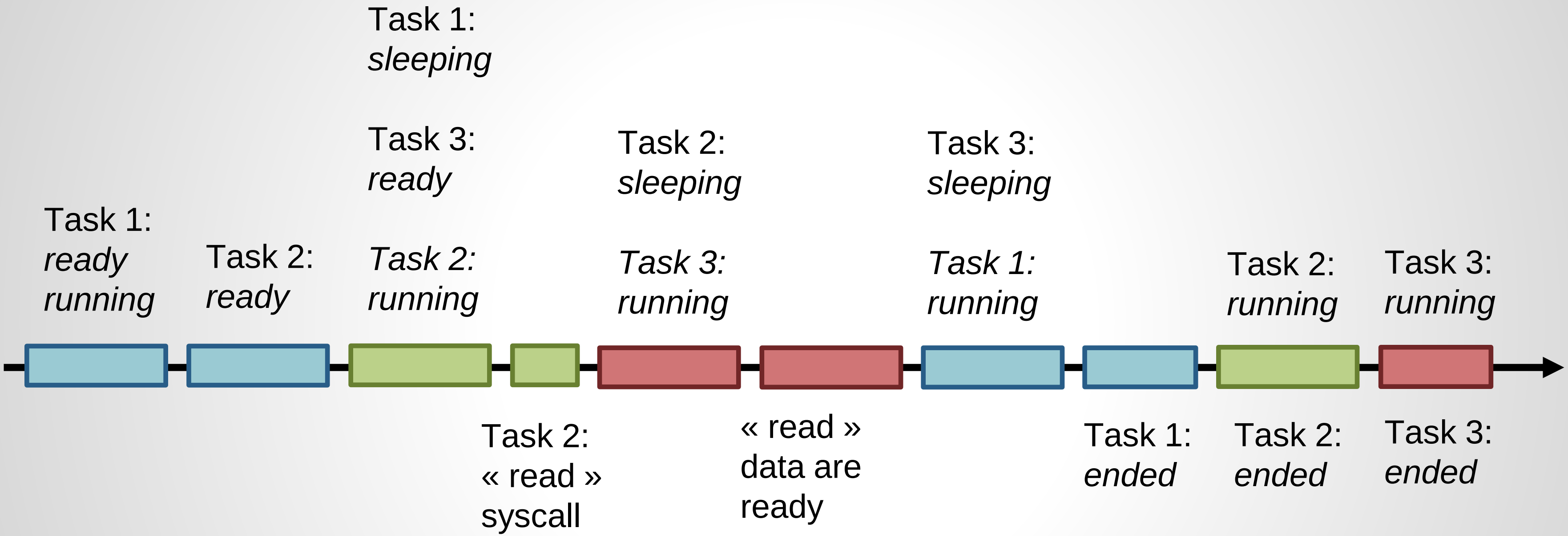
Example: Tasks to execute

Task 1	Task 2	Task 3
Long task without any I/O	Medium task with one I/O : A « read » syscall is made on a file from a disk device	Medium task without any I/O
Task 1 is sent first	Task 2 is sent few time after Task 1	Task 3 is sent few time after Task 2

Example: Cooperative Multitasking



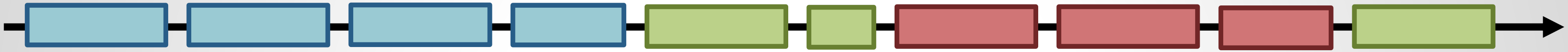
Example: Preemptive Multitasking



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

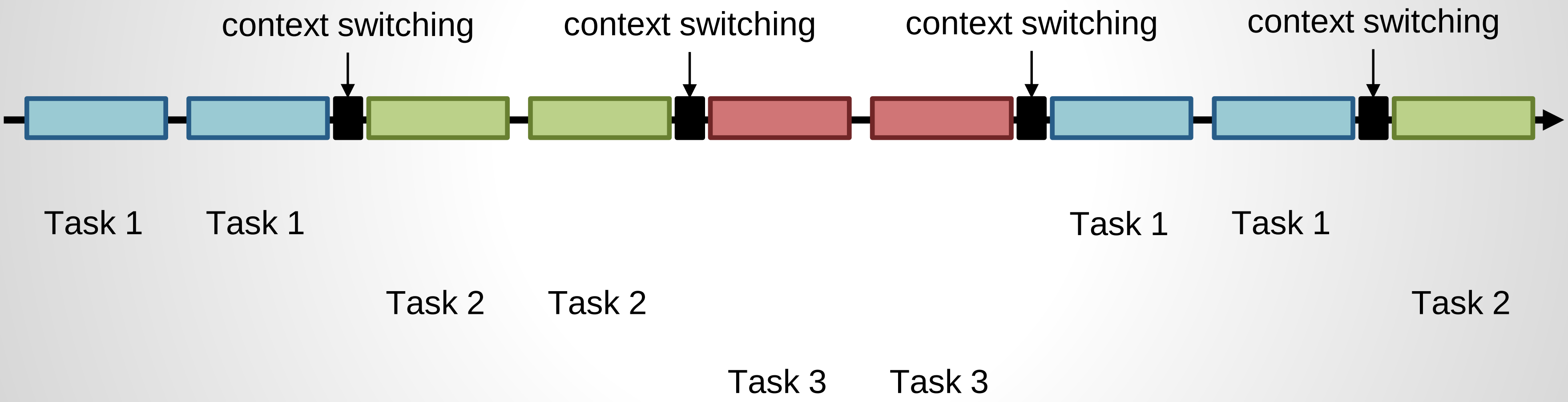
Time Sharing: how

- 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

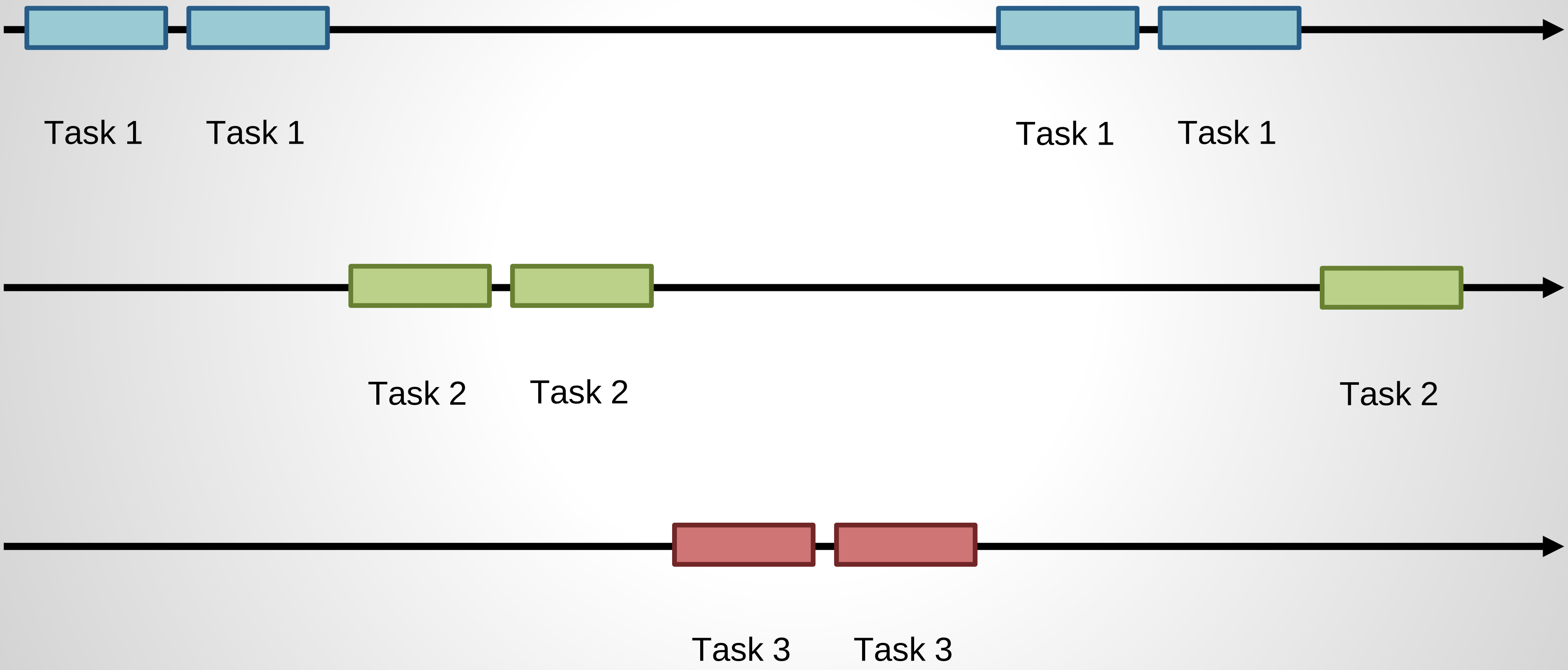
- When changing of process a *context switching* happens
  - It’s a pretty heavy operation: all of the registers, pages used, … (the context of the process) must be saved/restored

- 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



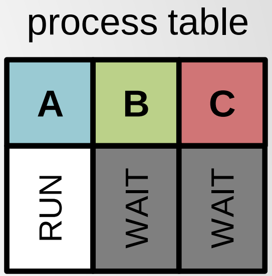
Time Sharing

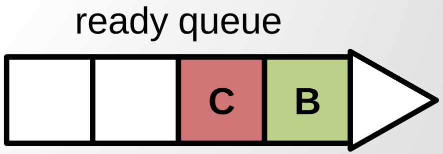
- Simplified example with 3 tasks using *Round Robin* algorithm and 2 quantums of time
- Each process is waiting for the 2 others to use their times
- What if there are more than 3 processes?...
  - Longer time to wait for each process to get the CPU

Scheduling

- Process table (list all processes)
    - https://en.wikipedia.org/wiki/Process_(computing)

- Queue with ready processes
- Queues with blocked processes





Types of scheduler

- FCFS
- Round Robin
- MPQ
- Lottery
- CFS
- RTS

FCFS First come First served

- Pros:
  - No preemption
  - FIFO for ready processes
  - Easy to learn and understand

- Cons:
  - Great variance in scheduling criteria
  - Accumulation effect

- Bad for Shared Time Systems
- OK/Good for Batch Systems
- SCHED_FIFO

Round Robin

- Same thing as FIFO, with a base time quantum
- Same Pros & Cons
- A little bit better for shared time systems

Multiple Priority Queue

- Split tasks into multiple priorities
- Different Scheduling policy for each priority
- Scheduling between the different priorities

Completely Fair scheduling

- Try to give the same amount of power for each processes
- Count with a fair clock the “waiting time”
- Higher priority = Time elapses faster
- Store processes by “waiting time” in a Red Black Tree
- Current Linux Scheduler

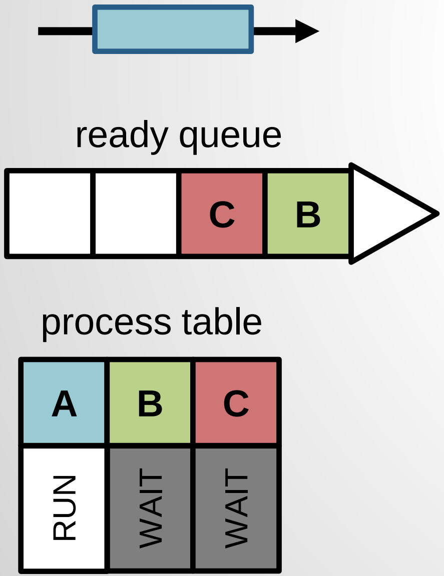
Simplified example of context switching

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

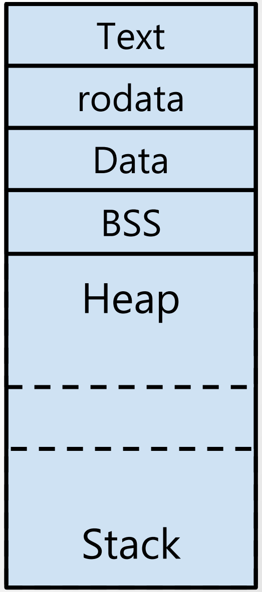
- At the end of the quantum:
  1. An interruption stops the running process A
  2. The process A is put in a « waiting » state in the OS
  3. The OS saves the context of the process A
  4. The OS puts the process A at the tail of the ready queue
  5. The OS takes the process at the head of the ready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

