
بسم الله الرحمن الرحیم

Piece 1

» Virtualization

» Process

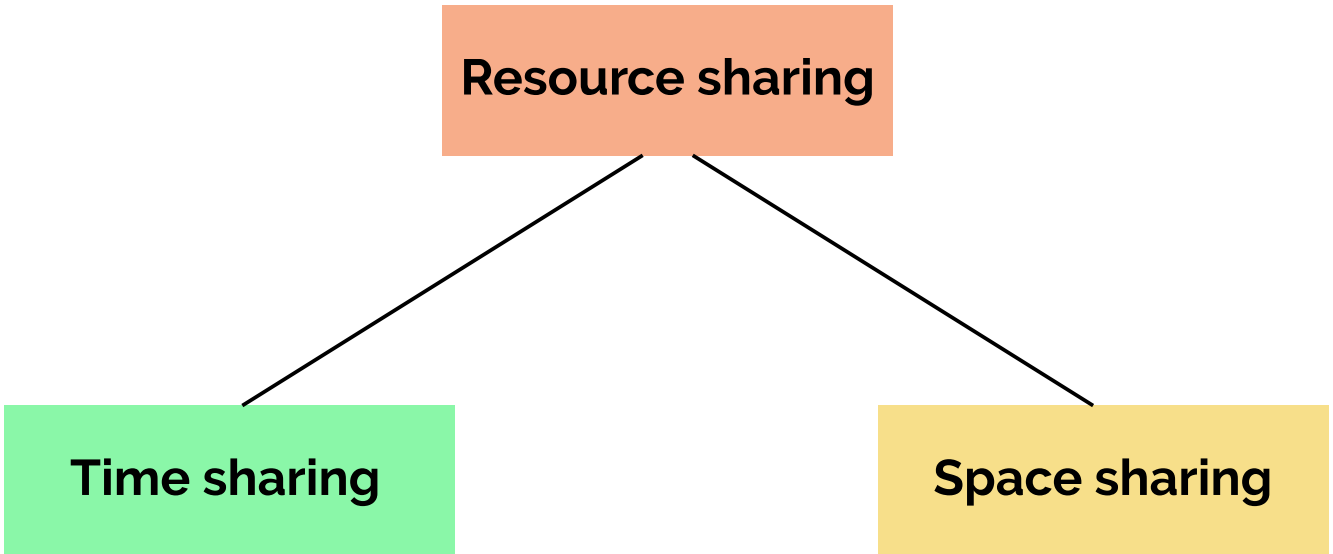
Processes

Process: one of the most fundamental abstractions that the OS provides to users

- A process is a running program.
- A process provides two important abstractions: (1) the illusion that the program has exclusive use of the processor, and (2) the illusion that the program has exclusive use of the main memory.

Often more than one program is running

- CPU time-sharing $\rightarrow$ illusion of many virtual CPUs
  - The potential cost is performance (why?)
- needs both low level machinery (**mechanism**) and high level intelligence (**policy**)
- mechanism: context switch, policy: scheduling



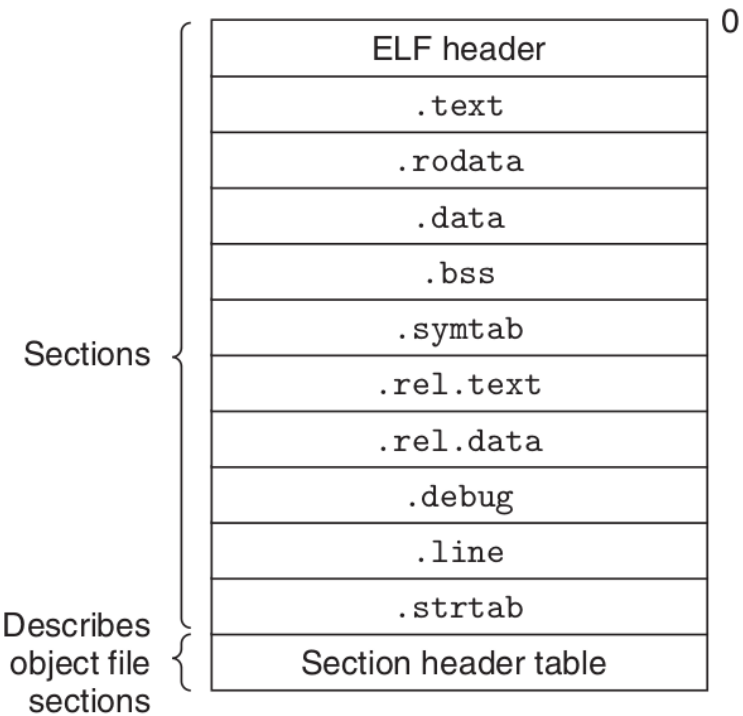
- Single core CPU
- Memory

Multi-core & Hyper-threaded ?

Aside: Program

A program is a lifeless thing: instructions (and maybe some static data) stored in a non-volatile storage such as hard disk.
» use `xxd`, `size`, `objdump -h` commands to see different parts of a binary executable file!

Aside: ELF



Typical ELF relocatable object file

Randal E. Bryant, David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, ch 7.

OS APIs for Processes

- Create: loads (code and date from HD/SSD into memory), allocates and initializes memory (stack and heap), initializes I/O (stdin, stdout, stderr), context switch (starts the program running at the entry point, `main()`)
  - The OS associates a **process identifier**, or **pid**, with each process.
- Destroy: forcefully terminates a process
- Wait: lets a process to complete
- other controls: suspend, ...
- status: running, ready (runnable), blocked (suspend)

We will study Posix APIs soon ...

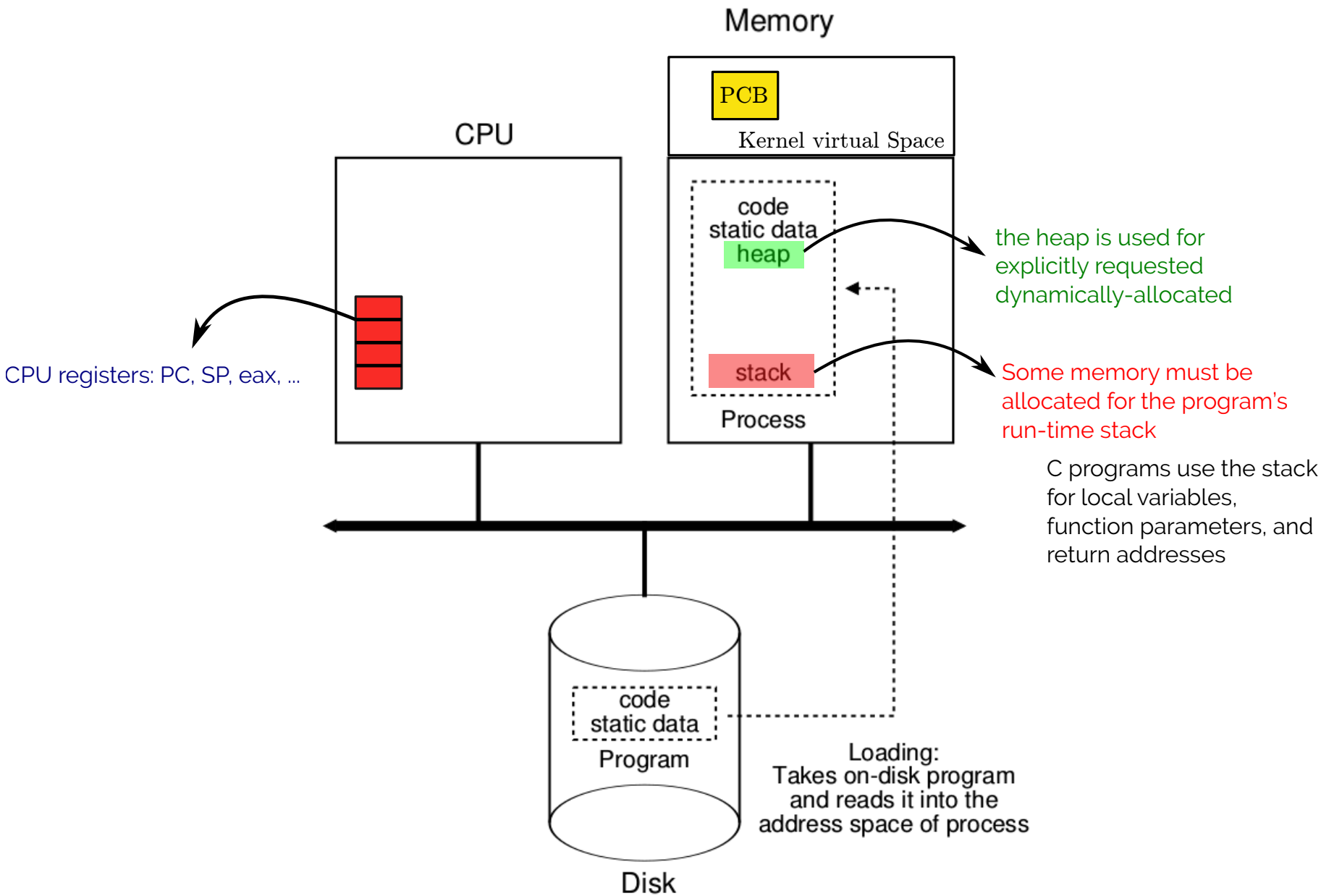
Process Creation: more details

Process block (PCB) initialization: Finding an empty record for the new PCB in the process list, assigning PID, ...

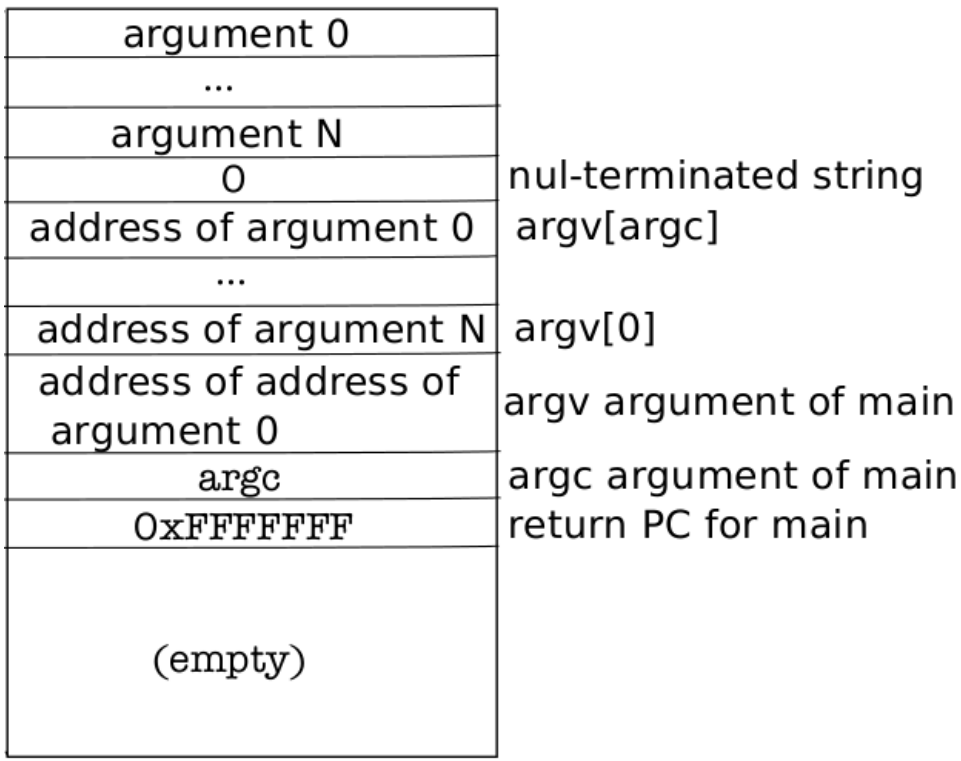
Address space setup: loading text & data, stack, heap (eagerly or lazily)

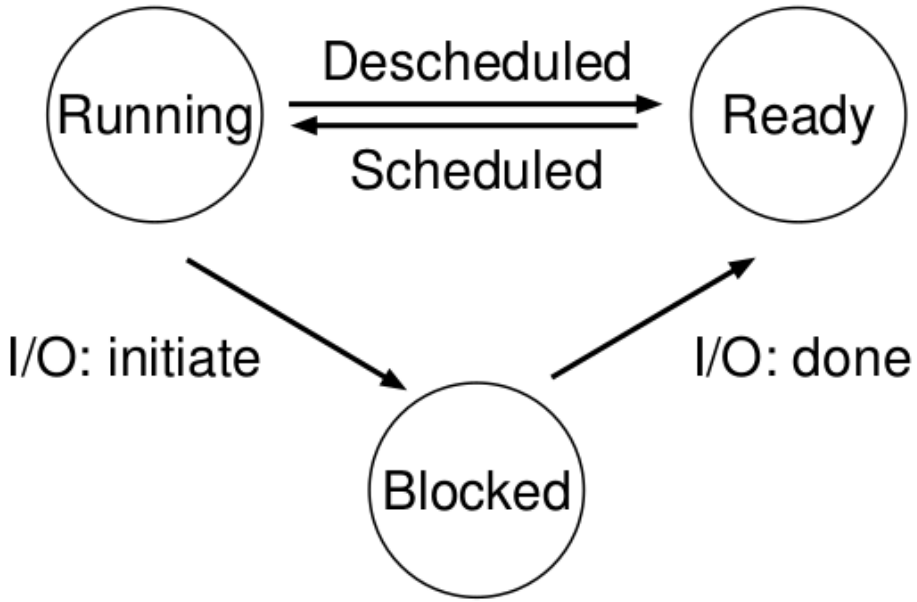
I/O setup: each process by default has three open file descriptors, for standard input, output, and error

Start the program running: by jumping to the main() routine (through a specialized mechanism)


Memory
PCB
Kernel virtual Space
CPU
code
static data
heap
stack
Process
CPU registers: PC, SP, eax, ...
the heap is used for explicitly requested dynamically-allocated
Some memory must be allocated for the program's run-time stack
C programs use the stack for local variables, function parameters, and return addresses
code
static data
Program
Loading:
Takes on-disk program and reads it into the address space of process
Disk

Aside: Initial stack in xv6





- running: its state word is contained in a processor which is running.
- ready: it could be placed in execution by a processor if one were free
- suspend: awaiting activation by an external event, such as the completion of an i/o function.

Example

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Tracing Process State: CPU Only

Example

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Tracing Process State: CPU and I/O

Data Structure

- The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information.

Example

- A **process list** contains information about all processes in the system.
- Each entry points to a **process control block (PCB)**, which is really just a structure that contains information about a specific process.

Aside: xv6

In this course, instead of practical OS kernels (with millions of code lines), we use xv6 (just few thousands of code lines) for Kernel hacking

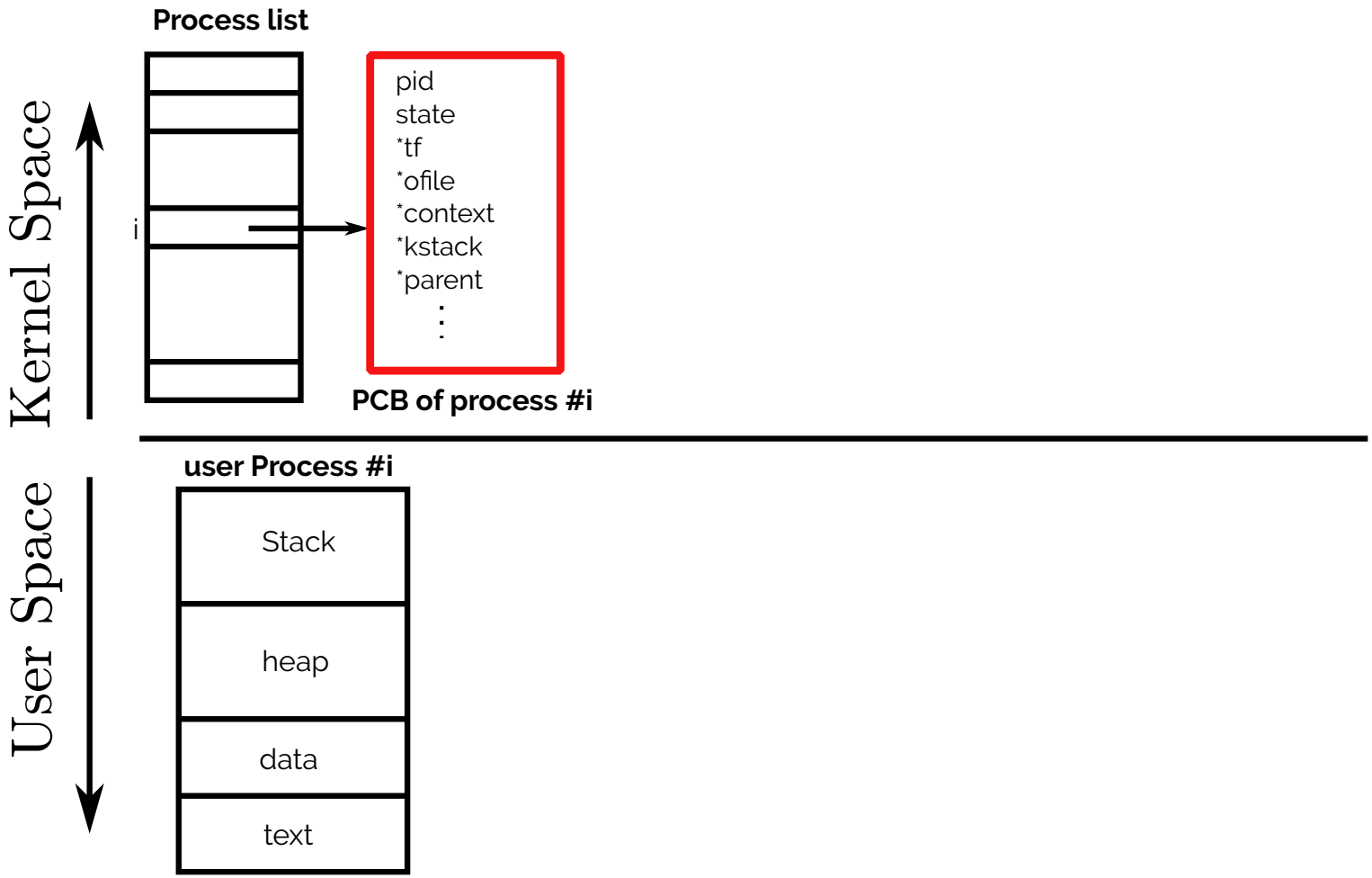
- A teaching OS developed at MIT
- simplified Unix version 6 (main change: from PDP-11 to Intel x86)
- well structured and documented
- github link: https://github.com/mit-pdos/xv6-public
- A commentary book: https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf
- visit https://pdos.csail.mit.edu/6.828/2012/xv6.html

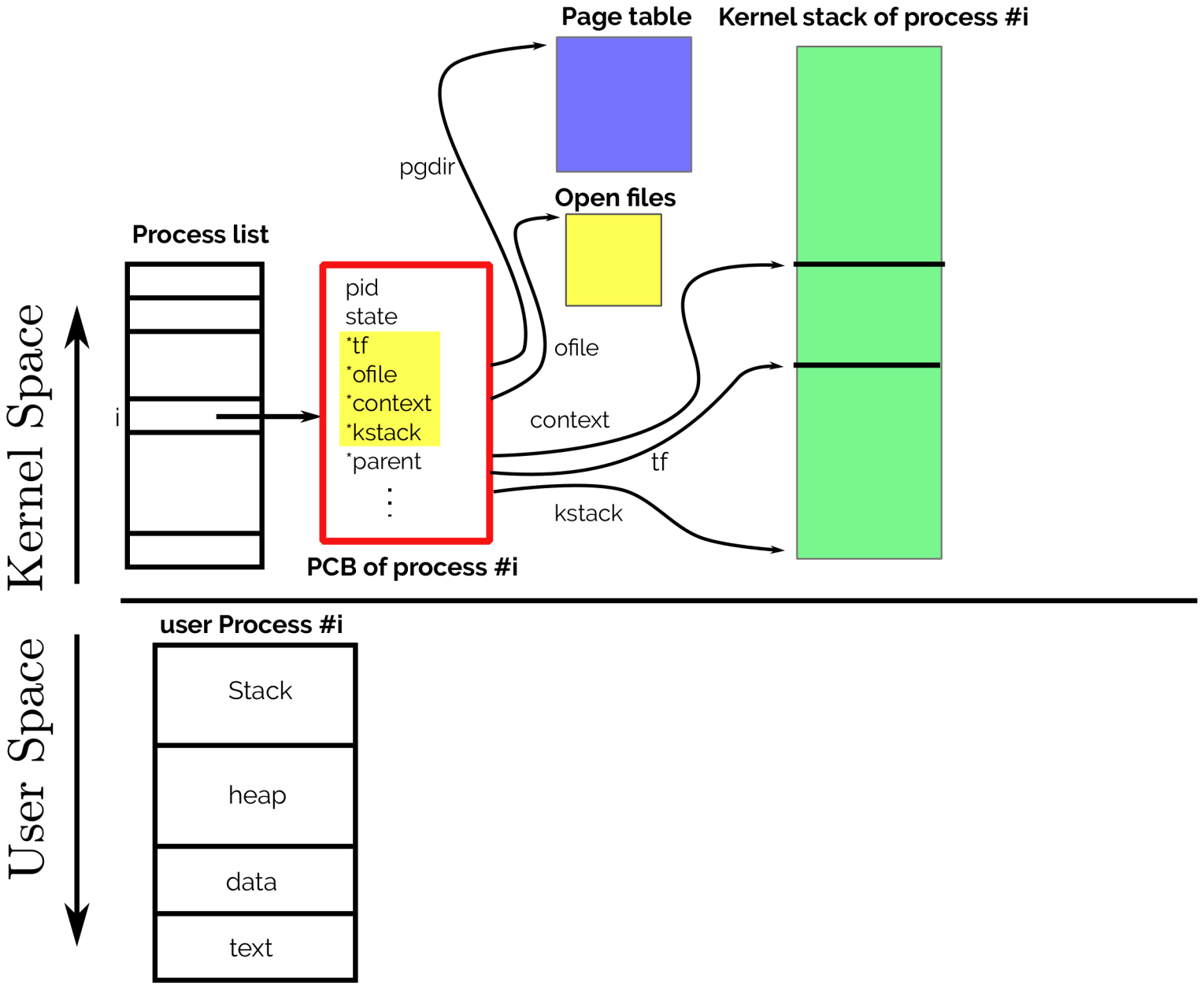
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

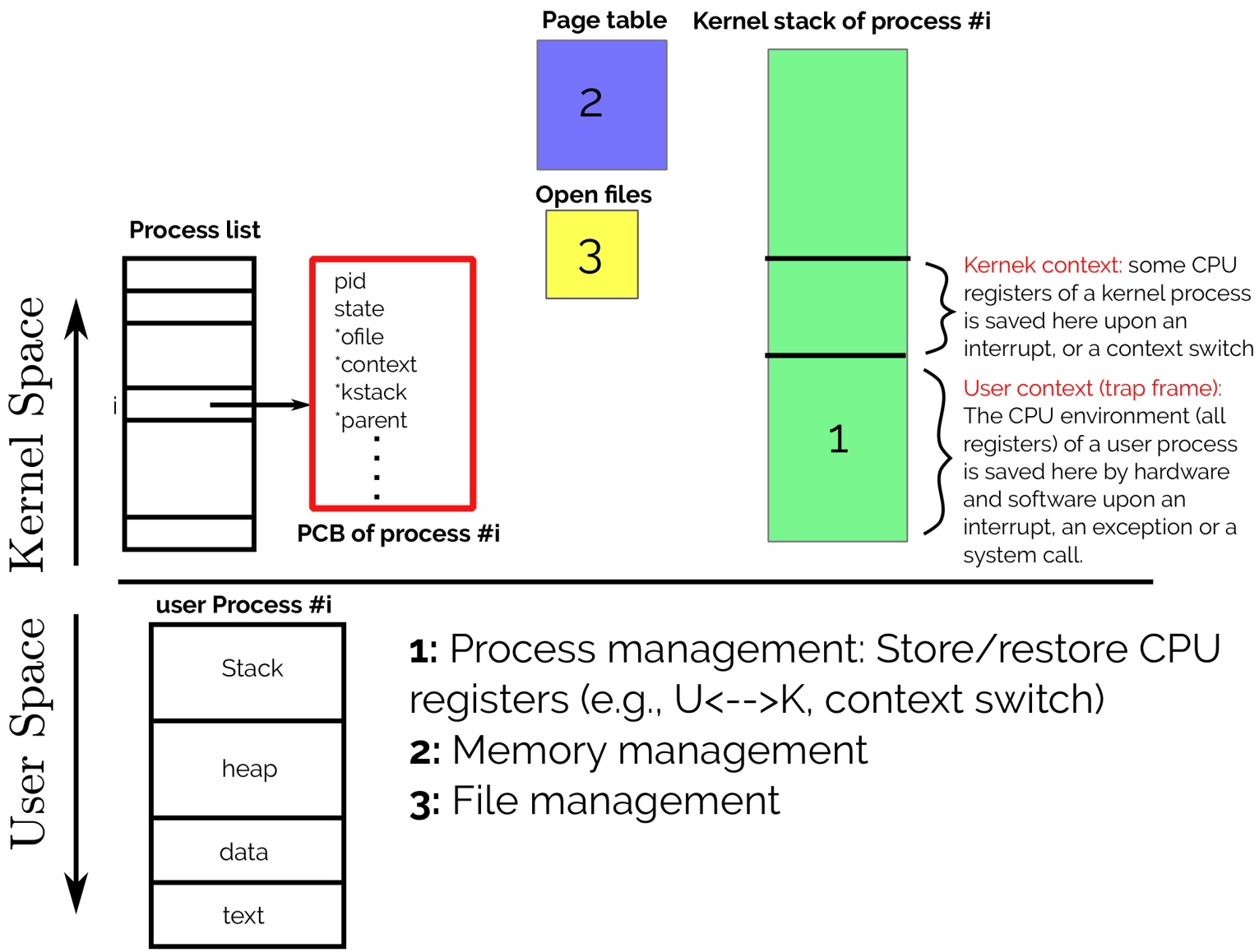
The xv6 Proc Structure

(**Process Control Block (PCB)/process descriptor**)

Exercise: USe `grep -nri` to locate where the `struct proc` is defined in the xv6 kernel.


Process list
Kernel Space
i
pid
state
*tf
*ofile
*context
*kstack
*parent
PCB of process #i
user Process #i
User Space
Stack
heap
data
text


Page table
Kernel stack of process #i
pgdir
Open files
Process list
pid
state
*tf
*ofile
*context
*kstack
*parent
ofile
context
tf
kstack
i
PCB of process #i
Kernel Space
User Space
user Process #i
Stack
heap
data
text



1: Process management: Store/restore CPU registers (e.g., U<-->K, context switch)
2: Memory management
3: File management

Aside: Process; Not a Unique Definition

Process = program in execution

Process = program in execution on a pseudo-processor

Process = the sequence of actions performed by the program

Process = sequence of program states

Process = a program status block

Example

Which state-to-state transition is not possible?

- ☐ Ready $\rightarrow$ Blocked
- ☐ Running $\rightarrow$ Blocked
- ☐ Ready $\rightarrow$ Running
- ☐ Blocked $\rightarrow$ Ready

(Posix) Process API

Process Creation in UNIX Systems

From a programmer's point of view, the user is a peripheral that types when you issue a read request.– Peter Williams

Three system calls

- `fork()` : Create a process
- `exec(filename, *argv)` : Load a file and execute it
- `wait()` : Wait for a child process to exit (die)

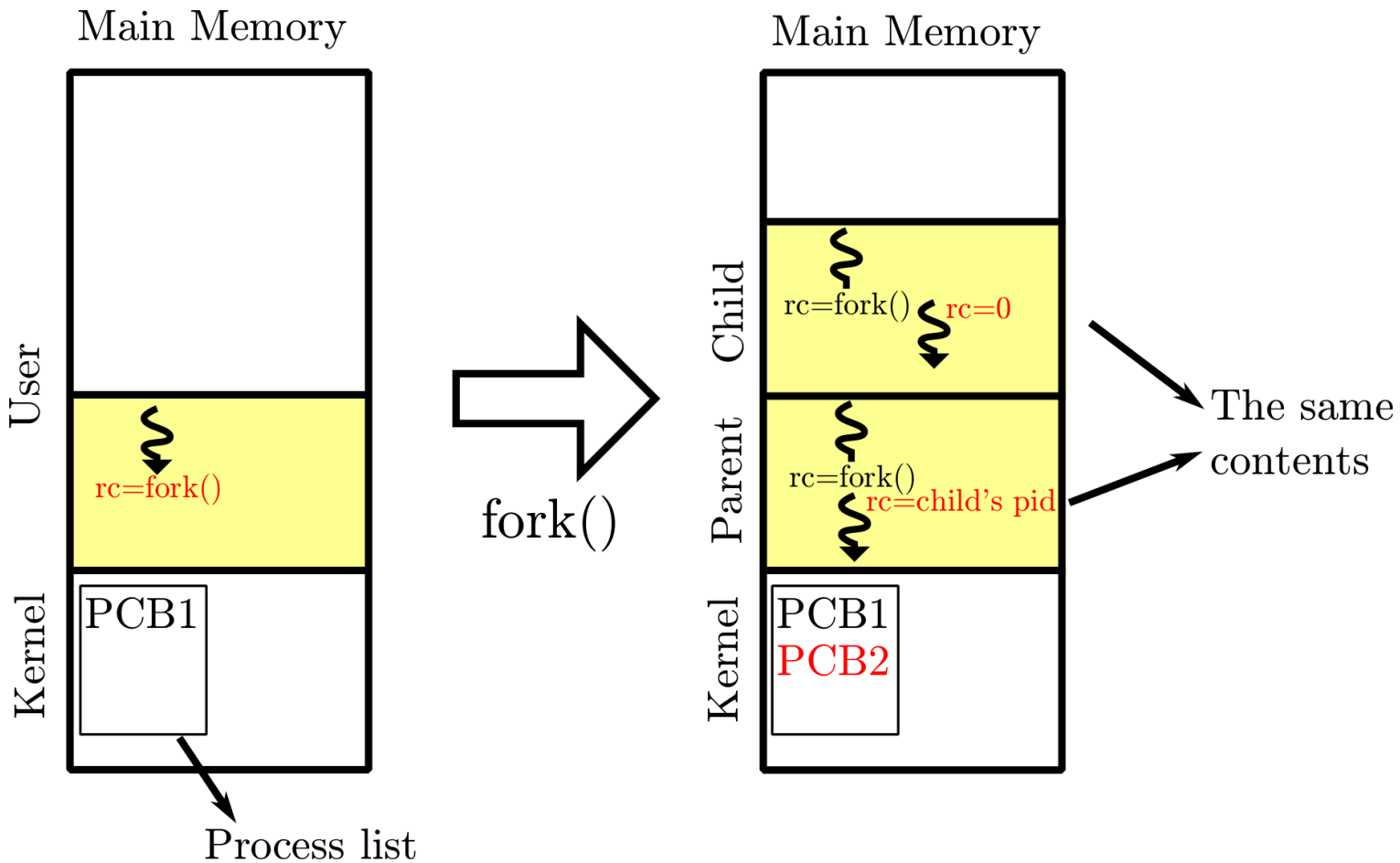
fork

- A process may create a new process using the `fork` system call.
- `fork` creates a new process, called the child process, **with exactly the same memory contents as the calling process**, called the parent process.
- `fork` returns twice: in both the parent and the child.
  - In the parent, `fork` returns the child's pid; in the child, it returns zero.

Note

Fork is a kind of cloning

Although the child has the same memory contents as the parent initially, the parent and child are executing with different memory and different registers: changing a variable in one does not affect the other. For example, when the return value of wait is stored into pid in the parent process, it doesn't change the variable pid in the child. The value of pid in the child will still be zero.


Main Memory
User
rc=fork()
Kernel
PCB1
Process list
fork()
Main Memory
Child
rc=fork()
rc=0
Parent
rc=fork()
rc=child's pid
Kernel
PCB1
PCB2
The same
contents

Aside: copy-on-write

We will revisit fork (after understanding virtual memory) to see how it really creates a new process with its own independent virtual address space.

They could be twins!

