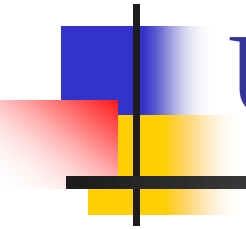
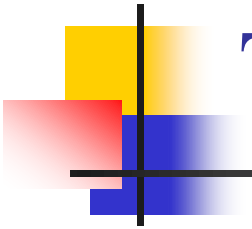
Lecture 03



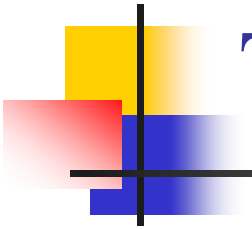
UNIX Architecture

What services the OS provides to the programmer



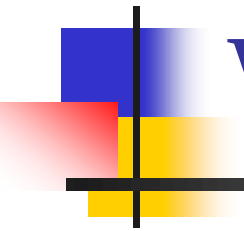
Today's Goal

- UNIX architecture and service overview
  - a global image for you to write a program through out this semaster



Today's material comes from

- Chap. 1 of [Stevens]
- Chap. 3 & 4 of your OS textbook [Silberschatz]
  - Chap. 3: operating system structure
  - Chap. 4: processes

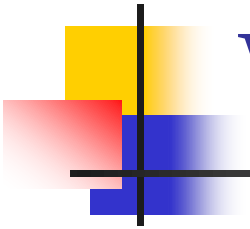


What an OS provides?

What services UNIX provides (the textbook version)

- program execution
- user protection
  - user ID, group ID, file permissions
- file management
- I/O device control
- proc… …rallel prog…
- netv…

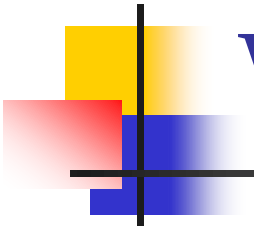
Why these services?



What an OS should provides?

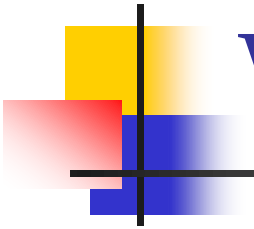
- for general users: ease to run application programs
- for programmers: ease of programming over all hardware resources
- security and users protection

What an OS should provides?



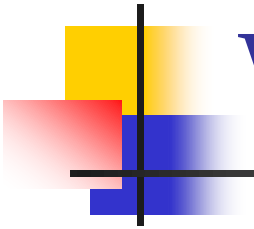
- for general users: ease to use application programs
  - program execution support (shell, command interpreter)
  - GUI (Graphics User Interface)
    - X-windows (on UNIX)
    - M$-Windows
- for programmers: ease of programming over all hardware resources
- security and users protection

What an OS should provides?

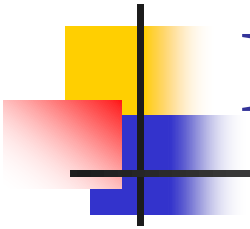


- for general users: ease to use application programs
- for programmers: ease of programming over all hardware resources
  - manage CPUs and processes/threads scheduling
  - memory and storage devices management (e.g. virtual memory, file system, etc.)
  - API (application programmer interfaces) to I/O devices
- security and users protection

What an OS should provides?



- for general users: ease to use application programs
- for programmers: ease of programming over all hardware resources
- security and users protection
  - access permission for files, I/O devices for multi-users
  - network security



In-Class Exercise

(1) create another account, say user2

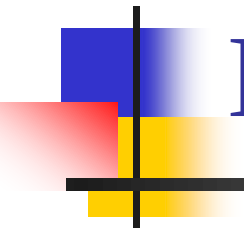
(2) trying to read a file in /home/user2

- should be not able to read

(3) do the following: (login as user2)

- chmod 777 /home/user2
- cd /home/user2
- chmod 666 /home/user2/test.c (the file to read)

(4) login as user1 and read /home/user2/test.c

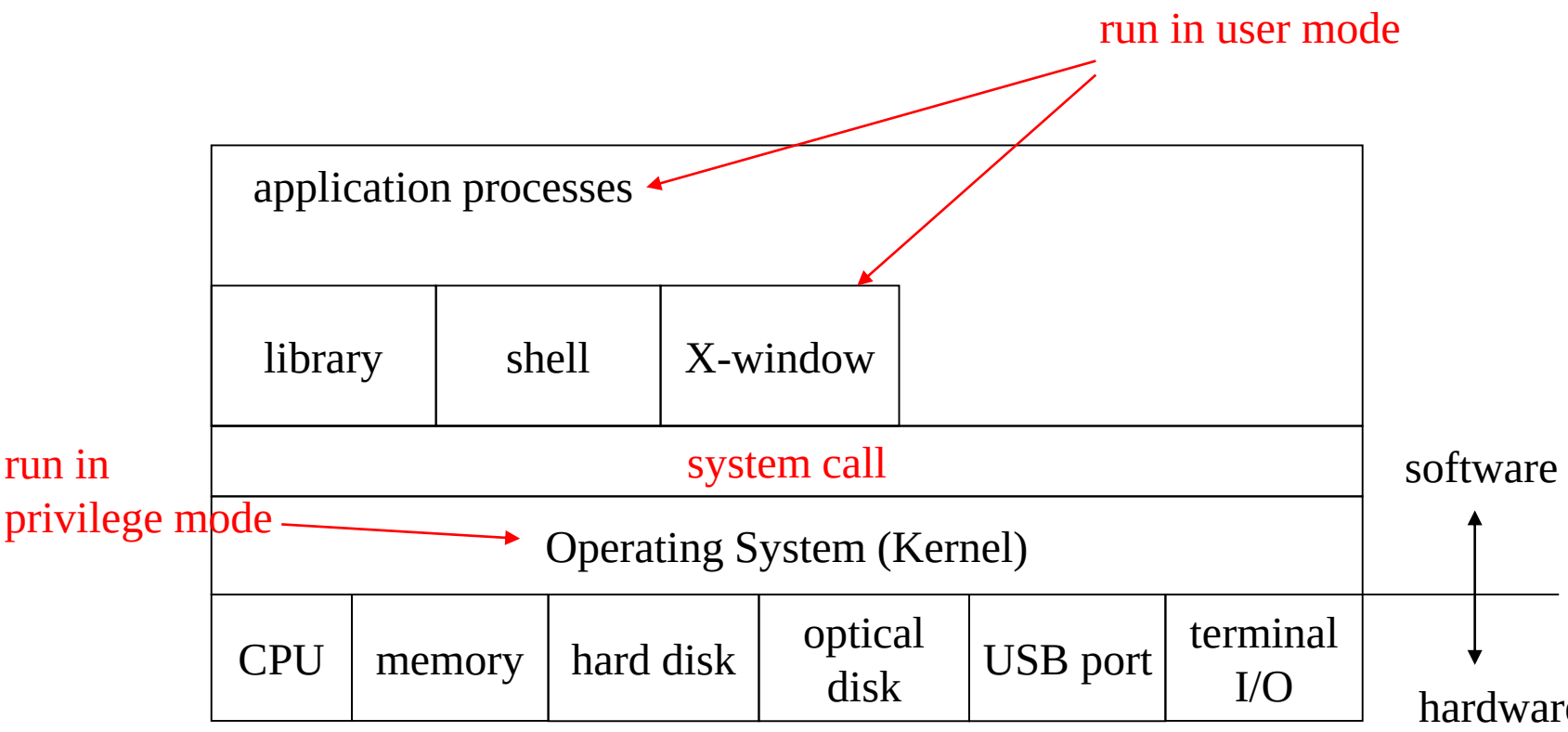


How the OS provides its services

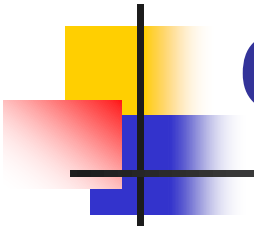
the UNIX architecture overview

UNIX Architecture

- Hardware support: user mode and privilege mode



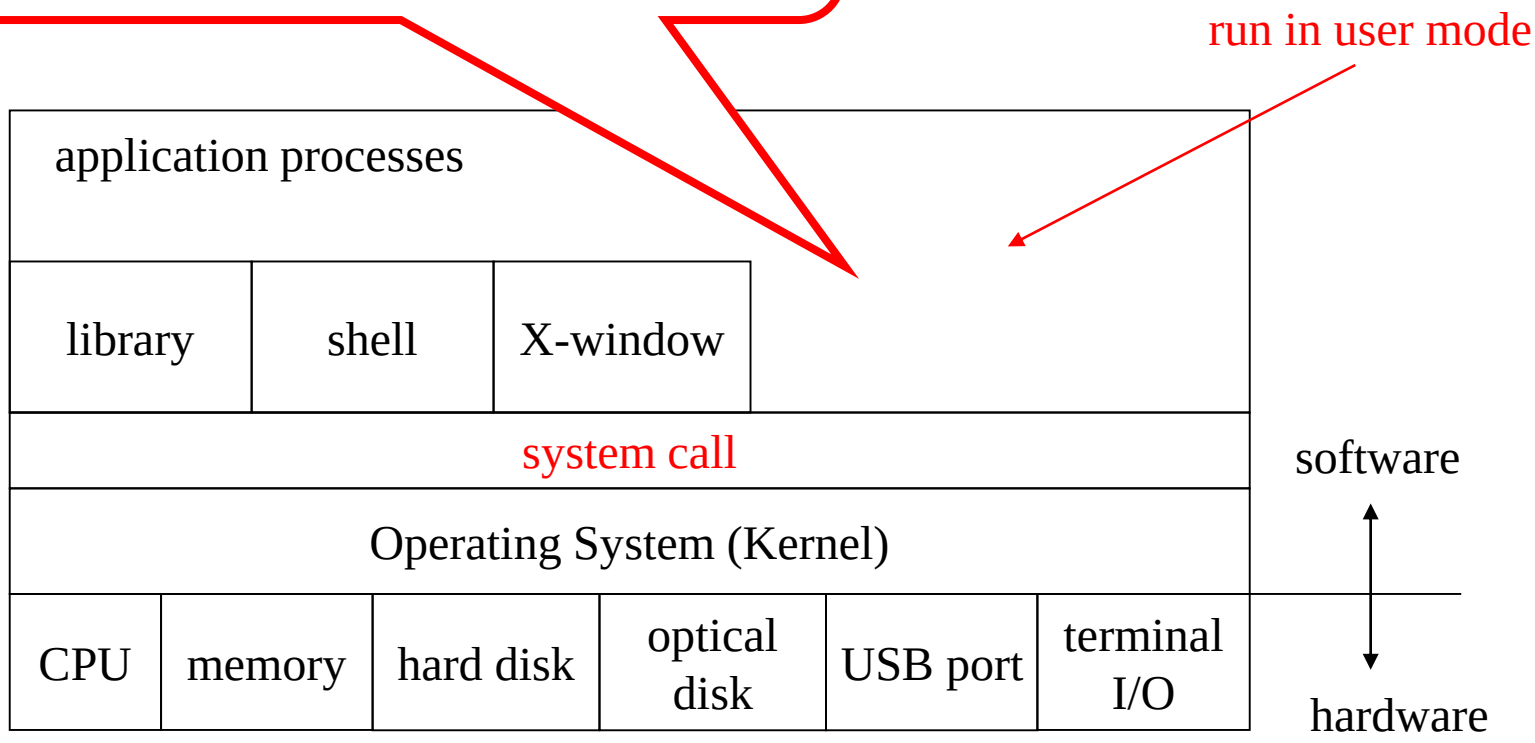
Execution modes supported by CPU



- CPU hardware has to support two execution modes
  - Privileged (kernel) mode: to execute the OS
    - everything the hardware can do
  - User mode: to execute application program
    - no I/O operation and low-level hardware control