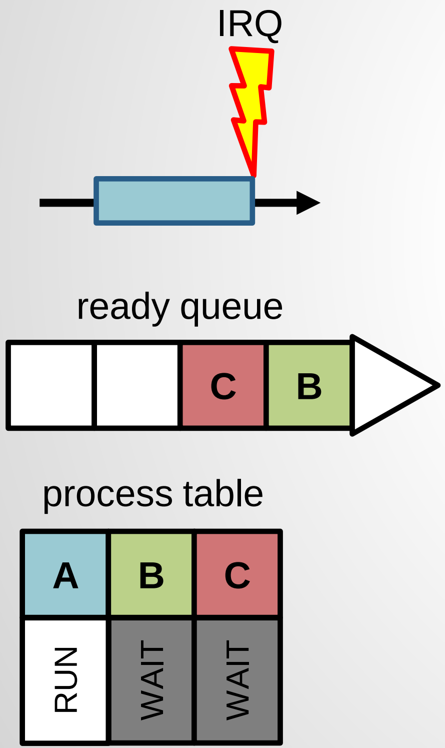



```
MOV   R2, R1
ADD   R2, 42
MUL   R2, 3
MOV   R1, R3
ADD   R1, R2
MUL   R1, 2
CMP   R1, R2
JGT   BIGGER
JMP   LOWER
```

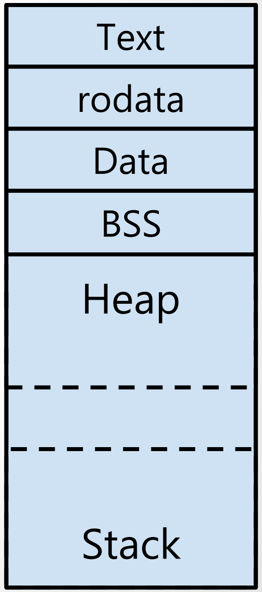


Example of scheduler: Round Robin

1. An interruption stops the running process A

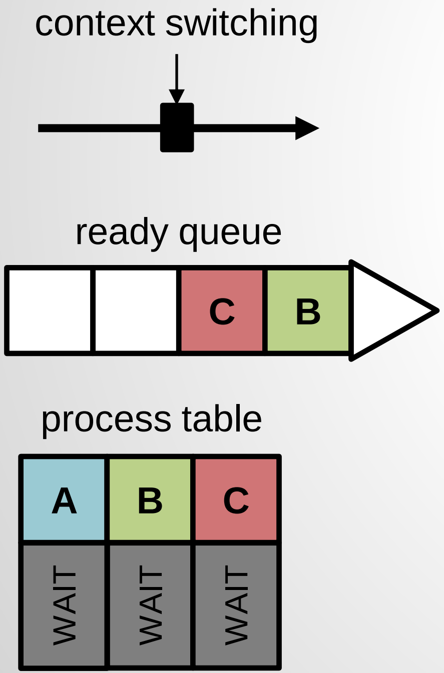


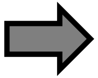

```
1. MOV  R2, R1
2. ADD  R2, 42
3. MUL  R2, 3
4. MOV  R1, R3
5. ADD  R1, R2
6. MUL  R1, 2
7. CMP  R1, R2
8. JGT  BIGGER
9. JMP  LOWER
```



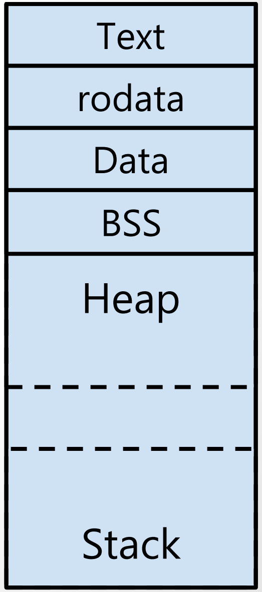
Example of scheduler: Round Robin

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



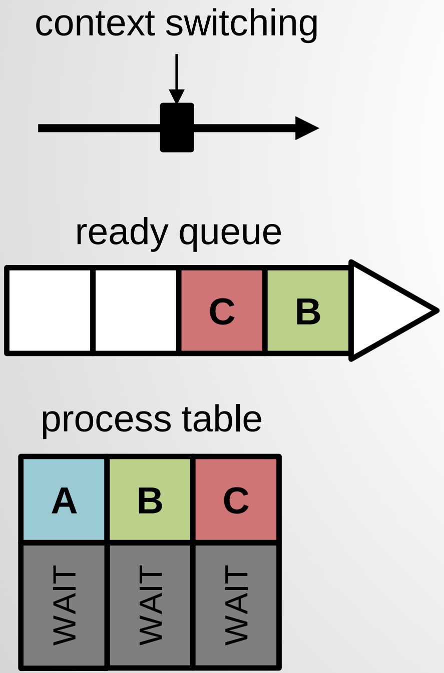


```
1. MOV  R2, R1
2. ADD  R2, 42
3. MUL  R2, 3
4. MOV  R1, R3
5. ADD  R1, R2
6. MUL  R1, 2
7. CMP  R1, R2
8. JGT  BIGGER
9. JMP  LOWER
```

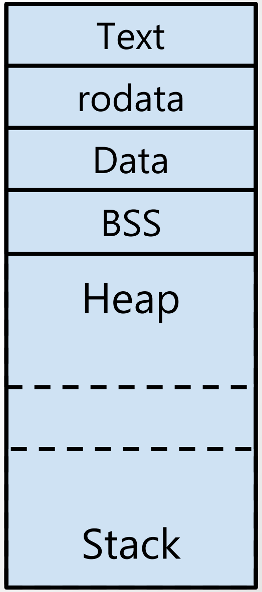


Example of scheduler: Round Robin

3. The OS saves the context of the process A

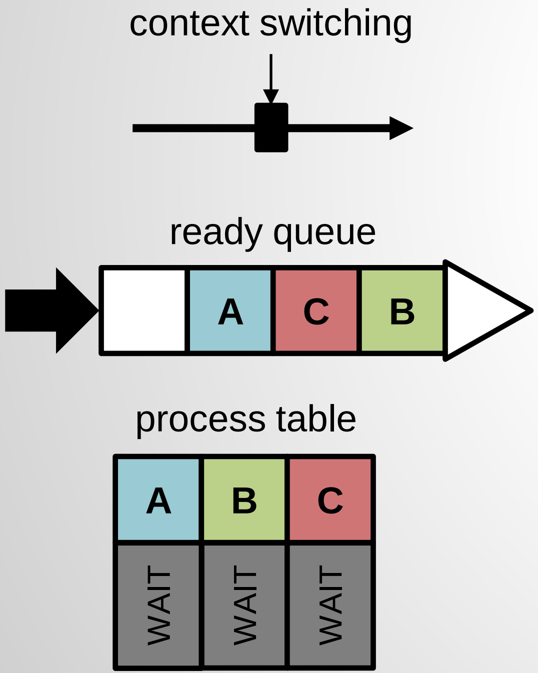


Context of Process A:
- stopped at instruction 5
- registers had values:
  R1 = XXX
  R2 = YYY
  R3 = ZZZ
  …
- …



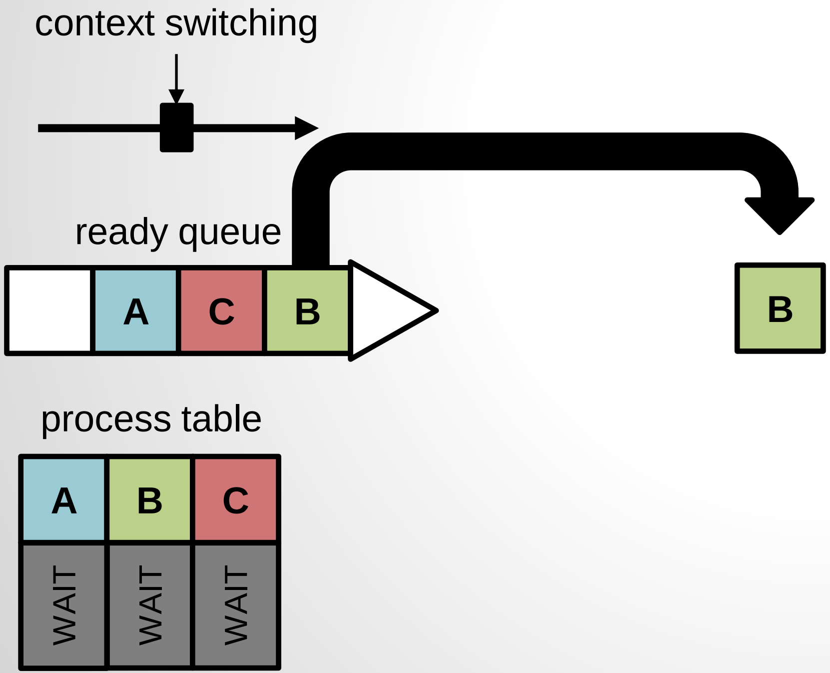
Example of scheduler: Round Robin

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



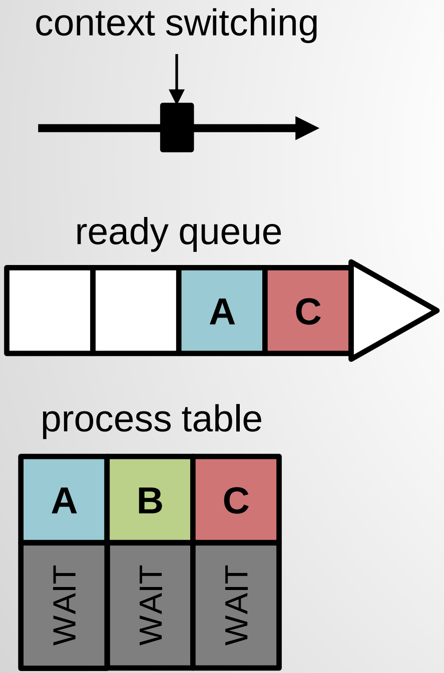
Example of scheduler: Round Robin

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



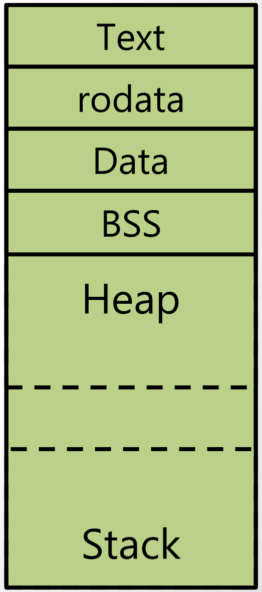
Example of scheduler: Round Robin

6. The OS loads the context of the process B



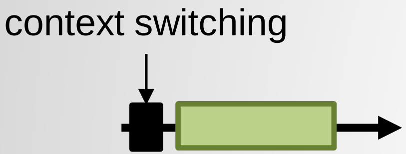
Context of Process B:
- stopped at instruction 2
- registers had values:

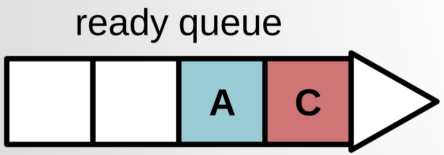
  R1 = 111
  R2 = 222
  R3 = 333
  …
- …

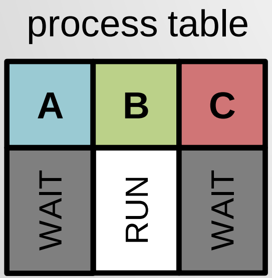


Example of scheduler: Round Robin

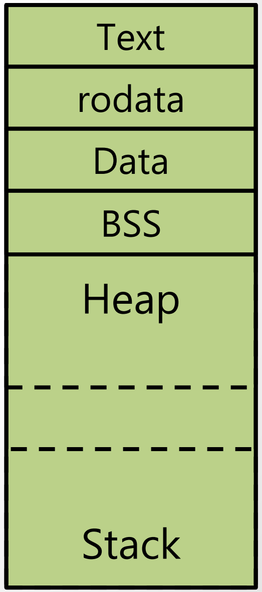
8. The process B is put in « running » state





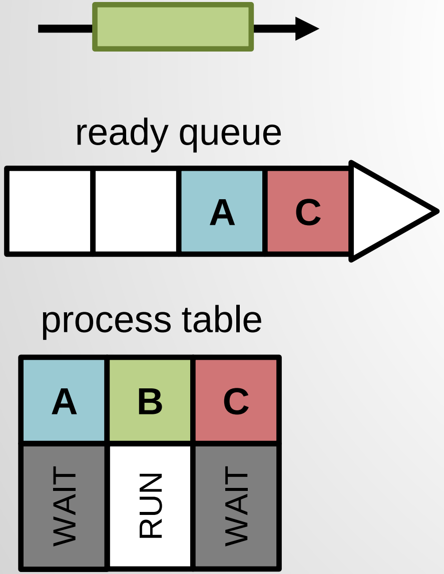



```
MUL   R4, 66
SUB   R1, 10
DIV   R1, 2
CMP   R1, R4
JNE   DIFFER
MUL   R1, 2
CMP   R1, R4
JNE   CATCH
CLR   R1
```