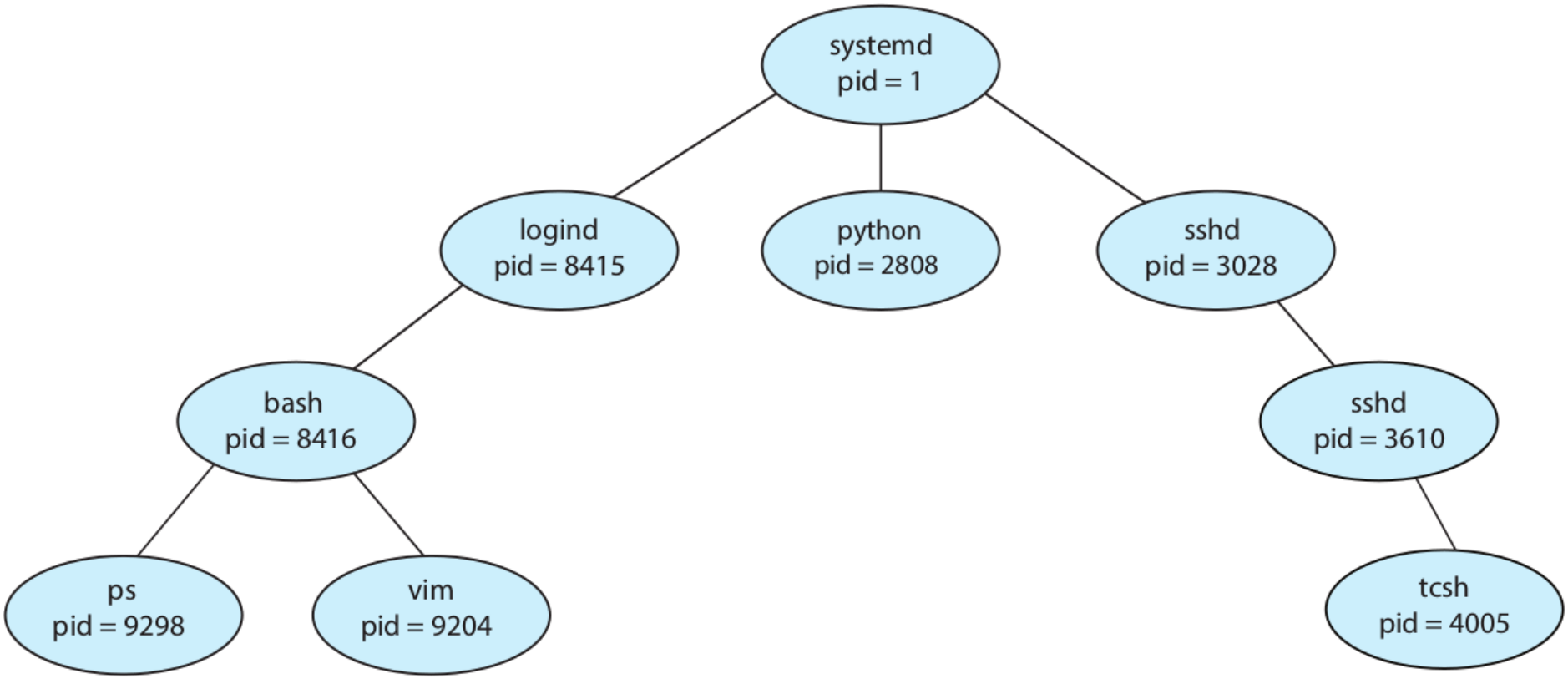
A man (right) has his father's facial expression and haircut down to a tee, alongside the wide collared shirt and tank top, which he must have borrowed from his old man to recreate his dad's pose (left)

Example: fork

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
    return 0;
}
```

```
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ gcc p1.c -o p1
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ ./p1
hello world (pid:10068)
hello, I am parent of 10069 (pid:10068)
hello, I am child (pid:10069)
```

- `fork()` creates a child which is an (almost) exact copy of its parent.
- child doesn't start running at `main()` , rather, it just comes into life as if it had called `fork()` itself.
- child has its own address space, its own registers, its own PC, ...
- return value of `fork()` is different in parent and child:
  - the parent receives the PID of the newly-created child,
  - the child receives a return code of zero.



A tree of processes on a typical Linux system

ASIDE: PS,PSTREE

On UNIX and Linux systems, we can obtain a listing of processes by using the `ps` command. For example, the command `ps -el` will list complete information for all processes currently active in the system. In addition, Linux systems provide the `pstree` command, which displays a tree of all processes in the system.

exit

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call.
- At that point, the process may return a status value (typically an integer) to its waiting parent process (via the `wait()` system call).
- All the resources of the process –including physical and virtual memory, open files, and I/O buffers–are deallocated and reclaimed by the operating system.

wait

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent.
- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.
- For now, we need to be aware that a process that calls `wait` to wait for the first child to die!

Example: wait()

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1);
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
```

```
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ gcc p2.c -o p2
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ ./p2
hello world (pid:10655)
hello, I am child (pid:10656)
hello, I am parent of 10656 (wc:10656) (pid:10655)
```

Aside: Kernel handles and garbage collection

UNIX has various system calls that return a **handle** to some kernel object; these handles are used in later calls as an **ID**.

- The process ID returned by UNIX fork is used in later calls to UNIX wait;
- UNIX open returns a file descriptor that is used in other system calls.

It is important to realize that these handles are not pointers to kernel data structures; otherwise, ...
Further, handles are reference counted. That means ...

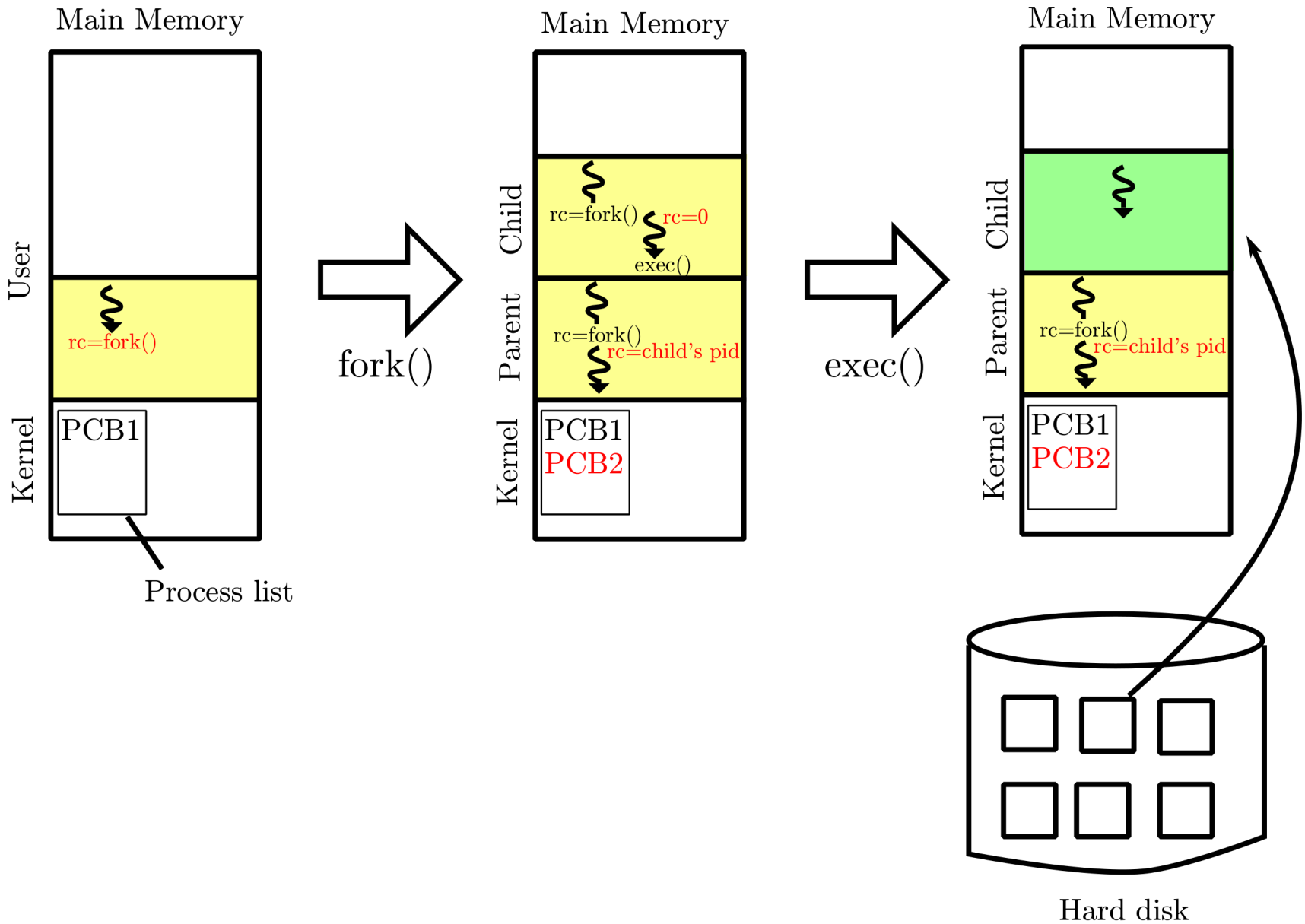
exec

one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the `exec` functions.

- When a process calls one of the `exec` functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID (along with open files, environment variables, working directory, user, etc.) does not change, because a new process is not created; exec merely **replaces the current process – its text, data, heap, and stack segments – with a brand-new program from disk.**
- Successful `exec()` never returns.

Note

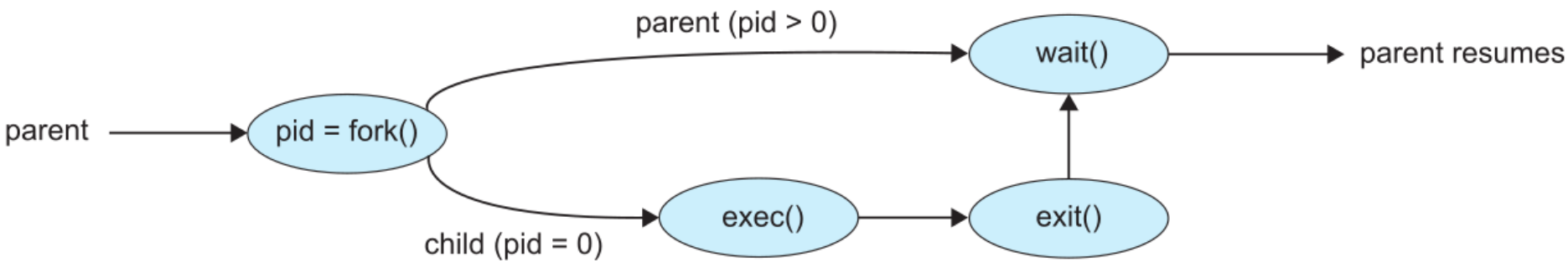
With fork, we can create new processes; and with the exec functions, we can initiate new programs.


Main Memory
User
rc=fork()
Kernel
PCB1
Process list
fork()
Main Memory
Child
rc=fork()
rc=0
exec()
Parent
rc=fork()
rc=child's pid
Kernel
PCB1
PCB2
exec()
Main Memory
Child
Parent
rc=fork()
rc=child's pid
Kernel
PCB1
PCB2
Hard disk

Example: exec()

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
```

```
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ gcc p3.c -o p3
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ ./p3
hello world (pid:10940)
hello, I am child (pid:10941)
 32 123 966 p3.c
hello, I am parent of 10941 (wc:10941) (pid:10940)
```


parent (pid > 0)
wait()
parent resumes
parent
pid = fork()
exec()
exit()
child (pid = 0)

Discussion Question

What happens if you run `exec ls` in a UNIX shell? Why?

Hint: Look for "builtin commands" in `$ man bash`

separate calls for creating a process and loading a program is a clever design.

Combination of `fork()` and `exec()` is the way that unix shell does a bunch of useful things rather easily.

- because this combination lets the shell run code after the call to `fork()` but before the call to `exec()` ; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.
- such as changing standard input/output, piping, ...

Example: I/o Redirection

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        assert(wc >= 0);
    }
    return 0;
}
```

```
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ gcc p4.c -o p4
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ ./p4
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ ls
Makefile  p1  #p1.c#  p1.c  p2  p2.c  p3  p3.c  p4  #p4.c#  p4.c  p4.output  README.md
~/Desktop/os/remzi/codes/ostep-code/cpu-api(master)$ cat p4.output
 34 114 884 p4.c
```

There are two uses for `fork` :

- When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time.

Example

This is common for network servers–the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.

- When a process wants to execute a different program.

Example

This is common for shells. In this case, the child does an exec right after it returns from the `fork` .

Beyond `fork()` , `exec()` , and `wait()`

- there are a lot of other interfaces for interacting with processes in UNIX systems.
- For example, the `kill()` system call is used to send **signals** to a process
  - to stop, continue, or even terminate the process.

Aside: Tools for Manipulating Processes

Linux systems provide a number of useful tools for monitoring and manipulating processes:

- `ps` Lists processes (including zombies) currently in the system.
- `top` prints information about the resource usage of current processes
- `kill` can be used to send arbitrary signals to processes
- `/proc` A virtual filesystem that exports the contents of numerous kernel data structures

سوال ۱: کد زیر قسمتی از برنامه manyForks.c را نشان می دهد.

```
1   int main(){
2     int i, pid;
3     for(i=0;i<=2;i++){
4        fork();
5        printf("%d\n",getpid());
6     }
7     pid = wait();
8     printf("%d\n",pid);
9     return(0);
10  }
```

فرض کنید پروسس اولیه که توسط shell ایجاد می شود pid برابر با ۱۰۰ دارد و همچنین pid های مربوط به پروسسهای بعدی به ترتیب با افزایش یک واحدی بدست می آیند.

نحوه عملکرد زمانبند را هم بدین صورت در نظر بگیرید: پس از اجرای هر فراخوانی سیستمی (هر fork ، wait و یا return) اجرا می شود و پروسسی را ازبین پروسسهای آماده برای اجرا انتخاب می کند که pid کمتری دارد.

الف) درخت پروسسها را ترسیم کنید.

ب) آنچه در ترمینال چاپ می شود را بنویسید.

ج) ترتیب به اتمام رسیدن پروسسها به چه نحو است؟

Class Question

Which output is possible after running the following program?

```
int main(){
  int x = 0;
  if (fork()==0){
    x = 10;
  printf("%d ",x);
  }   printf("%d ",x);
}
```

A) 10 10 0

B) 0 10

C) 10 0 10

D) 0

E) 10 10

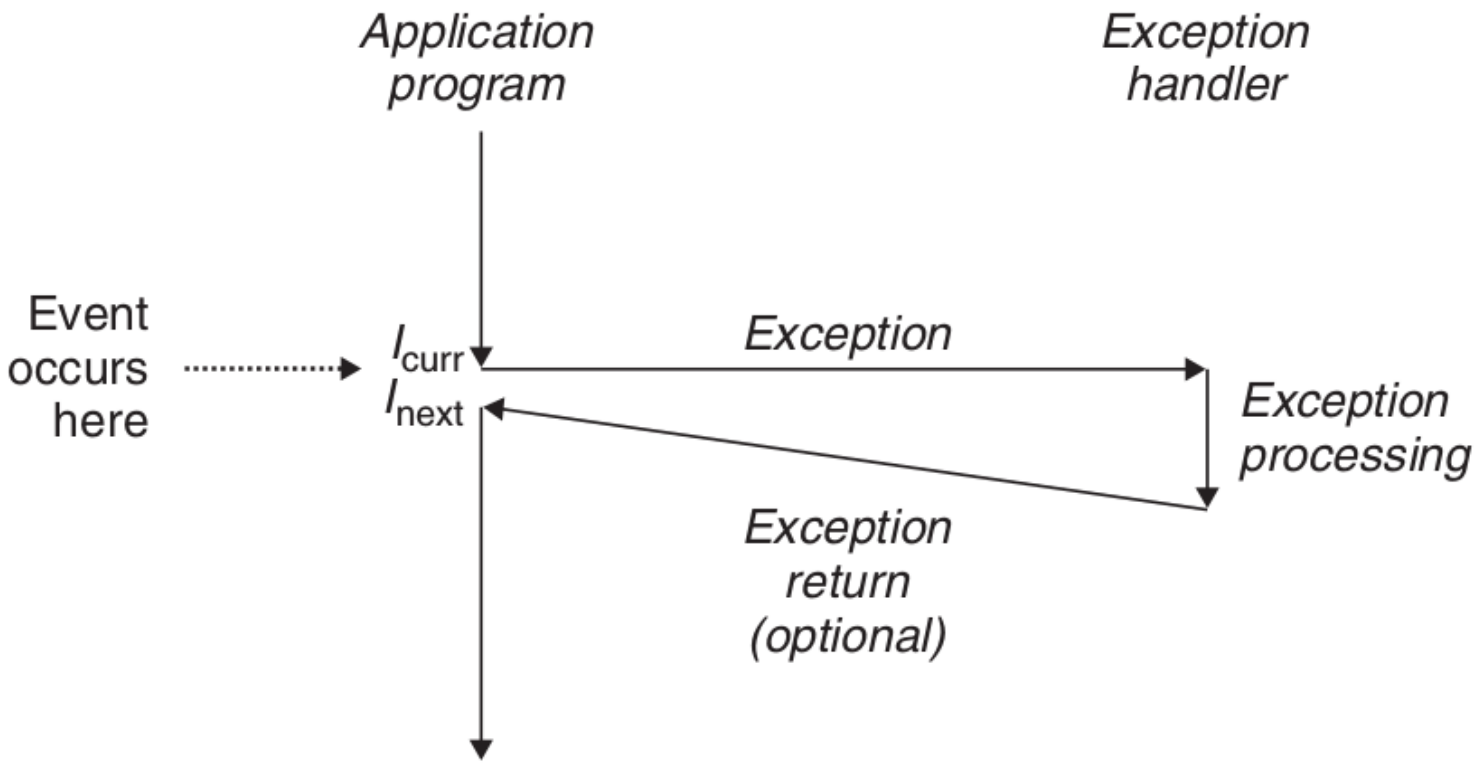
F) 0 10 10

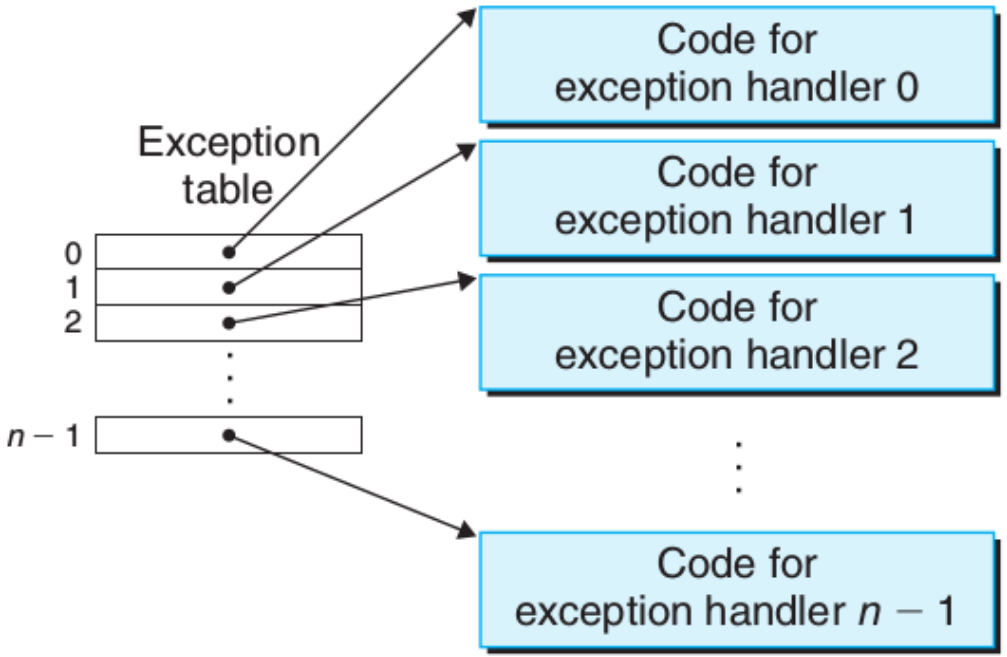
System Call & CPU Virtualization– Mechanism

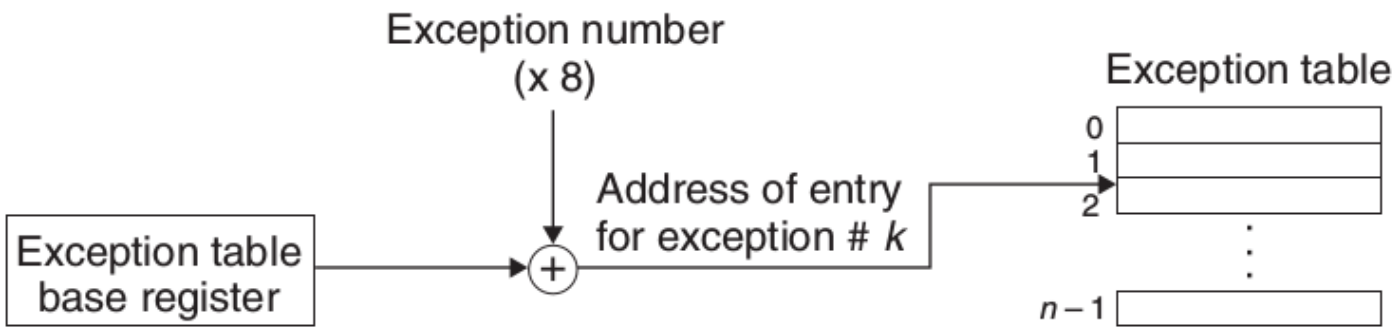
Hardware Support Primitives

Exception

An exception is an abrupt change in the control flow in response to some change in the processor's state.




Exception table
0
1
2
n – 1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
Code for exception handler n – 1


Exception number (x 8)
Exception table base register
Address of entry for exception # k
Exception table
0
1
2
n – 1

Classes of Exceptions

Asynchronous (Interrupts)

Synchronous: Traps, Faults, Aborts

Hardware Support Primitives

Hardware support primitives:

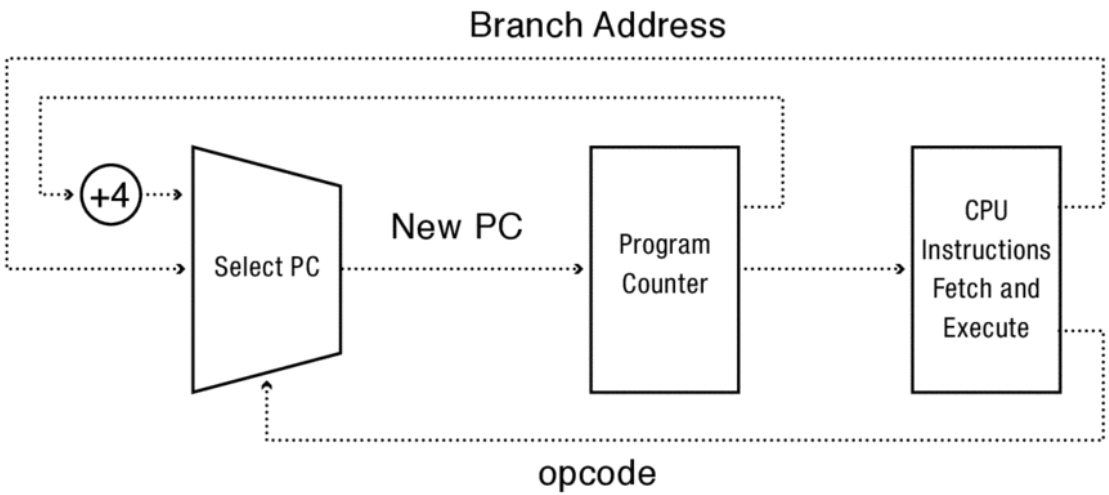
- Privileged instructions
- Interrupt vector
- Interrupt masking
- Memory Protection
- Atomic saving and restoring registers
- Interrupt stack
- Timer

oh lady, you alone is my master

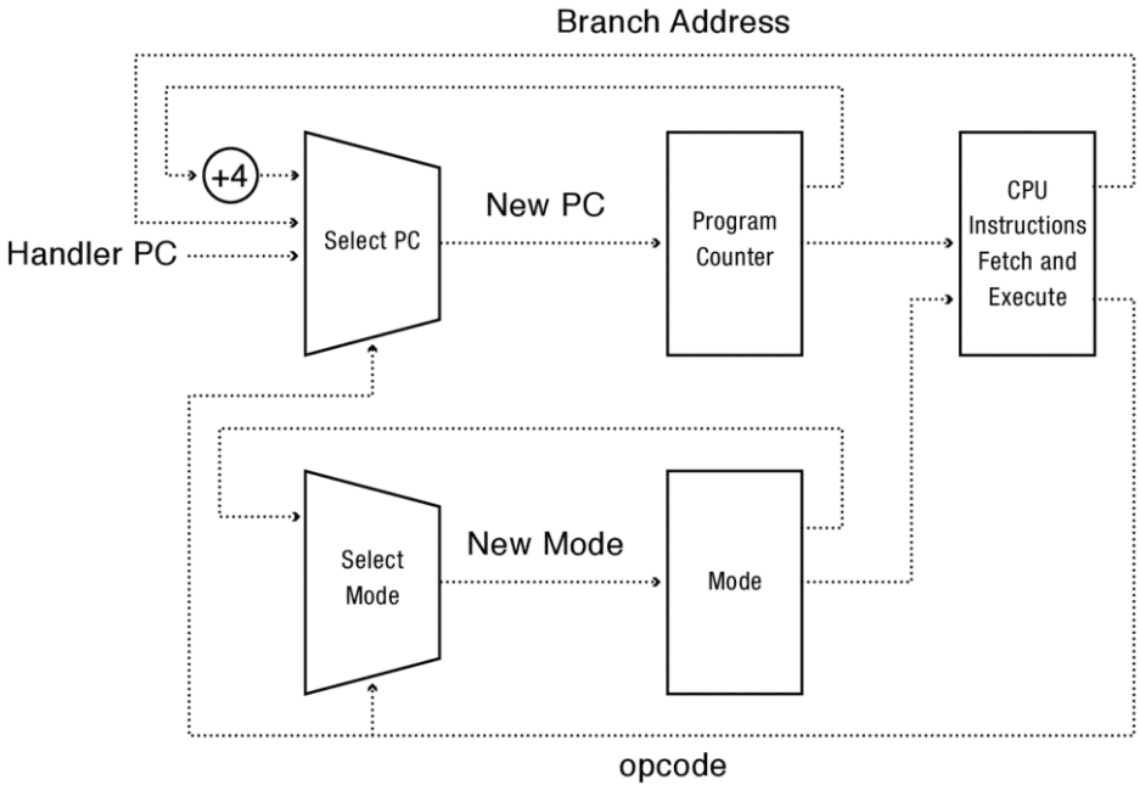
Privileged instructions & dual-mode operation

A single bit in the processor status register that signifies the current mode of the processor. In **user mode**, the processor checks each instruction before executing it to verify that it is permitted to be performed by that process. In **kernel mode**, the operating system executes with protection checks turned off.

The basic operation of a CPU



The operation of a CPU with kernel and user modes



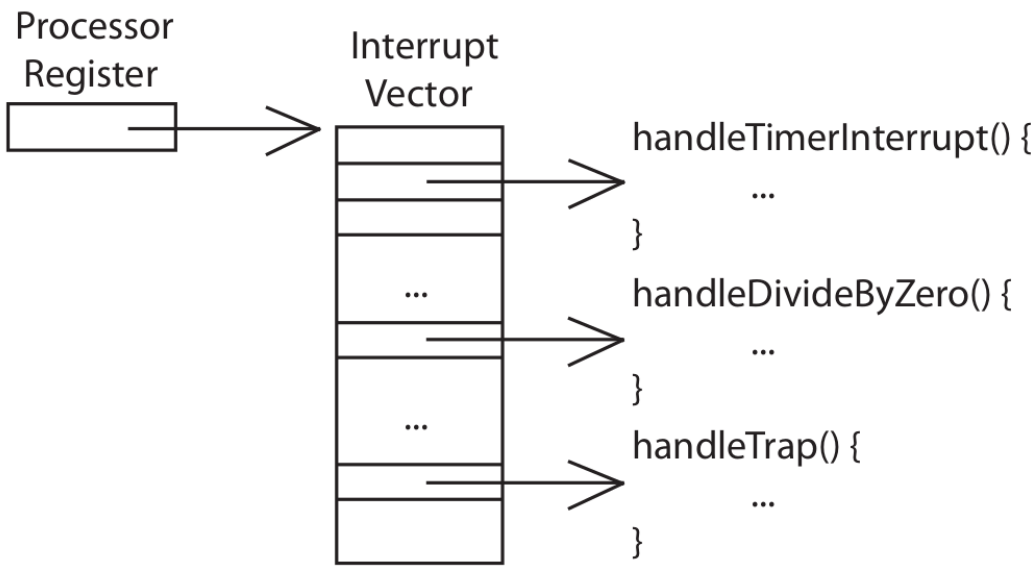
The kernel/user mode bit is one flag in the processor **status register**, set whenever the kernel is entered and reset whenever the kernel switches back to user mode.

- Instructions available in kernel mode, but not in user mode, are called privileged instructions.
- The operating system kernel must be able to execute these instructions to do its work
  - adjust memory access, and disable and enable interrupts, ...
- Thus, while application programs can use only a subset of the full instruction set, the operating system executes in kernel mode with the full power of the hardware.

Let me know your location

Interrupt vector

- To identify the code to run on an interrupt, exception or a system call, the processor will include a special register that points to an area of kernel memory called the interrupt vector.
- The interrupt vector is an array of pointers, pointing to the first instructions of different handler procedures in the kernel.



► OS at boot time informs the hardware of the locations of the interrupt vector, usually with some kind of special instruction. Once the hardware is informed, it remembers until the machine is next rebooted.

Aside: x86 interrupt vector

The format of the interrupt vector is processor-specific. The x86 allows for 256 different interrupts. Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses. Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt.

We can meet in a coffee-shop, if you like ...

Interrupt Masking

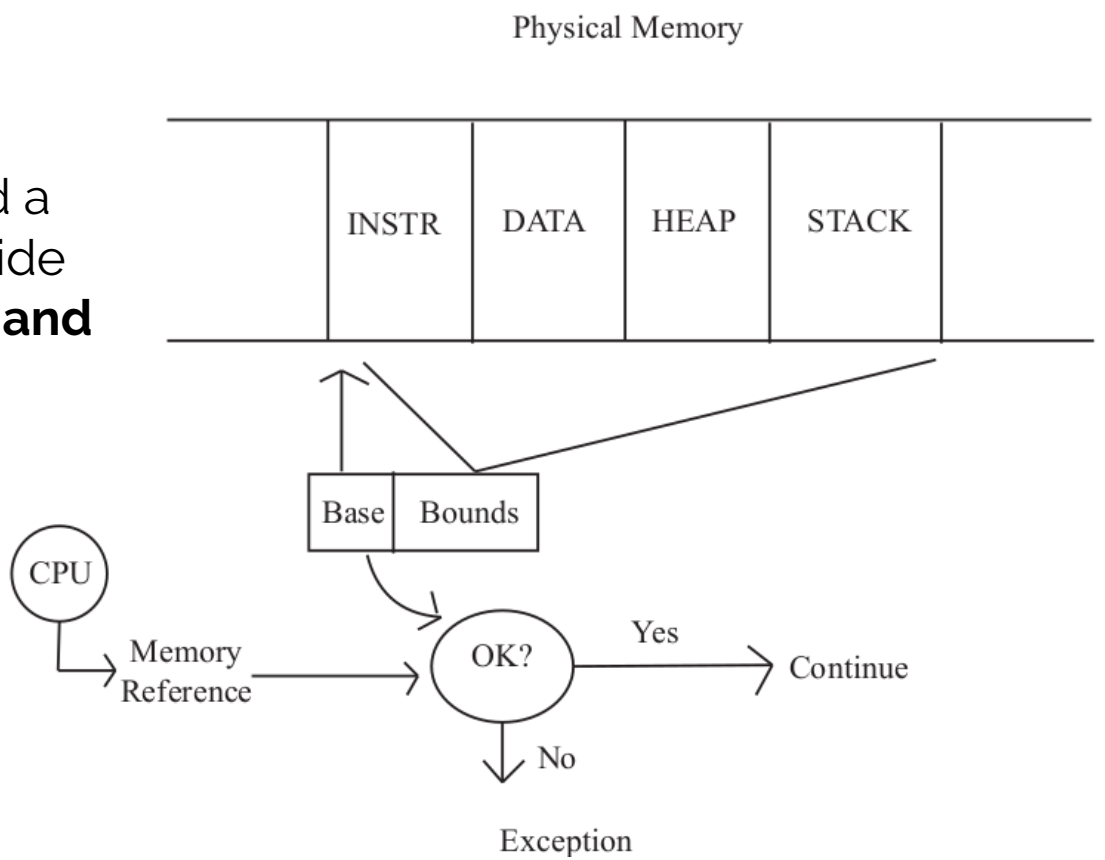
- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86:
    CLI: disable interrrupts
    STI: enable interrupts
- We'll need this to implement synchronization ...