UNIX Architecture

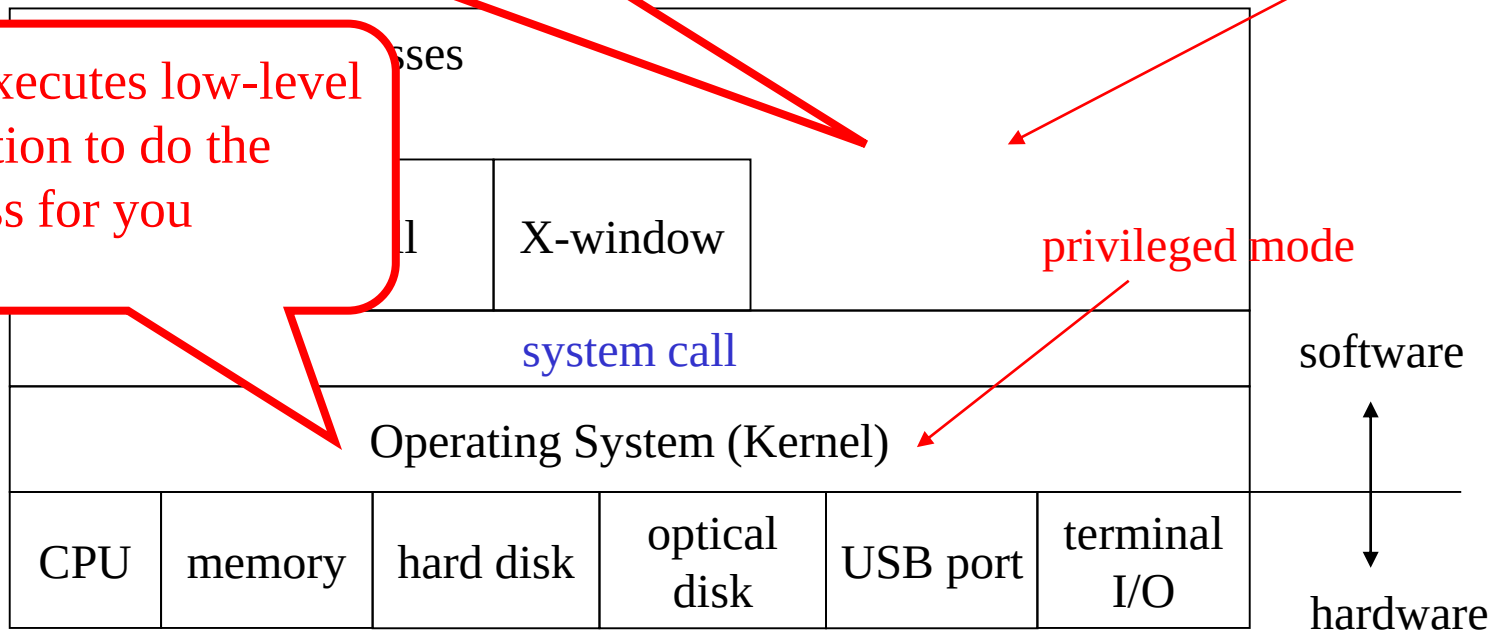
You cannot access the disk using assembly with direct I/O control



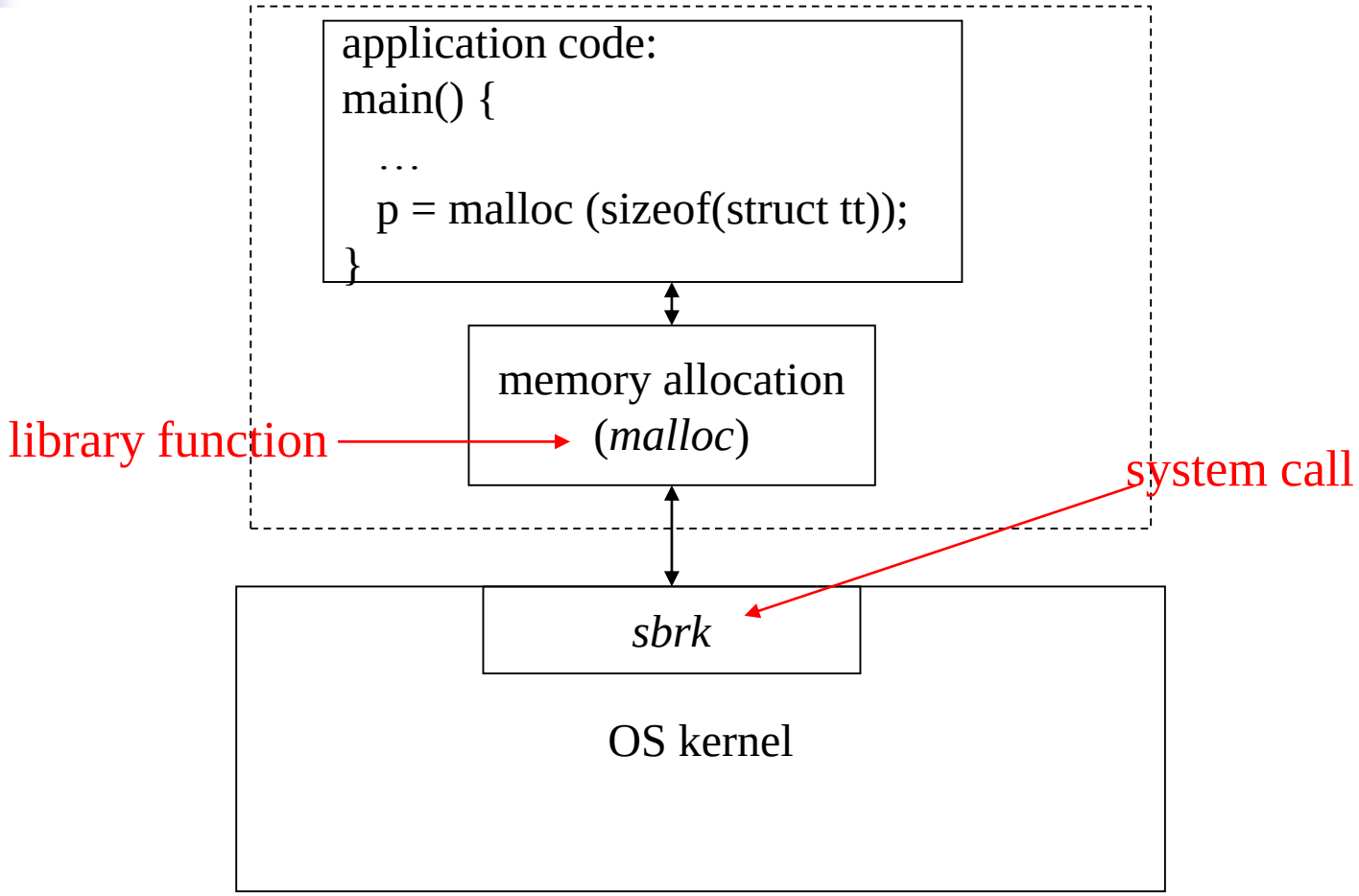
UNIX Architecture

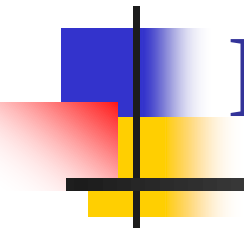
How should I access the disk?
- call the OS to do it for you
- through system call (e.g. write ())

The OS executes low-level I/O operation to do the disk access for you



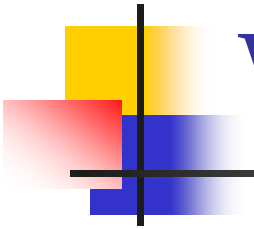
UNIX Architecture: from the view point of a single program