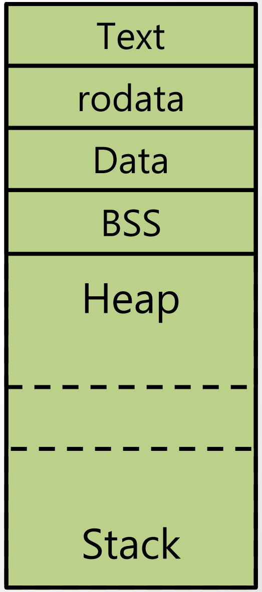


Example of scheduler: Round Robin

- The process B is running




```
1. MUL  R4, 66
2. SUB  R1, 10
3. DIV  R1, 2
4. CMP  R1, R4
5. JNE  DIFFER
6. MUL  R1, 2
7. CMP  R1, R4
8. JNE  CATCH
9. CLR  R1
```



Different kind of schedulers

- long term: plan for tasks in the future
- short term: plan for next task based on dynamic informations
- middle term: based on current load, plan for actions (swapping for example)

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)

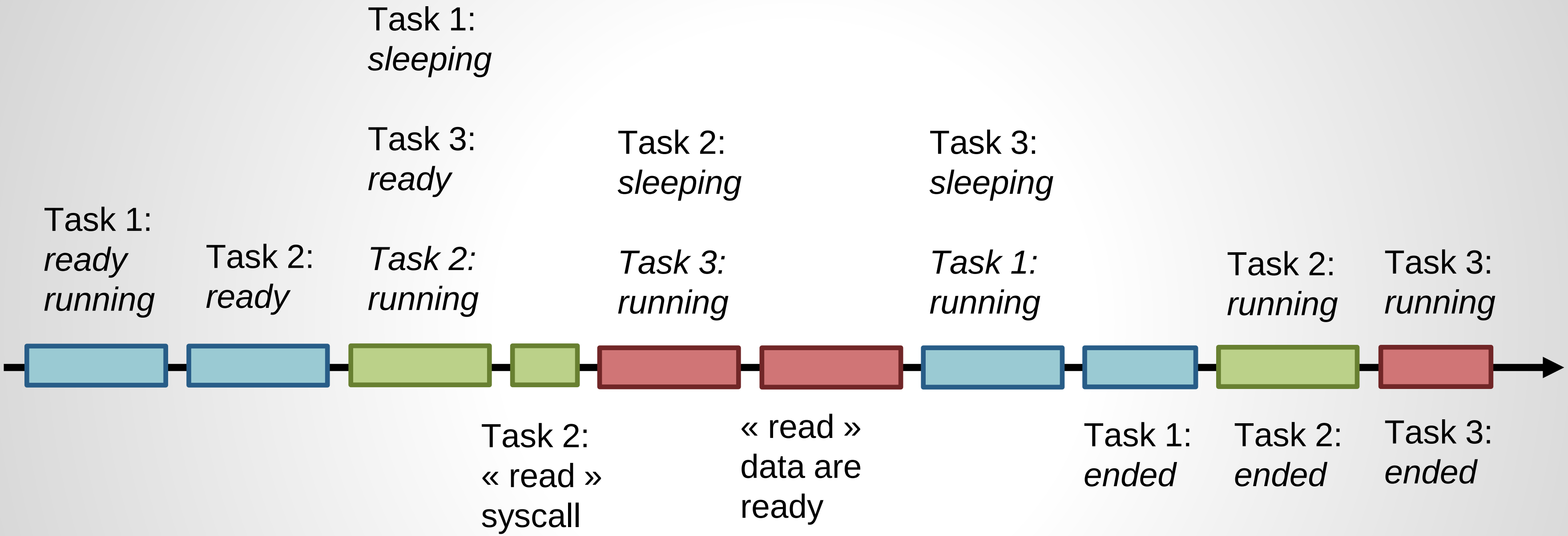
- -> Each task is « sure » to be executed

Example 1: Round Robin scheduler

Quantums of time per process: 2

Task 1	Task 2	Task 3
Long task without any I/O	Medium task with one I/O : A « read » syscall is made on a file from a disk device	Medium task without any I/O
Task 1 is sent first	Task 2 is sent few time after Task 1	Task 3 is sent few time after Task 2

Example 1: Round Robin scheduler



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

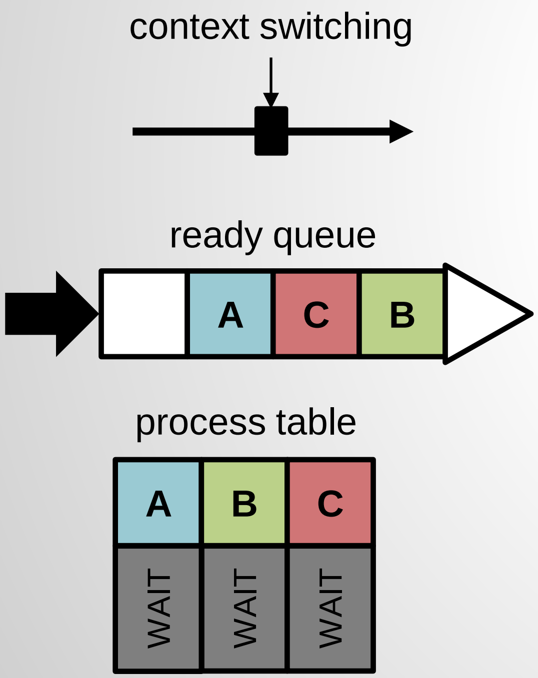
- Priorities are added!
  - Priority on GUI tasks (graphical client)
  - Priority on background tasks (servers)
  - Manual priorities
  - ...

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)

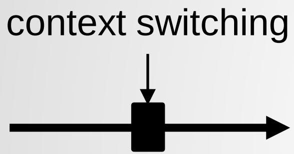
Example of scheduler: Round Robin + priority

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

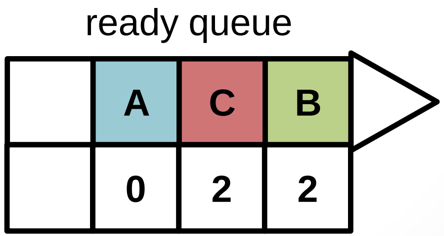


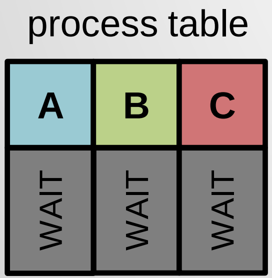
Example of scheduler: Round Robin + priority

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



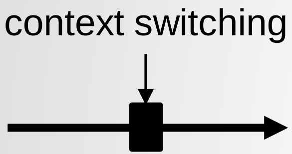
and updates the priorities



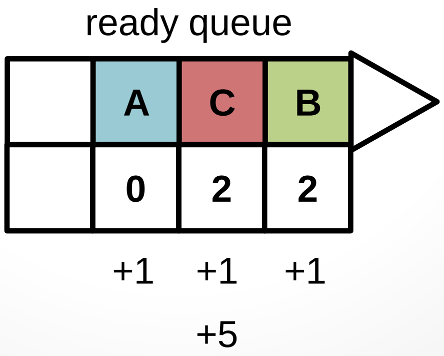


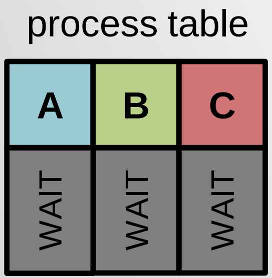
Example of scheduler: Round Robin + priority

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



and updates the priorities

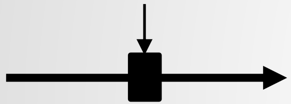




Example of scheduler: Round Robin + priority

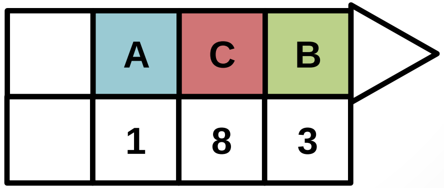
- The OS puts the process A at the tail of the ready queue

context switching

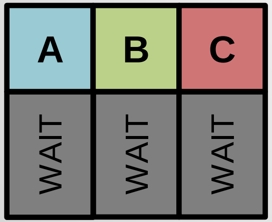


and updates the priorities

ready queue

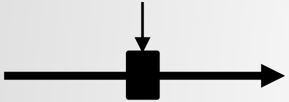


process table

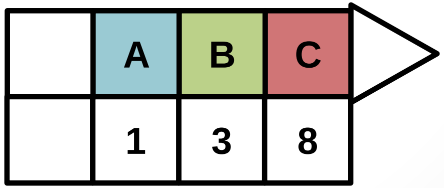


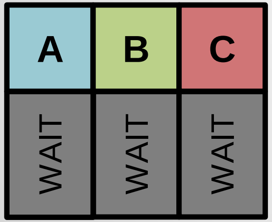
Example of scheduler: Round Robin + priority

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



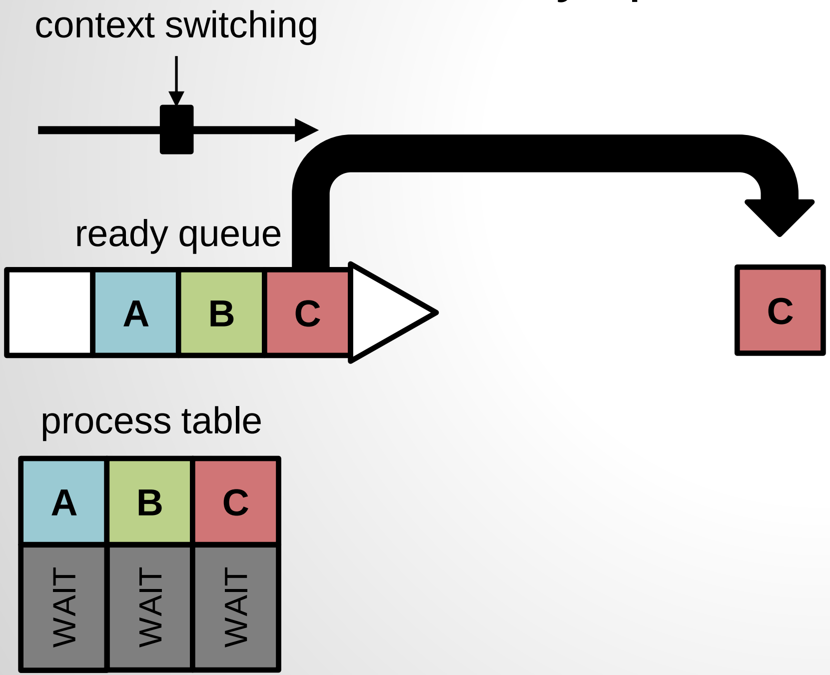
and updates the priorities





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...

sched(7)

- sched_setscheduler(2)
- sched_getscheduler(2)
- sched_yield(2)
- SCHED_FIFO: First in-first out scheduling
- SCHED_RR: Round-robin scheduling
- SCHED_DEADLINE: Sporadic task model deadline scheduling
- SCHED_OTHER: Default Linux time-sharing scheduling
- SCHED_BATCH: Scheduling batch processes
- SCHED_IDLE: Scheduling very low priority jobs

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?