I protect you ...

Memory Protection

All memory accesses outside of a process's valid memory region are prohibited when executing in user mode.

Early computers pioneered a simple mechanism to provide protection, known as **base and bounds**.



More elaborated methods in the next chapter ....

Always, I cleanup the room before you arrive ...

Atomic saving and restoring registers

- The interrupted process's registers must be saved before the handler changes any of them so that the process can be restarted exactly as it left off.
- This is tricky because we have to save the registers without changing them in the process.
- Hardware provides some support to help with this process.

Note

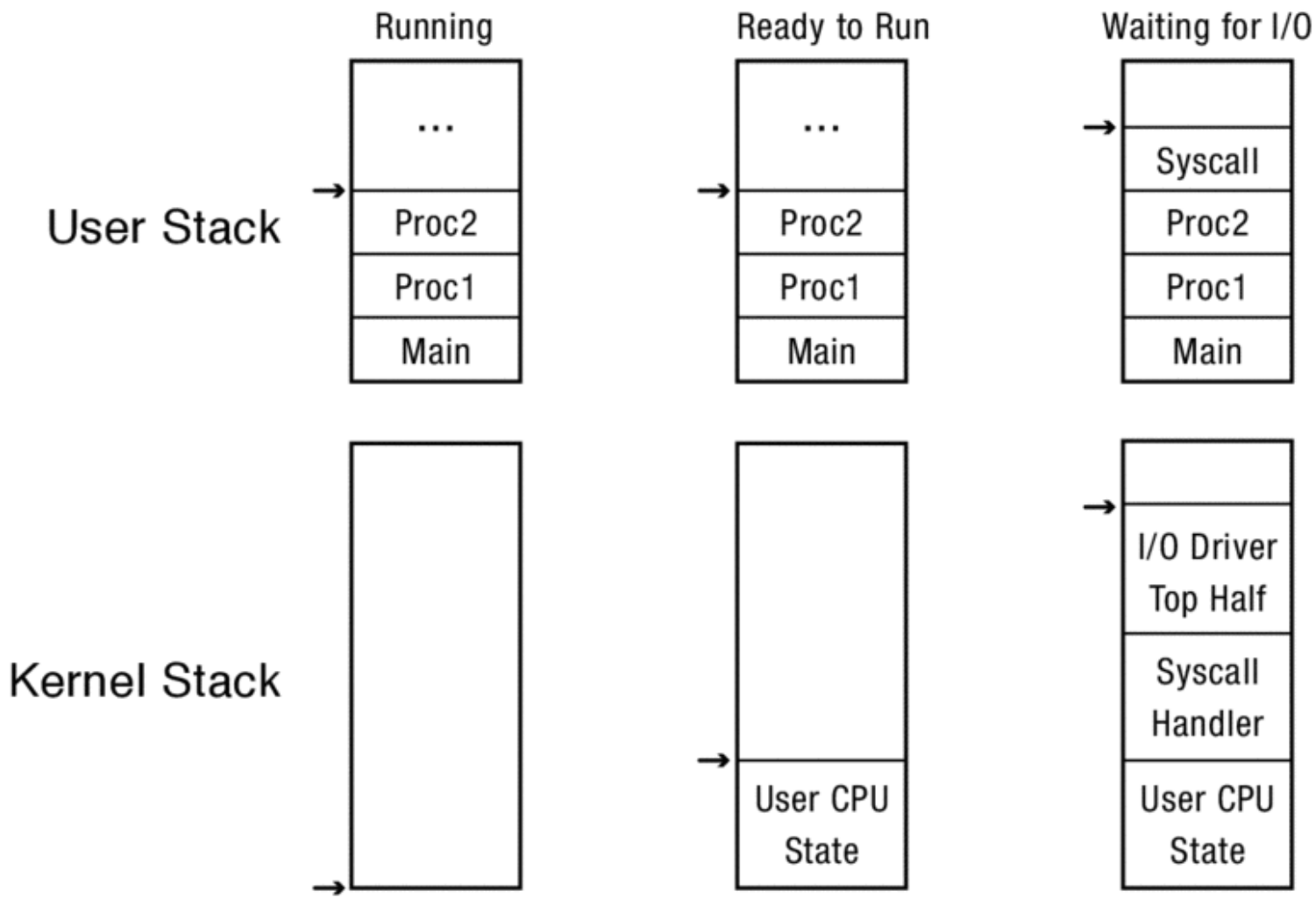
This hardware feature is needed because it is essential to save these before running the handler software: once the handler software starts running, the stack pointer, instruction pointer, and processor status word will be those of the handler, not those of the interrupted process.

I know you are very organized. So I put each item in its place where you told me before ...

Interrupt stack

- Most operating system kernels allocate a kernel interrupt stack for every user-level process (thread) inside kernel memory.
- The processor has a special, privileged register pointing to the currently running interrupt stack.
- When an interrupt, exception, or trap occurs, the hardware changes the stack pointer to point to the base of the kernel's interrupt stack.
- The hardware automatically saves some of the interrupted process's registers by pushing them onto the interrupt stack, before calling the kernel's handler.
- When the kernel handler runs, it can then push any remaining registers onto the stack before performing the work of the handler.

- When returning from the interrupt, exception or trap, the reverse occurs:
  - first we pop the registers saved by the handler, and then the hardware restores the registers it saved onto the interrupt stack, returning to the point where the process was interrupted.


Running
Ready to Run
Waiting for I/O
User Stack
...
Proc2
Proc1
Main
Syscall
Kernel Stack
User CPU State
I/O Driver Top Half
Syscall Handler

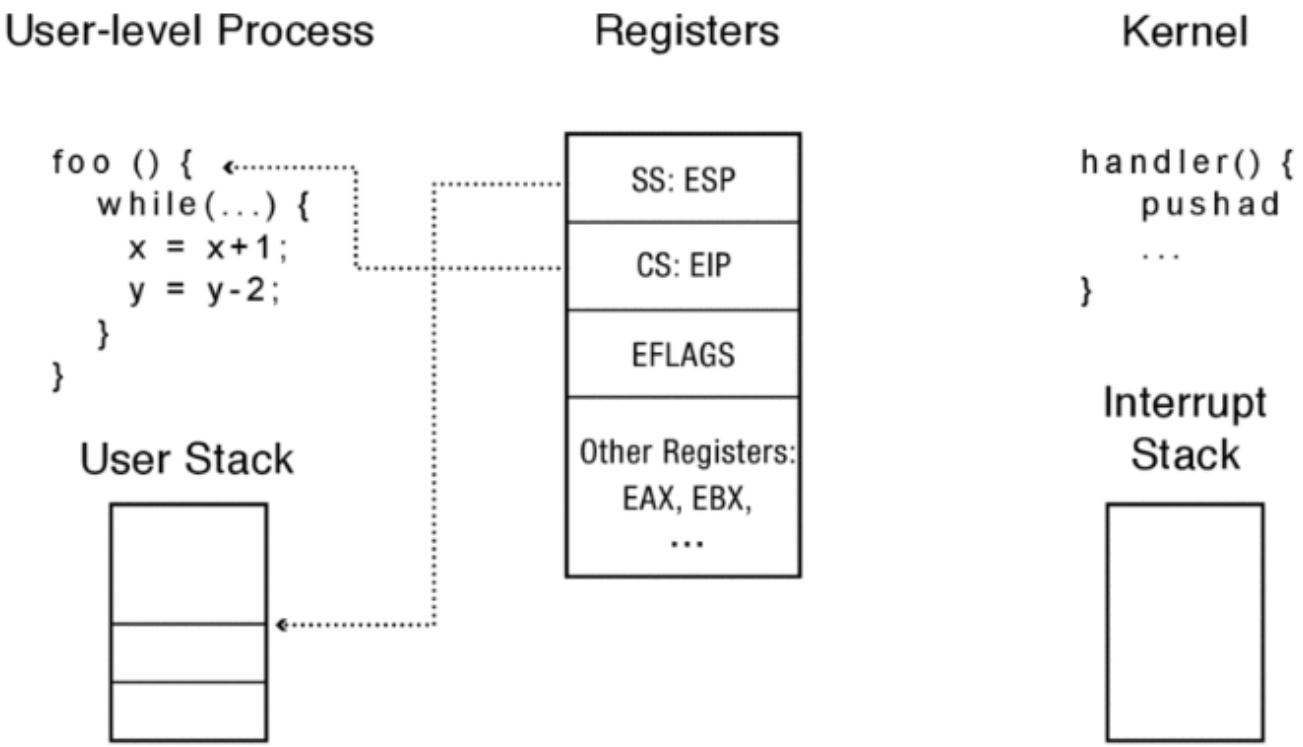
Aside: Interrupt handlers: top and bottom halves

In order to avoid disabling interrupts for a long time (which may result in missing interrupts), device drivers are divided into parts: **bottom half** (minimum necessary operations by hardware with disabled interrupt) and **top half** (remaining operations to be done by OS with enabled interrupts).
More info: Anderson's book

Example

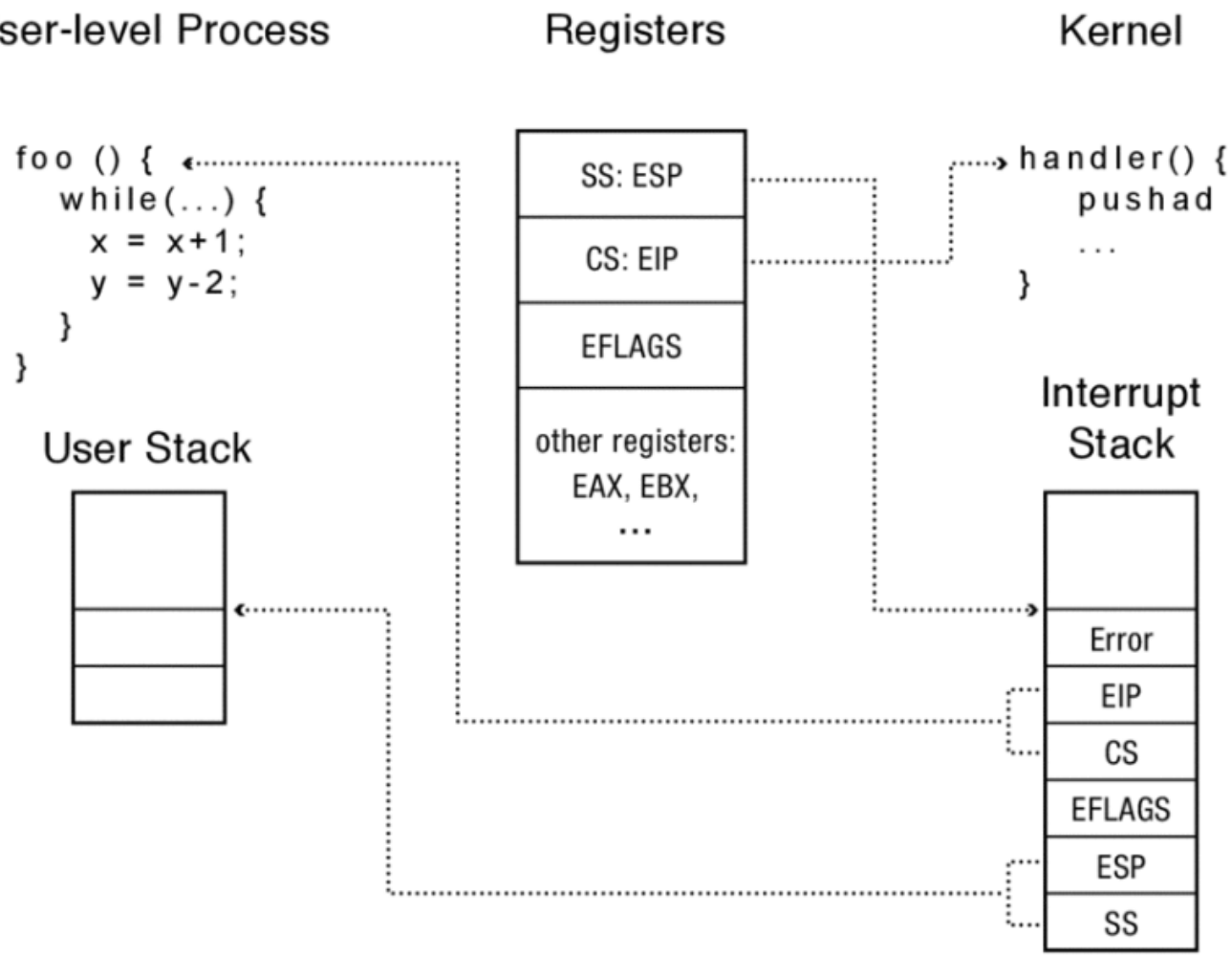
An interrupt-triggered mode switch on the x86 architecture

Before an interrupt



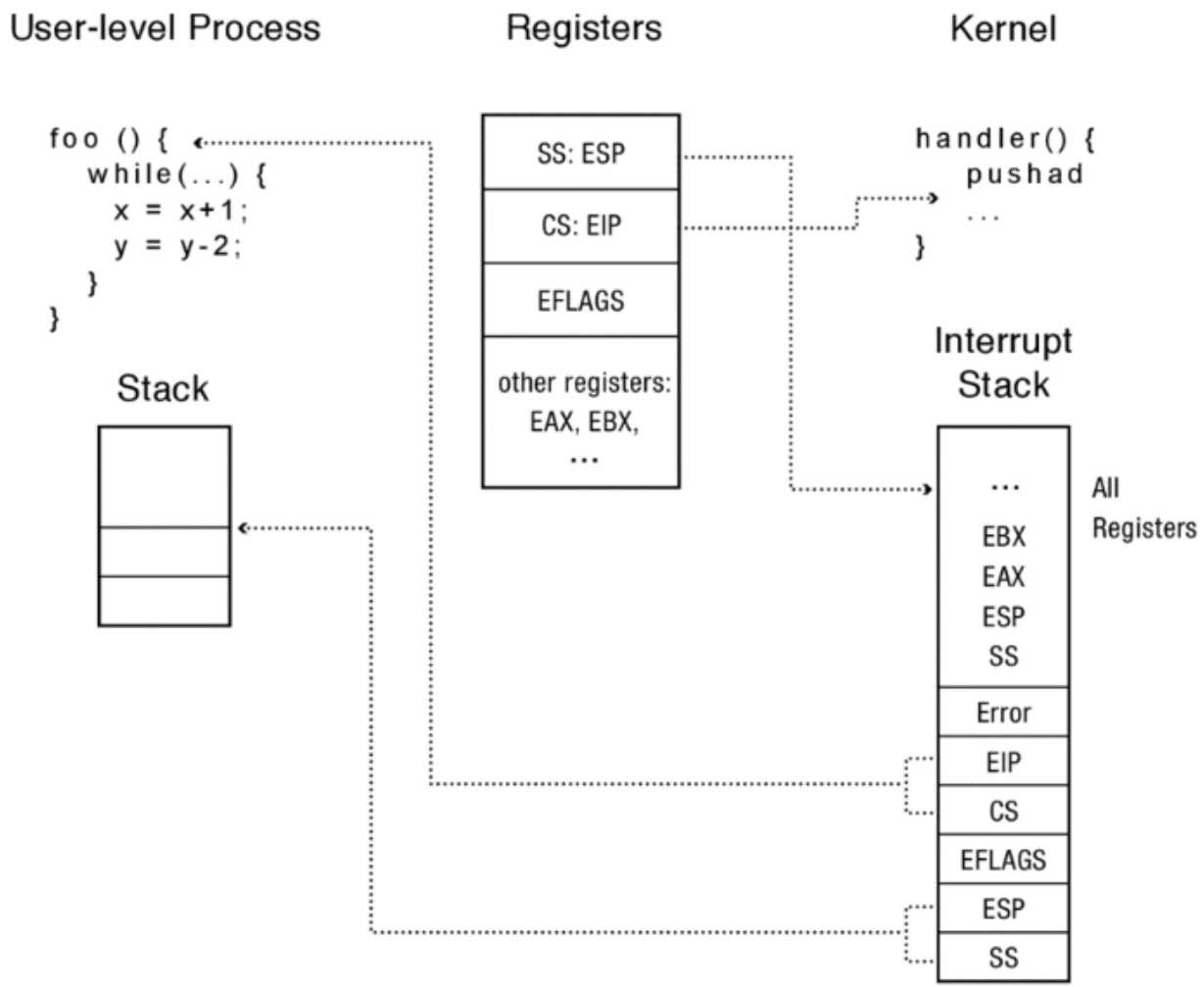
`SS` is the stack segment, `ESP` is the stack pointer, `CS` is the code segment, and `EIP` is the program counter. The program counter and stack pointer refer to locations in the user process, and the interrupt stack is empty.

After jumping to the interrupt handler



The hardware saves the user context on the kernel interrupt stack and changes the program counter/stack to locations in kernel memory.

After the interrupt handler has started executing



The handler first saves the current state of the processor registers, since it may overwrite them.

1. **Mask interrupts.**
2. **Save three key values.** The hardware internally saves the value of the stack pointer (the x86 `esp` and `ss` registers), the execution flags (the x86 `eflags` register), and the instruction pointer (the x86 `eip` and `cs` registers).
3. **Switch onto the kernel exception stack.** The hardware then switches the stack pointer to the base of the kernel exception stack, specified in a special hardware register.
4. **Push the three key values onto the new stack.**
5. **Invoke the interrupt handler.** First, it can use the `pushad` instruction to save the remaining registers onto the stack.

After the end of handler

- Handler restores saved registers (`popad`)
- Atomically (with one instruction) return to interrupted process/ thread (`iret`)
  - Restore program counter
  - Restore program stack
  - Restore processor status word
  - Switch to user mode

Note

To avoid confusion and reduce the possibility of error, most operating systems have a common sequence of instructions for entering the kernel - whether due to interrupts, exceptions or system calls - and a common sequence of instructions for returning to user level, again regardless of the cause.

Please let me know how often I can meet you again
every minute,
every hour,
every day, ...

Timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
    [–] Interrupt frequency set by the kernel
    - Not by user code!

Class Question

At which sate is the k-stack of a process empty of CPU's registers?

A) Ready

B) Blocked

C) Running

D) Never

Class Question

At which sate is the k-stack of a process empty of CPU's registers?

A) Ready

B) Blocked

✓ Running (at user mode)

D) Never

System Call and CPU Virtualization

Direct execution (without any software gate) of user process on hardware is **fast**

OS	Program
Create entry for process list Allocate memory for program Load program into memory Set up stack with argc/argv Clear registers Execute call `main()`	
	Run `main()` Execute **return** from `main`
Free memory of process Remove from process list	

Yes, it is fast but with protection and CPU monopolization concerns.

THE CRUX:
HOW TO EFFICIENTLY VIRTUALIZE THE CPU WITH CONTROL
The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support will be required. The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

► Next we see how to combine hardware supports and OS software to address these problems.

System Call

For protection, we need to limit, but

What to limit?

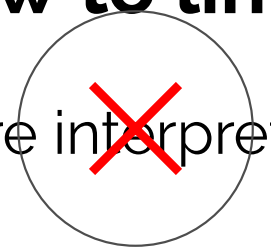
- General memory access
- Disk I/O
- Special x86 instructions like `lidt` !

How to limit?

Limited Execution: A software interpreter/gate (as a part of OS)

How to limit?

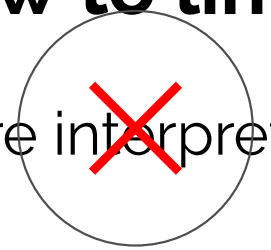
Limited Execution: A software interpreter/gate (as a part of OS)



slow

How to limit?

Limited Execution: A software interpreter/gate (as a part of OS)



slow

Limited Direct Execution (LDE): Get HW help, put processes in "user mode" ✔

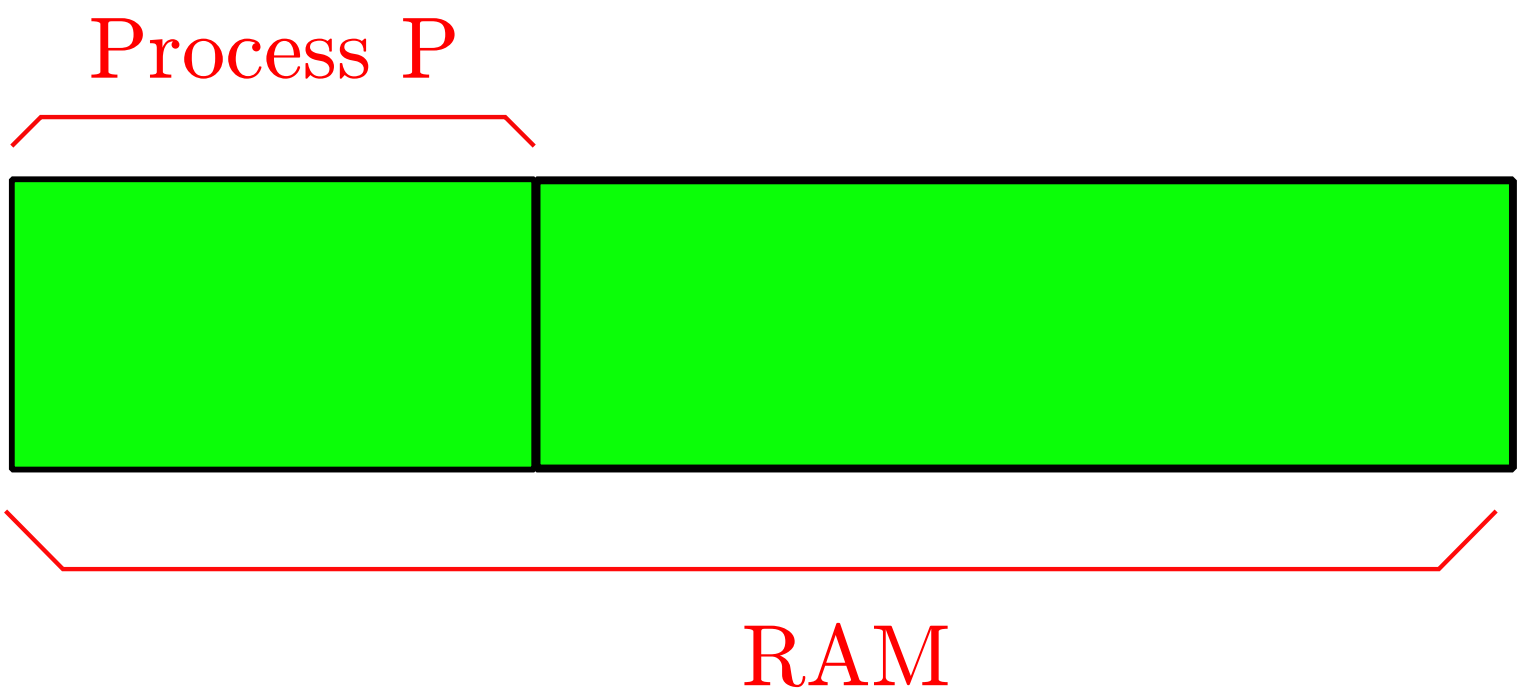
- Speed-up: hardware checks the instructions and allows safe (unprivileged) instructions and just pass dangerous (privileged) instructions to OS and change the mode to the kernel mode
- In the Kernel mode, all instructions are allowed
- LDE is also known as **dual mode**

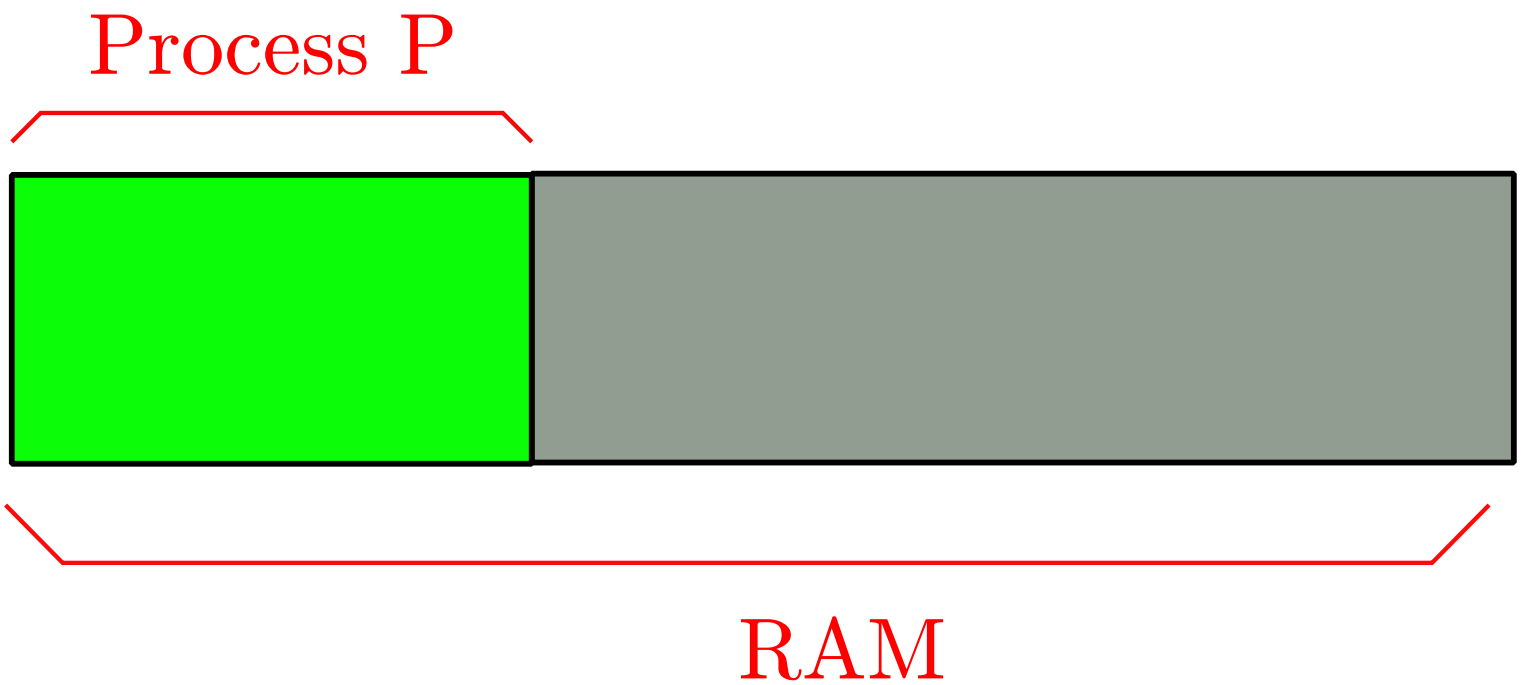
System call

- System calls provide the illusion that the operating system kernel is simply a set of library routines available for use by user programs.
- From a user point of view, system calls are like procedure calls.
  - In fact, system call is a procedure call, but hidden inside that procedure call is the famous **trap** instruction.
- But from implementation point of view System calls share much of the same mechanism for switching between user and kernel mode as interrupts and exceptions.
  - In fact, one frequently used Intel x86 instruction to trap into the kernel on a system call is called `int` , for “software interrupt."
  - one notable exception: the kernel should assume the parameters to each system call are intentionally designed to corrupt the kernel.

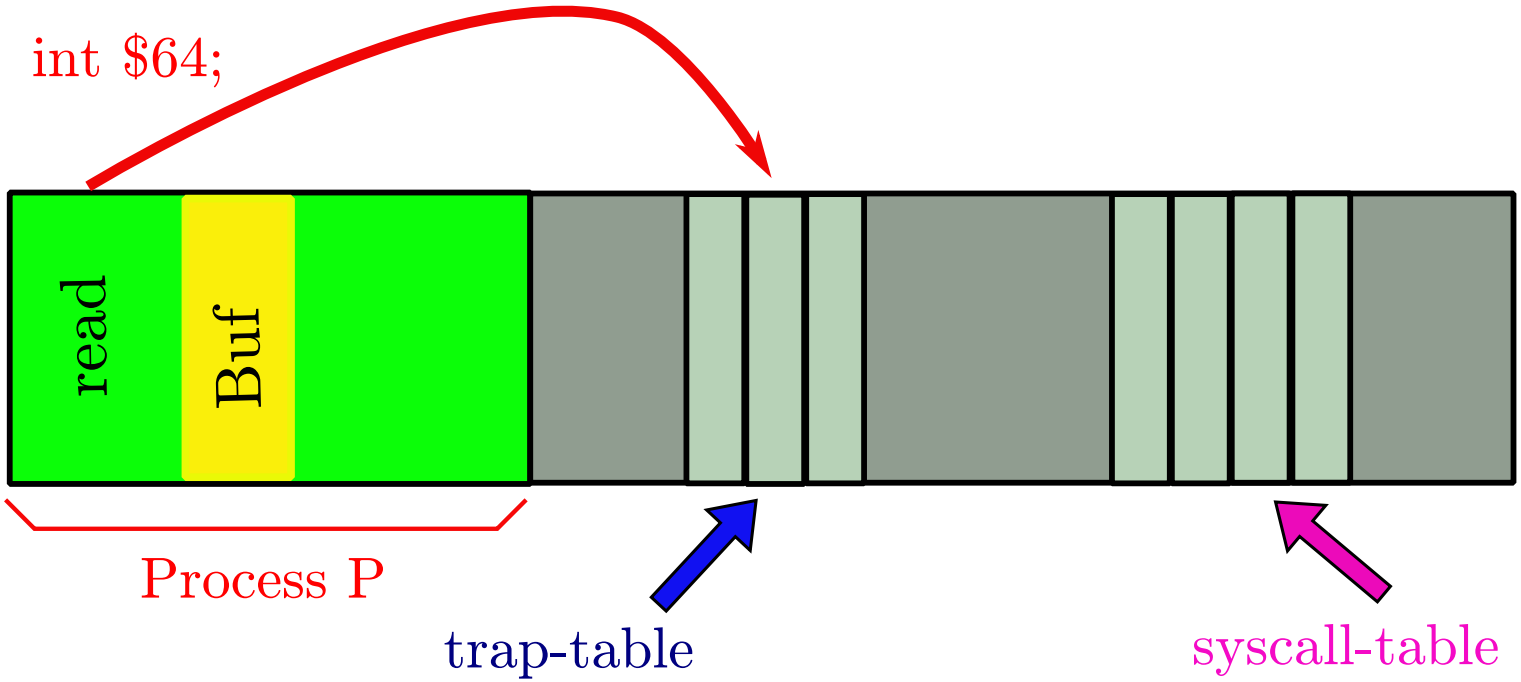
Example

read system call





P can only see its own memory because of user mode (other areas, including kernel, are hidden)



P wants to call read()

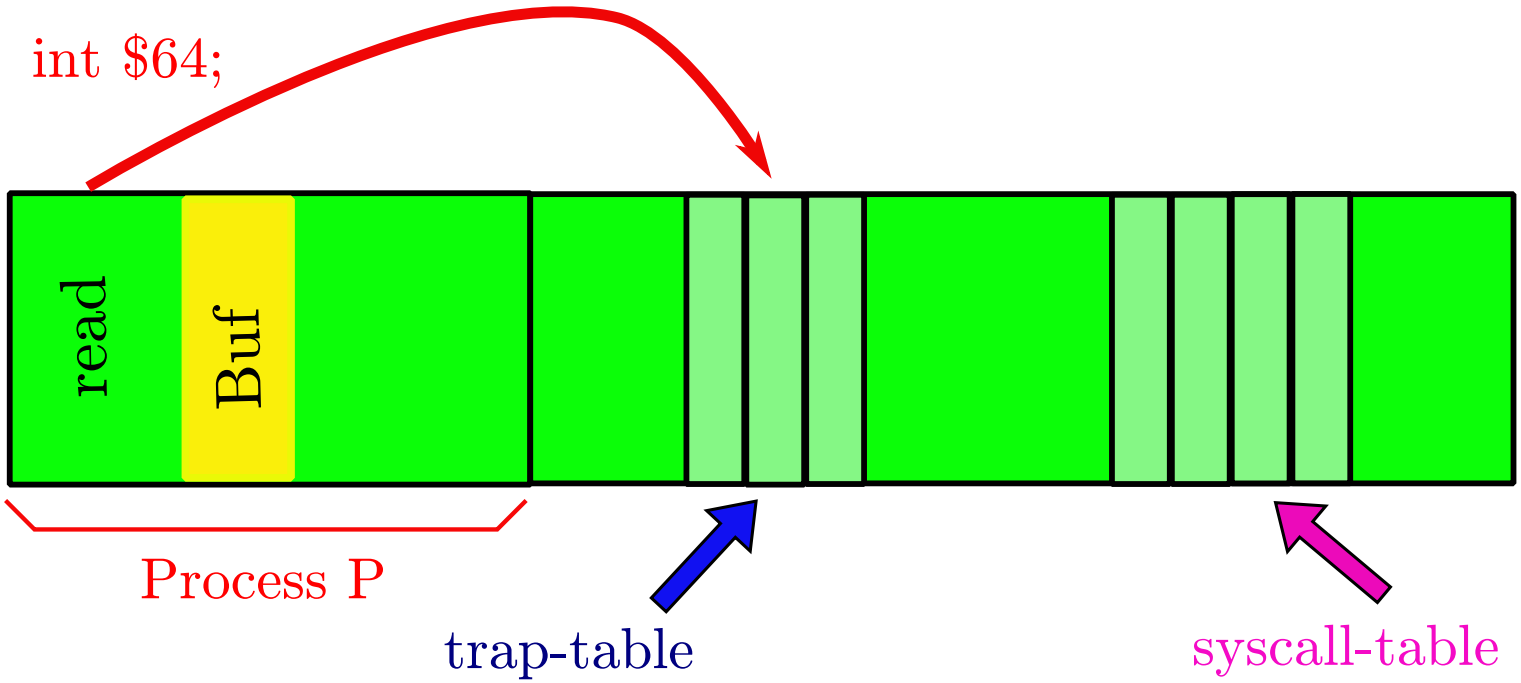
```
movl $6, %eax;
int $64;
```

syscall-table index

trap-table index

} inside read()