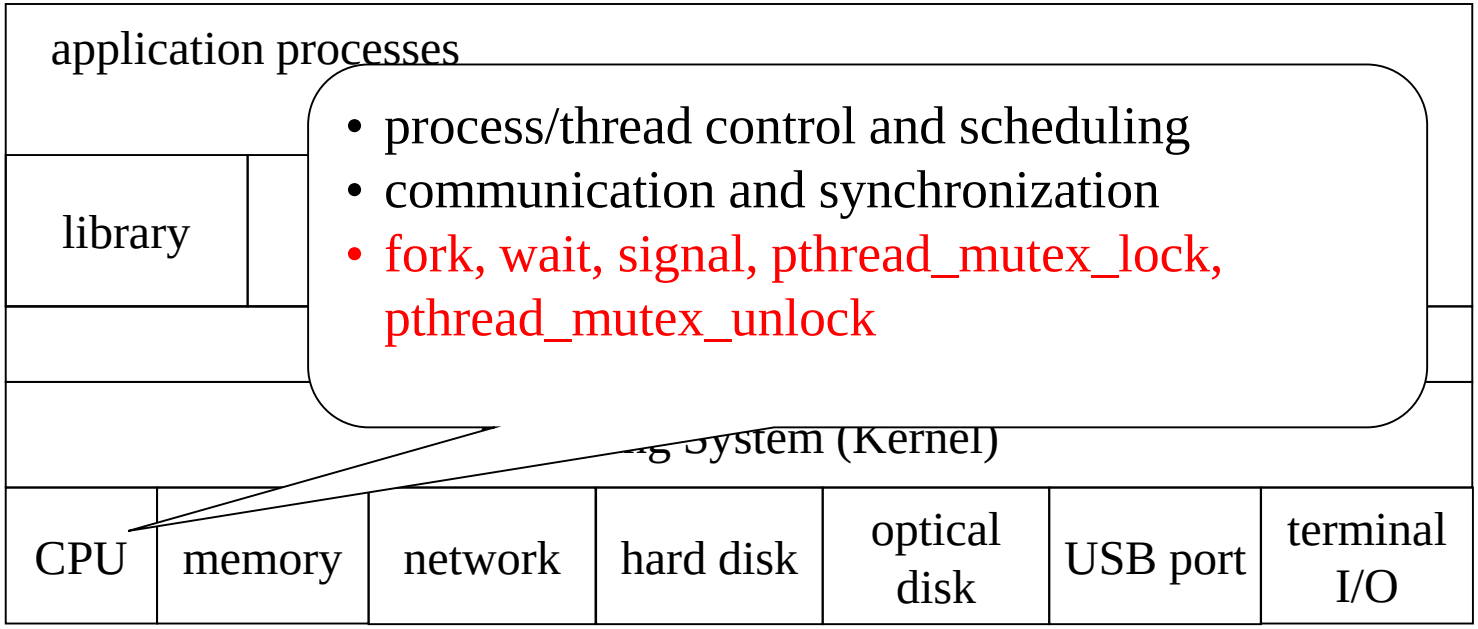
Basic concepts of a process

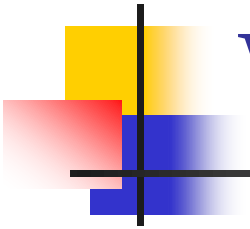
What an OS should provides?



- for general users: ease to use application programs
- for programmers: ease of programming over all hardware resources
  - manage CPUs and processes/threads scheduling
  - memory and storage devices management (e.g. virtual memory, file system, etc.)
  - API (application programmer interfaces) to I/O devices
- security and users protection

What services UNIX provides

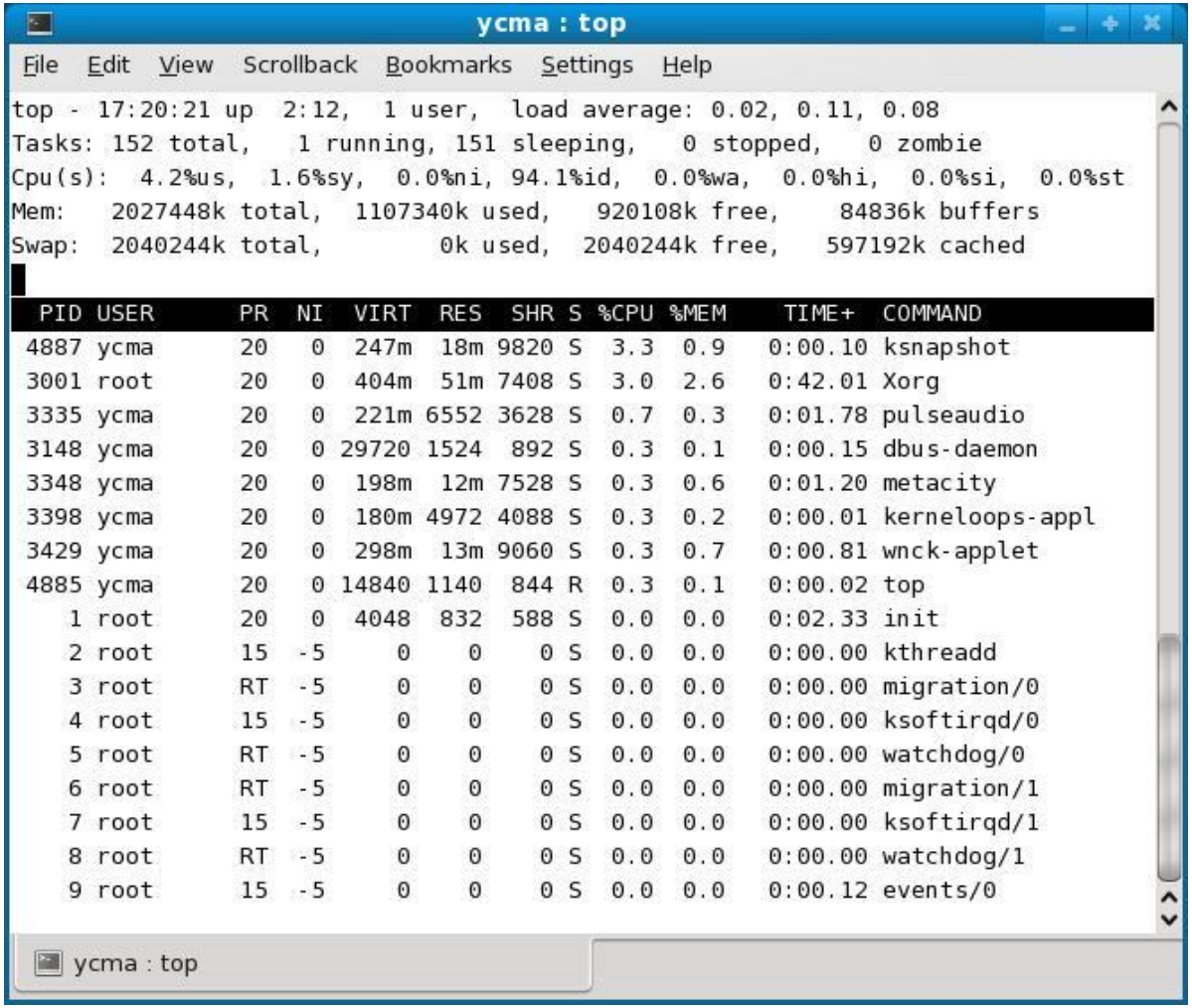




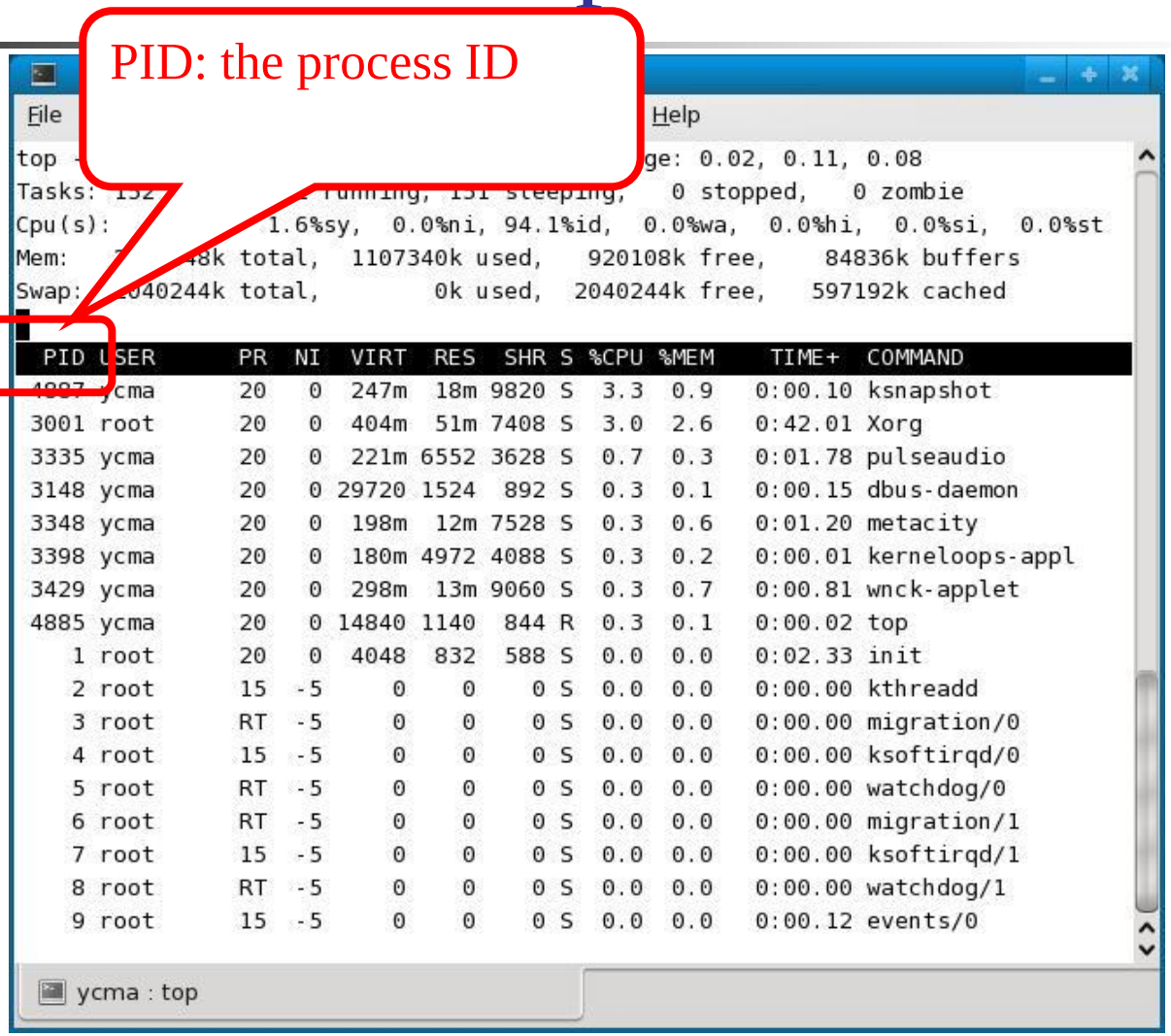
What is a process

- an instance of program been executed
  - process ID
  - it's own virtual memory space
  - files opened, I/O devices accessed, etc.
  - …and a lot…
- there may be 2 or more processes corresponding to the same program
- Example:
  - run an infinite-loop program
  - find out what a process is in "top"