Real-time systems: the exception

- Embedded systems usually have constraints and requirements with the time
  - Some do not use schedulers from regular computing

- « Hard » real time
  - A task « must » end before a deadline
  - A task « must » end within less than X quantums
  - …

- « Soft » real time
  - A task « should » end before a deadline…
  - …but if there is nothing else to run, it’s okay to be late

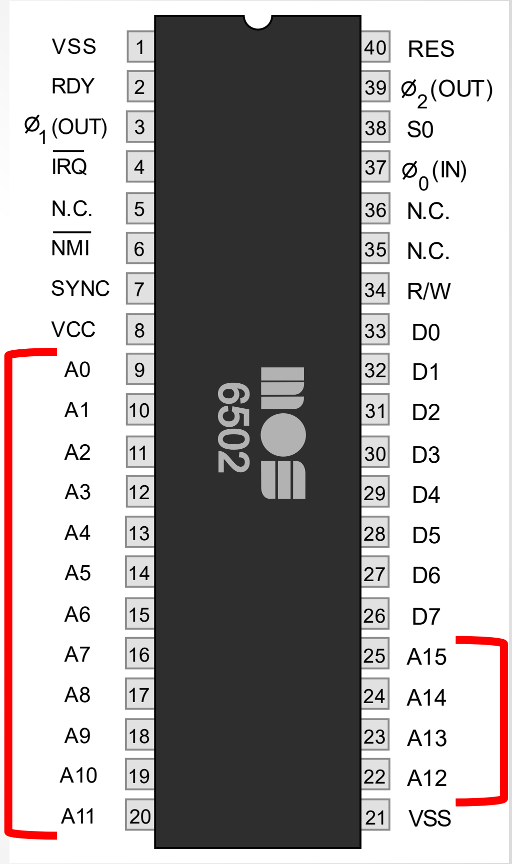
Real-time systems: the exception

- Schedulers in these cases require a fine tuning
  - And are in fact completly different than the « time sharing » ones…
- Applications are written with specific languages, libraries, and OS in order to comply with the constraints
  - ADA
- As constraints and requirements are strong… the « system » (in the abstract sense) must be deterministic

Memory Management

Memory Protection

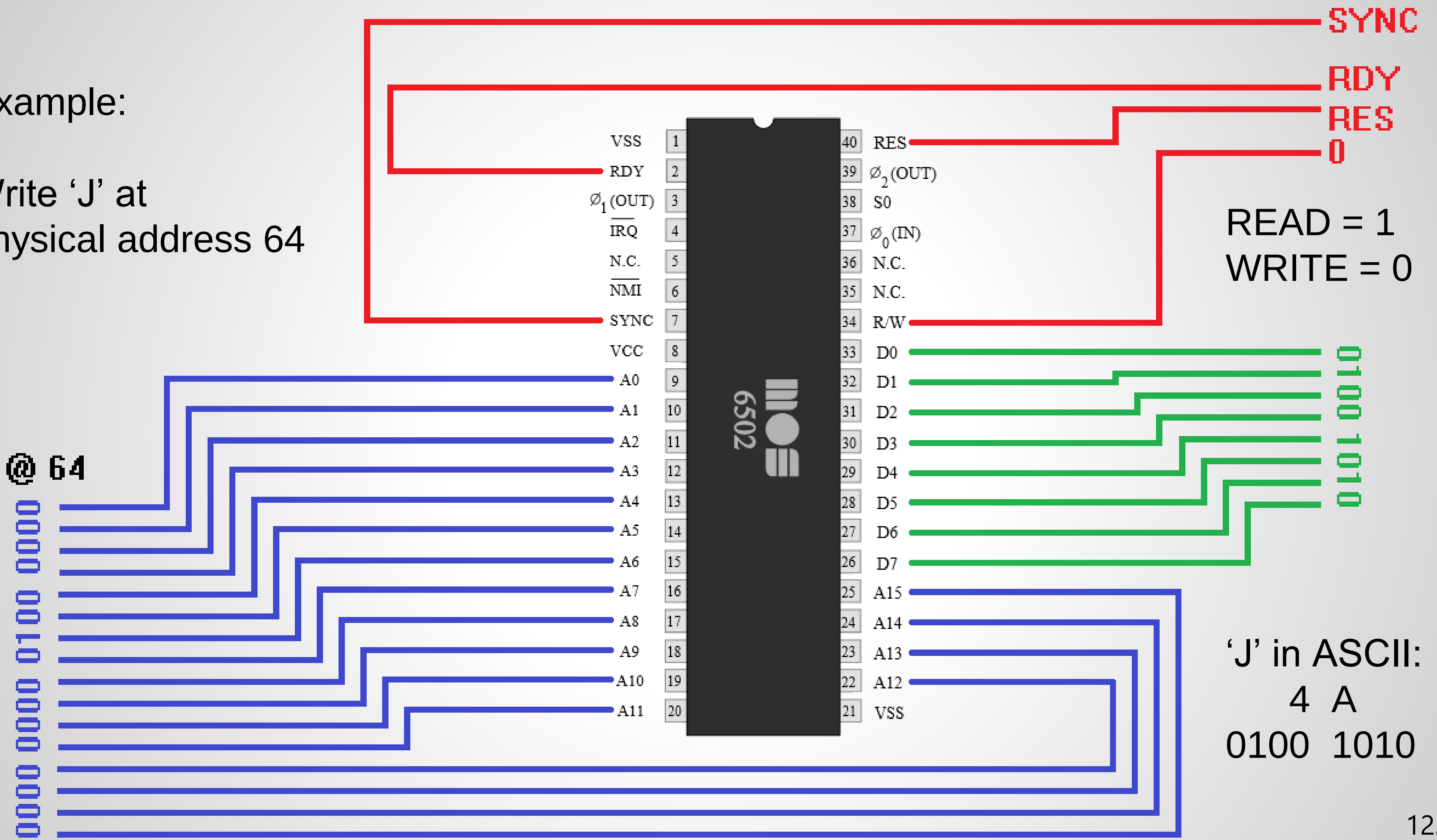
- Physical Addresses
- Chips are accessed by wires at physical addresses
    - *Through the address bus*



(6502 reproduit par Bill Bertram)

Example:

Write ‘J’ at
physical address 64



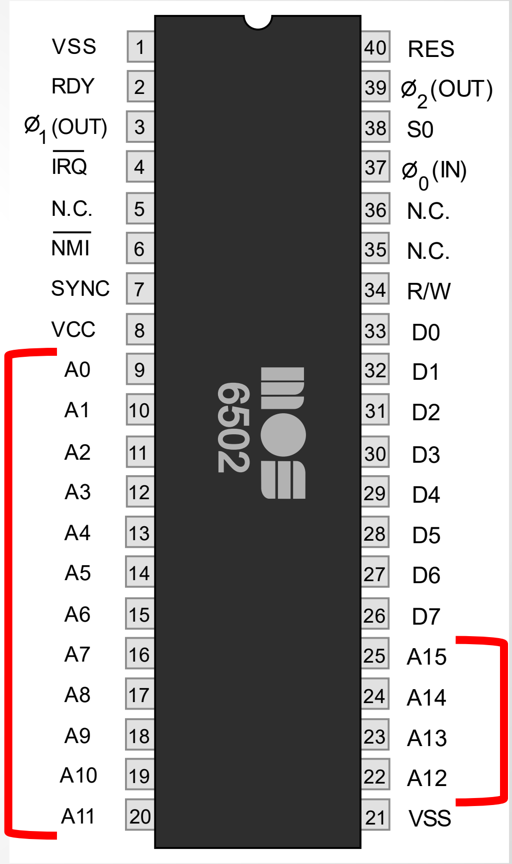
READ = 1
WRITE = 0

‘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

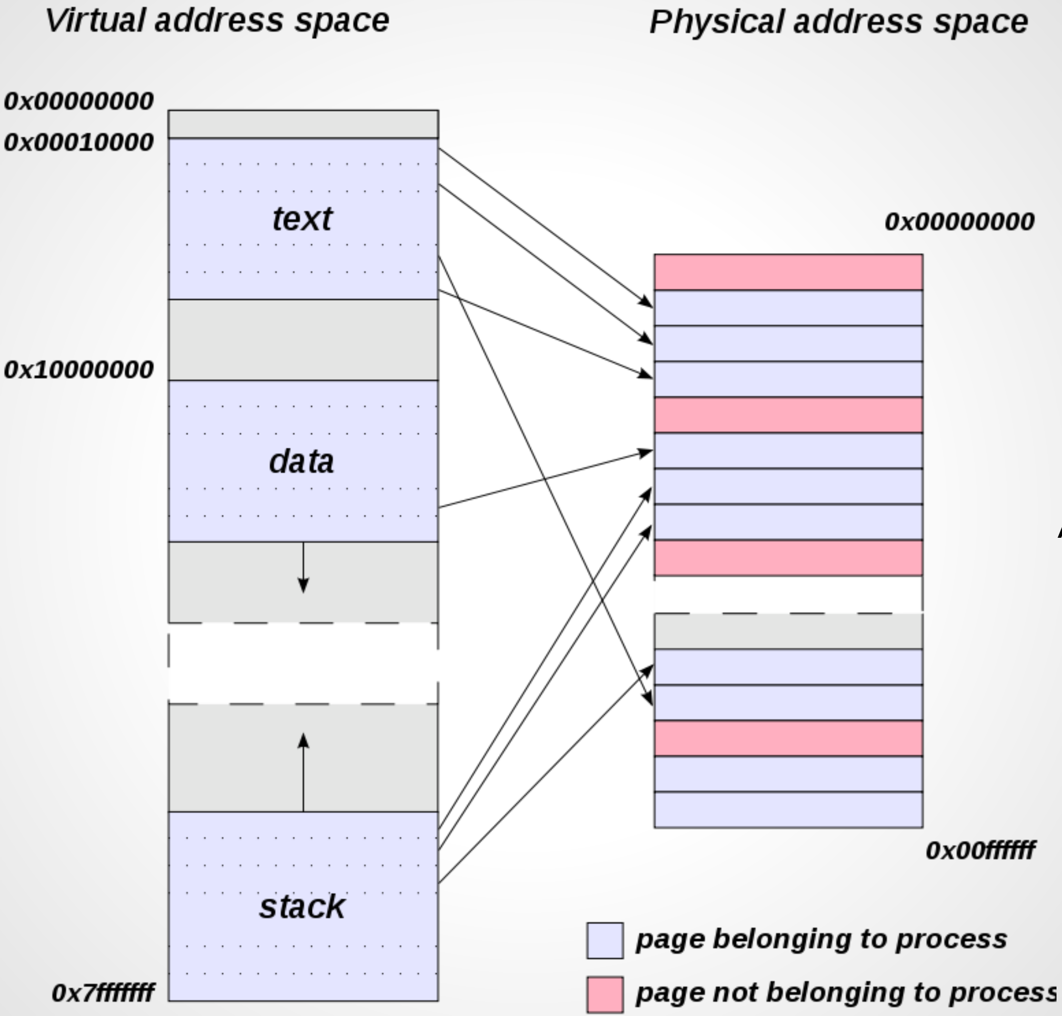
- Requires a mechanism for protection
  - Keep a list of « areas » per processes
  - Keep the rights per processes