Mode switch → **Kernel mode:** we can do anything! (process execution in kernel mode)



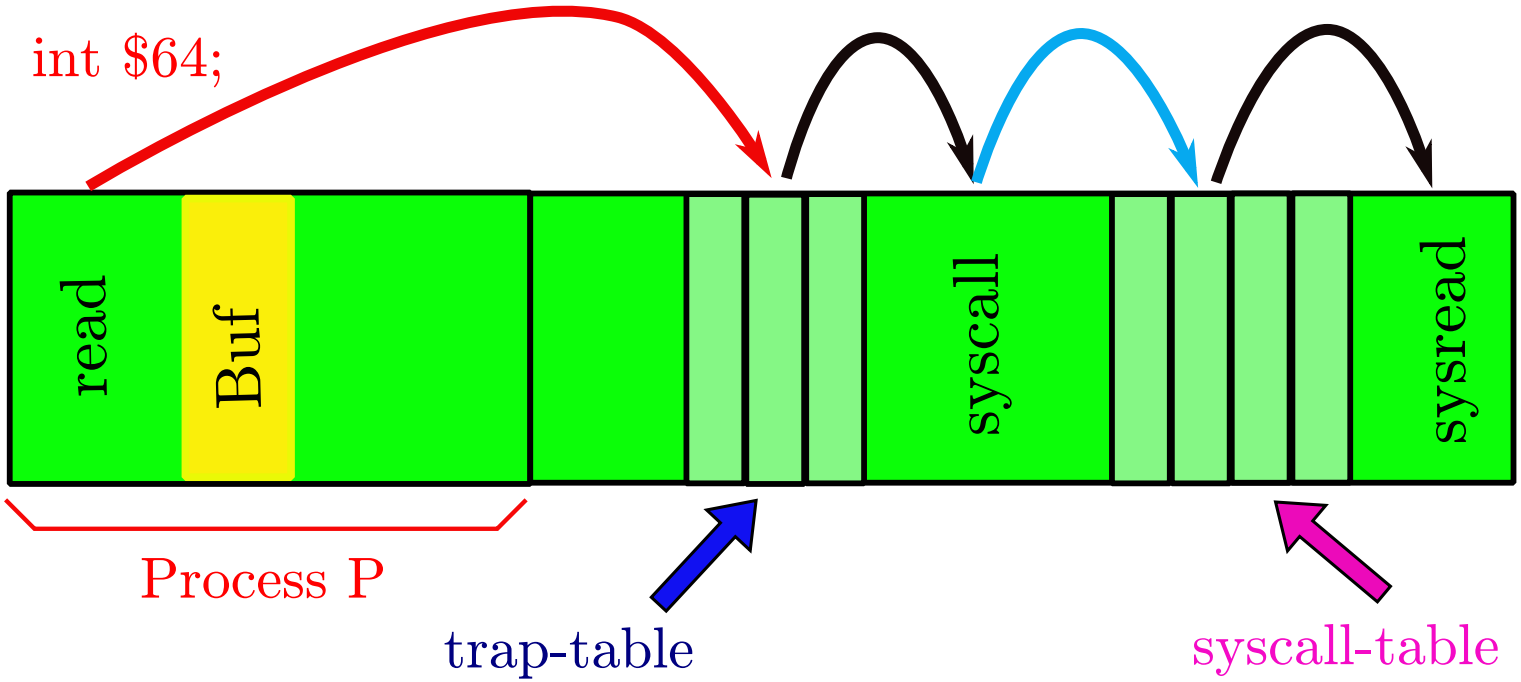
P wants to call read()

```
movl $6, %eax;
int $64;
```

syscall-table index

trap-table index

} inside read()



P wants to call read()

```
                    movl $6, %eax;
syscall-table index ↗
                    int $64;
trap-table index ↗
```

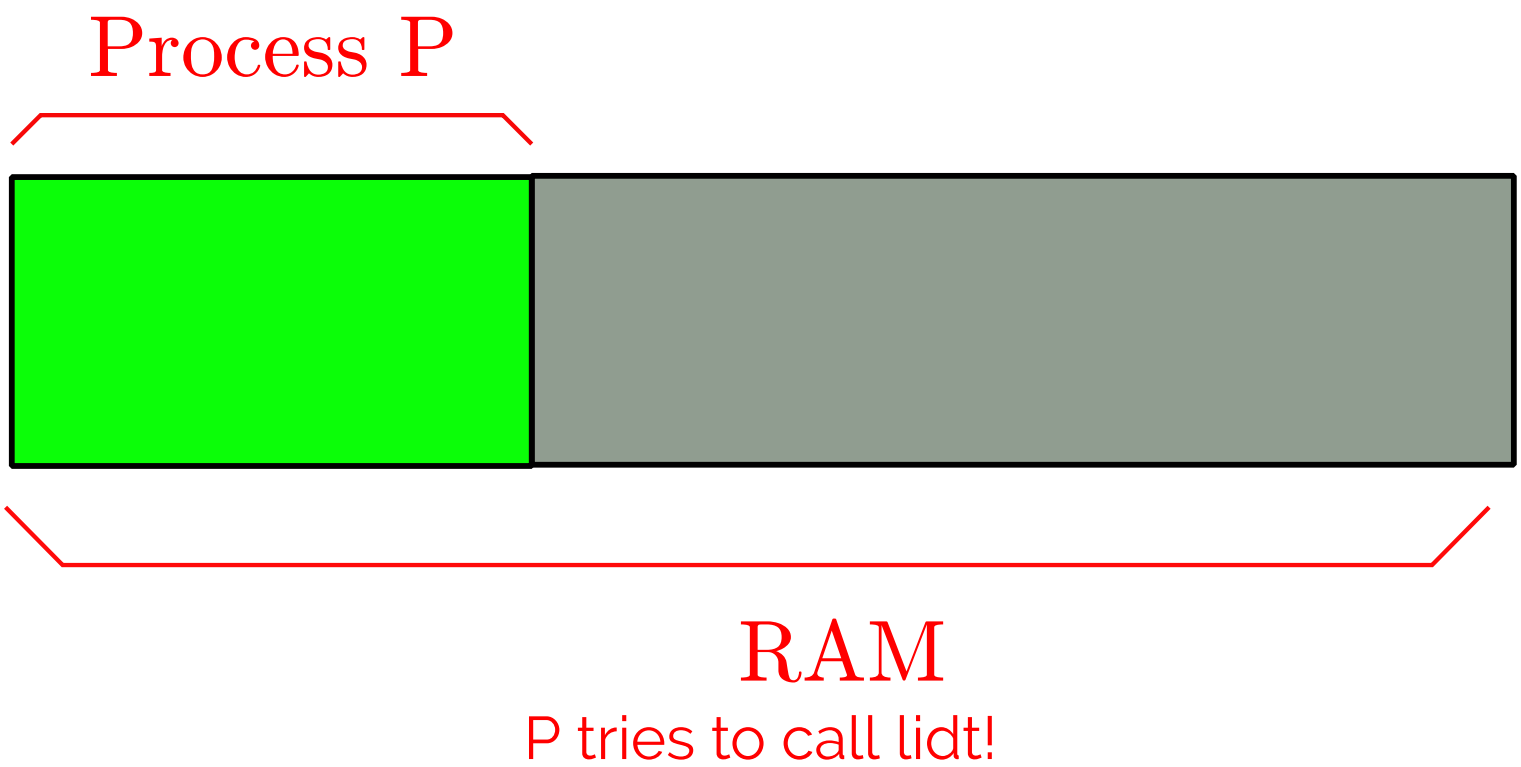
} inside read()

- The handler acts as a stub on the kernel side

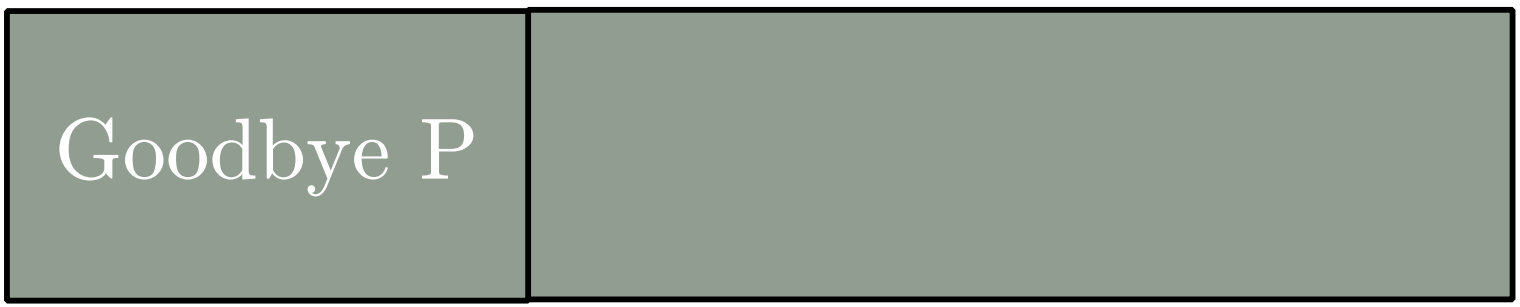
1. Locate arguments
   - In registers or on user stack
2. Copy arguments (before check!)
   - From user memory into kernel memory
3. Validate arguments
4. calling the implementation of the routine in the kernel.
5. Copy results back into user memory
   - If the system call reads data into a buffer in user memory, the stub needs to copy the data from the kernel buffer into

Example

lidt instruction in user mode



User mode → instructions are checked by CPU → A privileged instruction → Exception (switch mode to kernel, call the exception handler)



CPU warns OS, OS kills P

Limited Direct Execution Protocol

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	

Limited Direct Execution Protocol (continue)

		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

سوال ۲:

الف) تصور کنید شما می‌خواهید سیستم‌عاملی را بر روی یک پردازنده ایجاد کنید که تنها از وقفه‌ها (interrupt) و استثناها (exception) پشتیبانی می‌کند اما دستوری مرتبط با فراخوانی سیستمی (trap) ندارد (مثلا دستور int ندارد). آیا می‌توانید به روشی هوشمندانه توسط وقفه‌ها و یا استثناها، فراخوانی‌های سیستمی را نیز پیاده‌سازی نمایید؟

ب) تصور کنید شما می‌خواهید سیستم‌عاملی را بر روی یک پردازنده ایجاد کنید که تنها از فراخوانی‌های سیستمی و استثناها پشتیبانی می‌کند اما از وقفه پشتیبانی نمی‌کند. آیا می‌توانید به روشی هوشمندانه توسط فراخوانی‌های سیستمی و یا استثناها، وقفه‌ها را نیز پیاده‌سازی نمایید؟

سوال ۱ مقایسه فراخوانی سیستمی (system call) با فراخوانی تابع (procedure call) :

الف) دستور int و call را با هم مقایسه کنید.

ب) دستور iret و ret را با هم مقایسه کنید.

ج) نحوه مدیریت stack در فراخوانی سیستمی و فراخوانی تابع چه تفاوتی با هم دارند؟ در هر مورد مشخص کنید با چه مقادیری مقداردهی اولیه می‌شوند و توسط نرم‌فزار و یا سخت‌افزار این کار صورت می‌پذیرد.

د) برنامه زیر را اجرا کنید و خروجی آن را تحلیل نمایید.

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#define LOOP 10000
void proc(void)
{
  return;
}
int main(int argc, char const *argv[])
{
  clock_t start, end;
  double cpu_time_function, cpu_time_syscall;

  start = clock();
  for (int i = 0; i <LOOP; ++i)
    proc();
  end = clock();
  cpu_time_function = ((double) (end - start)) / CLOCKS_PER_SEC;

  start = clock();
  for (int i = 0; i < LOOP; ++i)
    getppid();
  end = clock();
  cpu_time_syscall = ((double) (end - start)) / CLOCKS_PER_SEC;

  printf("Procedure call (%d times): %f sec\n",LOOP, cpu_time_function);
  printf("System call  (%d times): %f sec\n", LOOP, cpu_time_syscall);
}
```

CPU Virtualization

Do we have enough CPUs?

Linux commands:

```
ps -A | wc
```

```
top
```

```
cat /proc/cpuinfo | grep 'model name'
```

How do we share?

CPU?
Memory?
Disk?

How do we share?

CPU? (a: time sharing)

This section

Memory? (a: space sharing)

Disk? (a: space sharing)

Goal: processes should NOT even know they are sharing (each process will get its own virtual CPU)

Context Switch

Problem: when to switch process contexts?
LDE $\rightarrow$ OS can't run while process runs (without I/O or mistake)

How can the OS do anything while it's not running?
A: it can't

Solution: switch on interrupts. But which interrupt?

- Cooperative : `yield` system call *utopian!*
- Non-cooperative : timer interrupt

Limited Direct Execution Protocol (Timer Interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

» Keep documenting before leaving one job to another

Scheduler

Definitions

Definition

Workload: set of job descriptions (e.g., arrival time, length, cpu-bounded, I/O bounded)

Scheduler: algorithm that decides when each job runs (e.g., FIFO, SJF, RR, ...)

Metrics: measurement of scheduling quality

- Performance metrics (e.g., turnaround time, response time)
- Fairness (e.g. Jain's fairness index)

Scheduling Basics

Metric

Definition (Turnaround time)

Turnaround time: the time at which the job completes minus the time at which the job arrived in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Workload Assumptions

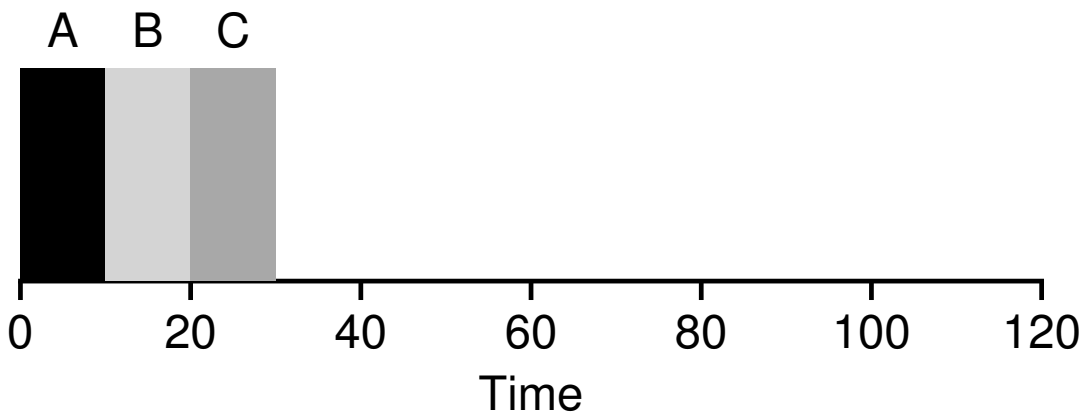
1. Each job runs for the same amount of time.
2. Jobs (processes) arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (no I/O).
5. The run-time of each job is known.

First In, First Out (FIFO)

- The most basic algorithm
- Also known as first come, first serve (FCFS)
- easy and performs well under our assumptions

Example

Three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} \approx 0$)



What will the average turnaround time be for these jobs?

$\frac{10+20+30}{3} = 20$

Workload Assumptions

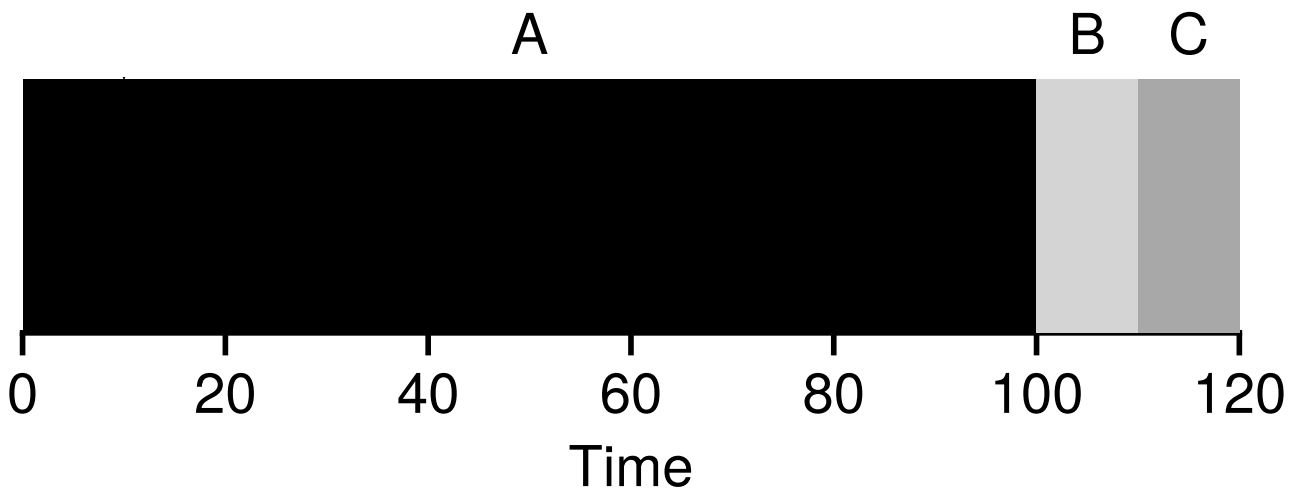
1. ~~Each job runs for the same amount of time.~~
2. Jobs (processes) arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (no I/O).
5. The run-time of each job is known.

Then, FIFO may perform poorly!

Convoy Effect

Example

Three jobs arrive in the system, A, B, and C, at roughly the same time ($T_{arrival} \approx 0$). But this time A runs for 100 seconds while B and C run for 10 each.



What will the average turnaround time be for these jobs?

$\frac{100+110+120}{3} = 110$

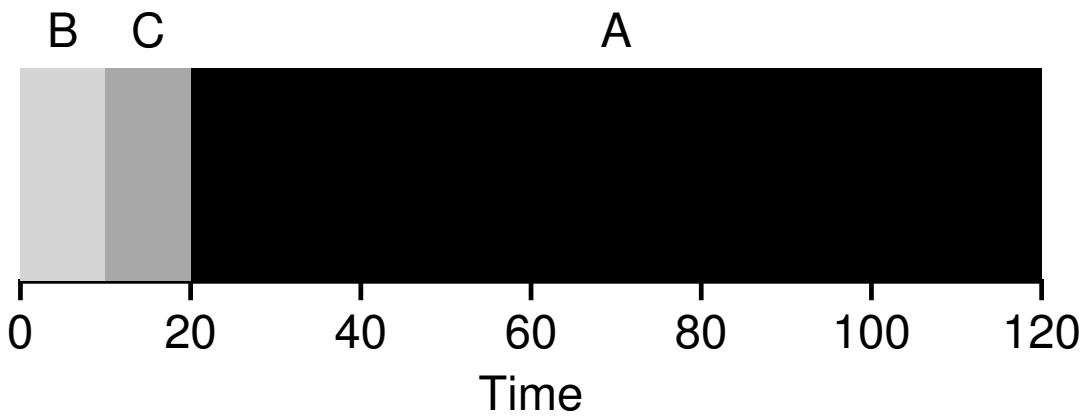
» FIFO performs poorly

Shortest Job First (SJF)

- Runs the shortest job first, then the next shortest, and so on.

Example

Last example but with SJF as our scheduling policy.



SJF reduces average turnaround from 110 seconds to 50 ($\frac{10+20+120}{3} = 50$)

- Given that all jobs arriving at the same time, SJF is optimal in minimizing average turnaround time.

» SJF principle: 1-10 item queues in super markets

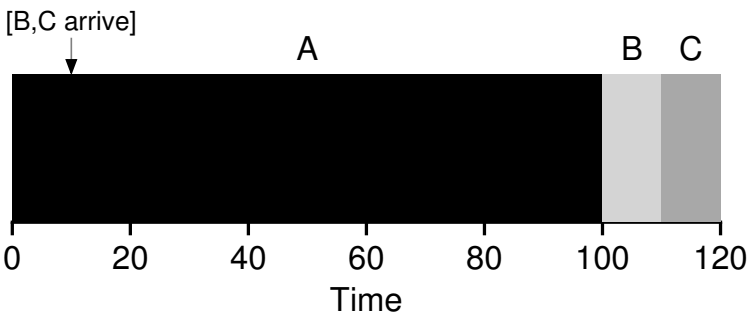
Workload Assumptions

1. ~~Each job runs for the same amount of time.~~
2. ~~Jobs (processes) arrive at the same time.~~
3. Once started, each job runs to completion.
4. All jobs only use the CPU (no I/O).
5. The run-time of each job is known.

Then, SJF may perform poorly! (Stuck Behind a Tractor Again)

Example

Assume A arrives at $t = 0$ and needs to run for 100 seconds, whereas B and C arrive at $t = 10$ and each need to run for 10 seconds



Average turnaround time for these three jobs is 103.33 seconds

Shortest Time-to-Completion First (SCTT)

- To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that.

Workload Assumptions

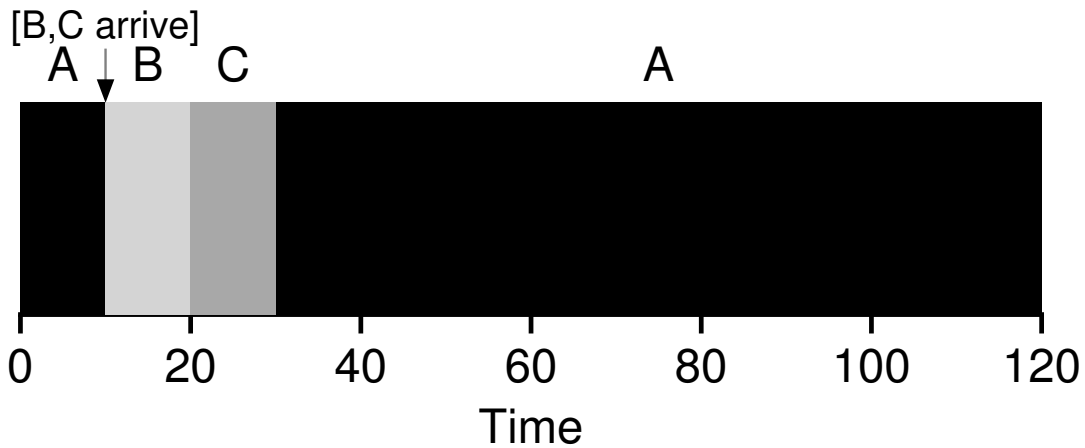
1. ~~Each job runs for the same amount of time.~~
2. ~~Jobs (processes) arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. All jobs only use the CPU (no I/O).
5. The run-time of each job is known.

- We also need some machinery within the scheduler itself: the ability to **preempt** a job when new jobs arrive and decide to run another job.

There is a scheduler which does exactly that: add preemption to SJF, known as the **Shortest Time-to-Completion First(STCF)** or **Preemptive Shortest Job First(PSJF)** scheduler.

- Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.

Example



Average turnaround time: 50 seconds ($\frac{(120-0)+(20-10)+(30-10)}{3}$)

- Given that jobs arriving at different time, SCTF is optimal in minimizing turnaround.

New Metric: Response Time

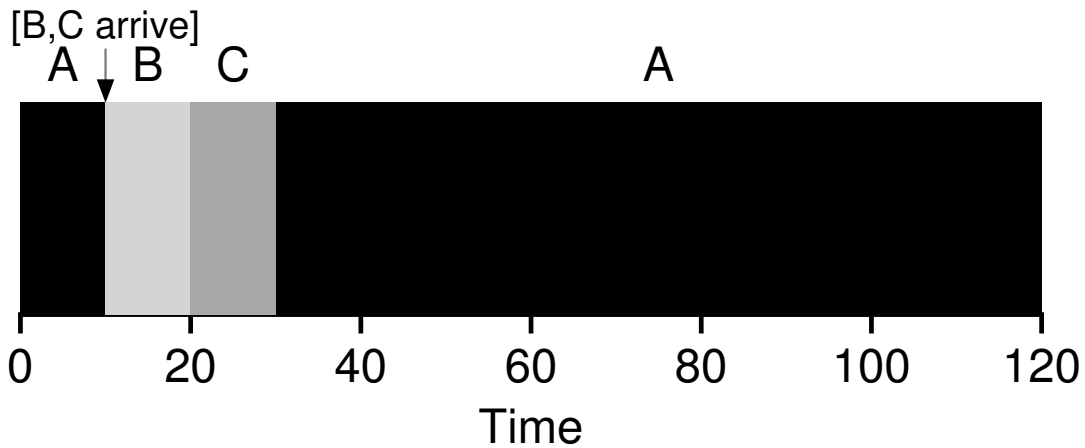
- SJF/SCTF are great in early **batch computing systems**.
- However, the introduction of **time-shared machines** changed all that. Now users may demand interactive performance from the system as well. And thus, a new metric was born: response time.

Definition (Response time)

Response time: the time from when the job arrives in a system to the first time it is scheduled.

$$T_{response} = T_{firstrun} - T_{arrival}$$

Example



Average response time: 3.3 seconds ($\frac{0+0+10}{3}$)

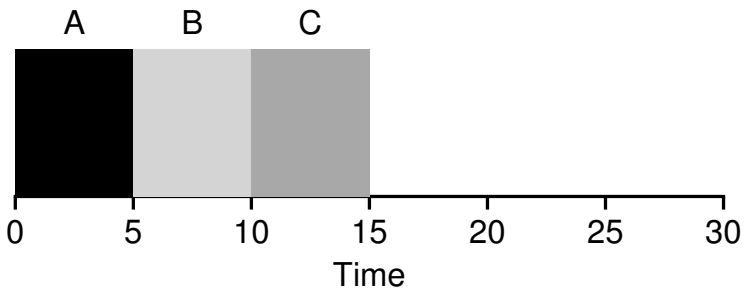
STCF and related disciplines are not particularly good for response time.

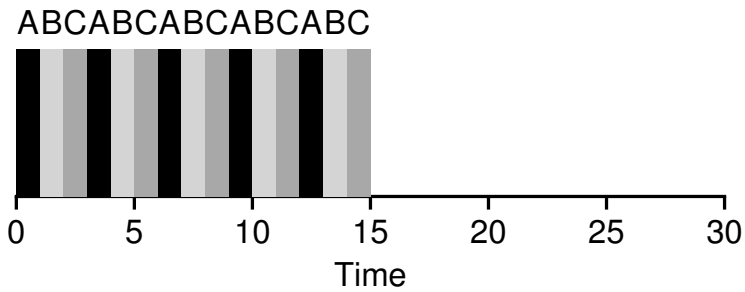
- If three jobs arrive at the same time,for example, the third job has to wait for the previous two jobs to run *in their entirety* before being scheduled just once.

Round Robin (RR)

- time slice (scheduling quantum): multiple of timer interrupt period.
- Every time slice, RR switches between jobs in the run queue.

Example





SJF average response time:
$\frac{0+5+10}{3} = 5$

RR average response time:
$\frac{0+1+2}{3} = 1$

- Time slice length: a trade-off
  - smaller, better for response time
  - longer, more amortizing the cost of switching

    costs of context switch: { saving registers; flushing CPU caches, ... }
- RR is good at response time but may be worse than FIFO in turnaround time.
  - In the last example: the turnaround time is 14 for RR and it is 10 for SJF.

» you can't have your cake and eat it too

سوال ۳: بهینه بودن زمانبند SJF را برای حالتی که کارهای محاسباتی صرف (cpu-bounded) تقریبا همزمان به سیستم وارد می‌شوند و مدت زمان مورد نیاز برای پردازش هر یک از آنها نیز معلوم است را اثبات نمایید.

سوال ۴:

الف) زمانبند چرخشی (RR) را در نظر بگیرید. فرض کنید اتلاف زمان برای تعویض محیط (context switch) ناچیز است. در شرایطی که کوانتوم زمانی این زمانبند اجازه اجرای تنها یک دستور از هر کار را بدهد، عملکرد این زمانبند به کدامیک از زمانبندهای FIFO و SJF نزدیک‌تر است؟ (فرض کنید کارها تقریبا همزمان به سیستم وارد می‌شوند)

ب) به سوال بخش الف با فرض اینکه کوانتوم زمانی به اندازه کافی بزرگ است که قبل از به اتمام رسیدن آن هر کاری که نوبت آن بوده پایان یابد پاسخ بدهید.

Looking backward and forward

We have developed two types of schedulers. The first type (SJF, STCF)optimizes turnaround time, but is bad for response time. The second type(RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed!

Workload Assumptions

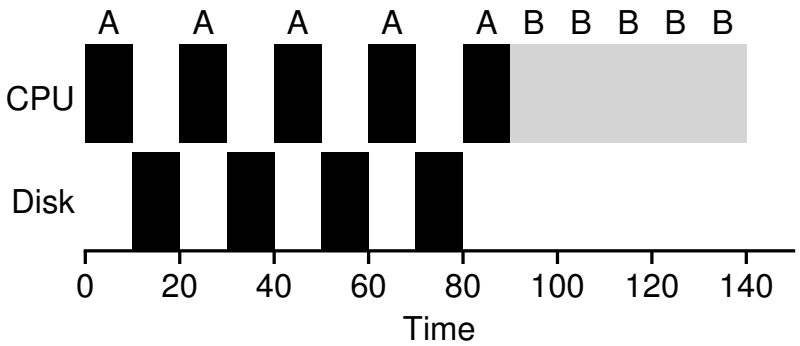
1. ~~Each job runs for the same amount of time.~~
2. ~~Jobs (processes) arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. ~~All jobs only use the CPU (no I/O).~~
5. The run-time of each job is known.

The idea of incorporating I/O is to overlap I/O time of one process with the runtime of another process.
Considering one of the scheduling methods, the I/O is incorporated by

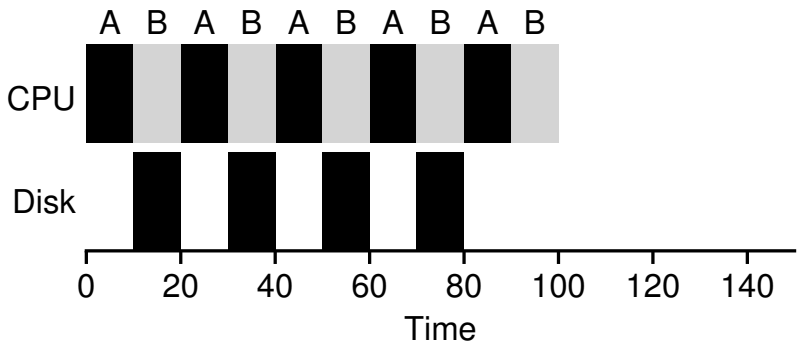
- At the start of an I/O: blocking the process, re-scheduling the CPU among the ready-to-run process
- After I/O completes (in the interrupt handler): moving the process that issued the I/O from blocked back to the ready state and run the scheduler
- A common approach is to treat each sub-job of an I/O bounded job as an independent job.

Example

STCF with two jobs, A and B, each need 50 ms of CPU time. But A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O.



No overlap: Poor Use Of Resources



Overlap Allows Better Use Of Resources

Workload Assumptions

1. ~~Each job runs for the same amount of time.~~
2. ~~Jobs (processes) arrive at the same time.~~
3. ~~Once started, each job runs to completion.~~
4. ~~All jobs only use the CPU (no I/O).~~
5. The run-time of each job is known.

Multi-Level Feedback Queue (MLFQ)

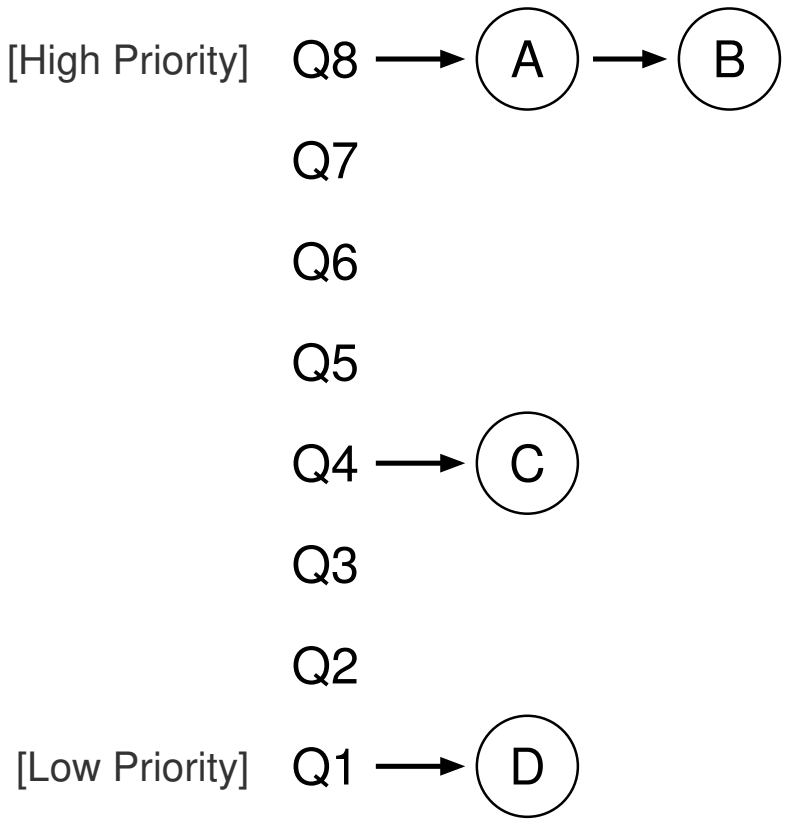
Multi-level Feed-back Queue (MLFQ)

- One of the most well-known scheduling algorithm: Linux, MAC, Windows
- First described by Corbato et al. in 1962 in CTSS system.

THE CRUX:
HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?
How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without *a priori* knowledge of job length?

MLFQ: Basic Rules

MLFQ has a number of distinct **queues**, each assigned a different **priority level**.



Rule #1 If Priority(A) $>$ Priority(B), A runs (B doesn't).

Rule #2 If Priority(A)=Priority(B), A & B run in RR.

MLFQ varies the priority of a job based on its observed behavior

Rule #3 When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule #4a If a job uses up an entire time slice while running, its priority is reduced(i.e., it moves down one queue).

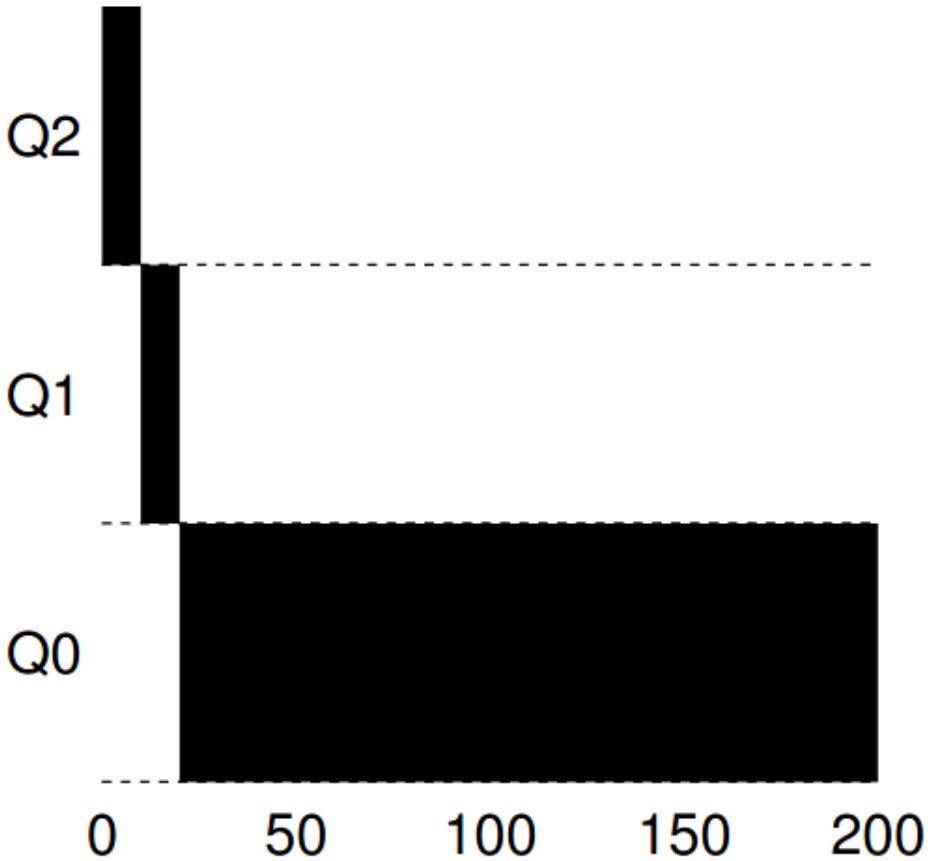
Rule #4b If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this way, MLFQ will try to **learn** about processes as they run, and thus **use the history of the job to predict its future behavior**.

» Learn from the past to predict the future

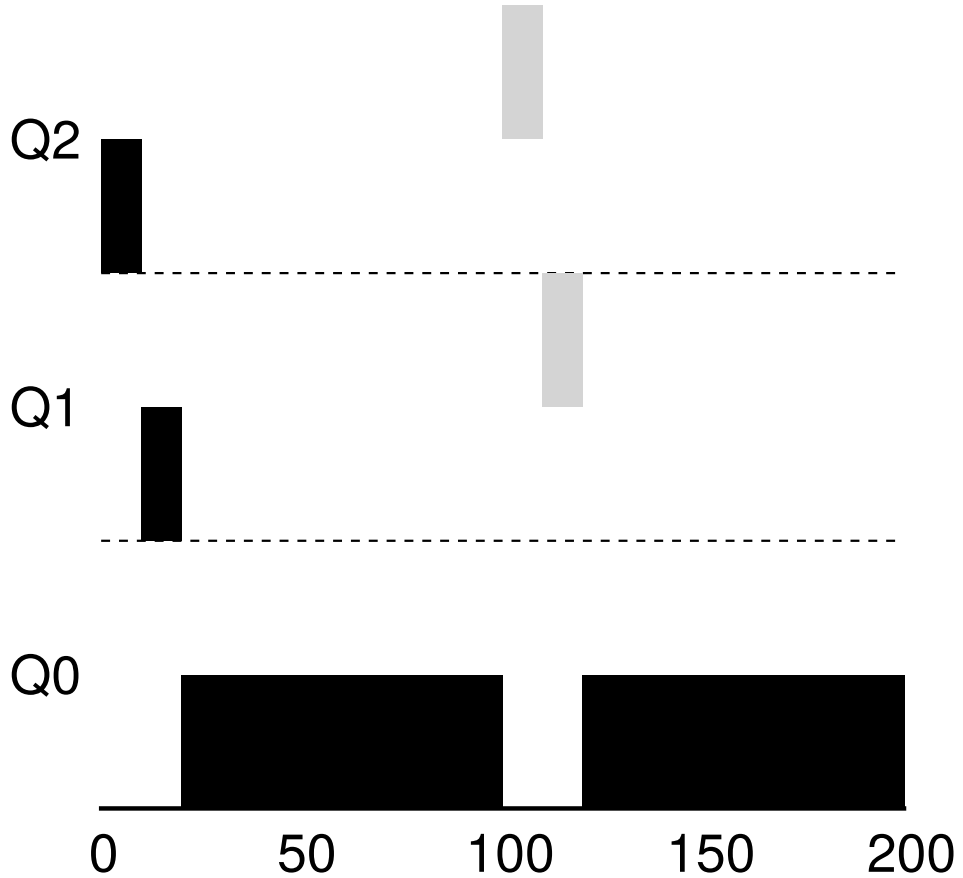
Example

A Single Long-Running Job



Example

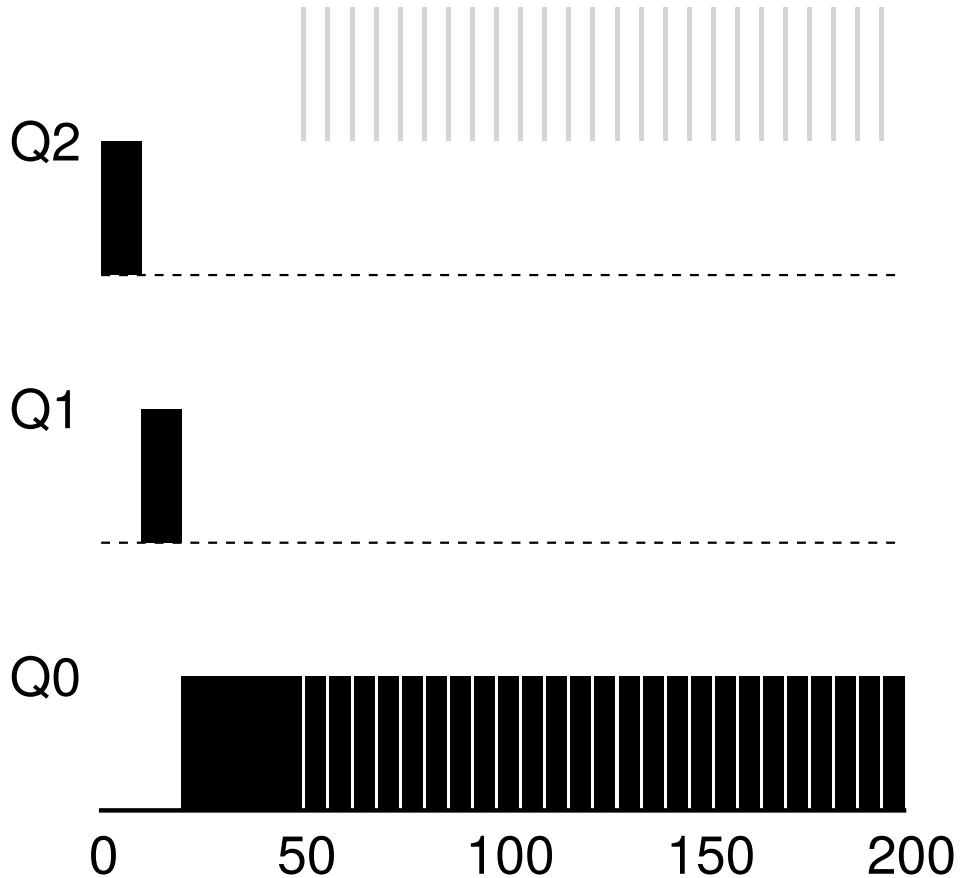
Along Came A Short Job



See how MLFQ tries to approximate SJF!

Example

What About I/O?

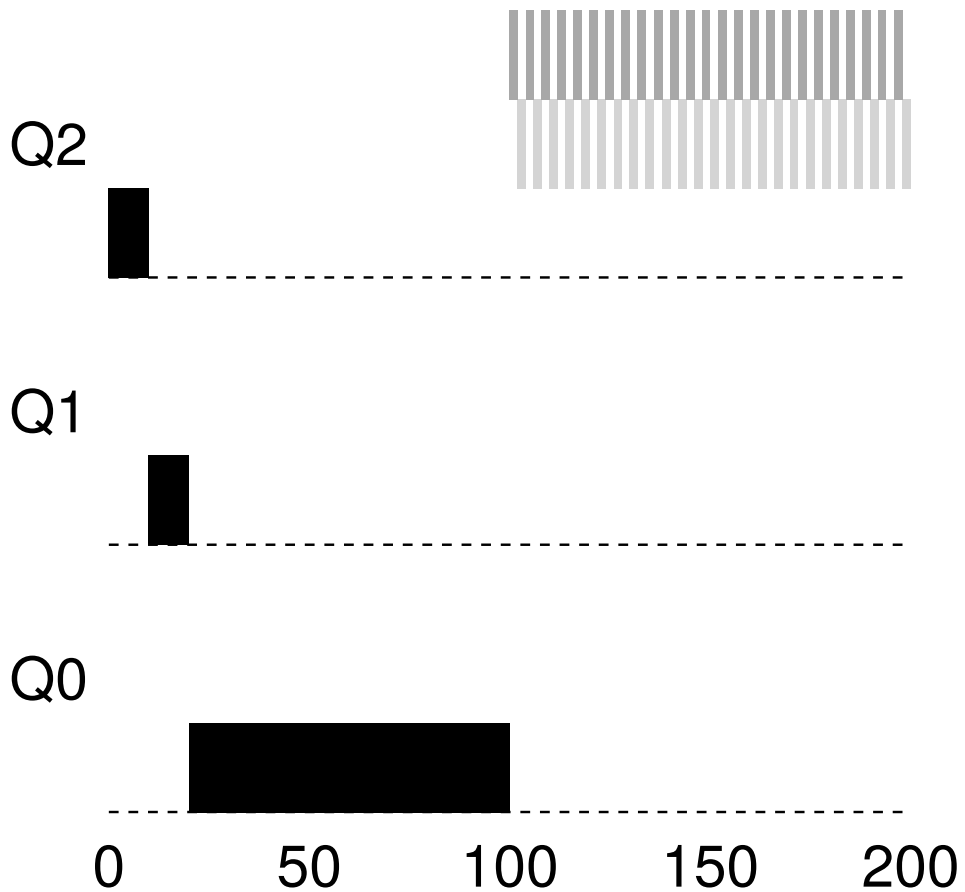


Problems With Our Current MLFQ:

- Starvation
- Gaming the scheduler
- A program may change its behavior over time

Starvation: if there are “too many" interactive jobs in the system, they will combine to consume all CPU time,and thus long-running jobs will never receive any CPU time (they starve).

Example

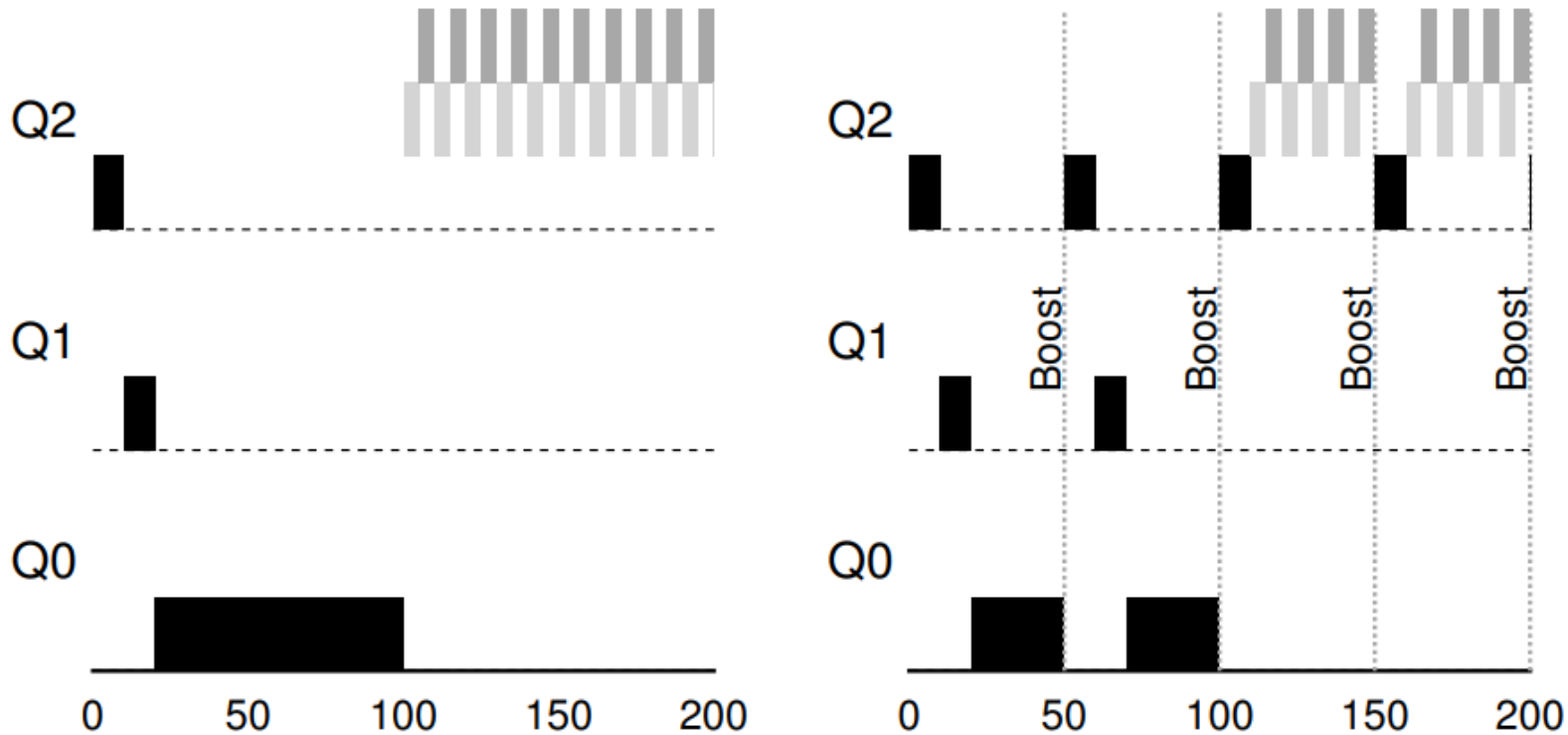


Rule #5 (Priority Boost) After some time period S, move all the jobs in the system to the topmost queue

This new rule solves two problems at once.

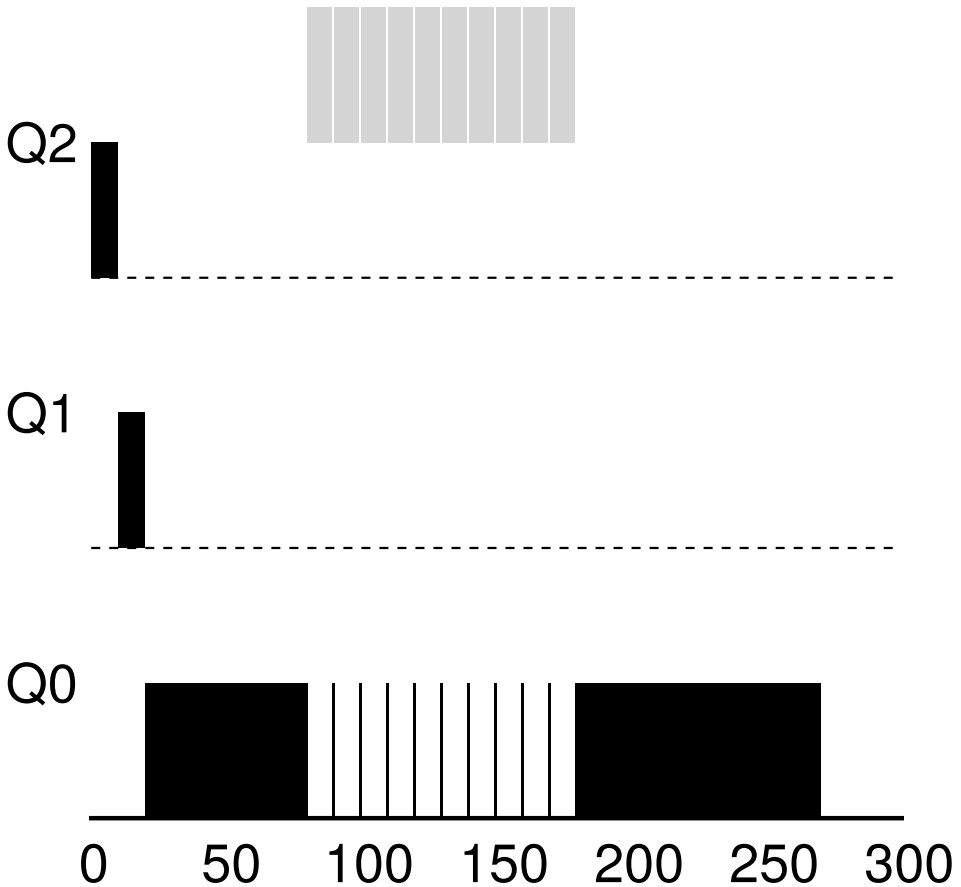
- Starvation
- behavioral change

Example


Q2
Q1
Q0
0
50
100
150
200
Q2
Q1
Q0
Boost
Boost
Boost
Boost
0
50
100
150
200

Gaming (attacking) the scheduler: before the time slice is over, issue an I/O operation(to some file you don't care about) and thus relinquish the CPU ...

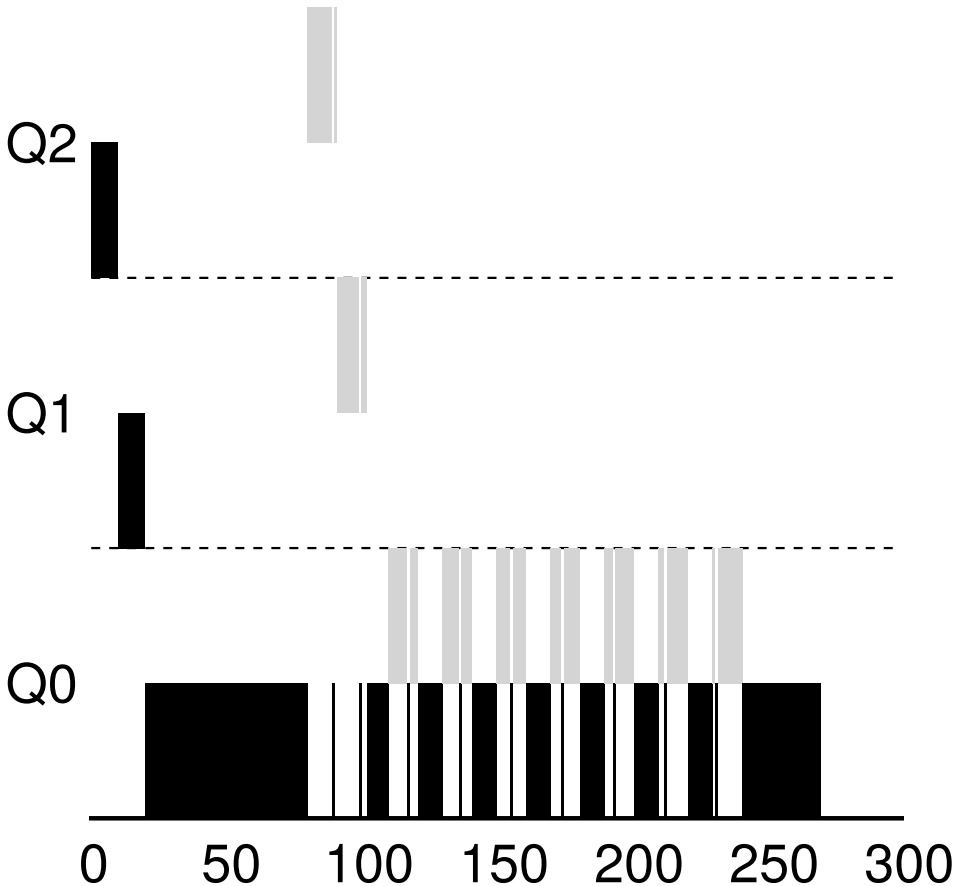
Example



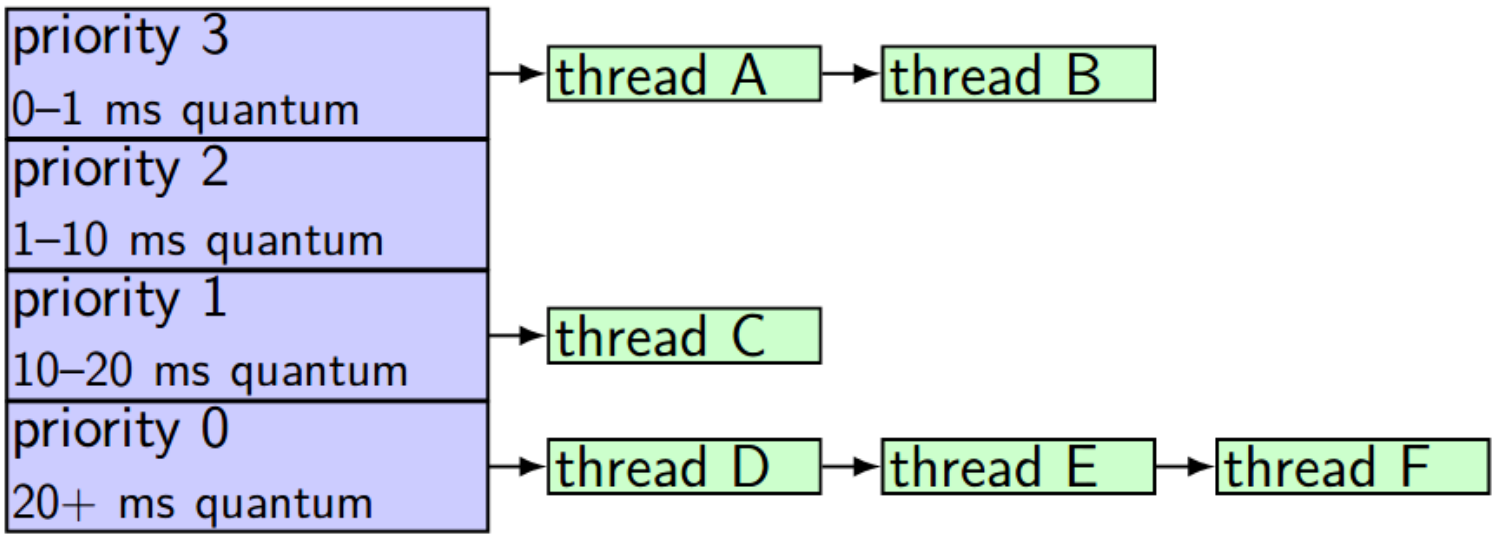
Rule #4 (anti-gaming) Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

A replacement of #4a & #4b!

Example

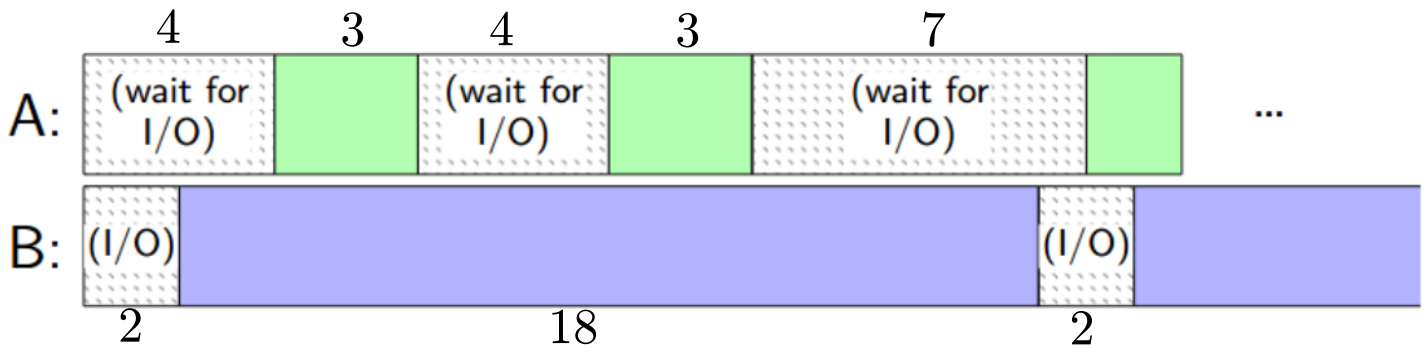


MLFQ usually has different time-slice lengths in different levels.



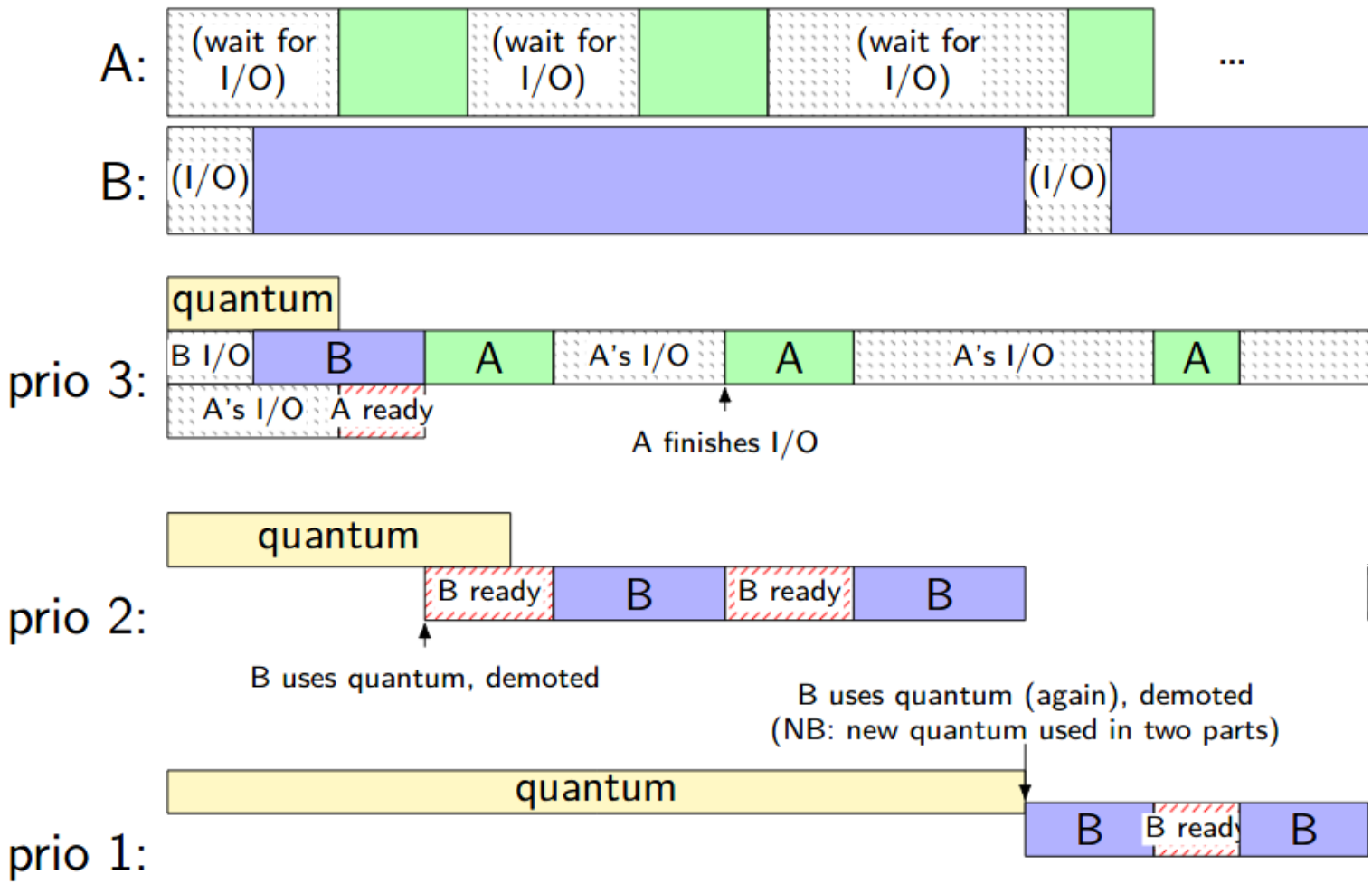
Example

(no anti-gaming is in place, 3 queues with quantum of 4,8,20)



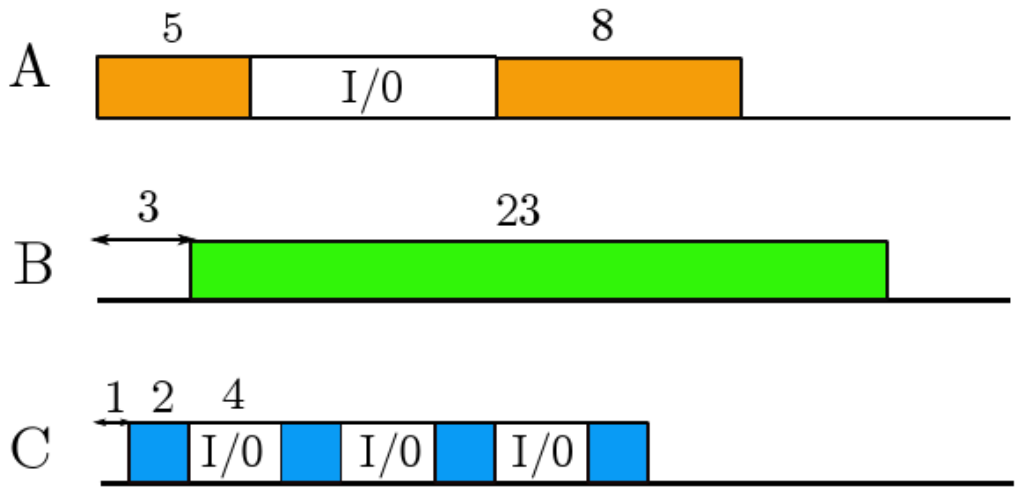
Example

(no anti-gaming is in place, 3 queues with quantum of 4,8,20)



سوال ۵:

حجم کار (workload) زیر را در نظر بگیرید.



ترتیب اجرای کارها را به ازای زمانبندهای زیر را بدست آورید و زمانبندها را با معیارهای زمان پاسخ و زمان چرخش با هم مقایسه کنید.

الف) SJF

ب) PSJF

ج) RR با کوانتوم زمانی ۲

د) MLFQ با سه سطح و کوانتوم‌های زمانی ۲، ۴ و ۸ (در هر سطح کارها به صورت RR ساده زمانبندی می‌شوند)

Lottery scheduling

Lottery Scheduling

Goal: proportional share

- » Guarantee that each job obtains a certain percentage of CPU time.
- » Not optimized for turnaround or response time

Approach:

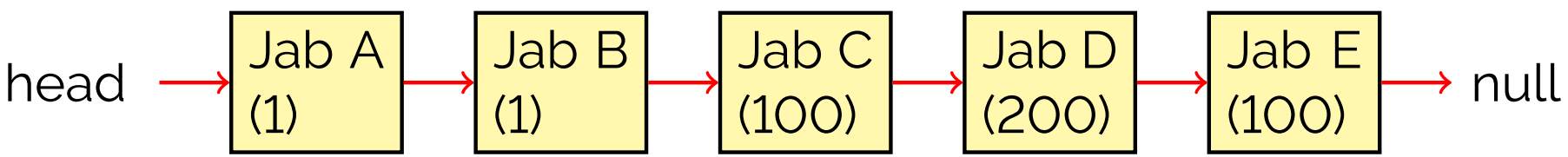
- give processes lottery tickets
- whoever wins runs
- higher priority $\Rightarrow$ more tickets

▶ A Random Algorithm

Lottery Code