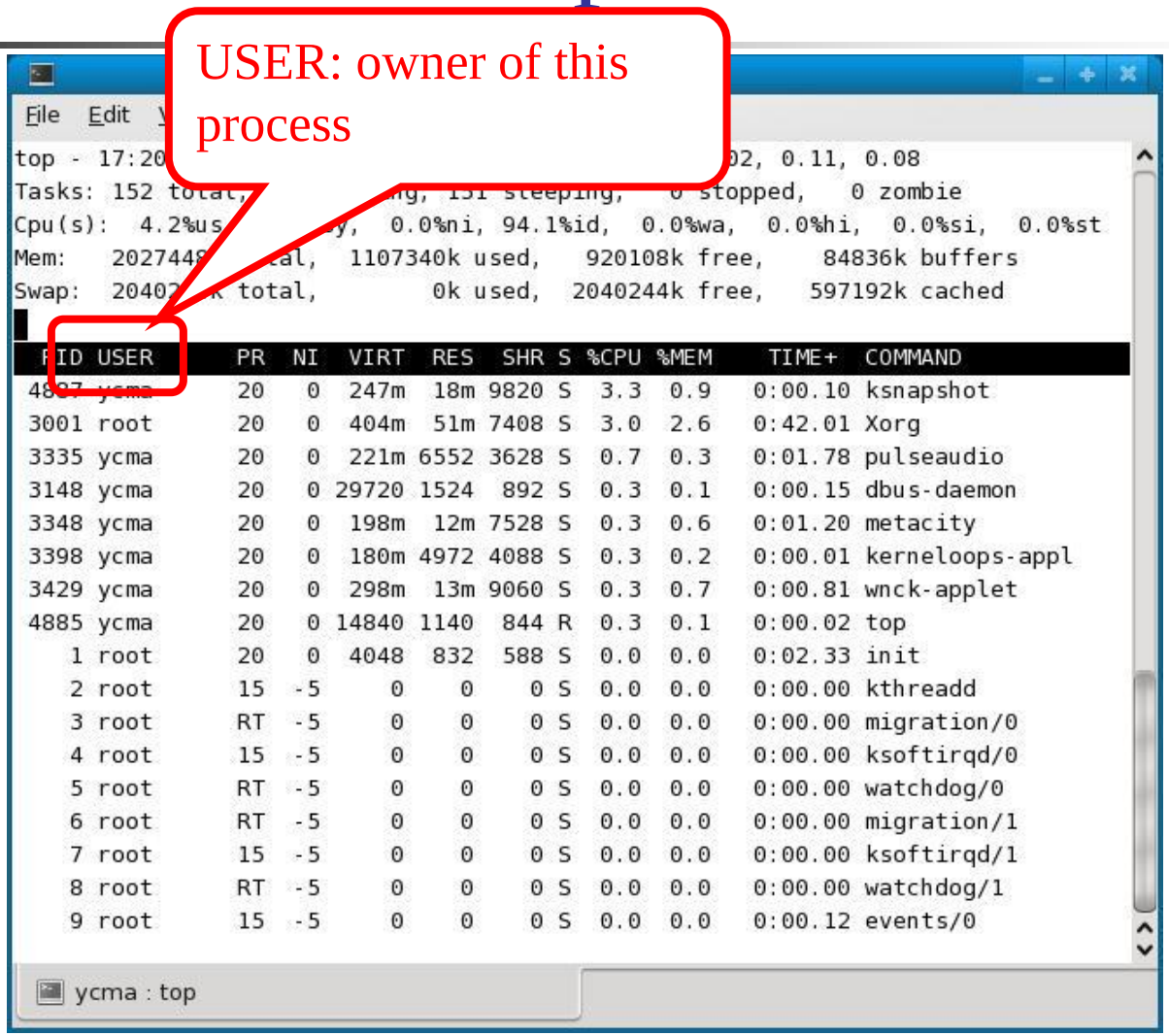
Attributes of a process



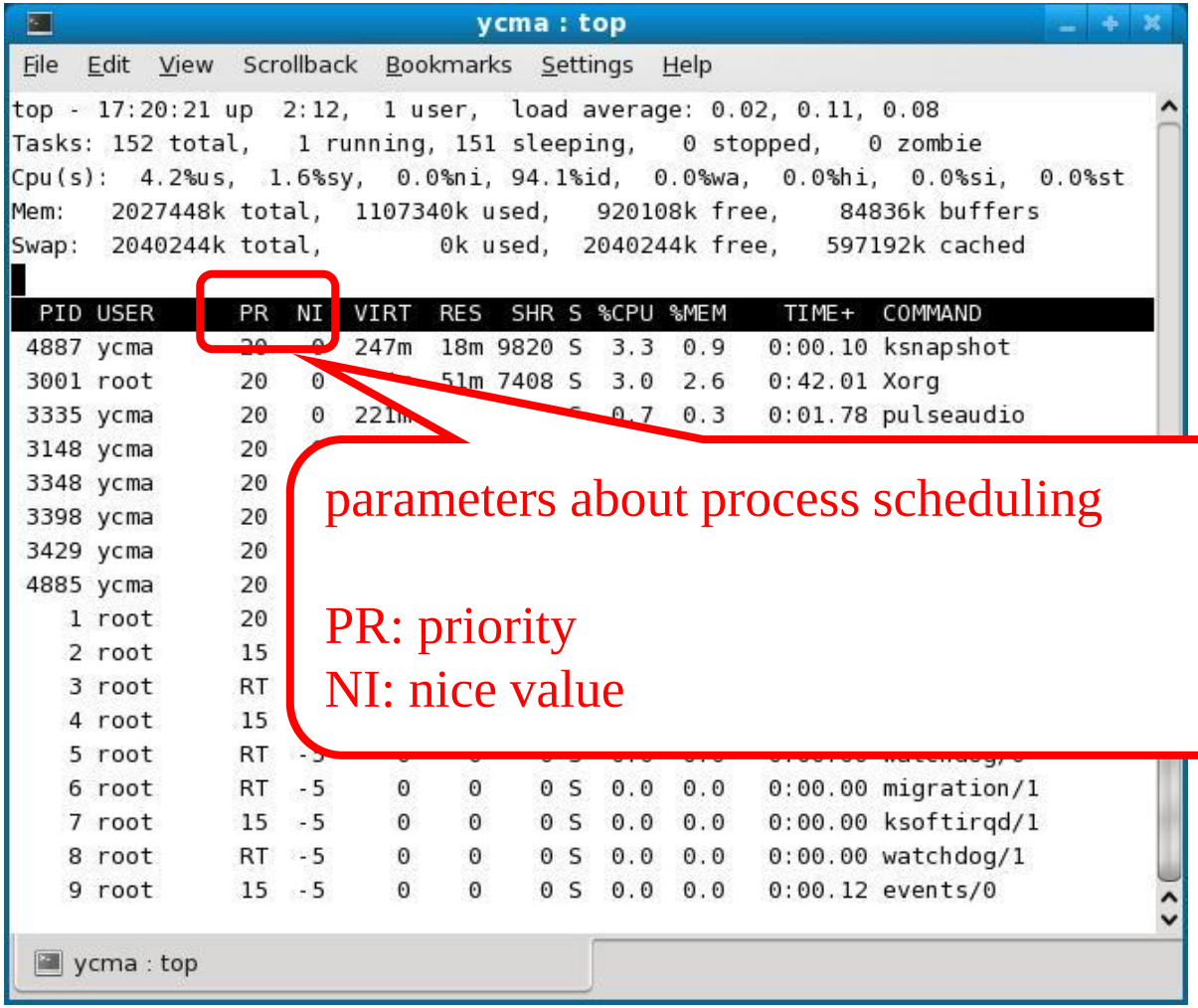
Attributes of a process



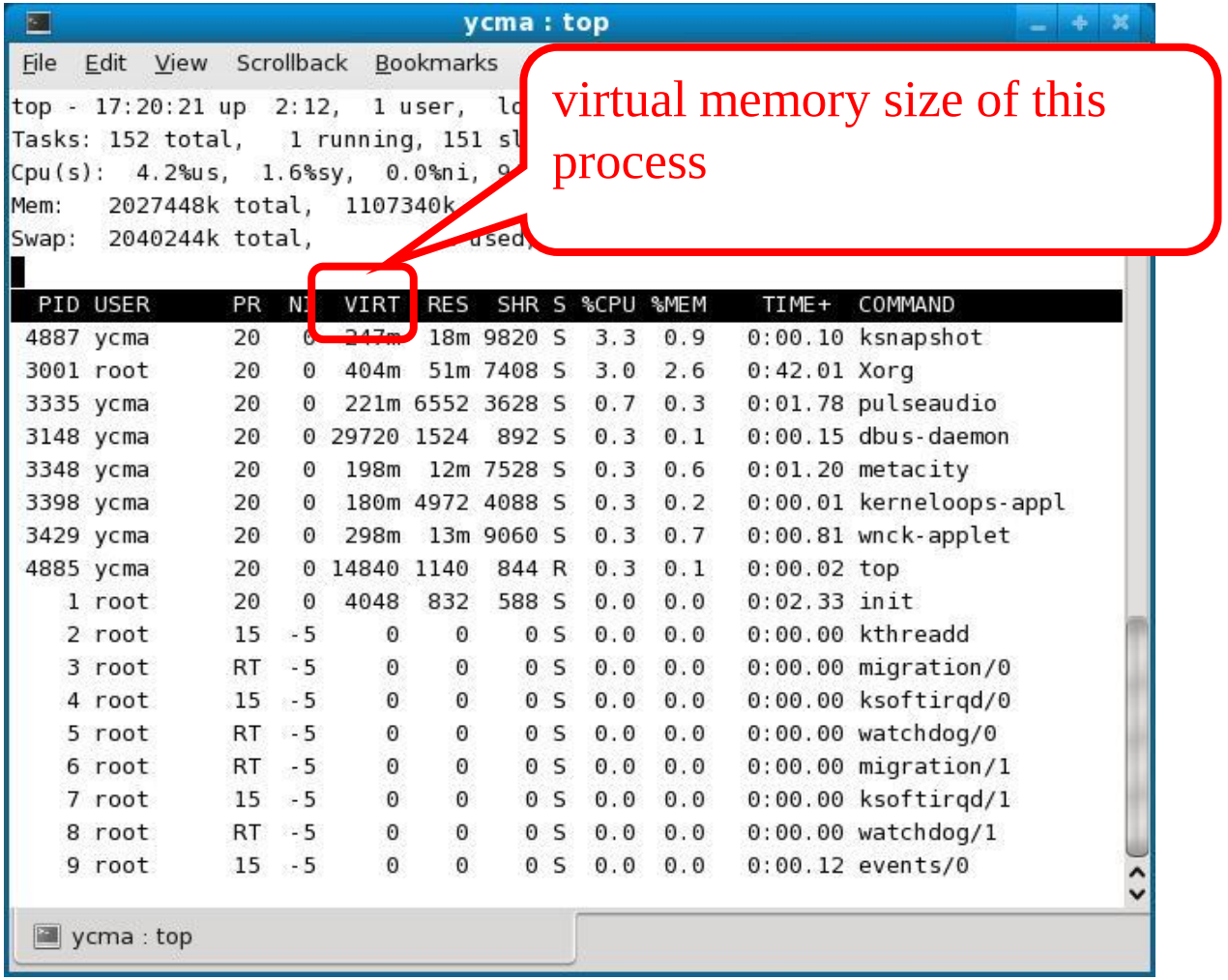
Attributes of a process



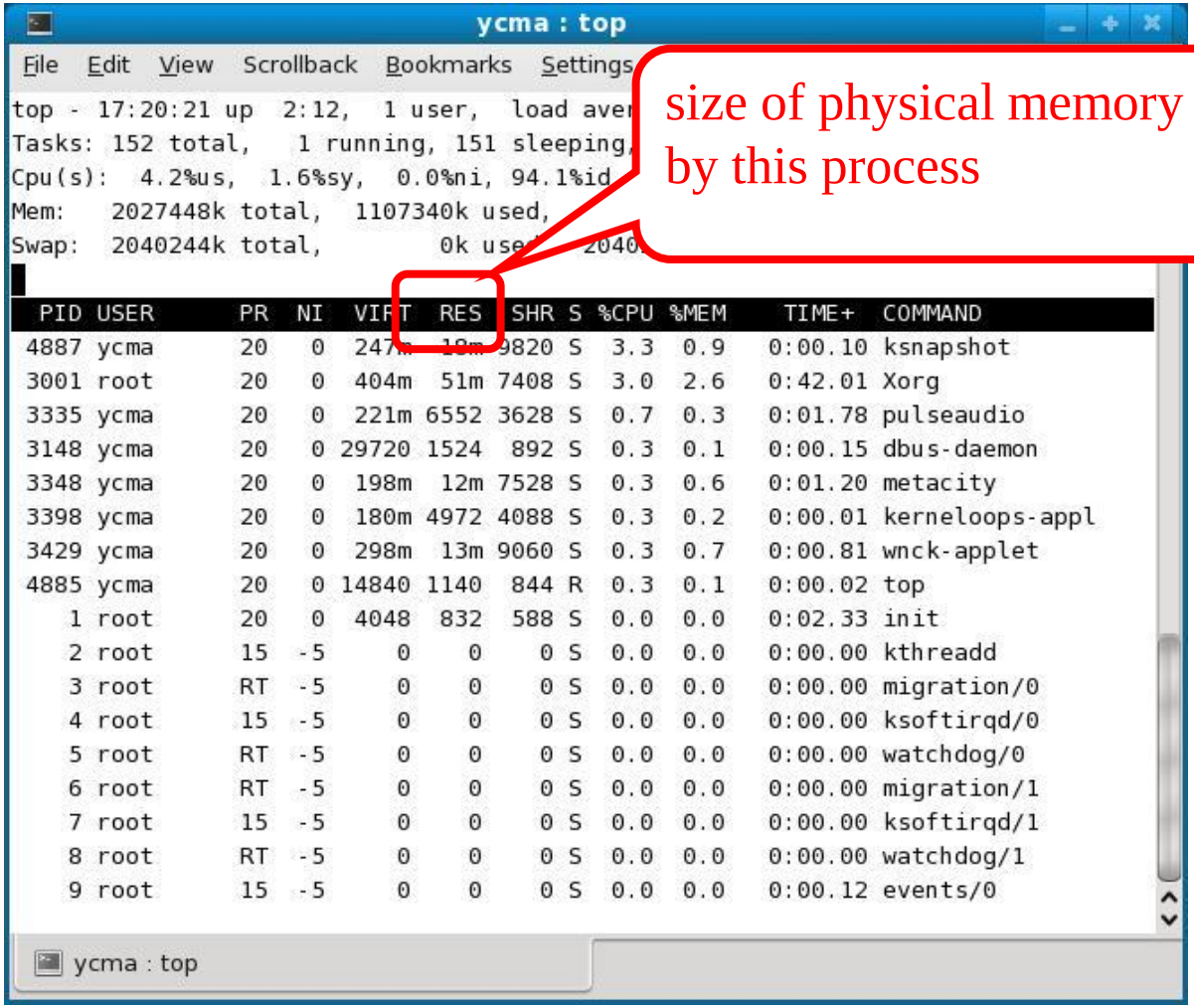
Attributes of a process



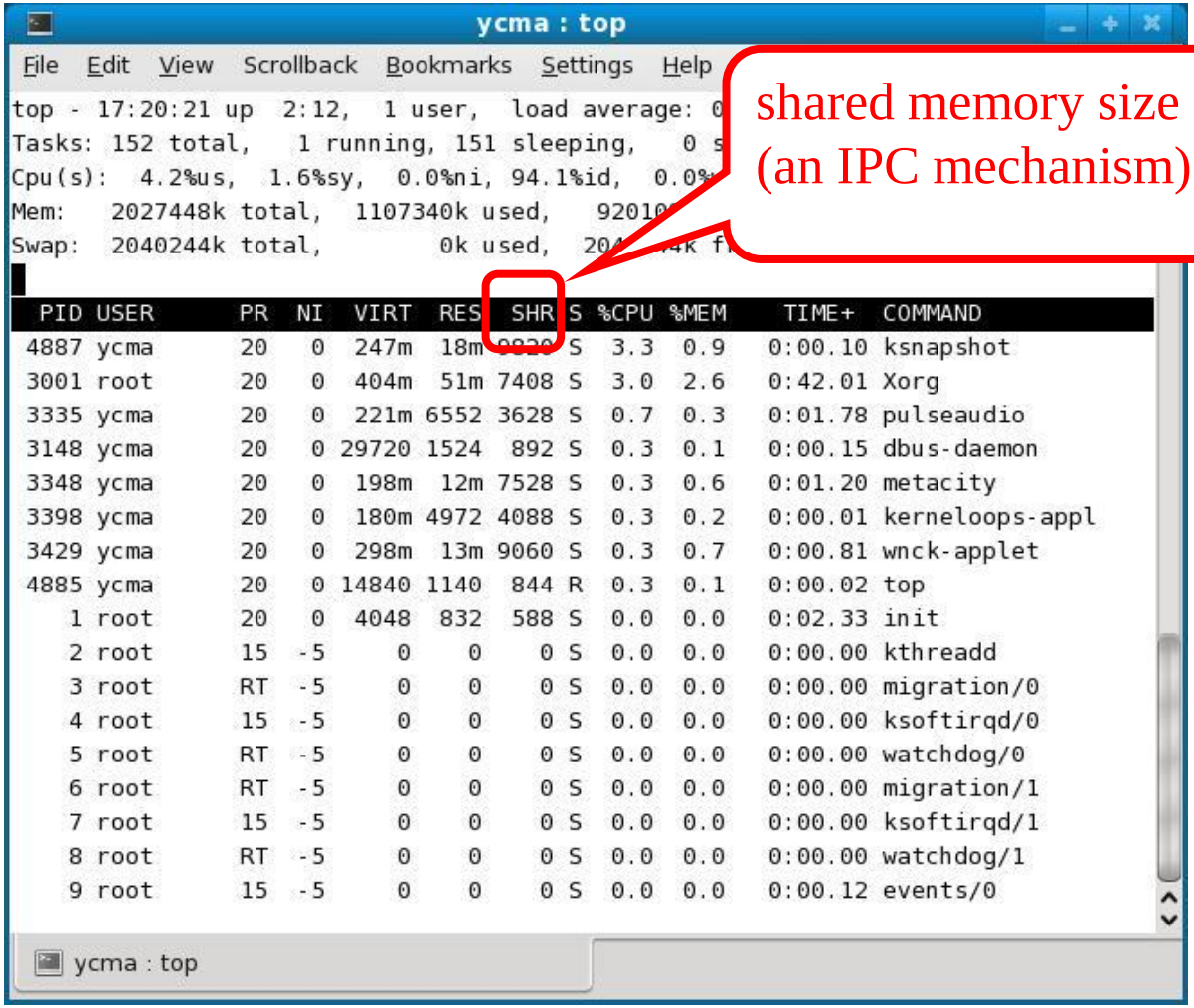
Attributes of a process