  - **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

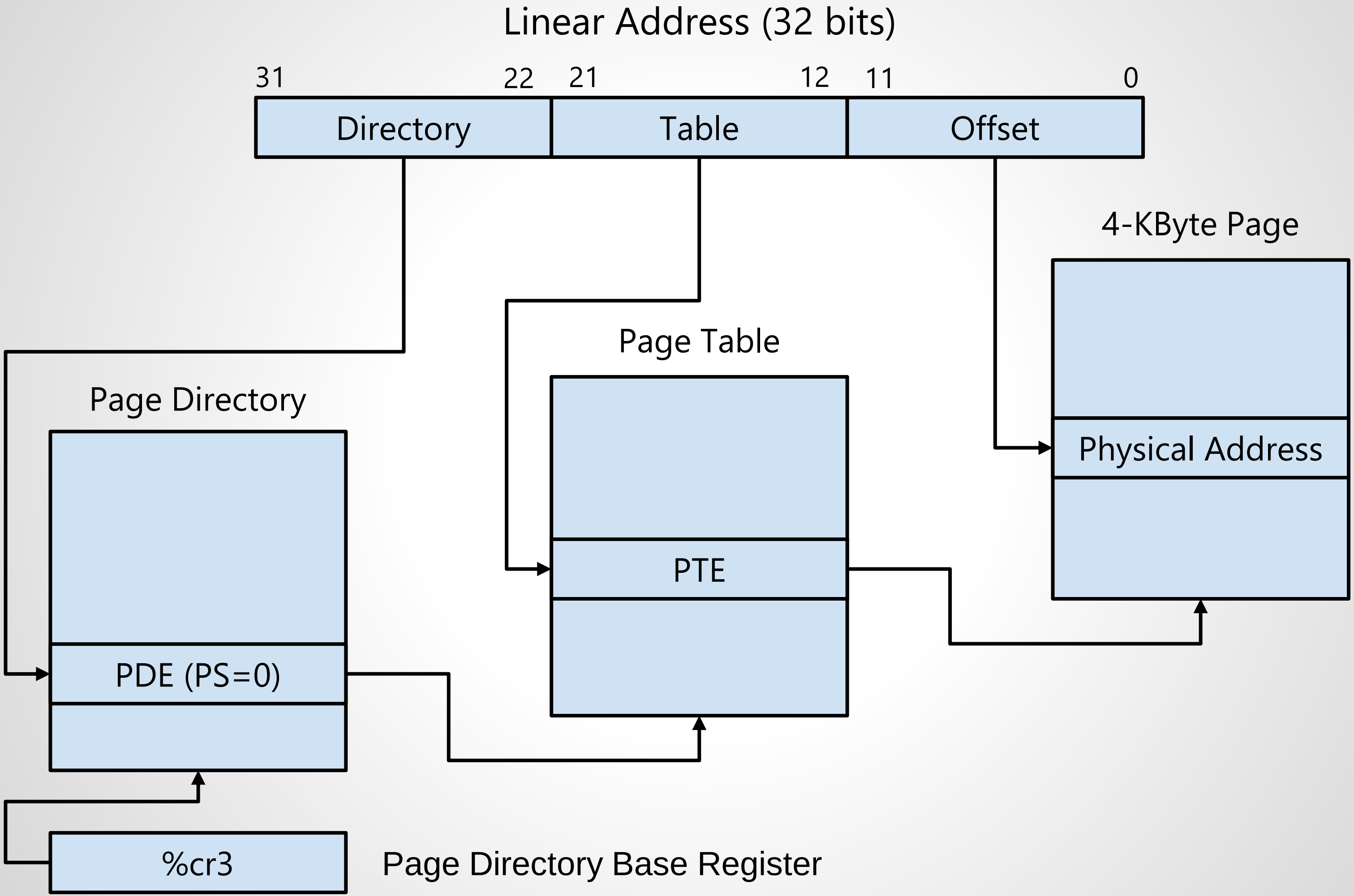
Address in the virtual address space

Userland programs use pointers in the virtual address space only

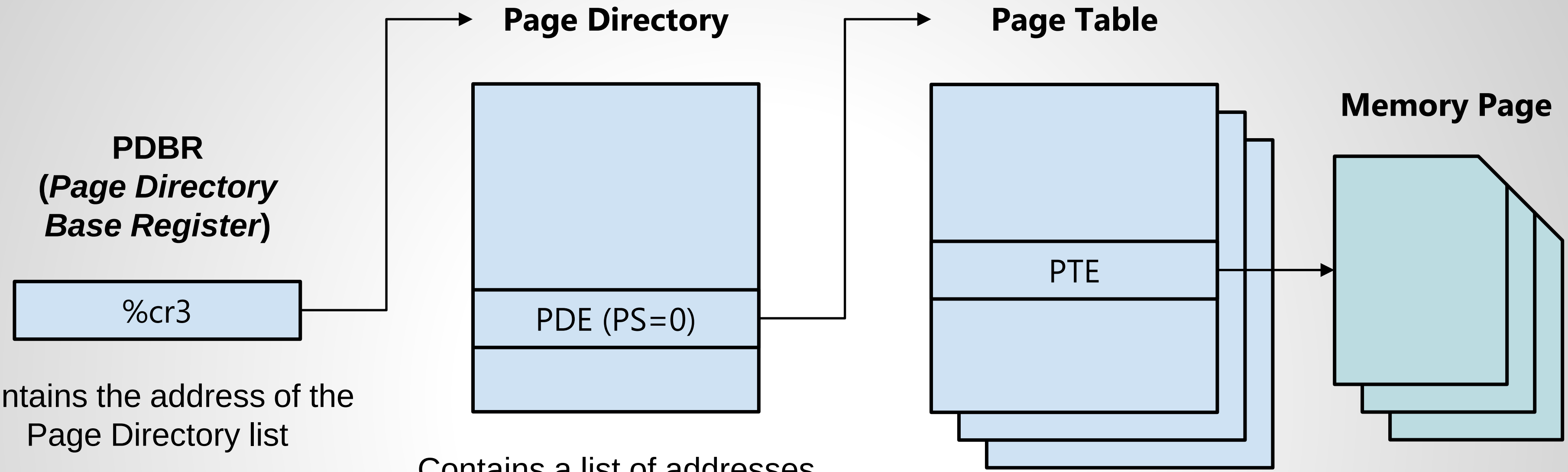
Physical address

Address in the physical address space

Kernel works with both virtual and physical addresses
(it can indirectly manipulates the physical addresses)


Linear Address (32 bits)
31
22
21
12
11
0
Directory
Table
Offset
4-KByte Page
Page Table
Page Directory
Physical Address
PTE
PDE (PS=0)
%cr3
Page Directory Base Register

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)

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

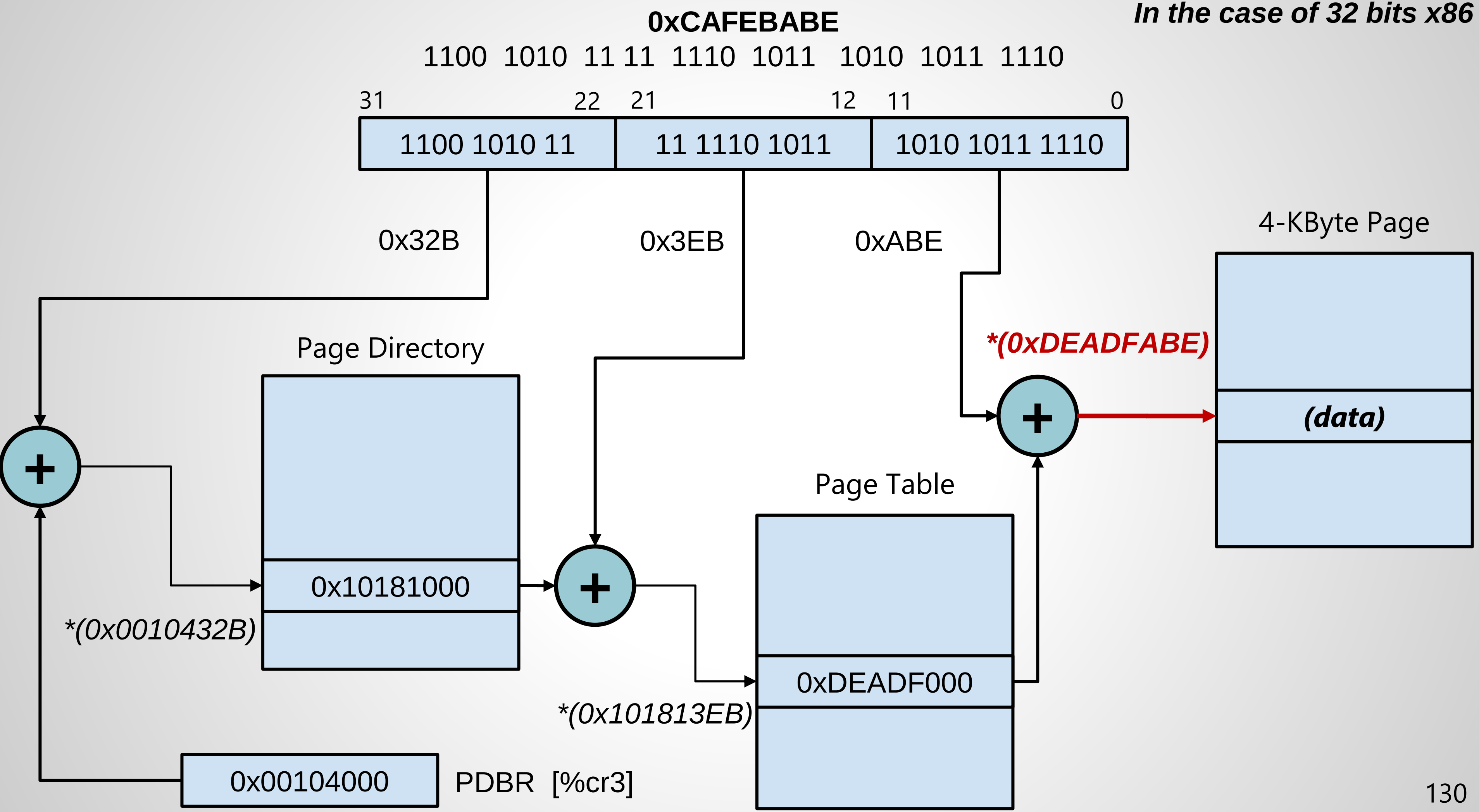
cr3 - 0x00104000

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



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), ...

Memory Virtualization

Swap:
When one (or more) program(s) consume(s) more memory pages than what the physical address space can handle…

- *Paging*: each unused page is put back on disk, and new pages are created (or targeted pages are reloaded from the disk)

- *Swapping*: the whole address space of a sleeping process is put on disk (and when he'll be awakened, his address space will be restored in memory)

  - Main memory, Primary storage, … : RAM
  - Auxiliary memory, Secondary storage, … : hard drive, flash memory, …

Memory

Therefore, what does malloc do ?

Memory usage

- malloc(3) is in userland (obvious)
  - Works in the « *heap* »
  - Uses mmap(2)
  - or brk(2)/sbrk(2) *(not anymore/only in specific environments)*

- Malloc manages how the pages are given to the user
  - Strategies for each usage :

  Few giant allocations, lot of small allocations, mixed usage...

- Reuse pages
  - Reduces the number of syscalls (context switching, ...)
  - Increases useful time for the user code

Memory usage

- Reminder:
  Each process believes he’s alone in memory with his libraries

- See the memory as a flat array from 0 to 4 GB/16 EB

Memory allocation

```
void*  mmap(void  *addr,
            size_t  len,
            int  prot,
            int  flags,
            int  fd,
            off_t  offset);
```

```
int  munmap(void  *addr,
            size_t  len);
```

Memory allocation

mmap(2)

void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

- Creates a new mapping in memory
  - Reserve one or more pages in memory, and return the 1st address
  - Kernel adds new entries in the page tables and directories
- If *addr* is NULL, the kernel chooses the location
  - Else, it searches for some pages around *addr*

Memory allocation

mmap(2)

void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

- Similar to files: some rights allow Read, Write, Execute
  - Objectives in the OS are to avoid Write and Execute in the same memory area...
  - ...avoids a program that can write code, and then, execute it

Memory allocation

mmap(2)

void* mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

- *fd* and *offset*: a file might be directly mapped into memory

Malloc implementations

- Storage techniques: *Linked list, Bitmaps, …*
- Policies: *First Fit, Best Fit, …*
- Methods: *Free list, Bucket, Buddy, Slab, Stack allocation*

- Mixed methods if various sizes must be malloc'ed

https://en.wikipedia.org/wiki/Memory_management

https://www.memorymanagement.org/index.html

http://brokenthorn.com/Resources/OSDev26.html

https://www.kernel.org/doc/gorman/html/understand/understand011.html

Malloc implementations: main concepts

- Memory chunks
  - The addresses returned by malloc to the user
- Meta-data about the memory chunks
  - Size, availability, address for the user
- Meta-data around the chunk, or elsewhere
- Mapped memory is rarely unmapped
  - You just keep the pages available for a future call to malloc
  - When your process ends, the address is fully released…
  - …don’t forget to help your malloc by not losing pointers!

Malloc storage techniques: Linked list

- Keep a list of free blocks *(Free list method)*
  - Those that was previously allocated

- Each block has a header
  - Size of the block (header + memory chunk)
  - Pointer to next free block

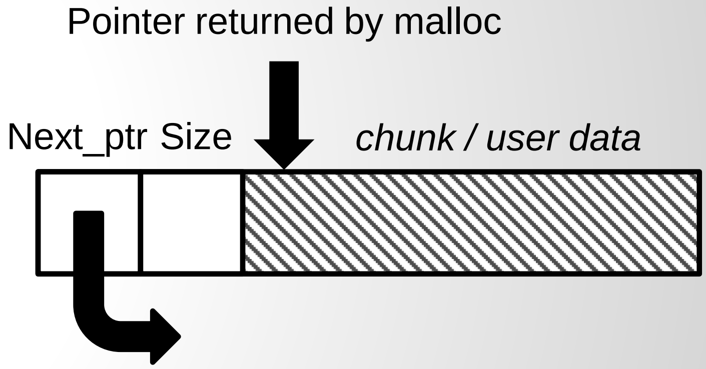
- Round the block (mandatory for memory)
  - Round around 64 bits / 8 bytes (easier for the CPU)

Don't forget : if you manage memory, use char in C*

Malloc storage techniques: Linked list

- *Next Pointer*: is a pointer
  64bits (8 Bytes)

- *Size*: is an int
  64bits (8 Bytes)