```
int counter = 0;
int winner = getrandom(0, totaltickets);
node_t current = head;
while(current) {
 counter += current->tickets;
 if (counter > winner)
  break;
 current = current->next;
}
// current is the winner
```

Who runs if winner is:
50
350
0



Other Lottery Ideas

Ticket Transfers

Ticket Currencies

Ticket Inflation

(read more in OSTEP)

Completely Fair Scheduling

Completely Fair Scheduling (CFS)

Goal: Another fair scheduling

- but deterministic
- very efficient (with $O(\log N)$ scheduling decision)
- favors interactive programs
- adjusts timeslices dynamically
- Widely used in Linux

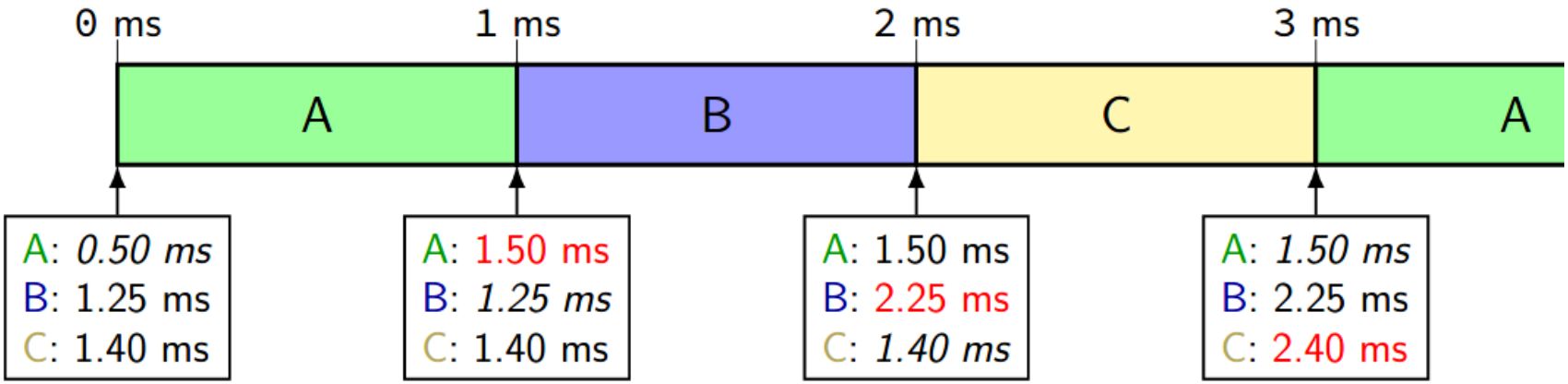
- each thread has a virtual runtime ($\approx$ run time)
- incremented when it is running based on how long it runs
- more/less important thread? multiply adjustments by factor
- adjustments for threads that are new or were sleeping
  - too big an advantage to start at runtime 0

A simple algorithm: run thread with lowest virtual runtime

data structure: balanced tree

Example

virtual time, always ready, 1 ms quantum

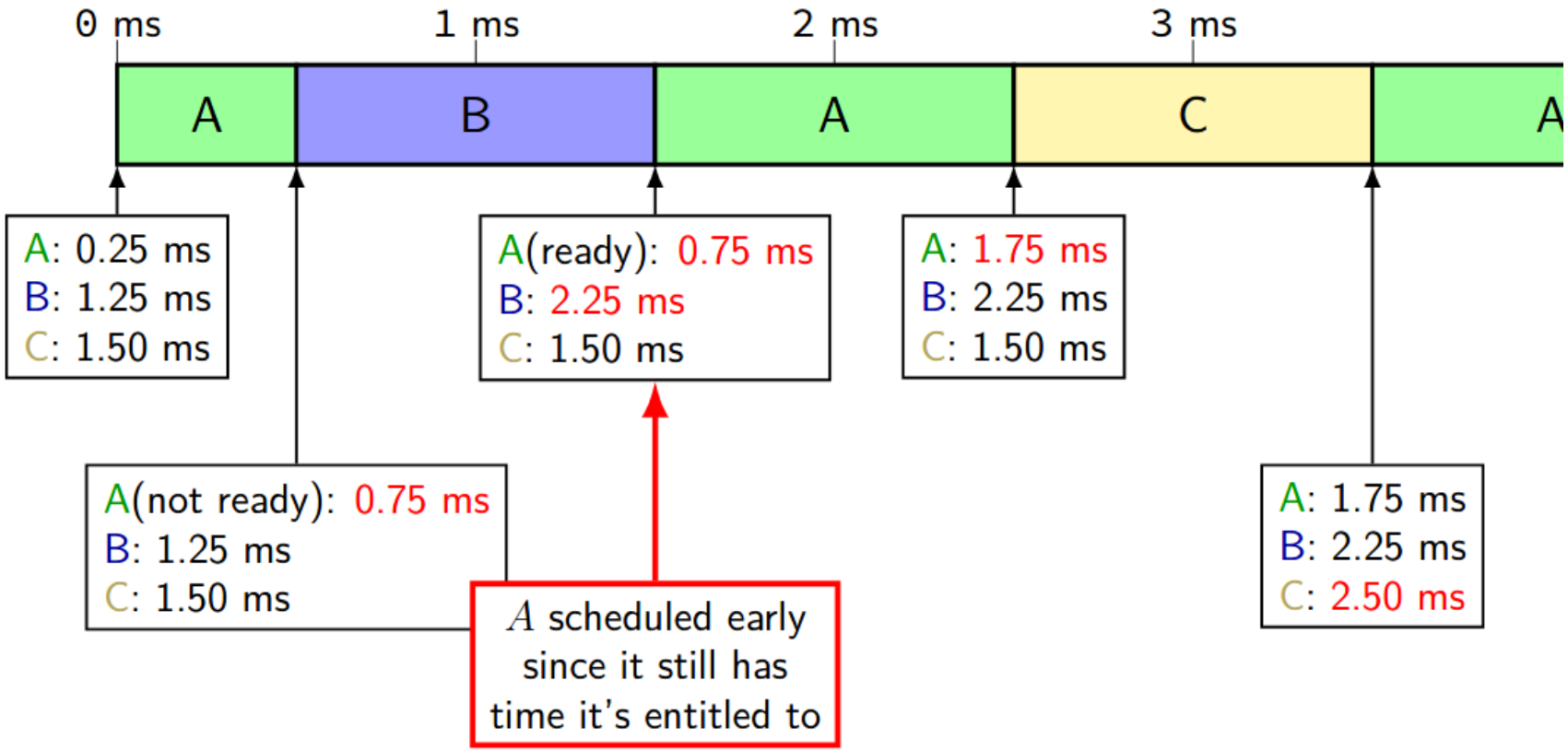


at each time: update current thread's time & run thread with lowest total time

same effect as round robin if everyone uses whole quantum

A doesn't use whole time ...

not runnable briefly? still get your share of CPU

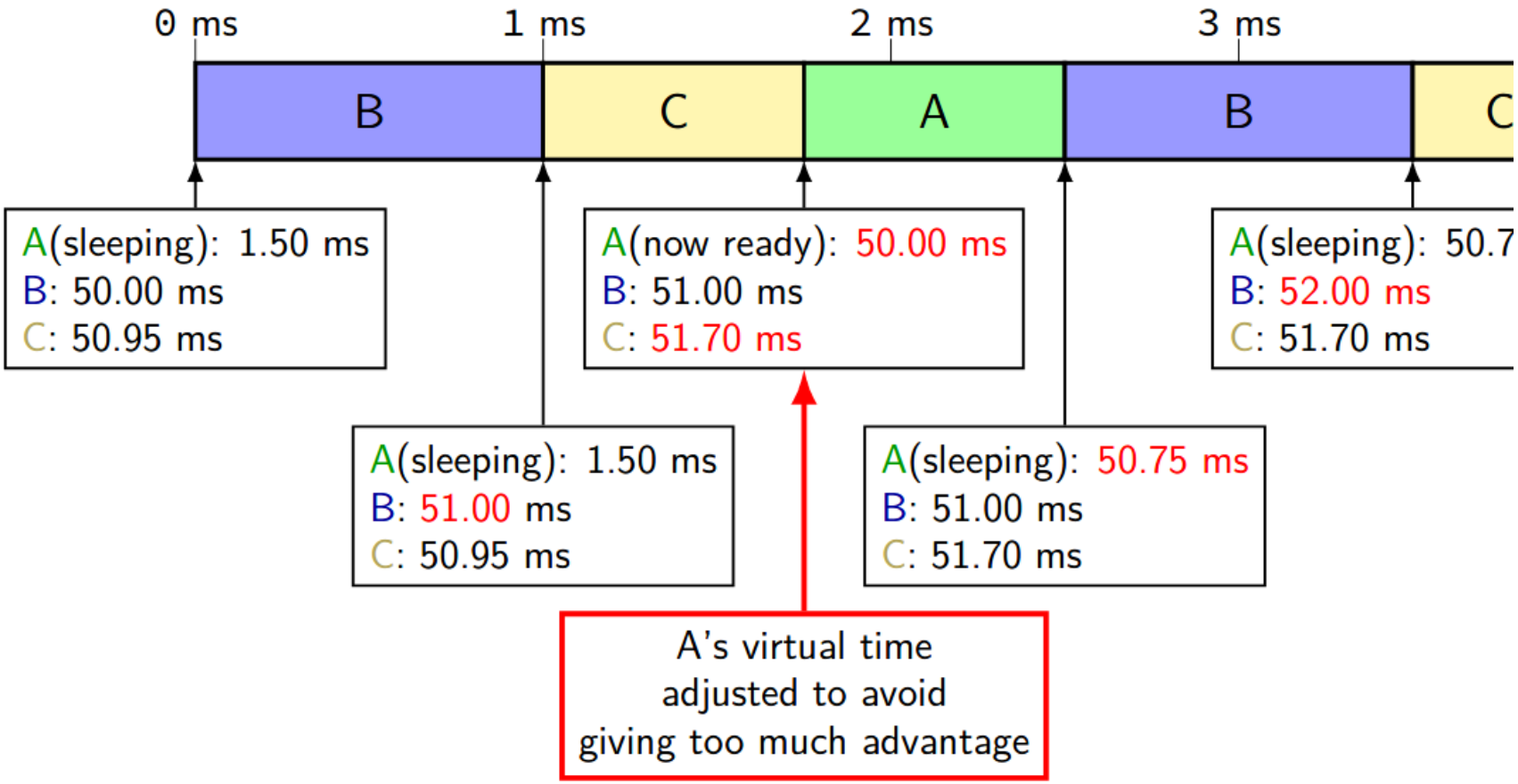


Example

A's long sleep...

not runnable for a while? get bounded advantage

vruntime = a little less than minimum virtual time (of already ready tasks)



handling proportional sharing

solution: multiply used time by a factor

e.g. 1 ms of CPU time costs process 2 ms of virtual time
higher factor $\Rightarrow$ process less favored to run

CFS quantum lengths

Adaptive for few threads (default: 8 process)

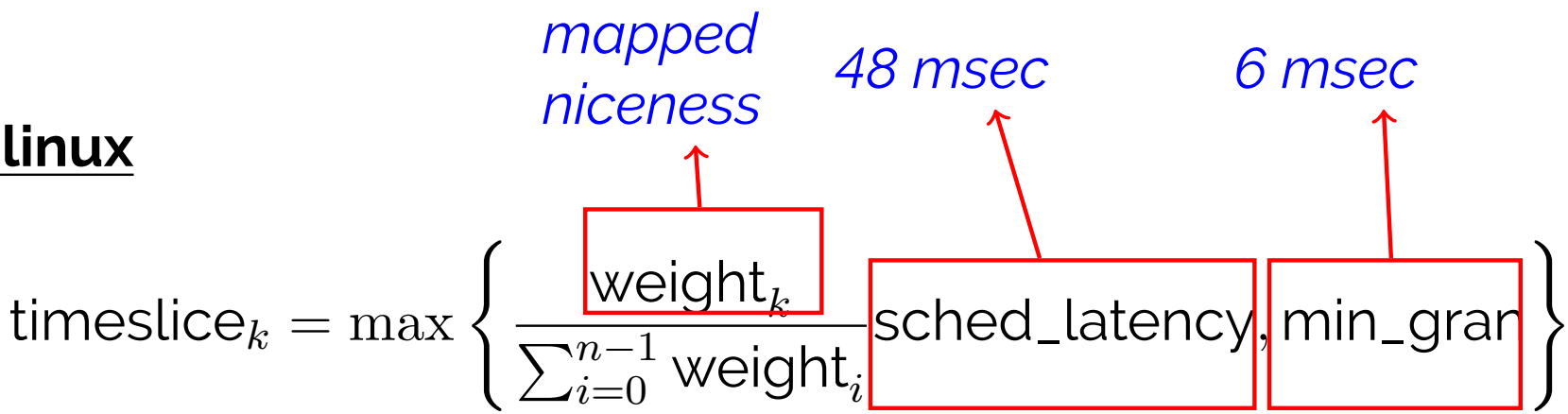
Fixed for more number of threads (default: 6 msec)

Constraining context switch

Switch if virtual time of new task $<$ current virtual time by threshold (default $=$ 1 msec)

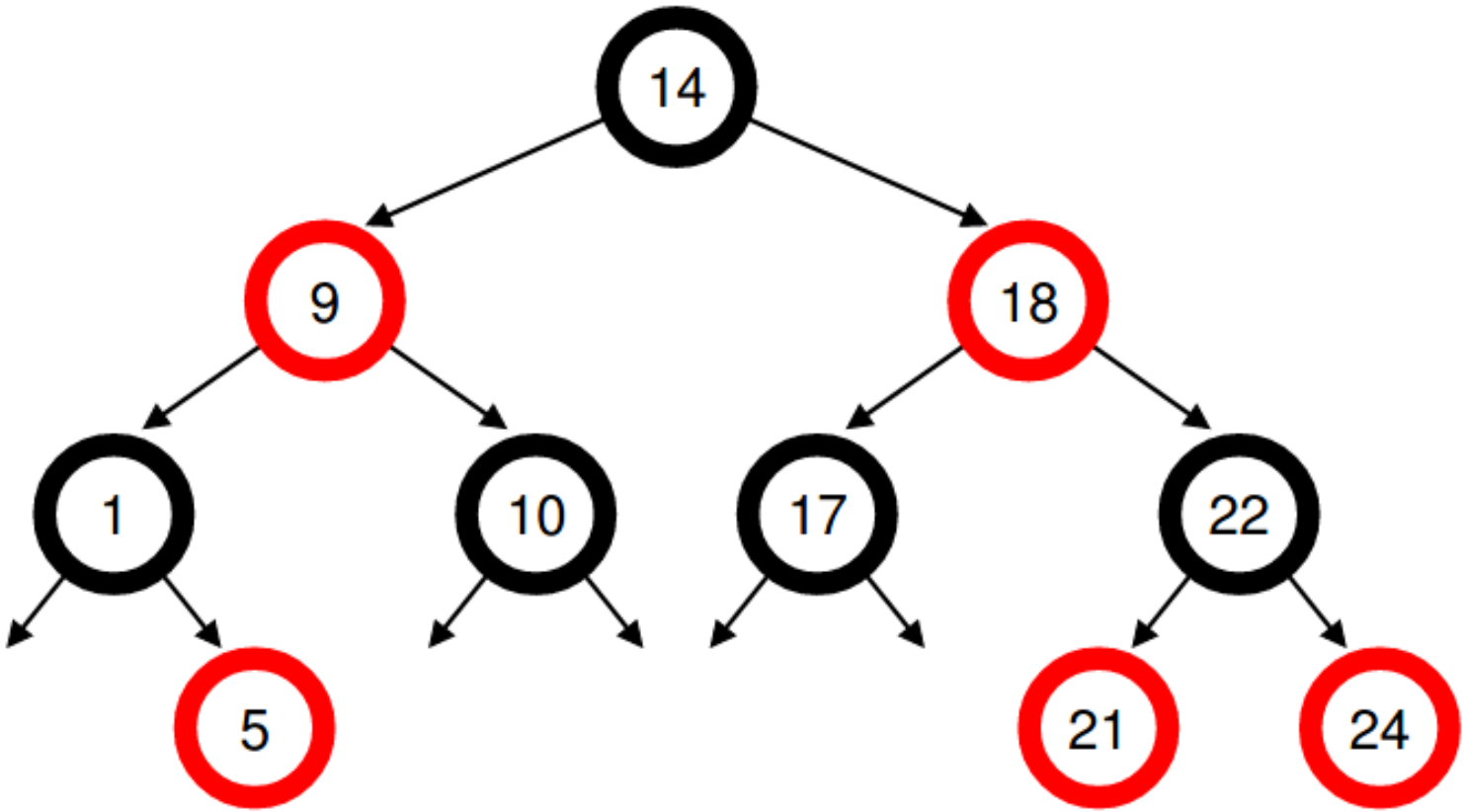
(otherwise, wait until quantum is done)

On linux



$$\text{vruntime}_i = \text{vruntime}_i + \frac{\text{weight}_0}{\text{weight}_i} \text{runtime}_i$$

On linux: using the red-black tree $\Rightarrow O(\log N)$



Class Question

In CFS, changing a process priority affects on

A) its virtual runtime calculation

B) its slice time interval

Earliest Deadline First

Earliest Deadline First

so far: "best effort" scheduling

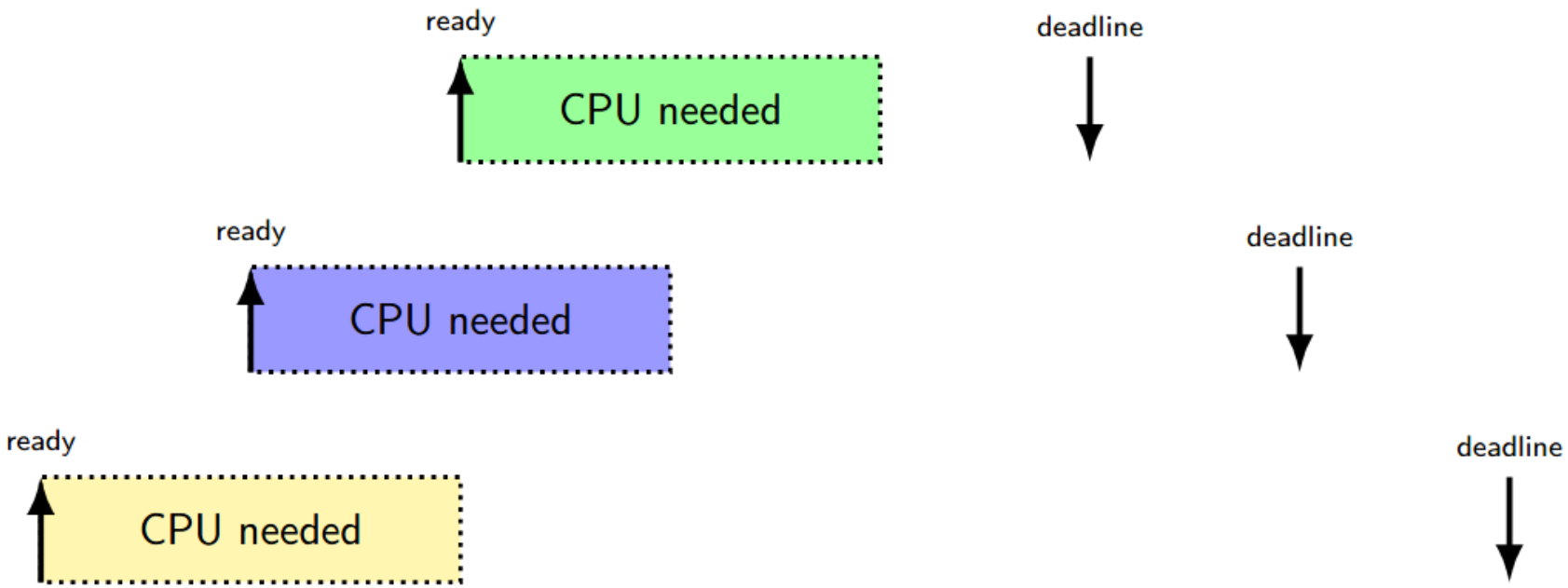
best possible (by some metrics) given some work

alternate model: need guarantees
deadlines imposed by real-world:

- process audio with 1ms delay
- computer-controlled industry
- car brake + engine control computer
- ...

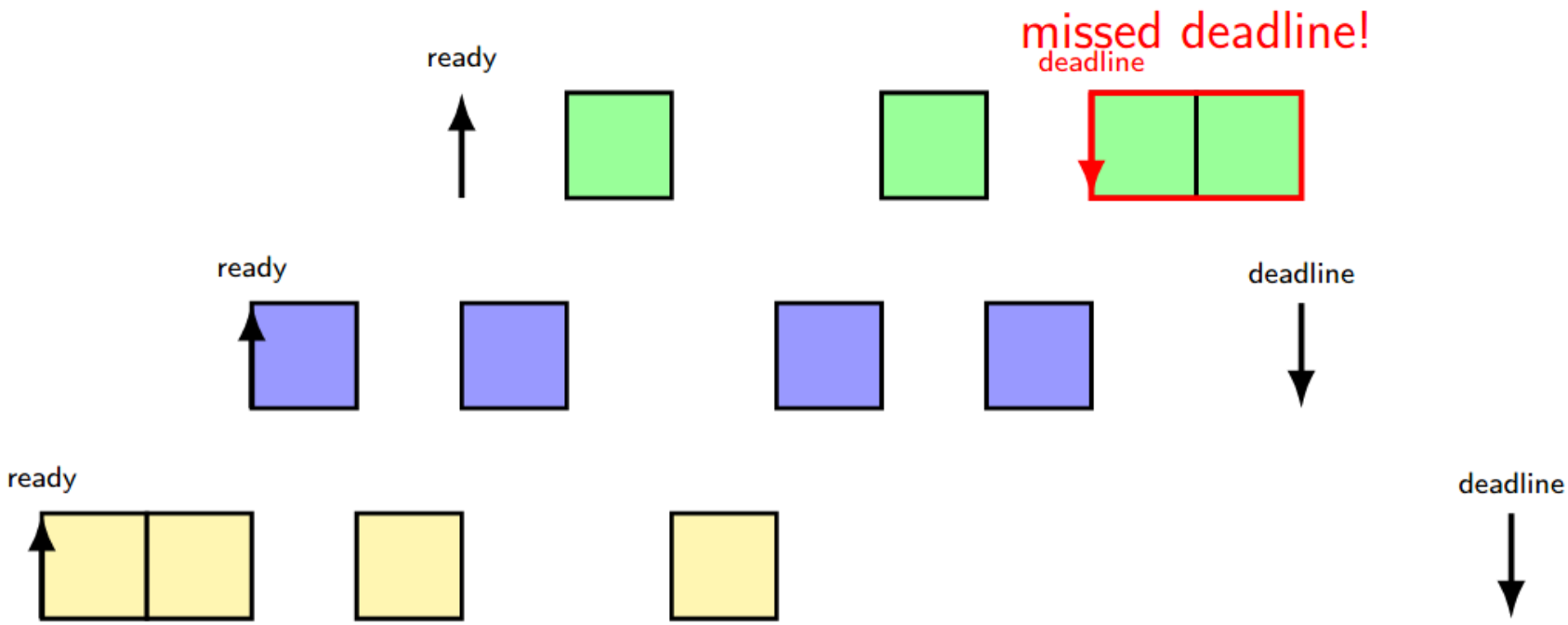
Example

real time example: CPU + deadlines



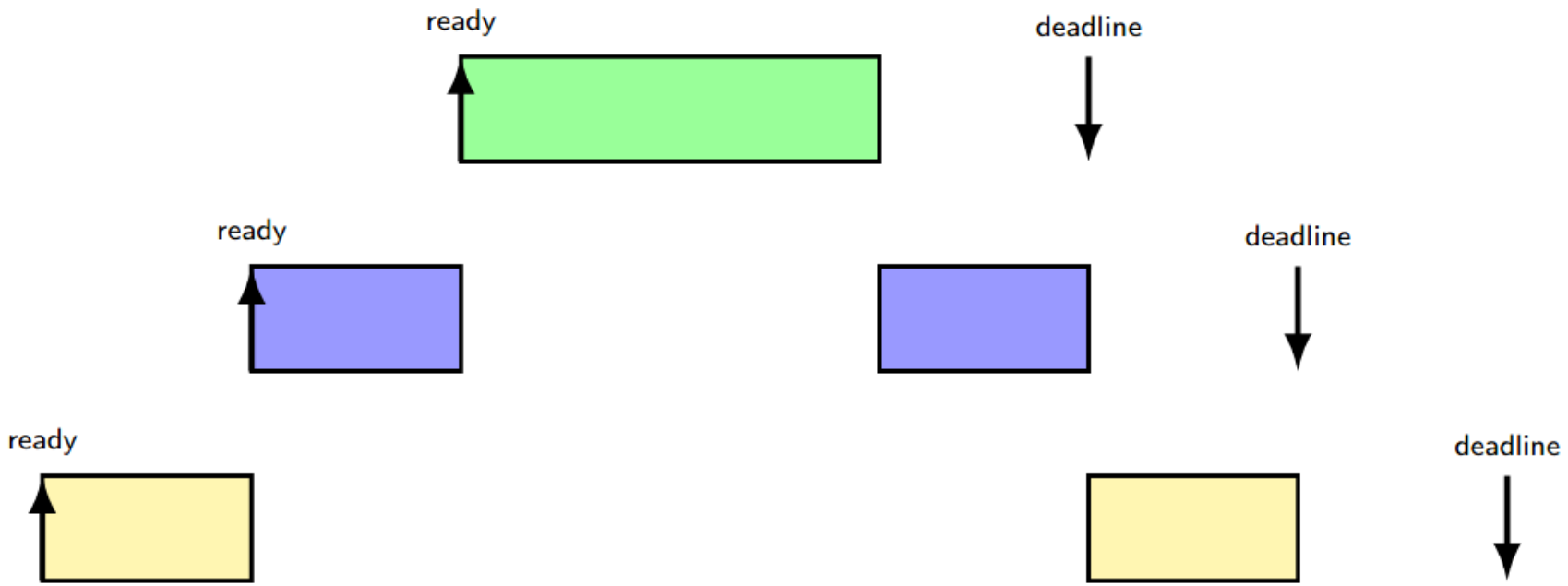
Example

Using RR scheduling



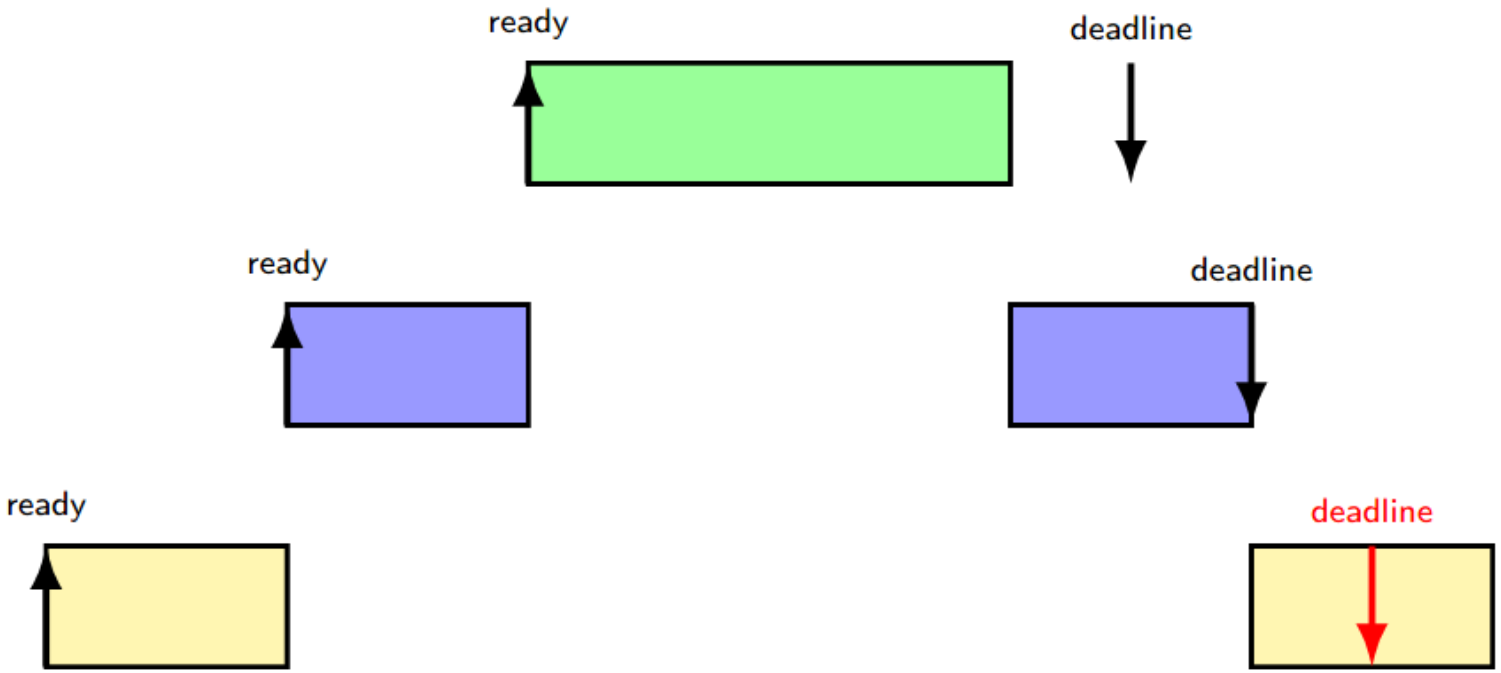
Example

Using earliest deadline first scheduling



Example

impossible deadlines



no way to meet all deadlines!⇒ **admission control**

earliest deadline first and...

earliest deadline first does not (even when deadlines met)

- minimize response time
- maximize throughput
- maximize fairness

exercise: give an example

Final Remarks

which scheduler should I choose?

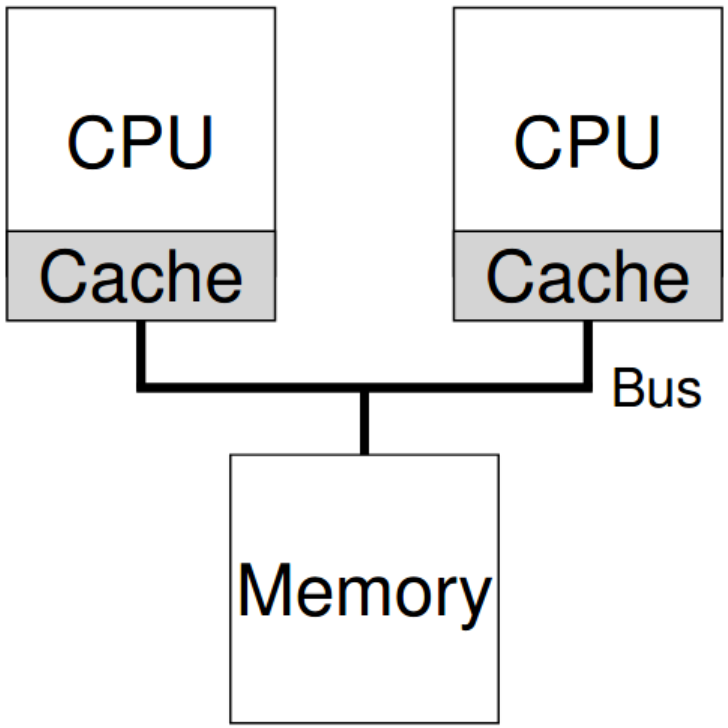
I care about ...

- CPU throughput: first-come first-serve
- average turnaround time: STCF
- Average response time: RR
- Both turnaround time & response time: MLFQ
- fairness: CFS
- deadline: earliest deadline first

A note on multiprocessors

extra considerations:

- want two processors to schedule without waiting for each other
- cache affinity: want to keep process on same processor (better for cache)
- cache coherence
- want load balance



Miscellaneous

Signal

Signal

A very short message that may be sent to a process or a group of processes.

The only information given to the process is usually a number identifying the signal

Software Interrupt

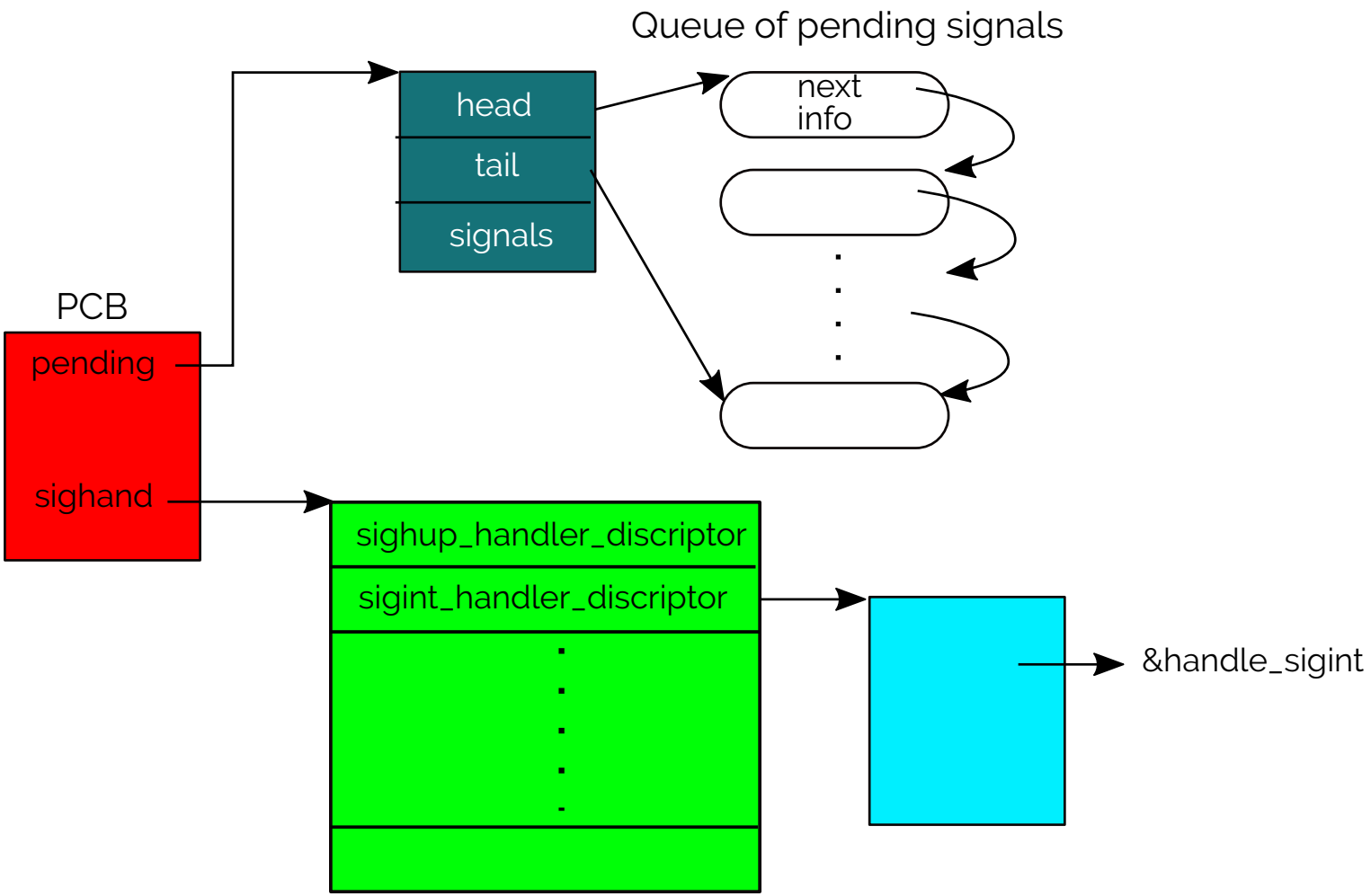
Syscalls for raising signals: kill, ...
Syscalls for catching: sigaction/signal, ...
More info:

- man 7 signal
- *Understanding the linux kernel*, by Bovet & Cesati

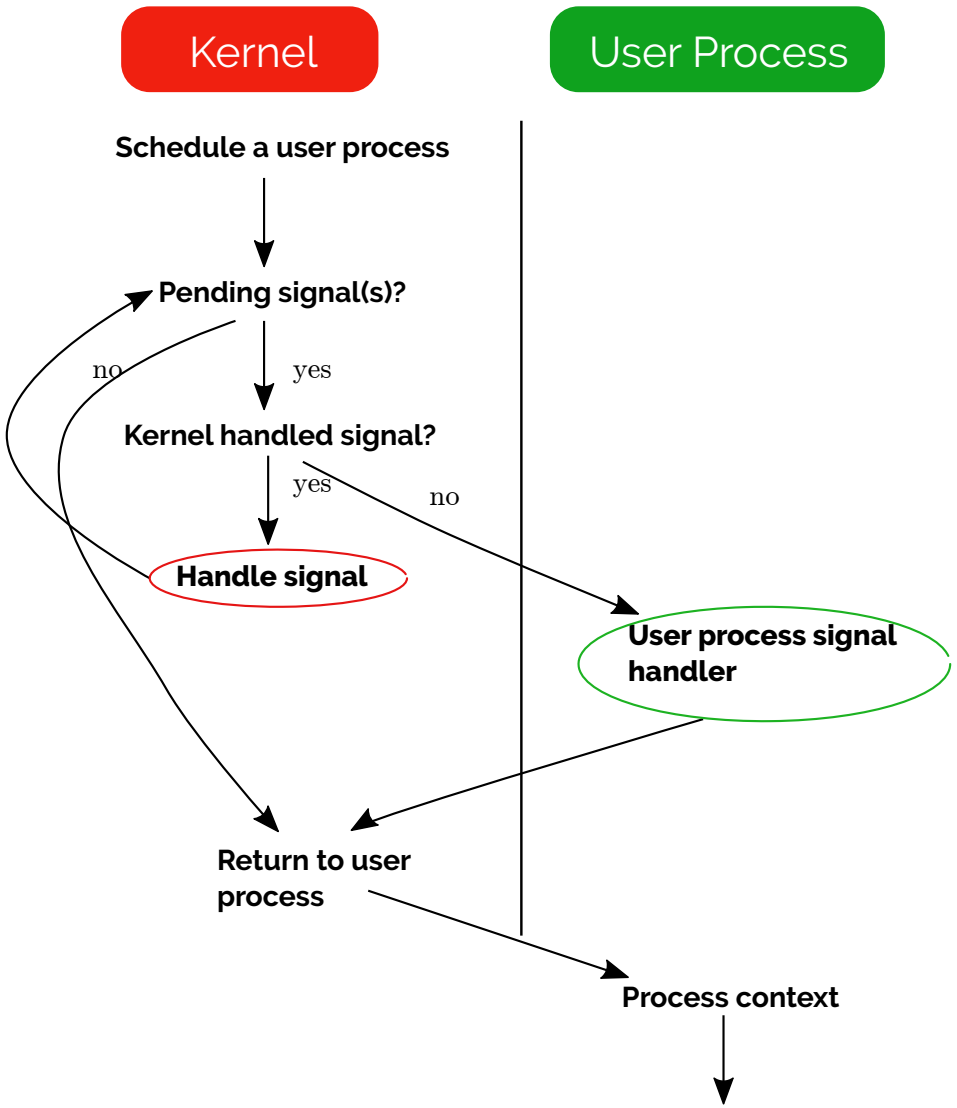
Signal

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

/* Print a message, then request another SIGALRM. */
void handle_sigalrm(int sig) {
  printf("Hello!\n");
  alarm(1); /* Request another SIGALRM in 1 second. */
}
/* User typed Ctrl-C. Taunt them. */
void handle_sigint(int sig) {
  printf("Ha ha, can't kill me!\n");
}

int main() {
  signal(SIGINT, handle_sigint);
  signal(SIGALRM, handle_sigalrm);

  alarm(1); /* Request a SIGALRM in 1 second. */
  while (1) pause();

return 0;
}
```