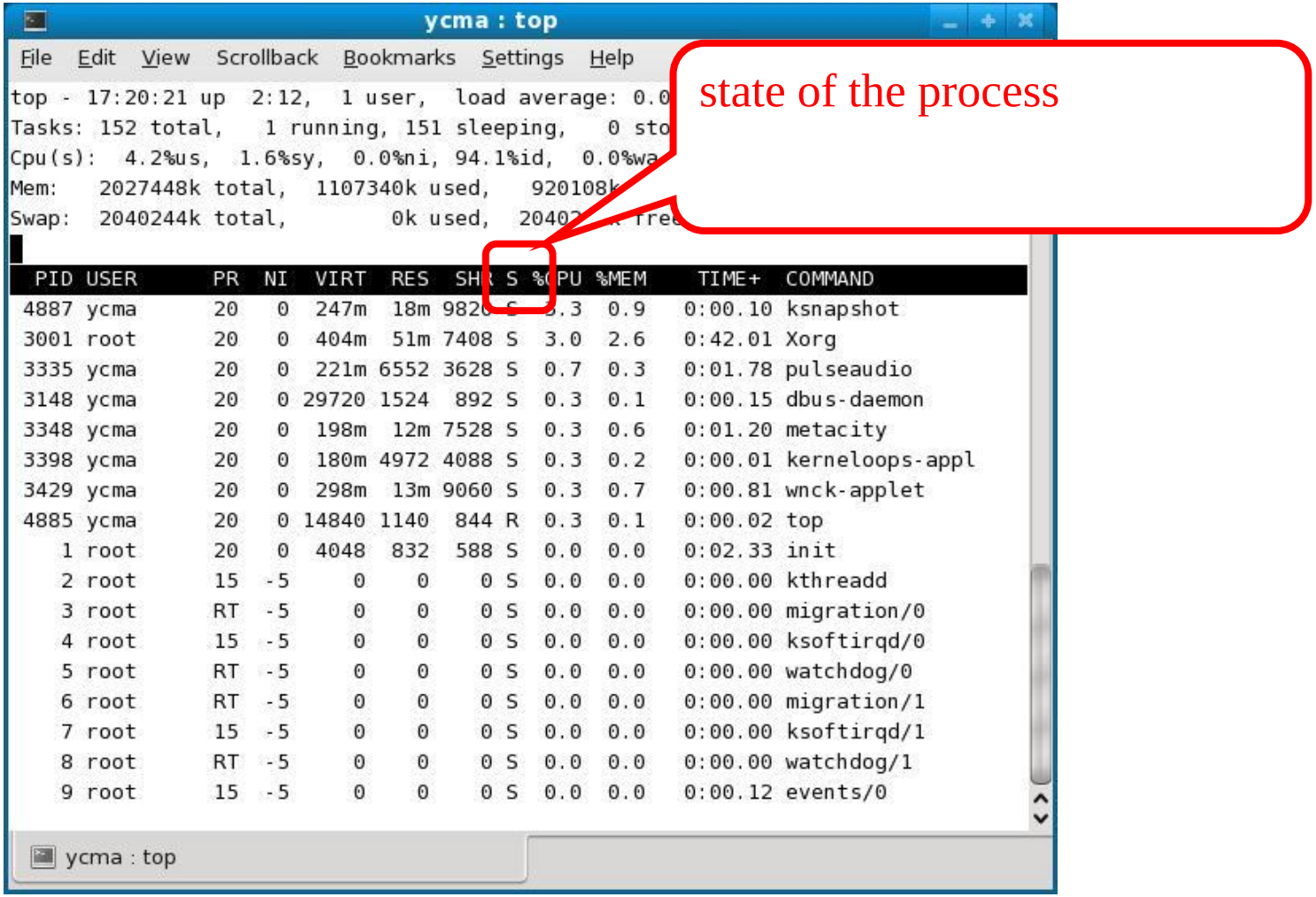
Attributes of a process

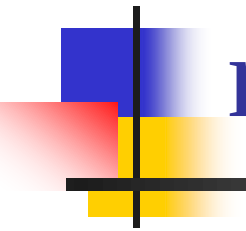


Attributes of a process



Attributes of a process



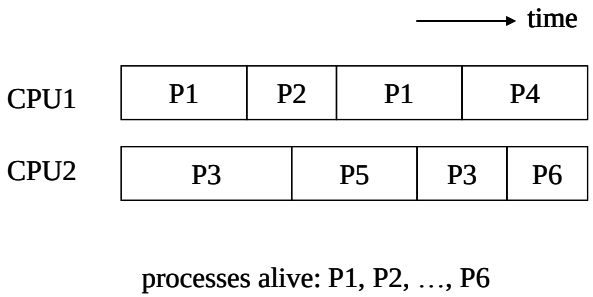


Process scheduling and management of CPUs (1)

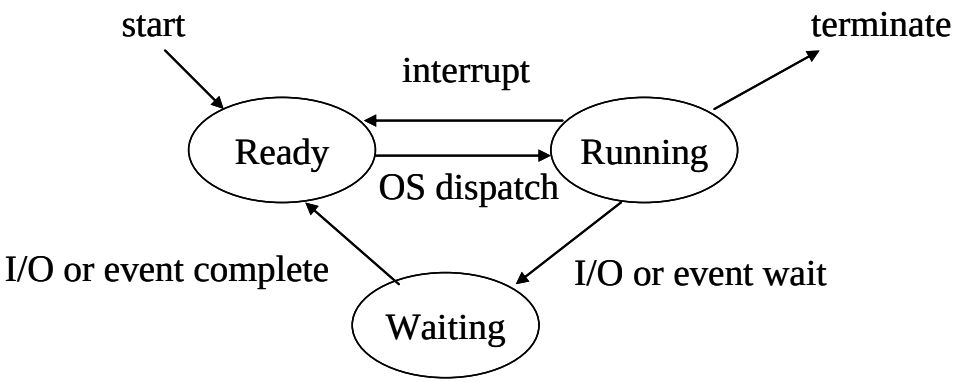
what the OS textbook tells you