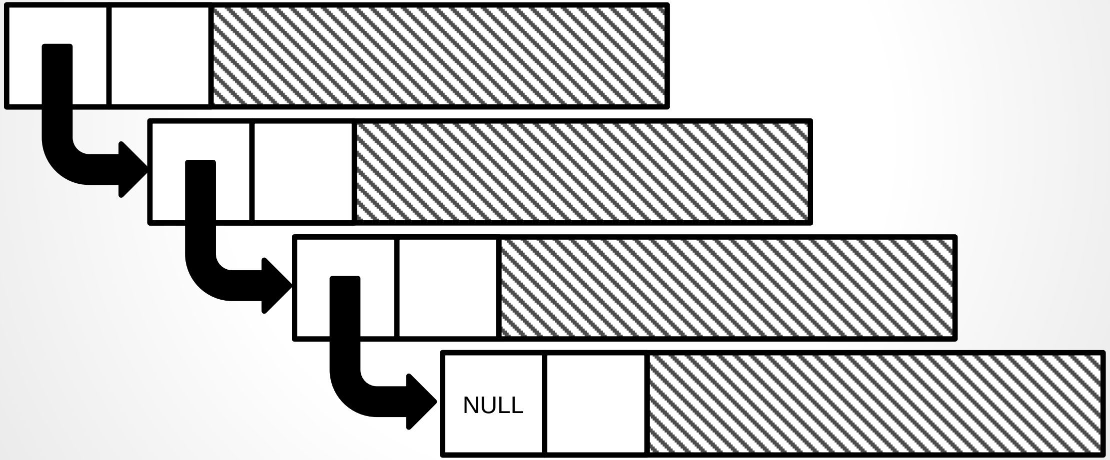


Don't forget:

- *Ease the work of your CPU, stay aligned…*
- *Sometimes it's forbidden not to be aligned*

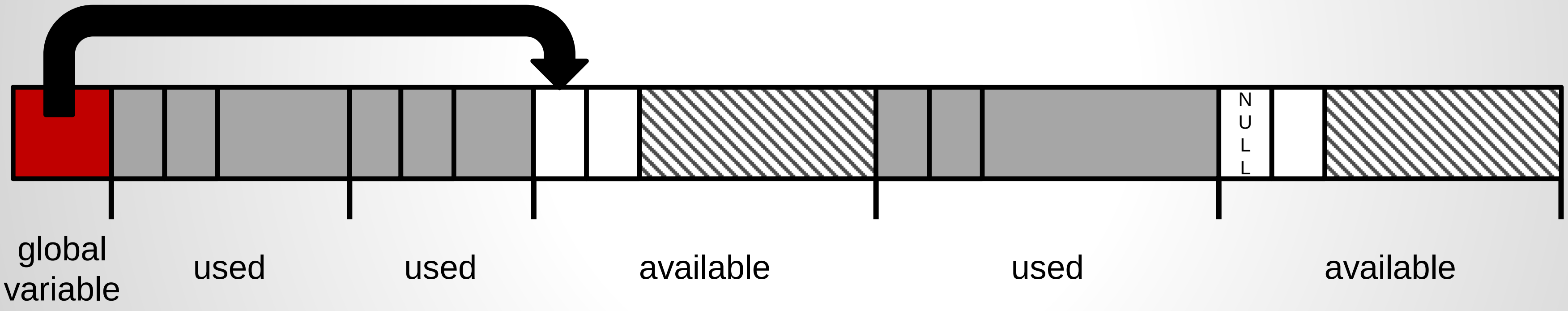
Malloc storage techniques: Linked list

Logical point of view



Malloc storage techniques: Linked list

Memory point of view

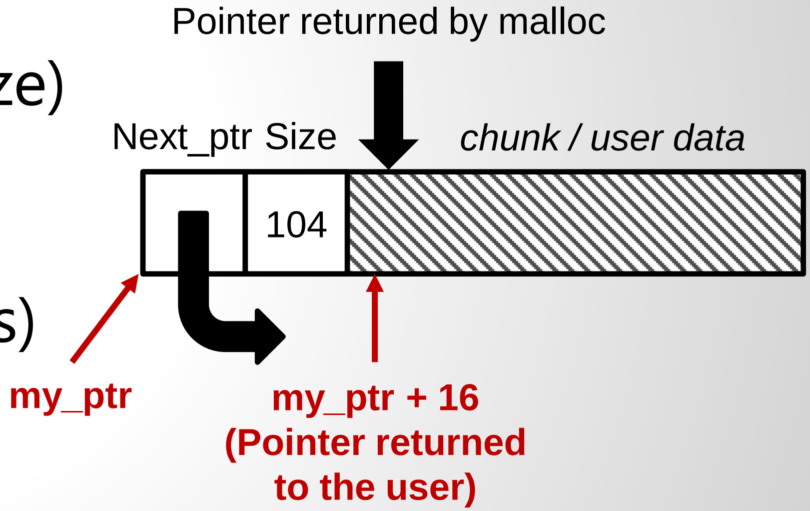


- Global variable used to point to the first free block
- Global variable is initiated on the first call to malloc

Malloc storage techniques: Linked list

Step 1: usr_ptr = malloc(84);

1. Add the header (Next_ptr + Size)
   84 + 8 + 8 = 100 (Bytes)



2. Round the total (64bits/8 Bytes)
   100 => 104 (Bytes)

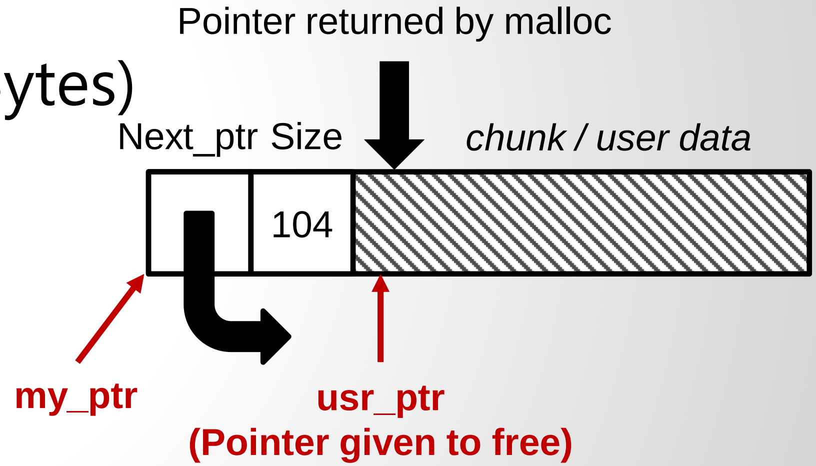
3. mmap the 104 Bytes, return the pointer + 16 Bytes
   return(my_ptr + 16);

Malloc storage techniques: Linked list

Step 2: free(usr_ptr);

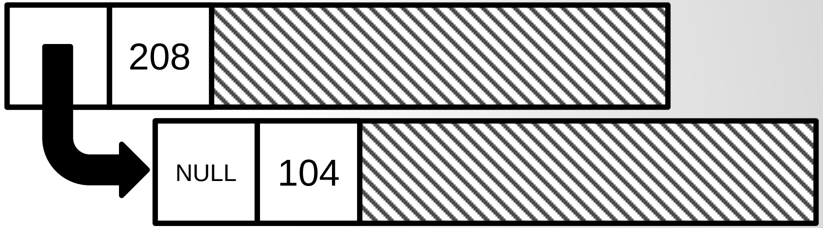
1. Find back the structure (– 16 Bytes)
   my_ptr = usr_ptr – 16;

2. Add it to the free list
   (insertion in a linked list)



Malloc storage techniques: Linked list

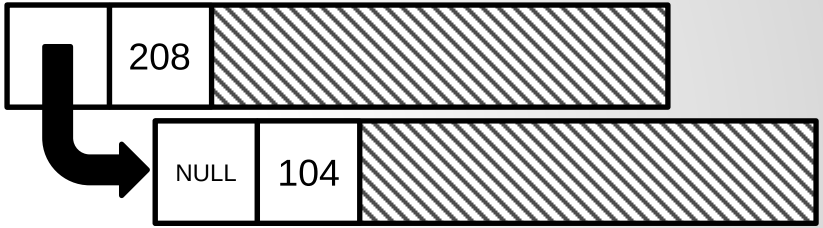
Step 3: oth_ptr = malloc(84);



1. Add the header
2. Round the total (104)
3. Search for an available block in the free list
   (browse the linked list of free blocks)
4. Use the available block OR create a new one
   (don't forget to delete the chosen block from the list)
5. … (repeat Step 1)

Malloc storage techniques: Linked list

Step 4: oth_ptr = realloc(oth_ptr, 200);

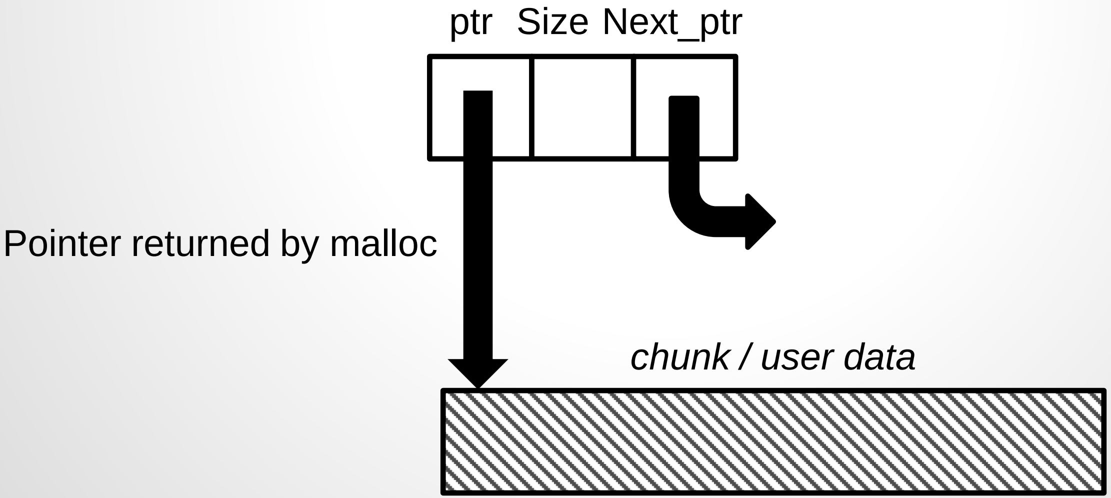


1. Add the header
2. Round the total (208)
3. Search for an available block in the free list
   (browse the linked list of free blocks)
4. Use the available block OR create a new one
   (don't forget to delete the chosen block from the list)
5. Copy the user data from the old block and free (Step 2)