Signal


Queue of pending signals
head
tail
signals
next
info
PCB
pending
sighand
sighup_handler_discriptor
sigint_handler_discriptor
&handle_sigint

Signal


Kernel
User Process
Schedule a user process
Pending signal(s)?
no
yes
Kernel handled signal?
yes
no
Handle signal
User process signal handler
Return to user process
Process context

Signal

- Standard or real time
- Catchable or not
- Pending or blocked

⋮

(see “man 7 signal”)

Pipe

Pipe

A unidirectional **interprocess communication channel (IPC)** (see “man 7 pipe”)

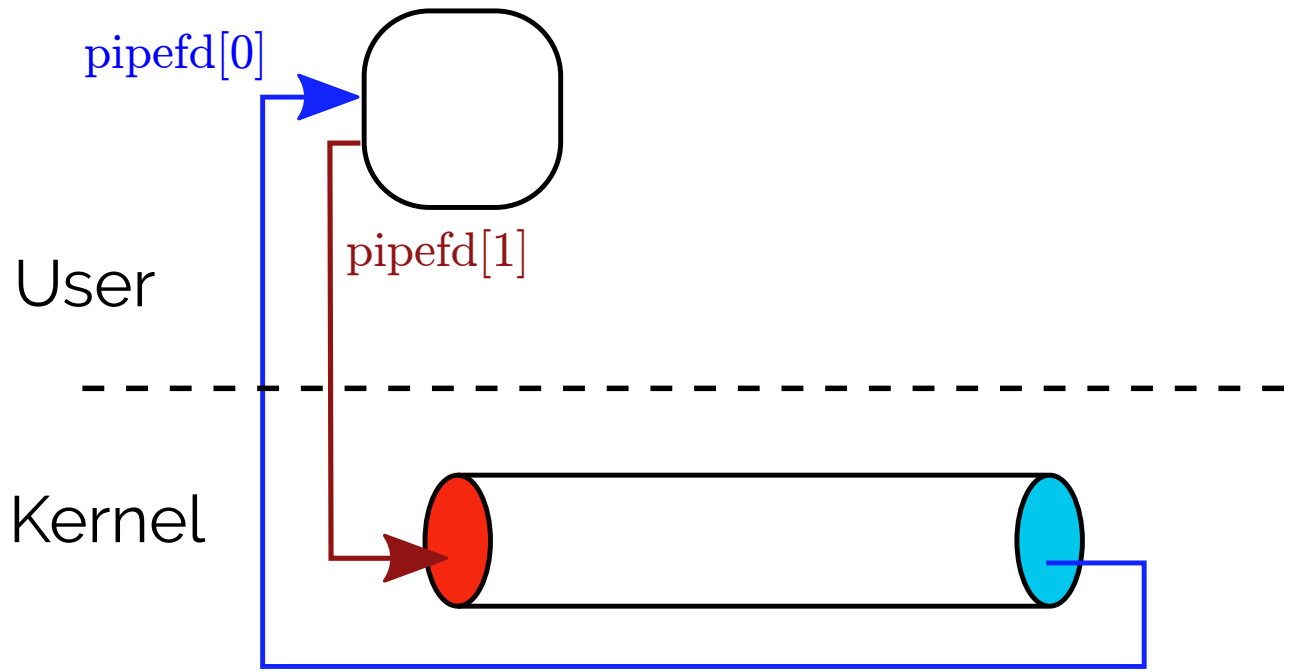
Resides in the Kernel

A pipe has a read end and a write end

- write: atomic, blocking/non-blocking on full pipe, **SIGPIPE** when no readers
- read: blocking/non-blocking(**SIGIO**) on empty pipe, EOF when no writer

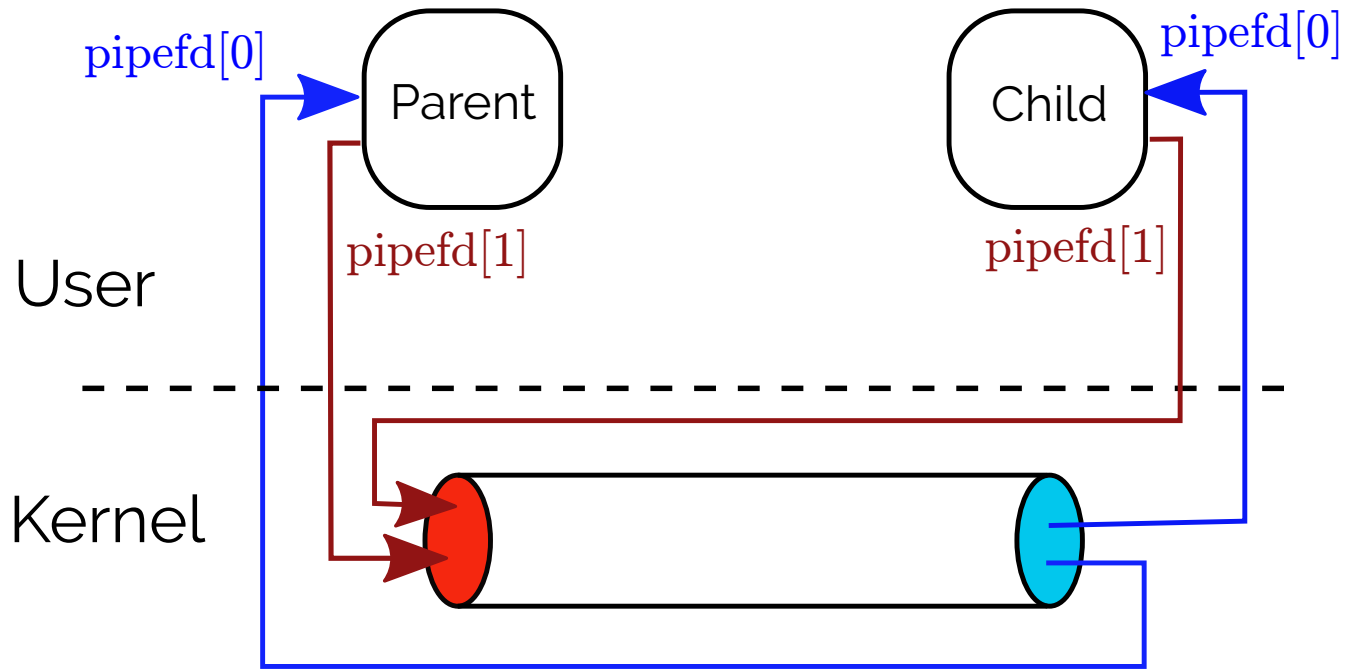
Pipe

```
int pipe(int pipefd[2]); // man 2 pipe
```



Pipe

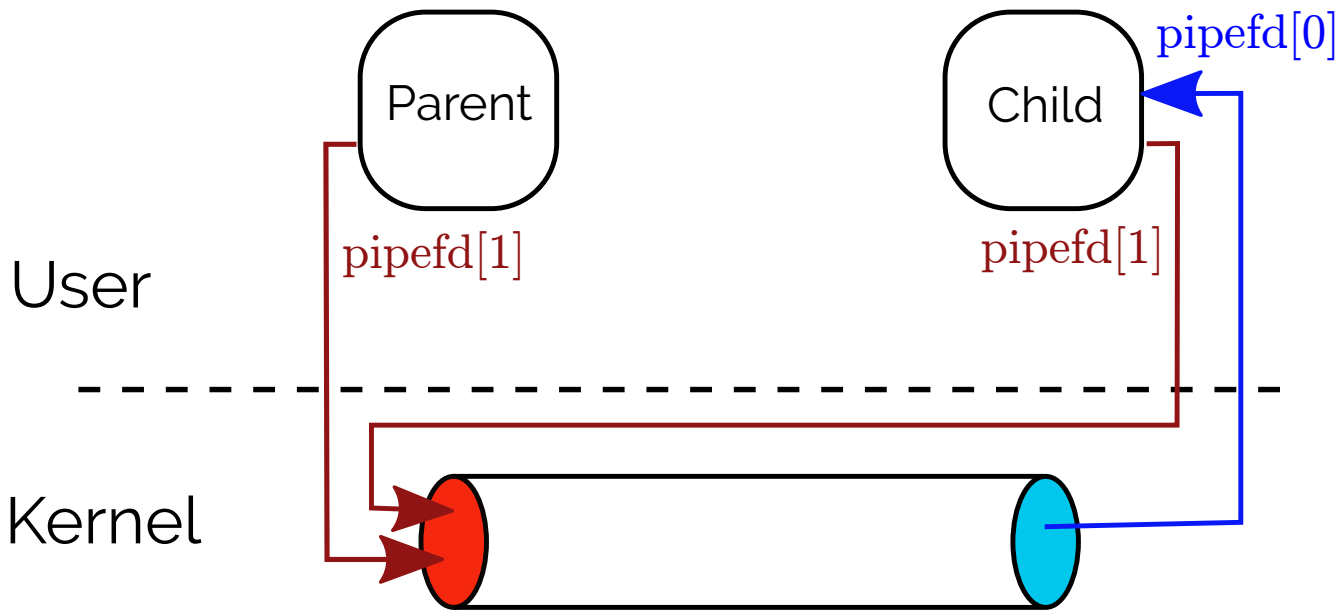
`fork()`



Pipe

If parent wants to be “producer”

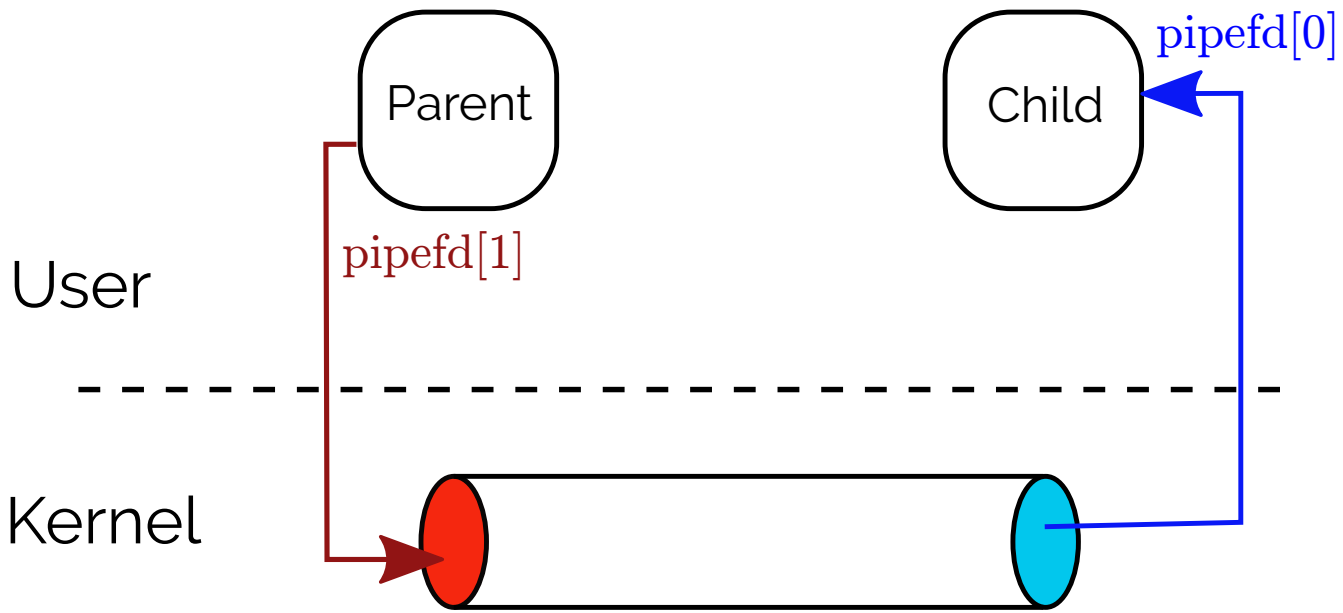
```
close(pipefd[0])
```



Pipe

Then child is a “consumer”

```
close(pipefd[1])
```



Shared Memory

Shared Memory

Bi-directional inter process communication by sharing a region of memory (see “man 7 shm_overview”)

Resides in RAM (not in the disk) inside the user space of processes

- the fastest way for processes to exchange and share data

A shared memory object will exist until the system is shut down, or until all processes have unmapped the object and it has been deleted with shm_unlink

Not managed by the kernel (be careful about the race!)

- Processes must synchronize their access to a shared memory object, using, for example, semaphores (semctl, semop, ...)

sender.c

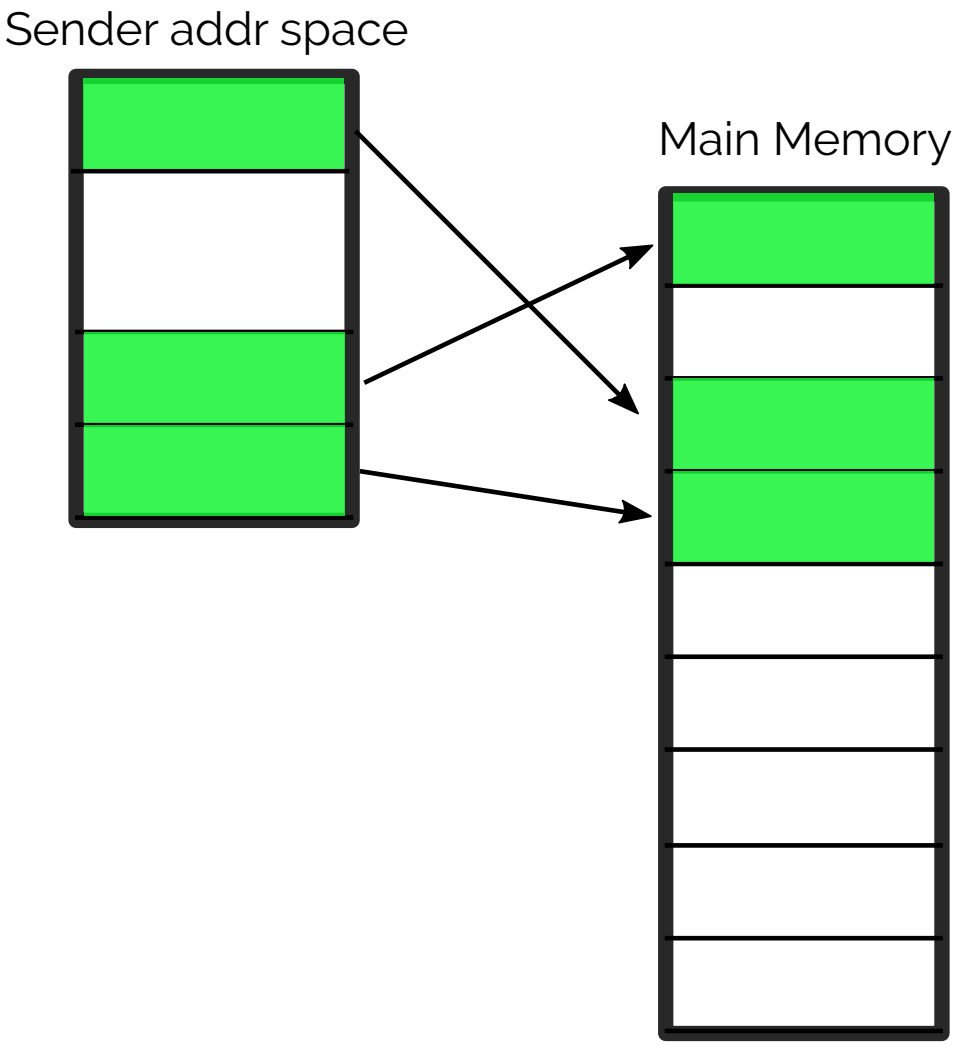
```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>
#include "protocol.h"


int main()
{
    int fd = shm_open(NAME, O_CREAT | O_EXCL | O_RDWR, 0600);
    if (fd<0) {
        perror("shm_open()");
        return EXIT_FAILURE;
    }

    int size = ARRAY_S + STR_L;

    ftruncate(fd, size);

    void *data = mmap(0, size, PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);

    if(data == (void *)-1){
        perror("mmap()");
        return EXIT_FAILURE;
    }

    int *array = (int *)data;
    char *msg = (char *)(data + ARRAY_S); // get address of messge

    printf("sender address: %p\n", data);

    // write array in shared memory
    for (int i = 0; i < NUM; i++) {
        array[i] = i*i;
    }

    // write messge in shared memory
    strncat(msg, "Hi Shared Memory!", STR_L);

    munmap(data, size);

    close(fd);
    return EXIT_SUCCESS;
}
```

receiver.c

```c
#include "protocol.h"
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

int main()
{
    int fd = shm_open(NAME, O_RDONLY, 0666);
    if (fd<0) {
        perror("shm_open()");
        return EXIT_FAILURE;
    }

    int size = ARRAY_S + STR_L;

    void *data = mmap(0, size, PROT_READ, MAP_SHARED, fd, 0);
    if(data == (void *)-1){
        perror("mmap()");
        return EXIT_FAILURE;
    }

    int *array = (int *)data;
    char *msg = (char *)(data + ARRAY_S);

    printf("receiver address: %p\n", data);

    for (int i = 0; i < NUM; i++) {
        printf("num%d: %d\n", i, array[i]);
    }

    printf("msg: %s\n", msg);

    munmap(data, size);

    close(fd);

    // delete file
    shm_unlink(NAME);
    return EXIT_SUCCESS;
}
```

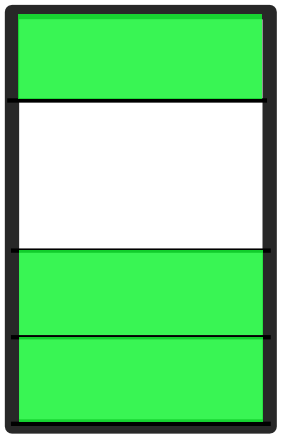
makefile

```
1   CFLAGS= -lrt -Wall -Wextra
2
3   all: receiver sender
4
5   receiver: receiver.c
6       $(CC) receiver.c -o receiver $(CFLAGS)
7
8   sender: sender.c
9       $(CC) sender.c -o sender $(CFLAGS)
10
11
12  clean:
13      rm receiver sender
```

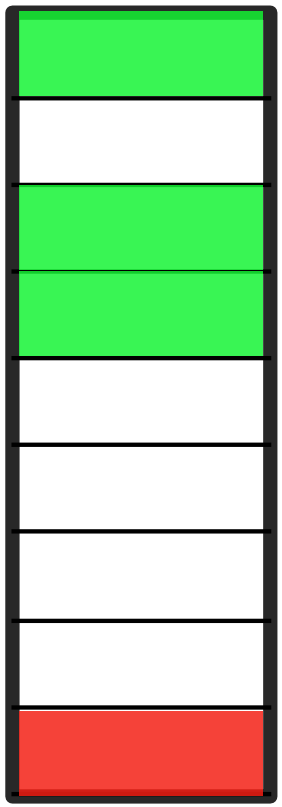

Sender addr space
Main Memory

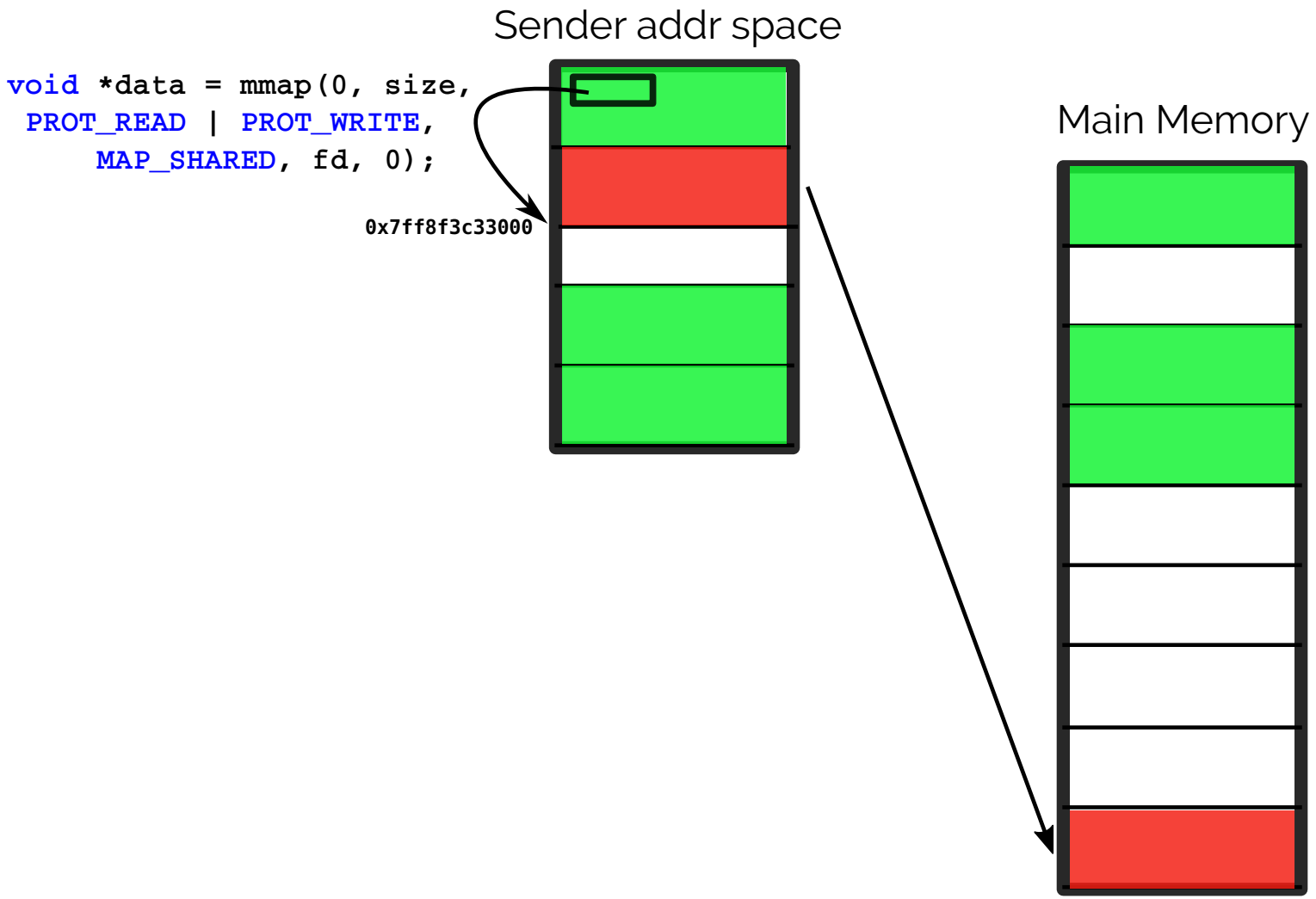
```
int fd = shm_open(NAME,
 O_CREAT | O_EXCL | O_RDWR, 0600);
    if (fd<0) {
        perror("shm_open()");
        return EXIT_FAILURE;
    }

    int size = ARRAY_S + STR_L;

    ftruncate(fd, size);
```