What the OS does to manage your CPUs and processes

- executing processes with time-sharing scheme

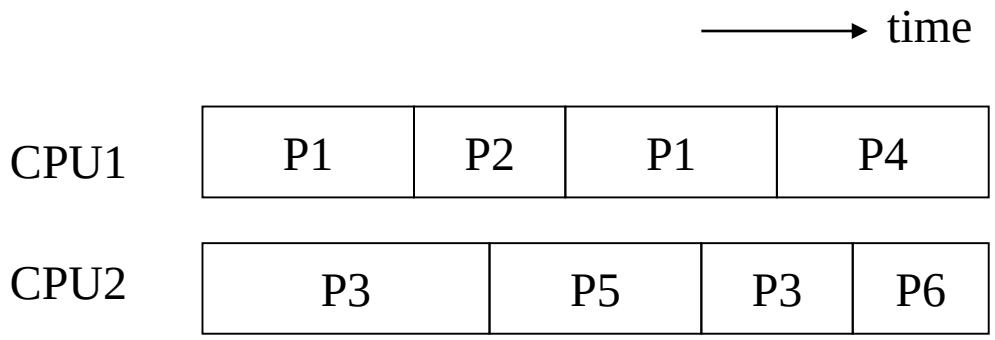


- a process may be in one of these states



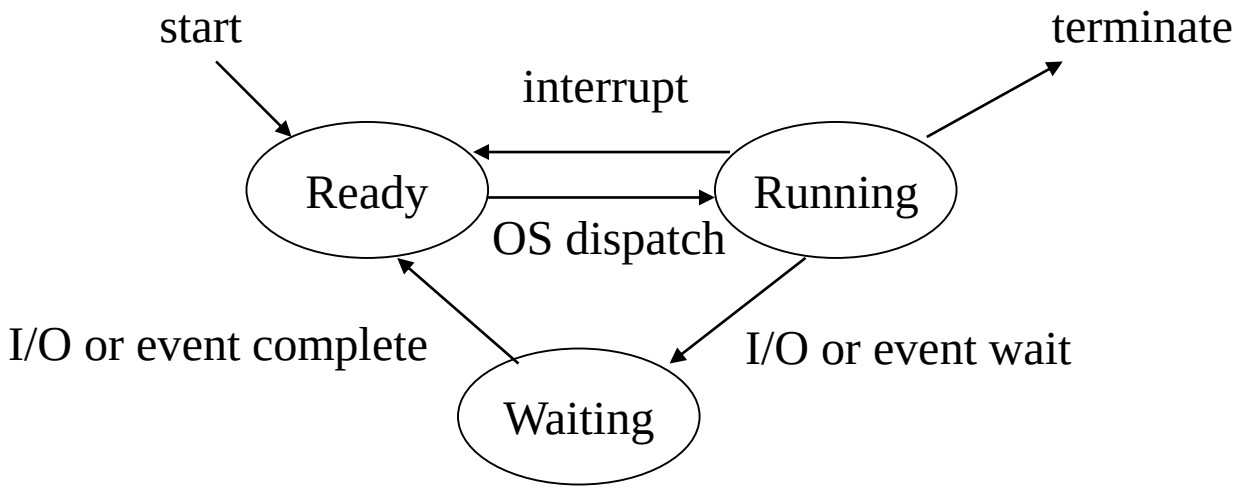
Time sharing

- the OS schedules a set of processes to run on CPUs
- at any time, only one process is running on an CPU