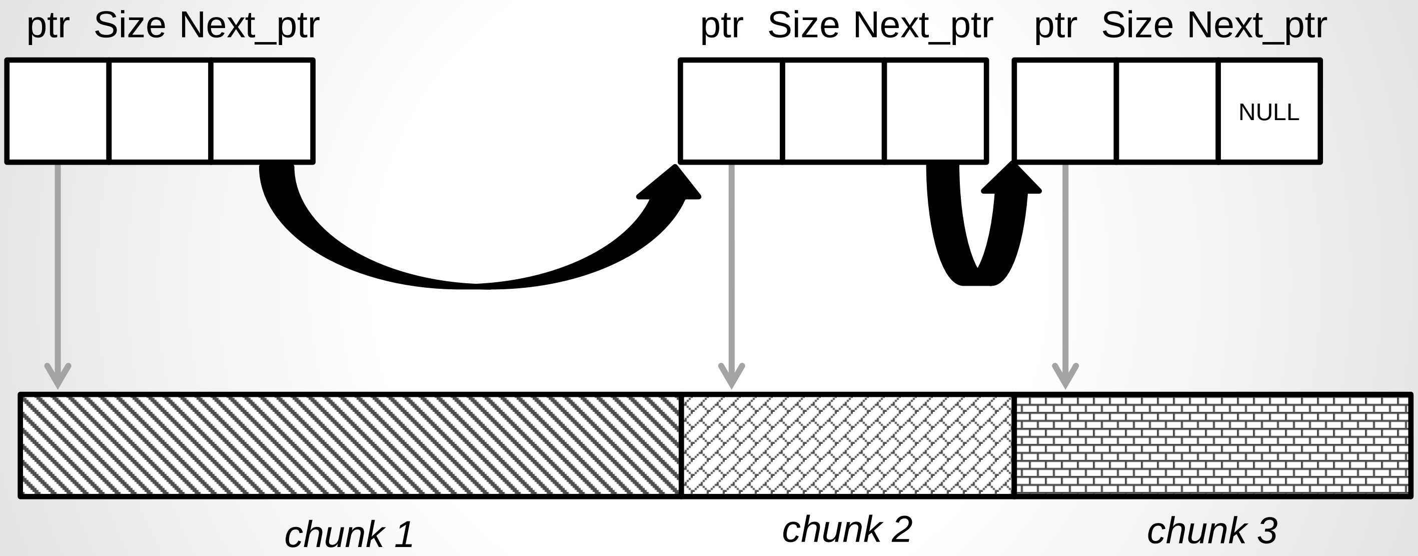
Malloc storage techniques: Linked list

- Variant: Header (meta-data) is stored somewhere else

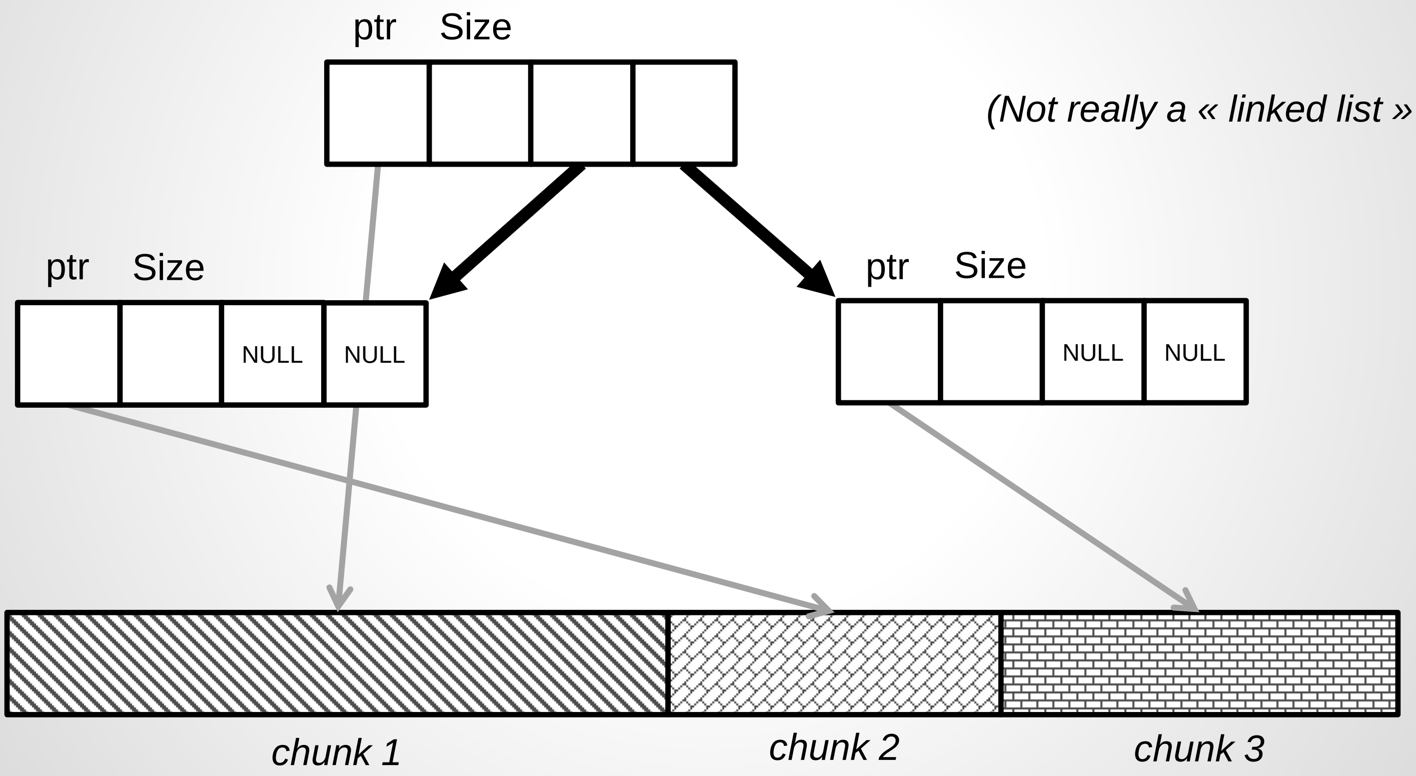
- Allows more ways to organize data



Malloc storage techniques: Linked list



Malloc storage techniques: Linked list



(Not really a « linked list » anymore…)

Malloc policies

- First fit
  - Reuse first hole available that is large enough

- Best fit
  - Reuse hole with exact match OR smallest hole large enough

- ~~Worst fit~~
  - ~~Reuse the largest hole available~~

Don't forget : if you manage memory, use char in C*

Malloc storage techniques: Bitmaps

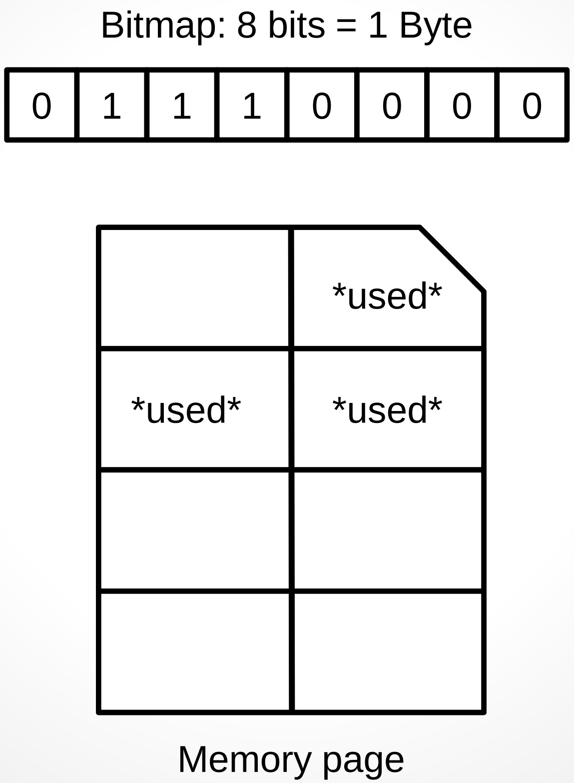
- Multiple small allocations on each page
  - Small allocations should be rounded to a 2^N
  - Bigger allocations can directly be done with mmap

- An array stores the states of each block in a bit
  - 1 = used 0 = free

- Mapping of 1 bit to N Bytes

Don't forget : if you manage memory, use char in C*

Malloc storage techniques: Bitmaps



Malloc storage techniques: Bitmaps

- Find a free chunk: check if the bitmap is full or not
  - (2^N) if 8, 16, 32, 64… bitmap is full
  - else, check for each bit which one is available

- Free a chunk: put ‘0’ in the bitmap at the right position
  - XOR and power of 2 are your friends

- Multiple bitmaps for multiple sizes of allocation
  - One bitmap per page
  - One (or more) page(s) for 8 Bytes of allocations, one for 16 bytes, …

Don’t forget : keep the alignment of memory!

Malloc methods: Buddy

- Divide a predefined space into smaller chunk
  - Divide « only » if the chunk asked is way bigger than what is available

- Every chunk is a power of 2
  - Easier for a lot of properties

- Define a minimal size and a maximal size of the chunks
  - Minimal chunk size could be the word size of the CPU (64 bits) or half of the word size (32 bits)

- Address gives you the size of the chunk
  - based on the min/max values, and the « level » like in a tree

Malloc methods: Buddy

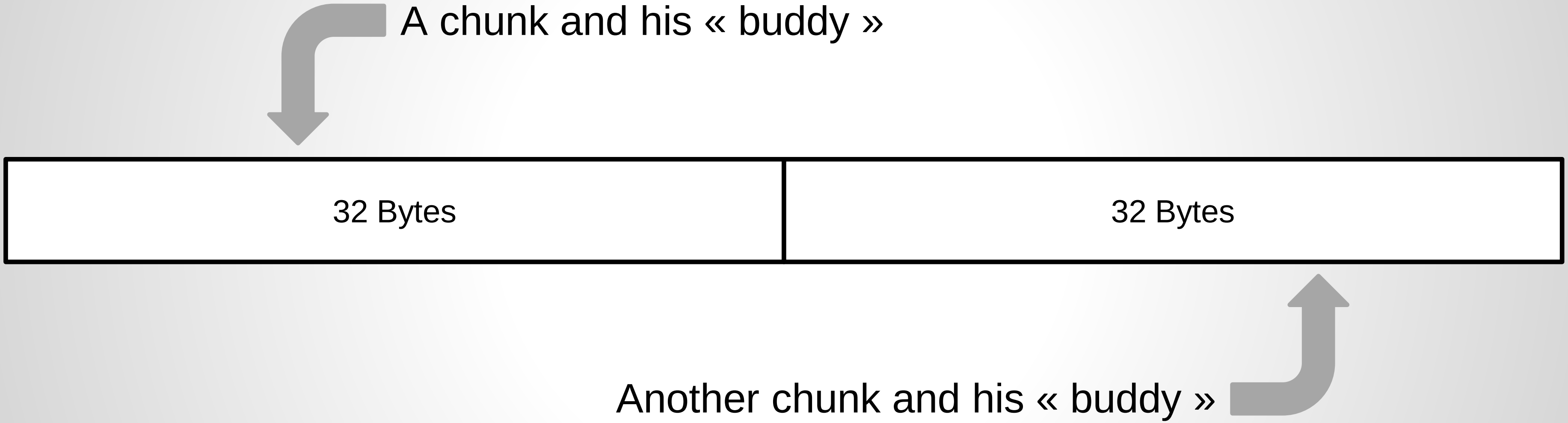
Level 0

64 Bytes

Chunk with the maximal size

Malloc methods: Buddy

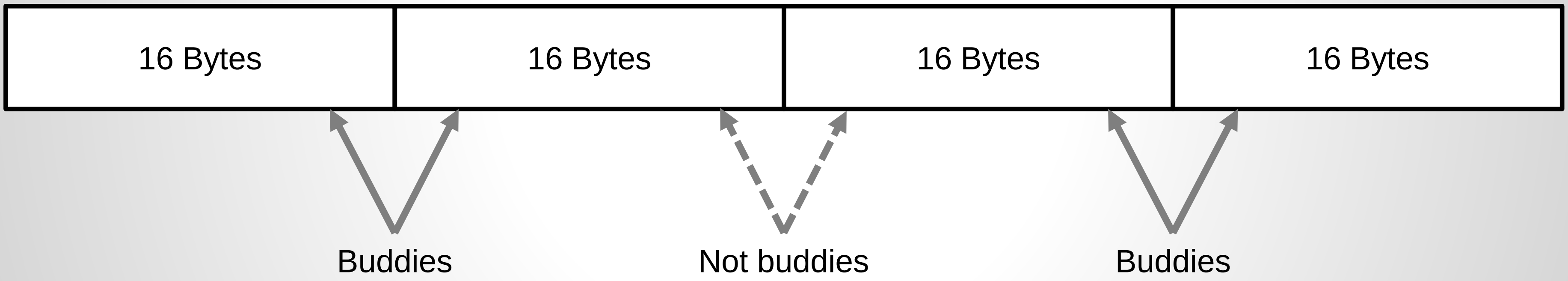
Level 1



As they are « buddies » to each other, they can be merged back if nothing is allocated inside

Malloc methods: Buddy

Level 2



Every neighbor cannot be merged back… only those which were a unique block in the previous level can

Malloc methods: Buddy

Level 3

8 Bytes	8 Bytes	8 Bytes	8 Bytes	8 Bytes	8 Bytes	8 Bytes	8 Bytes

Buddies Buddies Buddies Buddies

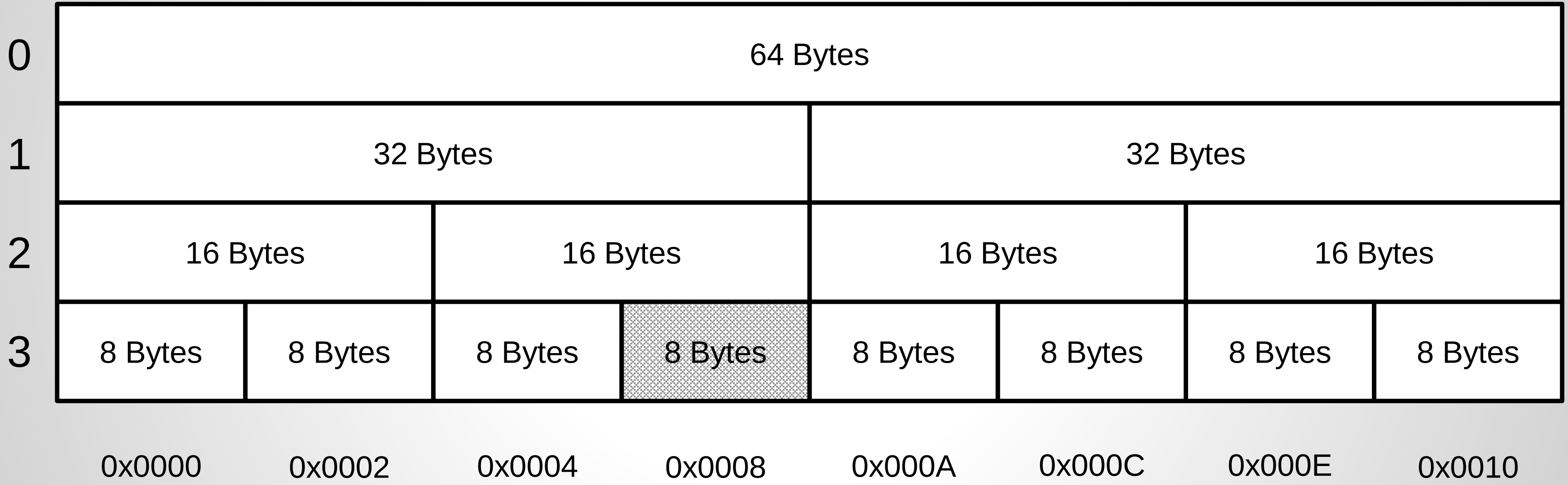
Now, there are a lot of small chunks…

In fact, chunks are not always splitted:
they are splitted IF AND ONLY IF they are required