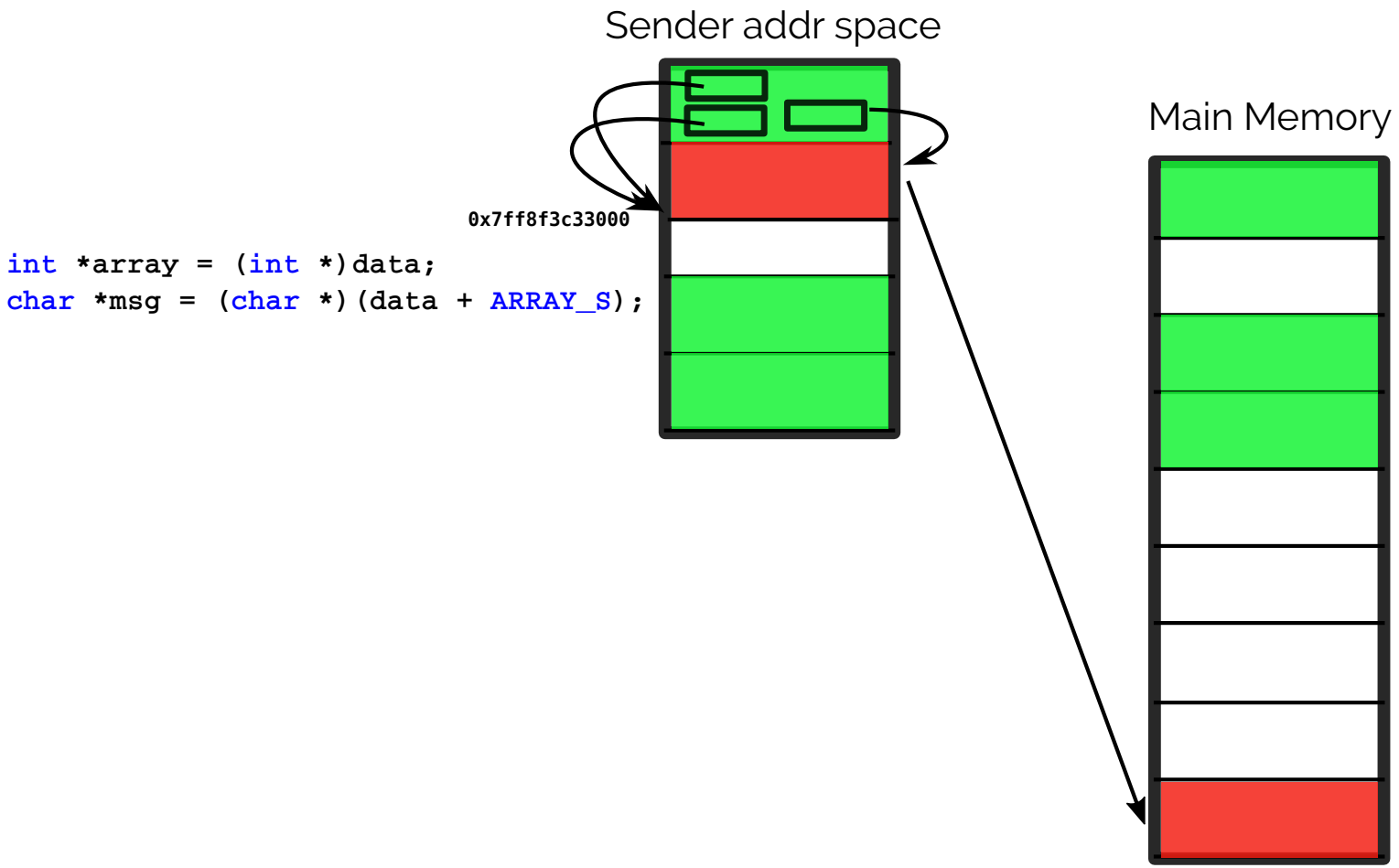
Sender addr space



Main Memory



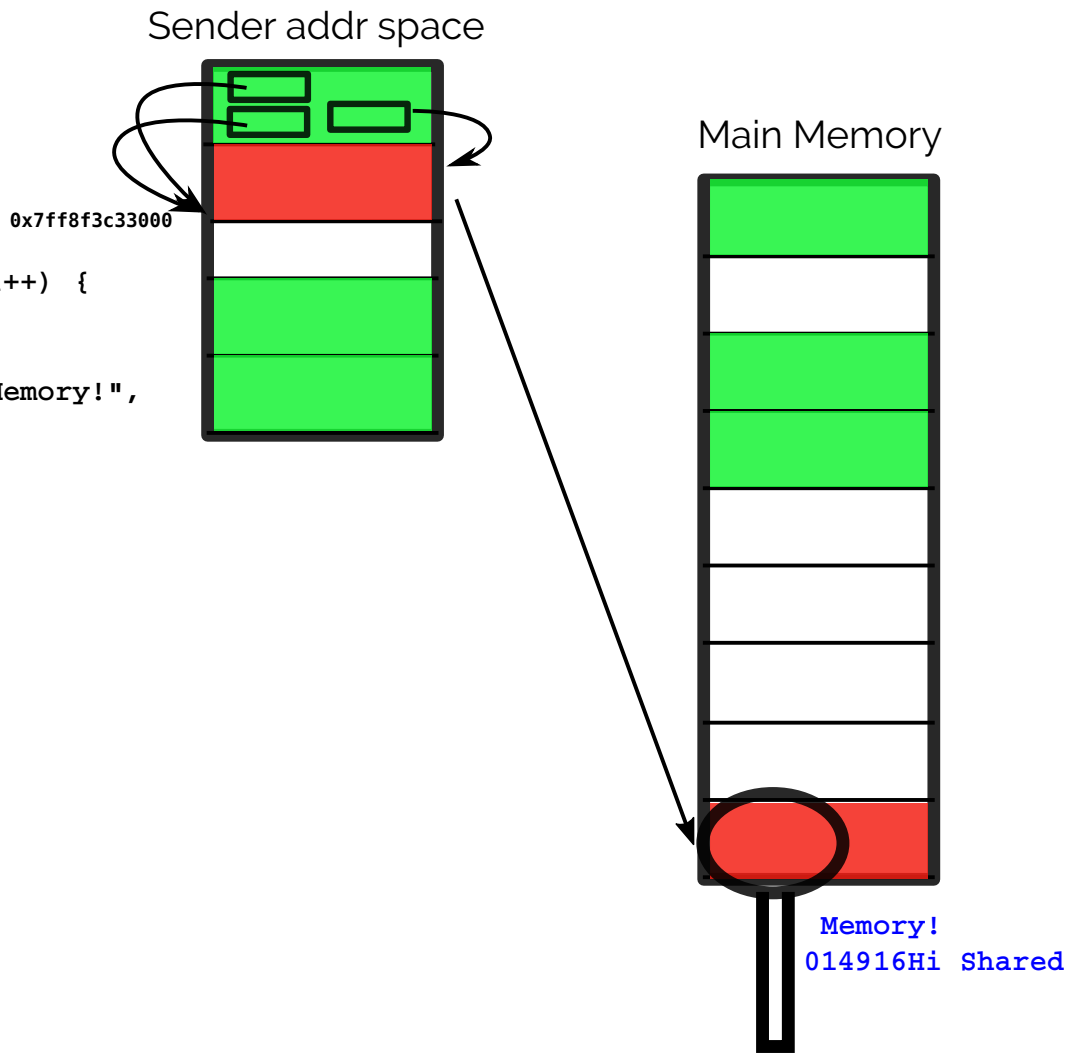

Sender addr space
void *data = mmap(0, size,
PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
0x7ff8f3c33000
Main Memory

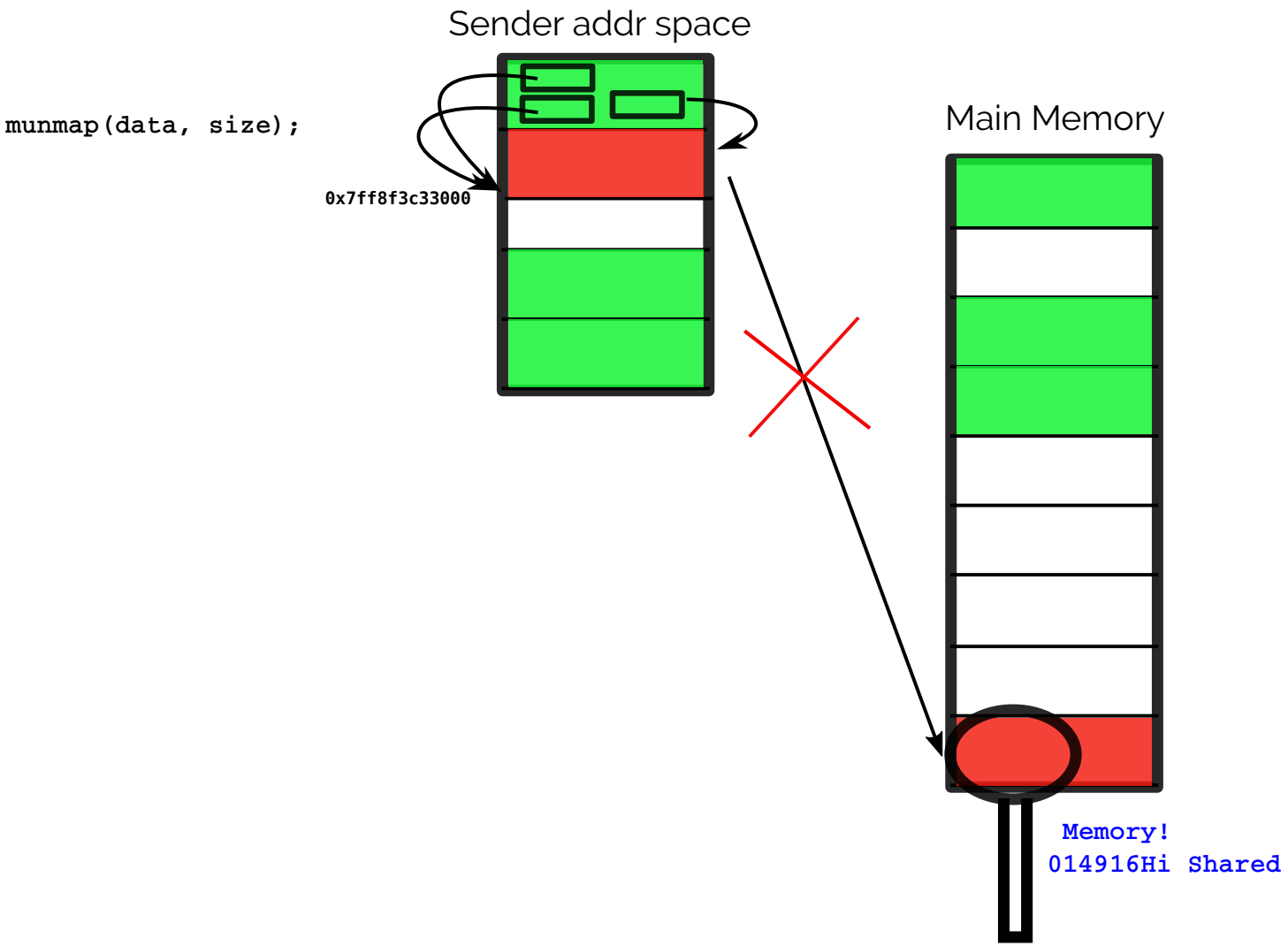

Sender addr space
Main Memory
0x7ff8f3c33000
int *array = (int *)data;
char *msg = (char *)(data + ARRAY_S);

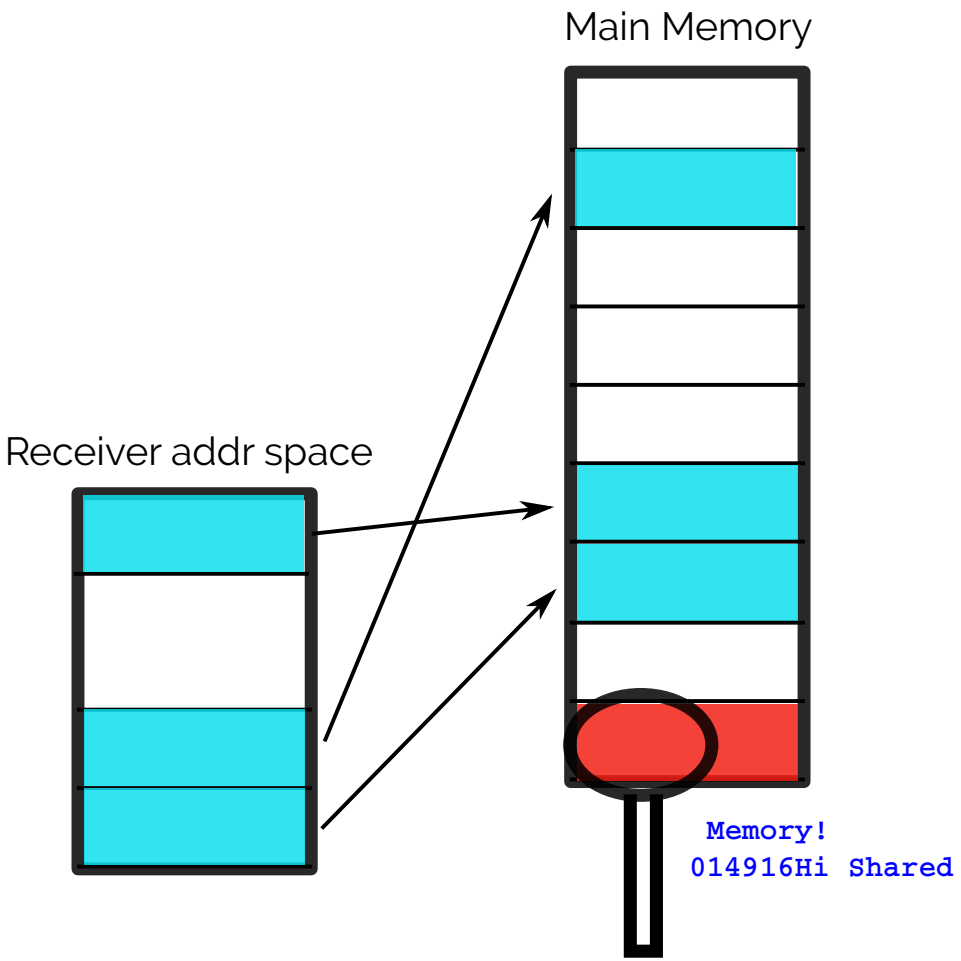
```
printf("sender address: %p\n", data);
```

```
sender address: 0x7ff8f3c33000
```

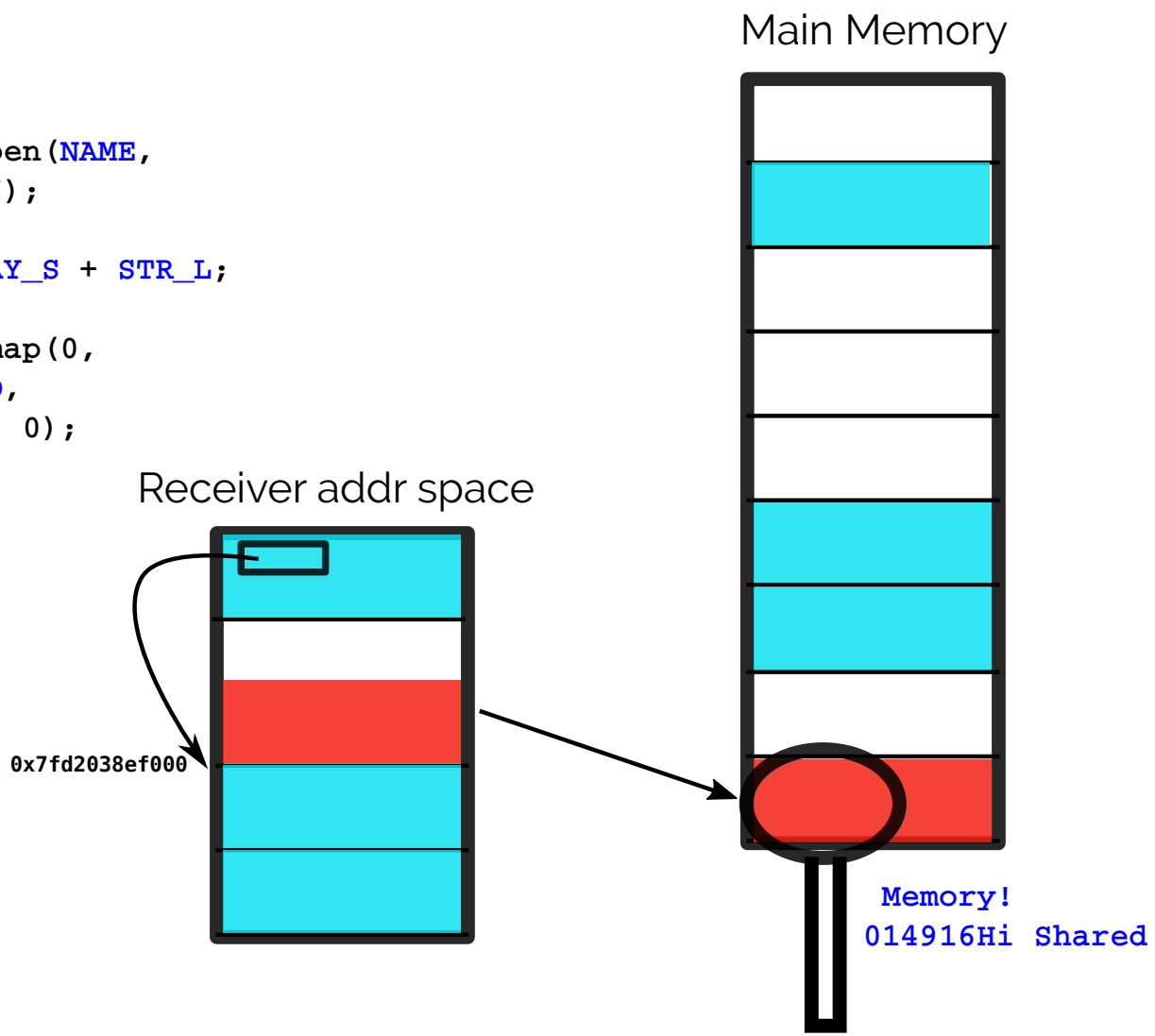
```
for (int i = 0; i < NUM; i++) {
        array[i] = i*i;
    }
 strncat(msg, "Hi Shared Memory!",
 STR_L);
```



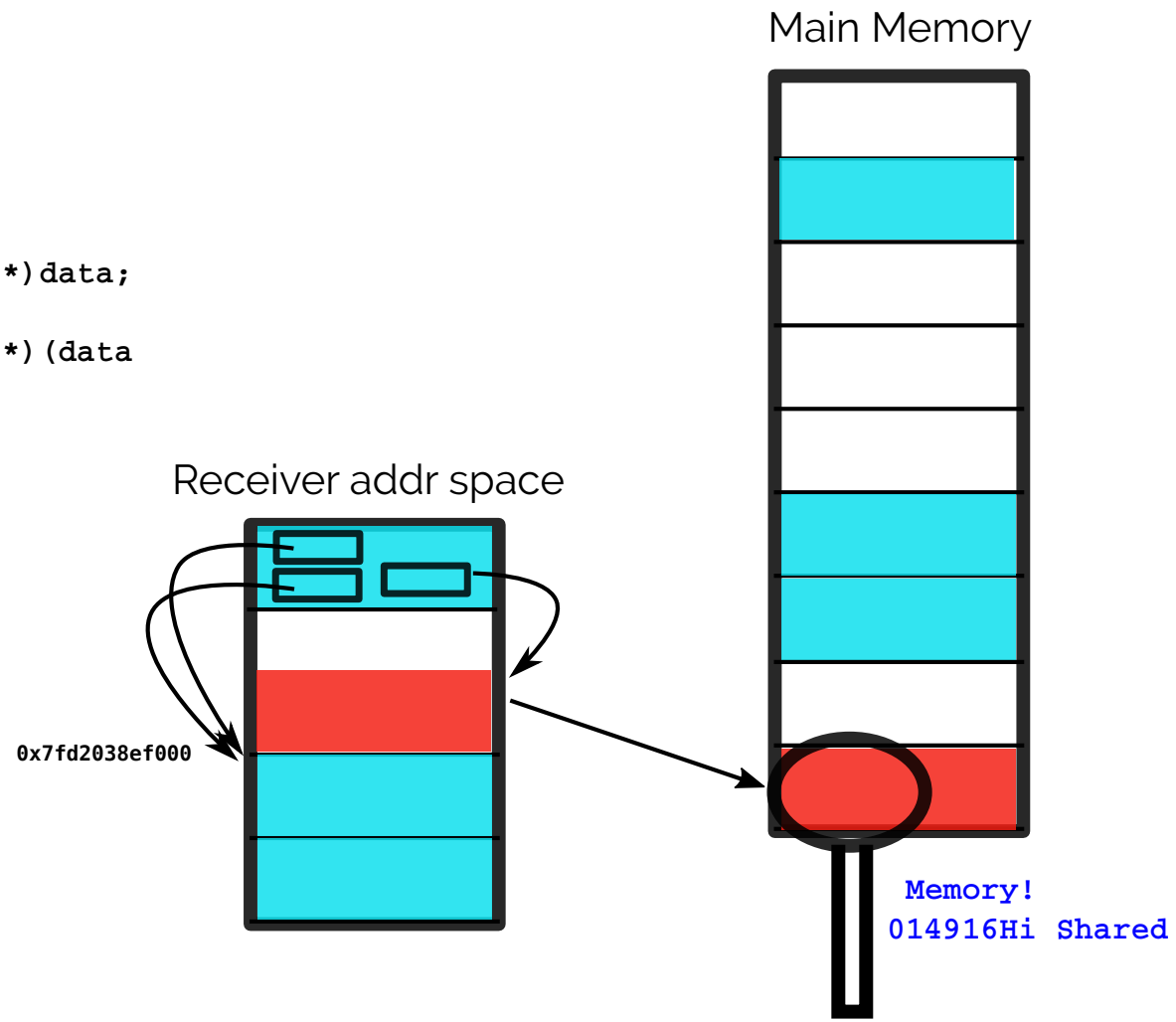

Sender addr space
munmap(data, size);
0x7ff8f3c33000
Main Memory
Memory!
014916Hi Shared


Main Memory
Receiver addr space
Memory!
014916Hi Shared

```
int fd = shm_open(NAME,
 O_RDONLY, 0666);

int size = ARRAY_S + STR_L;

void *data = mmap(0,
size, PROT_READ,
MAP_SHARED, fd, 0);
```



```
int *array = (int *)data;

char *msg = (char *)(data
 + ARRAY_S);
```

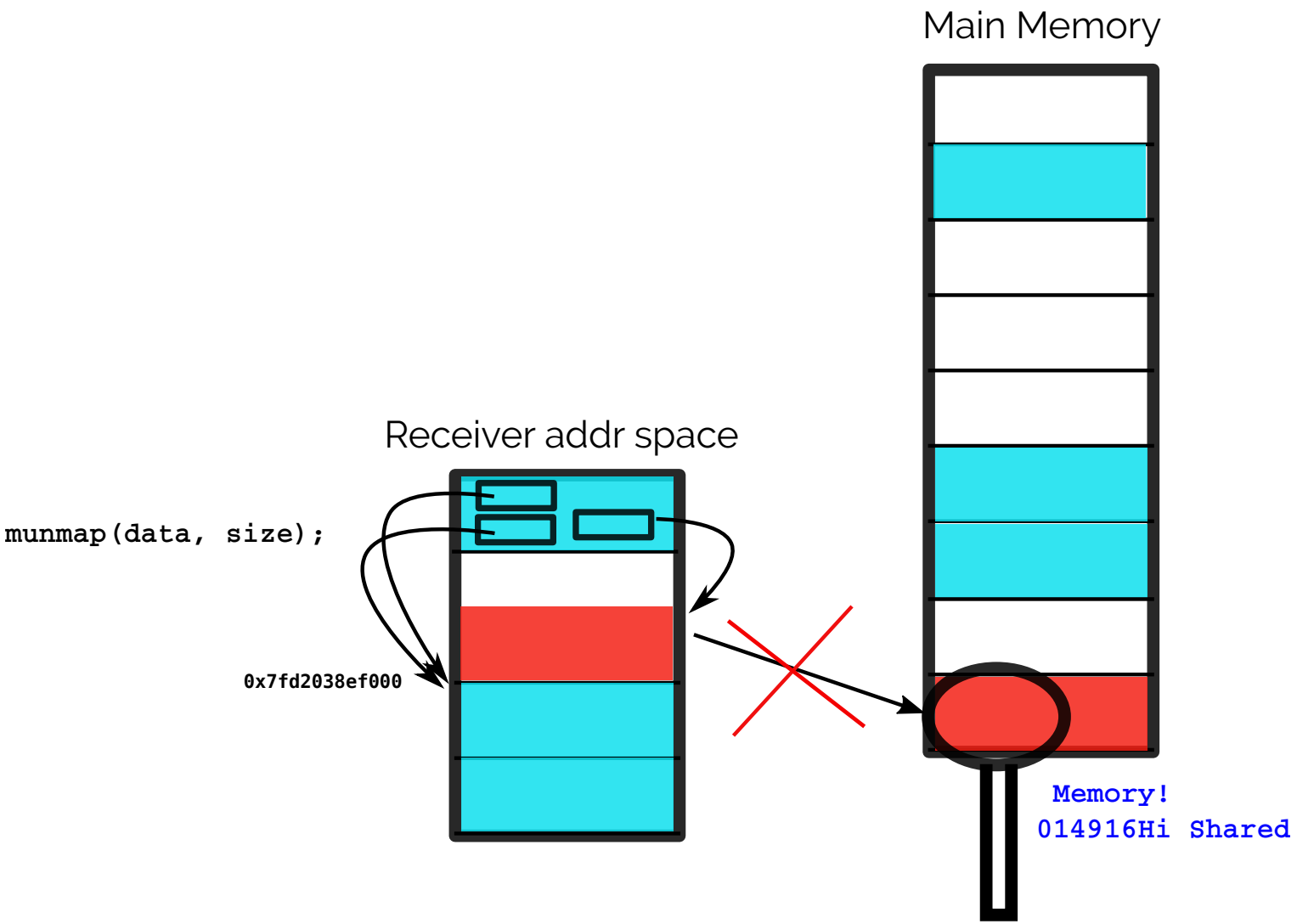


```
printf("receiver address: %p\n", data);
```

```
receiver address: 0x7fd2038ef000
```

```
for (int i = 0; i < NUM; i++)
{
 printf("num%d: %d\n", i,
array[i]);
}

printf("msg: %s\n", msg);
```

```
num0: 0
num1: 1
num2: 4
num3: 9
num4: 16
msg: Hi Shared Memory!
```


Main Memory
Receiver addr space
munmap(data, size);
0x7fd2038ef000
Memory!
014916Hi Shared

What happens if we try to reference the red region in the address space now?

- for example:

```
printf("msg: %s\n", msg);
```

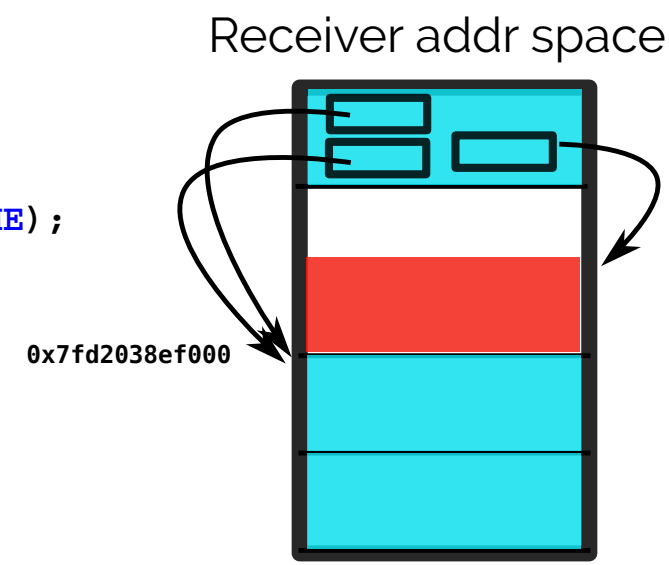
What happens if we try to reference the red region in the address space now?

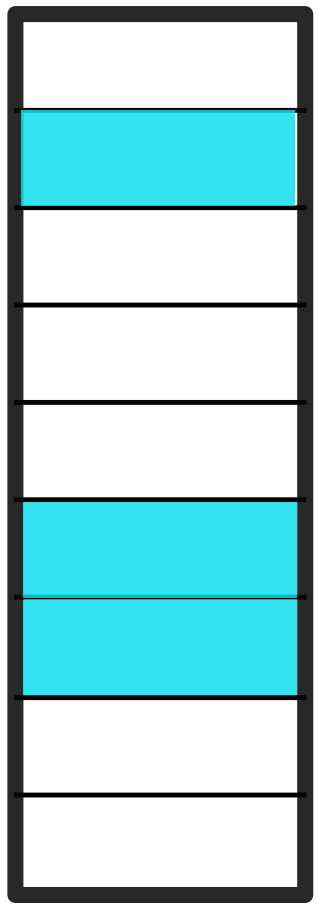
- for example:

```
printf("msg: %s\n", msg);
```

```
[1]    18439 segmentation fault  ./receiver
```

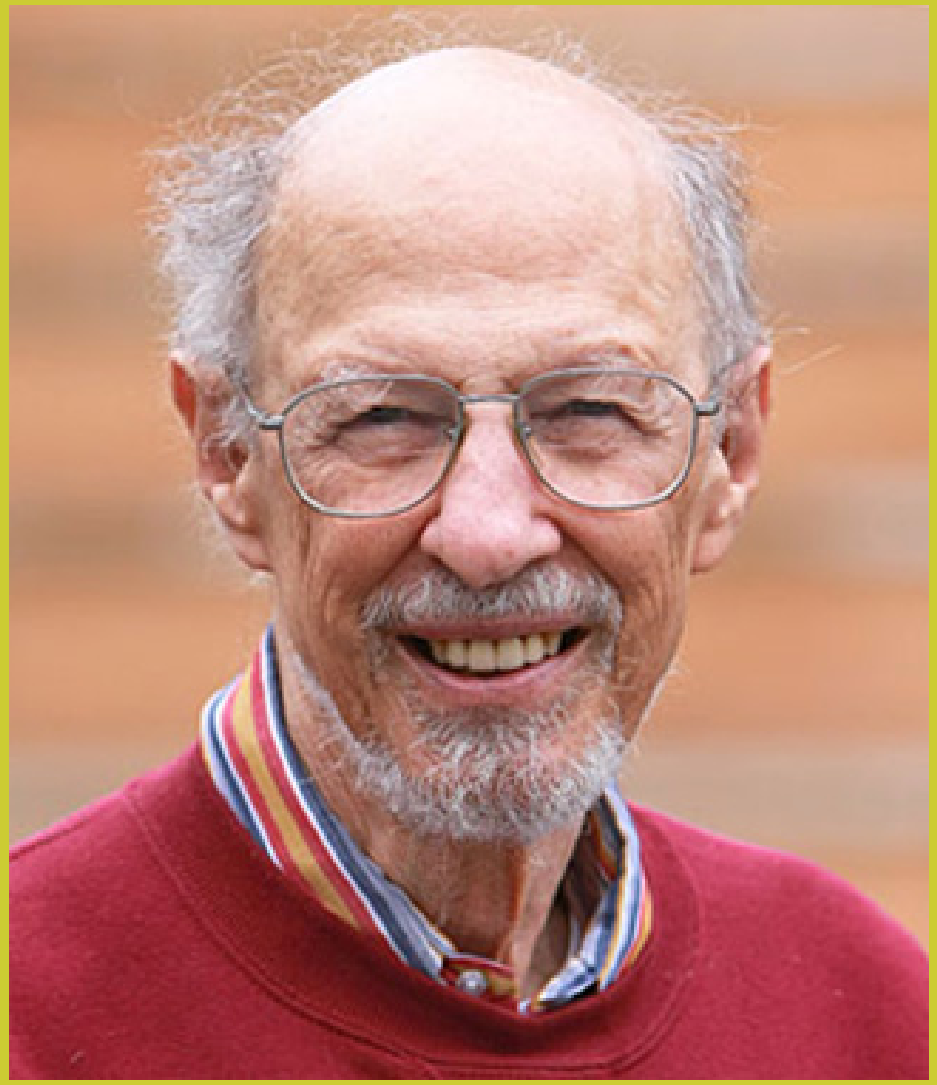
```
close(fd);
shm_unlink(NAME);
```





Fernando Jose Corbato - "Corby" (July 1, 1926 - July 12, 2019)

Nobel prize for development of time sharing operating systems (MIT CTSS, 1961). He also leaded the Multics project, and invented passwords for computer access.



Note

Disclaimer: This lecture slide set was initially developed for Operating System course in Electrical and Computer Engineering Dept. at Isfahan University of Technology. This lecture slide set is mainly for OSTEP book written by Remzi and Andrea at University of Wisconsin. Some slides have been also captured from Dr. Charles Reiss's course slides on operating systems at Virginia university.