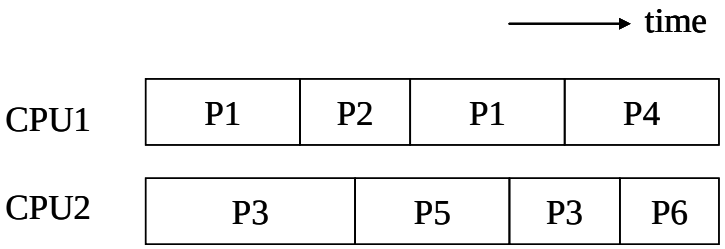
processes alive: P1, P2, …, P6

States of a process

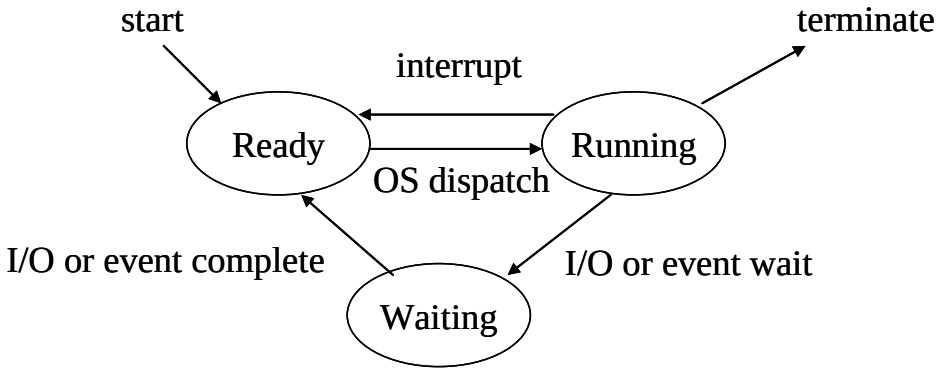


Process scheduling with states

- Consider process P1

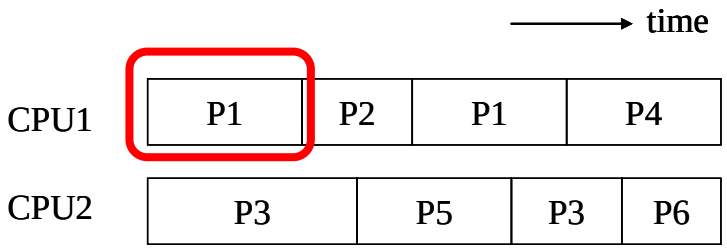


processes alive: P1, P2, …, P6

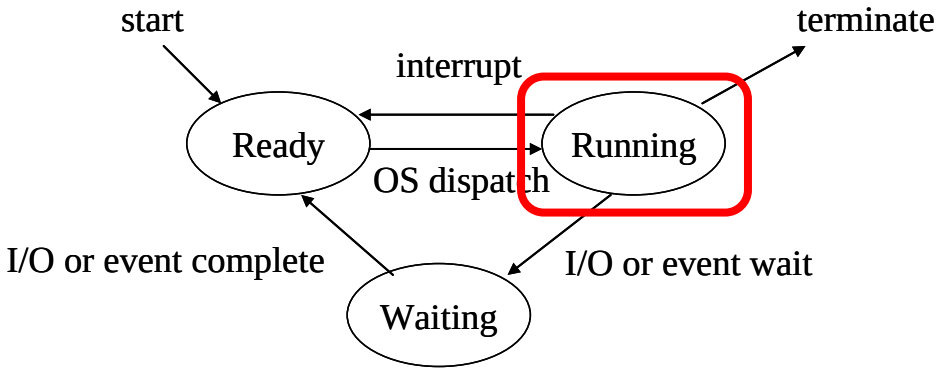


Process scheduling with states

- P1 in running state and occupies a CPU

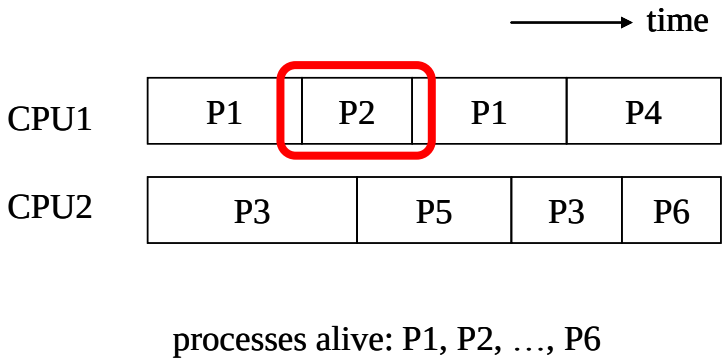


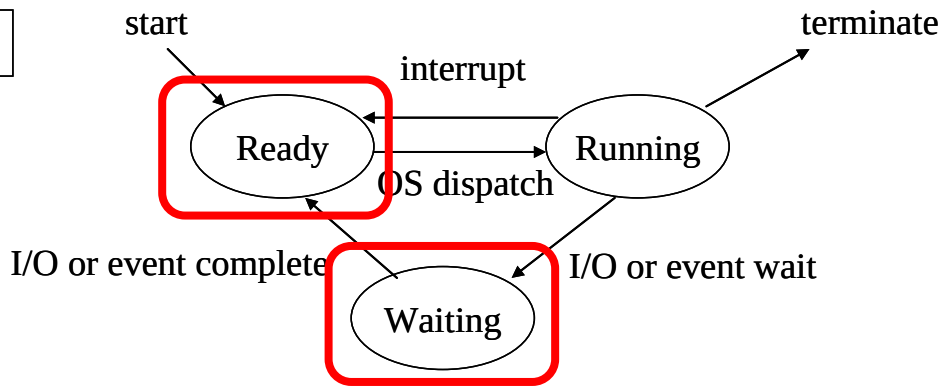
processes alive: P1, P2, …, P6



Process scheduling with states

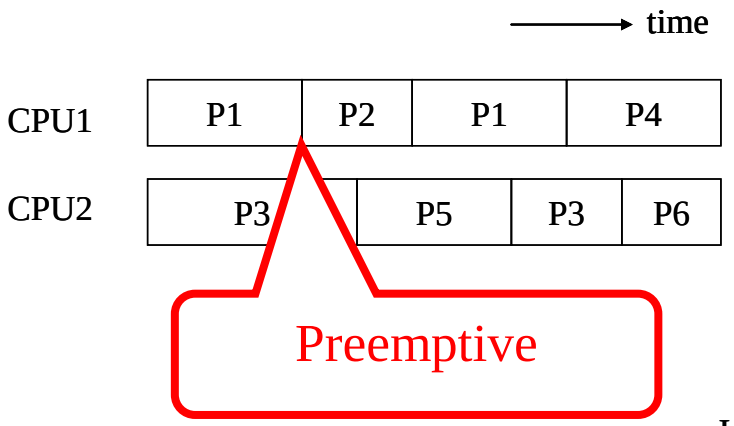
- P1 enters Waiting/Ready state due to
  - OS schedules, or
  - waiting for an I/O

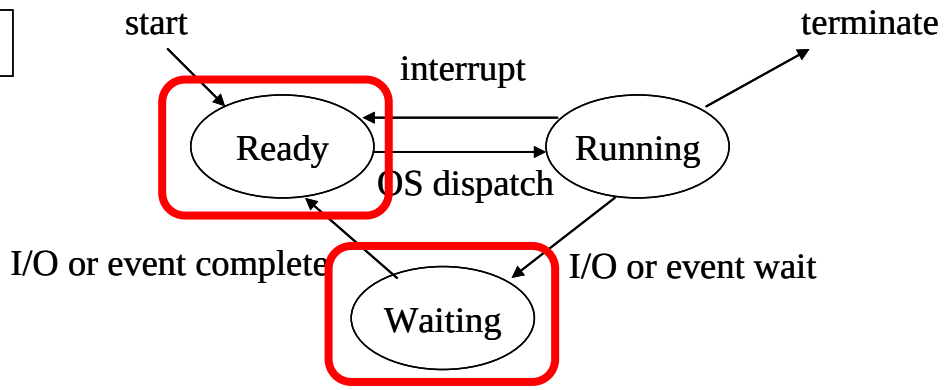




Process scheduling with states

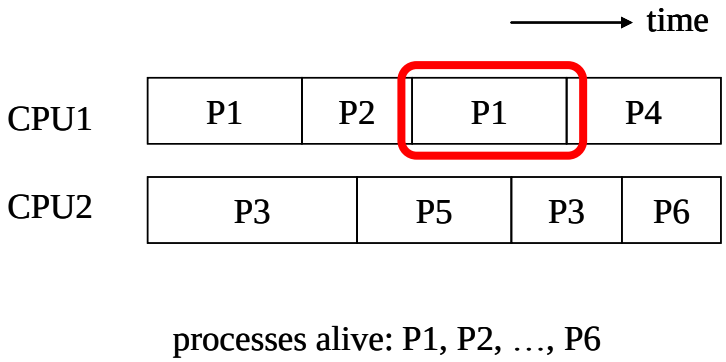
- P1 enters Waiting/Ready state due to
  - OS schedules, or
  - waiting for an I/O



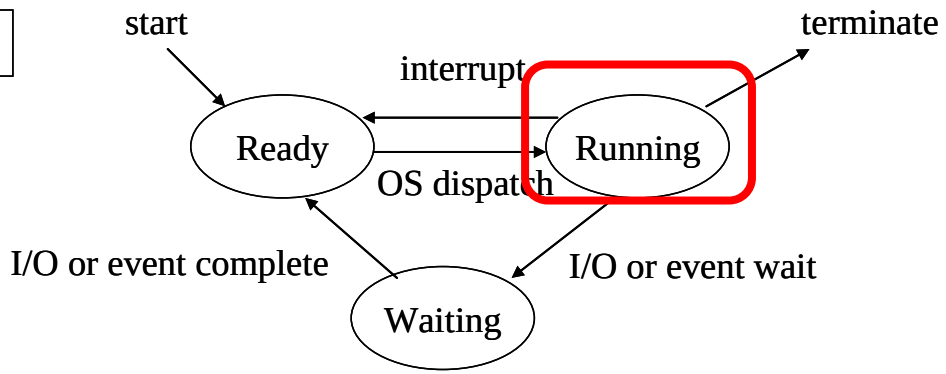


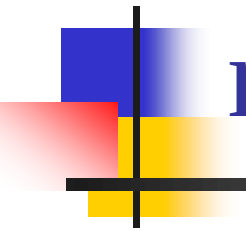
Process scheduling with states

- P1 re-gains the CPU due to
  - OS schedules, or
  - waiting for an I/O



processes alive: P1, P2, …, P6





Process scheduling and management of CPUs (2)