Malloc methods: Buddy

<table>
<tr><td>0</td><td colspan="8">64 Bytes</td></tr>
<tr><td>1</td><td colspan="4">32 Bytes</td><td colspan="4">32 Bytes</td></tr>
<tr><td>2</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td></tr>
<tr><td>3</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td></tr>
</table>

Malloc methods: Buddy



If I tell you that my pointer is at the address 0x0008,
and that we have « 8 Bytes » as minimal size…
…You can deduce it is an 8 Bytes chunk

Same:
Pointer at 0x004 => 16 Bytes

Malloc methods: Buddy

<table>
<tr><td>0</td><td colspan="8">64 Bytes</td></tr>
<tr><td>1</td><td colspan="4">32 Bytes</td><td colspan="4">32 Bytes</td></tr>
<tr><td>2</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td><td colspan="2">16 Bytes</td></tr>
<tr><td>3</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td><td>8 Bytes</td></tr>
<tr><td></td><td>0x0000</td><td>0x0002</td><td>0x0004</td><td>0x0008</td><td>0x000A</td><td>0x000C</td><td>0x000E</td><td>0x0010</td></tr>
</table>

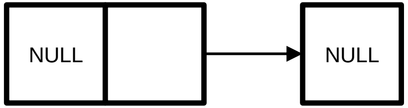
Now, you just have to store which cells are free!...

Example: extends the « free list » with a dimension storing the level

Malloc methods: Buddy

Step 1: malloc(14);

0


0


No other level is used

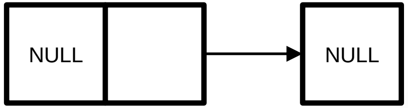
The main chunk is free

Example: max size at 64 Bytes

Malloc methods: Buddy

Step 2: malloc(14); « 14 » is not a power of 2, let’s round up at 16

0

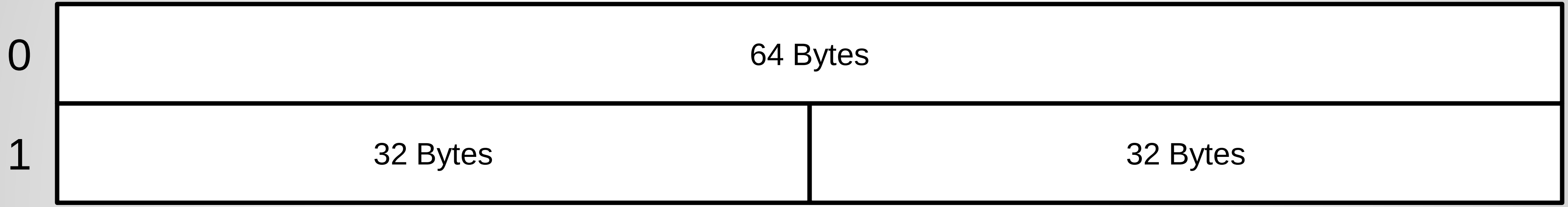

0


Level 0 is too large…
Let’s split it.

Example: max size at 64 Bytes

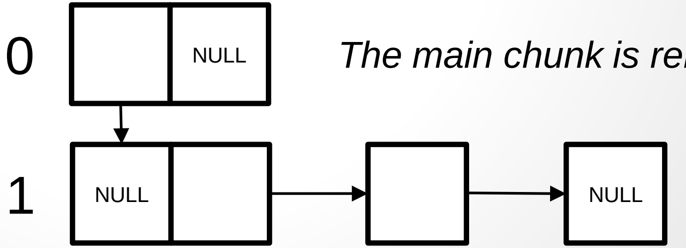
Malloc methods: Buddy

Step 3: malloc(14); [search for 16 Bytes]



Let’s split level 0 into two chunks of 32 Bytes…

Well, 32 Bytes is still too large.



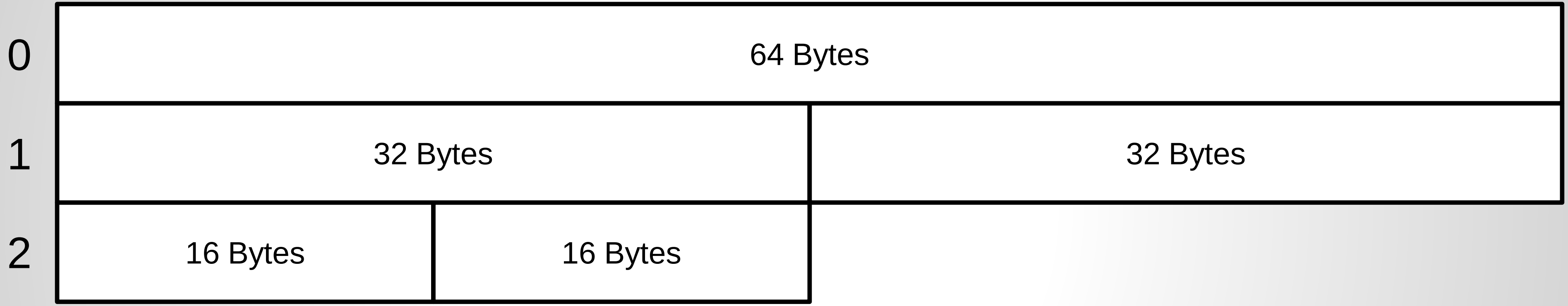
The main chunk is removed…

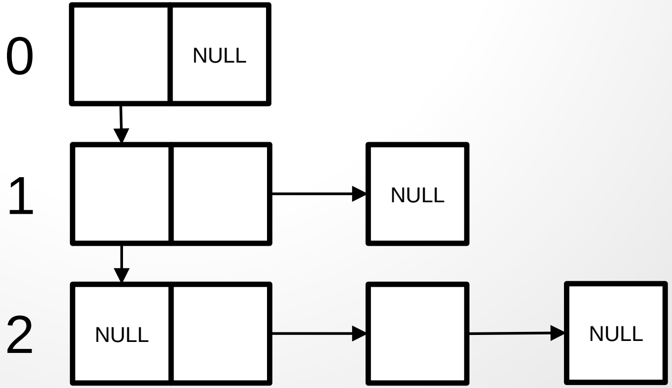
…and another level is created + 2 chunks

Example: max size at 64 Bytes

Malloc methods: Buddy

Step 4: malloc(14); [search for 16 Bytes] => FOUND !





Let’s split level one chunk of level 1 into two chunks…

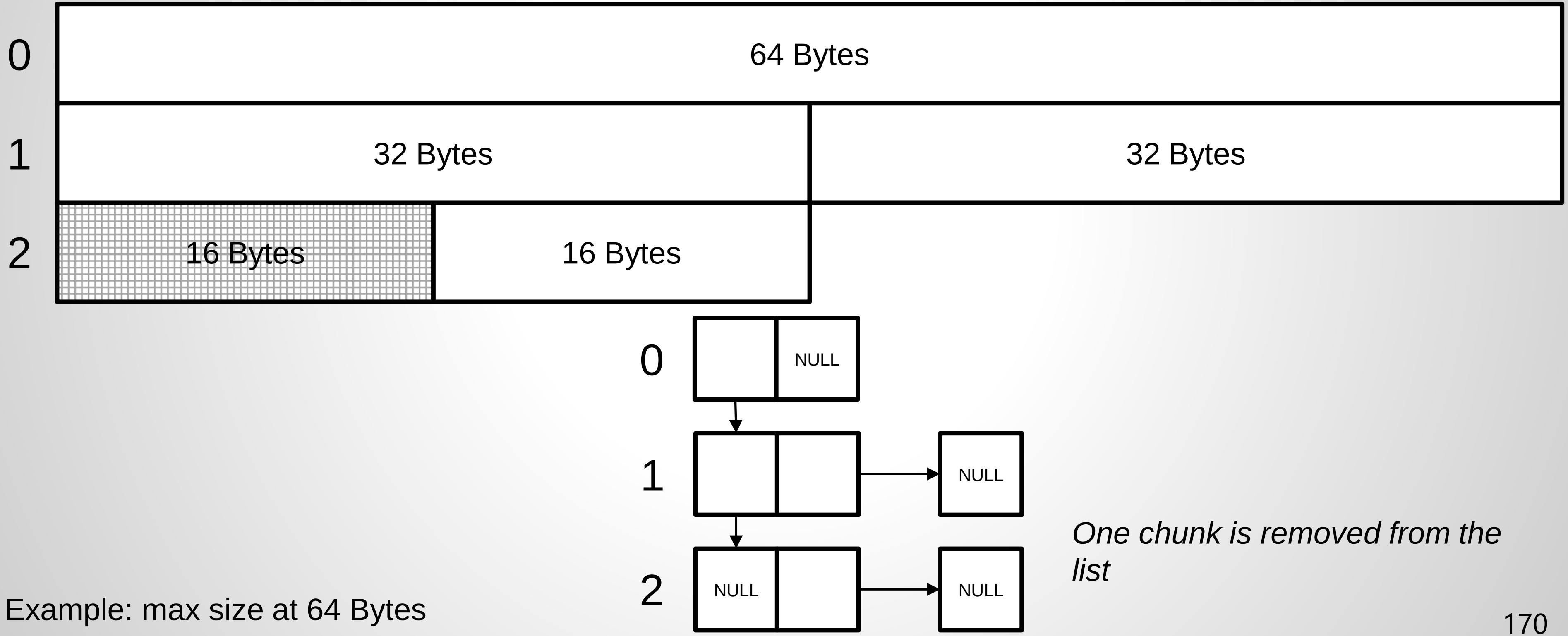
Now, 16 Bytes are perfect!
Let’s use one…

Example: max size at 64 Bytes

Malloc methods: Buddy

Step 5: malloc(14);

Reserves the address, and returns the pointer



One chunk is removed from the list

Example: max size at 64 Bytes

Malloc implementations

- There are a lot of ways to implement malloc

- You should definitely use the one offered by your system...

- ...except if you have *unconventional* needs.
  - Specific mallocs: jemalloc, tcmalloc, ...
  - Build your own allocator above mmap(2) & brk(2)/sbrk(2)

Quick overview of the threads

Multithreading

- Problems: How to…
  - Allows parallelism inside a process?
  - Reduces the cost of context switching?
- Solution
  - Thread (lightweight process): state, registers & stack.
    Shares other resources
  - Process: group of threads.
    Classical process = process with only 1 thread
- Functionalities
  - Same as a process: creation, termination, state, etc…
  - New issues: concurrent access on shared resources

Userland Threads

- Principle
  - Implemented as a library in userland
  - 1 thread table per process
- Pros
  - Usable on a system without support for threads
  - Fast context switching (no kernel trap)
  - Customizable scheduling algorithm
- Cons
  - Needs for unblocking syscalls
  - Threads can lock the CPU (they need to yield explicitly)
  - Threads are used to alleviate blocking

Kernel Threads

- Principle
  - Adds a thread table inside the process table
  - Every blocking call is implemented as a syscall
- Pros
  - Ease to create an application using them
  - No need for non blocking calls
- Cons
  - Creation/deletion/bookkeeping have a cost
  - Interrupt & blocking syscalls

Unified API: Pthread

- POSIX API used to run threads
- Simple unified interface for multi threaded environment on POSIX system
- Beware: everything is shared between threads...
  - Except the thread ID in the scheduler => each thread is independent
- pthreads(7)
  - pthread_create(3), pthread_join(3), pthread_yield(3), ...

Where? What?

Per Thread

- Thread ID
- Signal mask
- Errno
- Scheduling policy
- Capabilities
- CPU affinity

Per Process

- Process ID
- Parent Process ID
- Process Group
- User/Group ID
- File descriptors
- umask
- Current directory
- Limits
- ...