Let’s see a concrete example on Linux

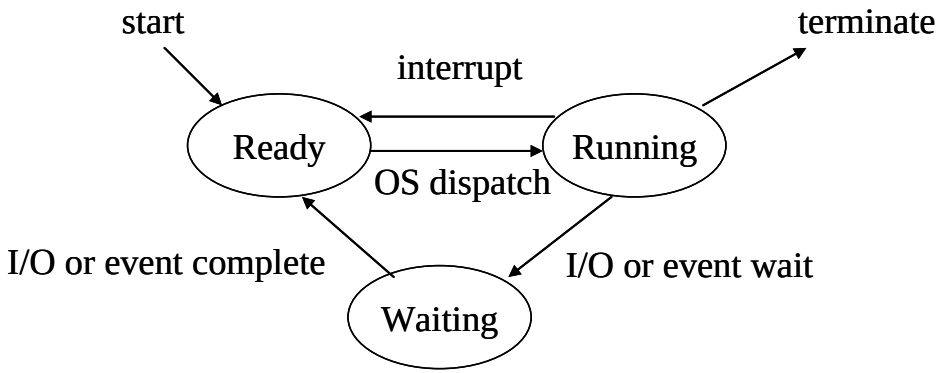
Example program

- Let's trace how the process state transfers with this program

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```



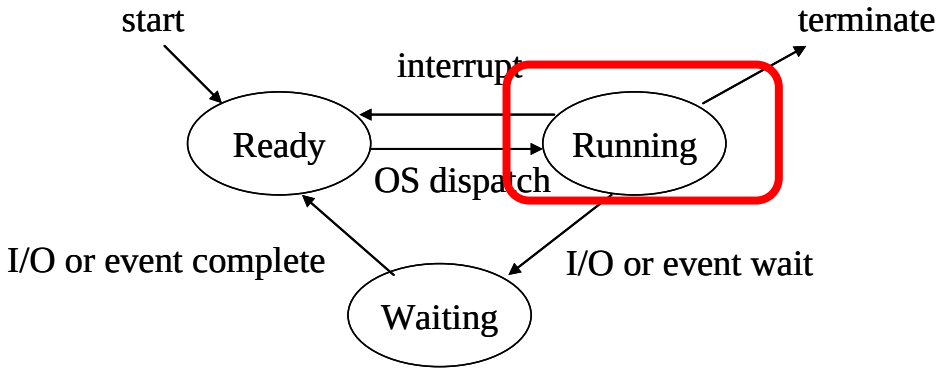
Example program

(1) At initial, the process is running

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```



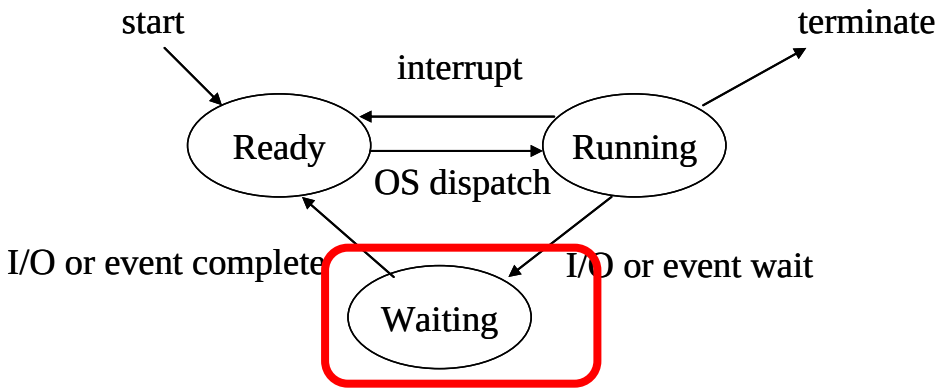
Example program

(2) Transfer to waiting state while waiting the user input from the keyboard

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```



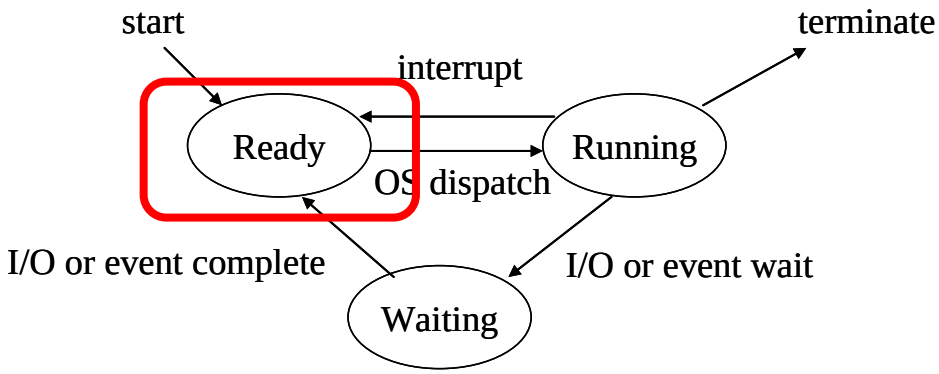
Example program

(3) Transfer to Ready state after the user inputs something from the keyboard

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```



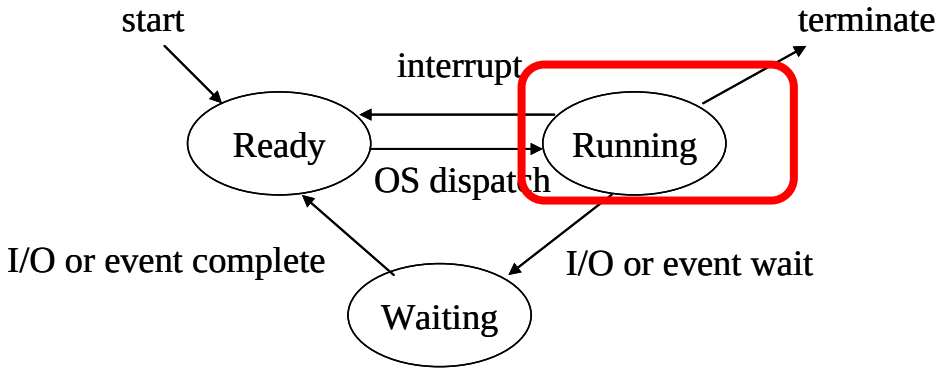
Example program

(4) Regain the CPU after the OS picks it up

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```



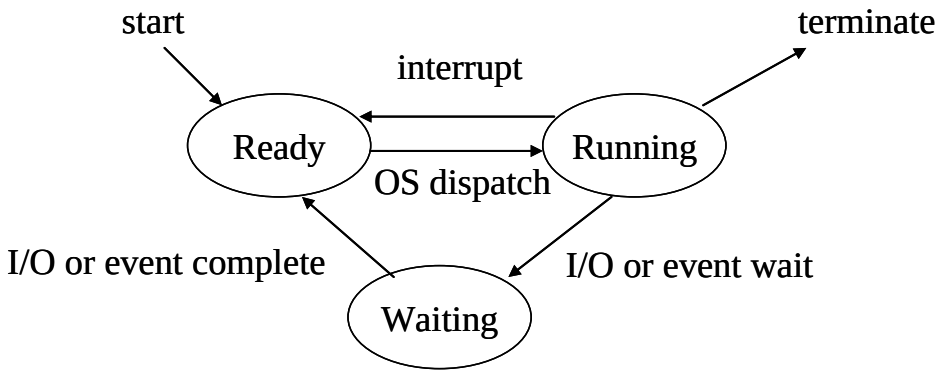
Example program

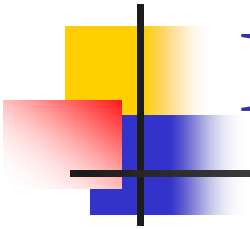
- Let's have a trial on Linux
  - demo/infinite_loop/demo2.c

```c
#include <stdio.h>

main ()
{
    int d;

    printf ("input an integer: ");
    scanf ("%d", &d);
    printf ("entering infinite loop\n");
    while (1);
    return 0;
}//main()
```

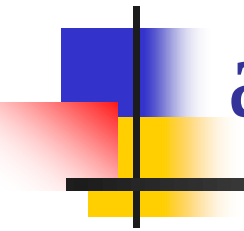




More on the process

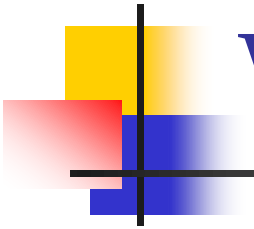
- inter-process communication
- process creation

Let’s explain other services of OS first



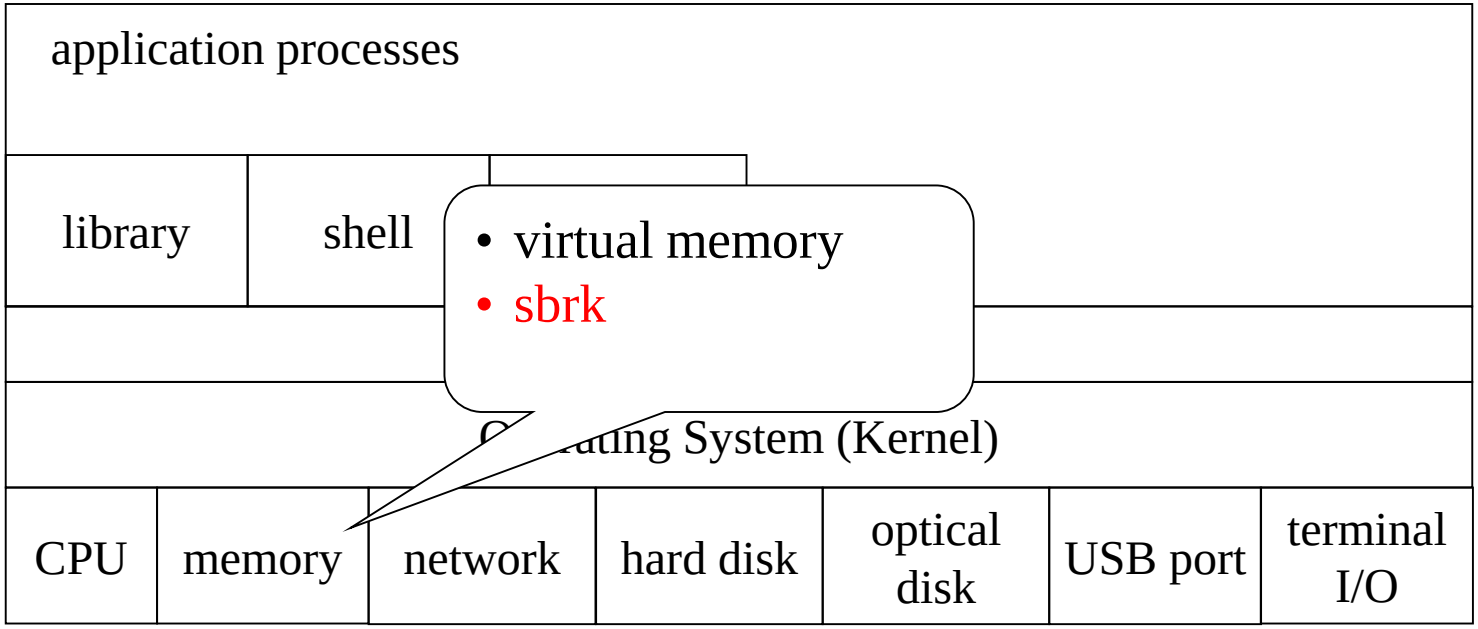
How the OS manages memory and storage devices

What an OS should provides?



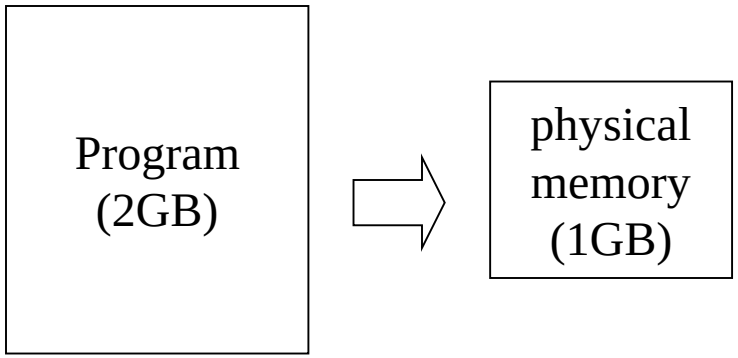
- for general users: ease to use application programs
- for programmers: ease of programming over all hardware resources
  - manage CPUs and processes/threads scheduling
  - memory and storage devices management (e.g. virtual memory, file system, etc.)
  - API (application programmer interfaces) to I/O devices
- security and users protection

What services UNIX provides



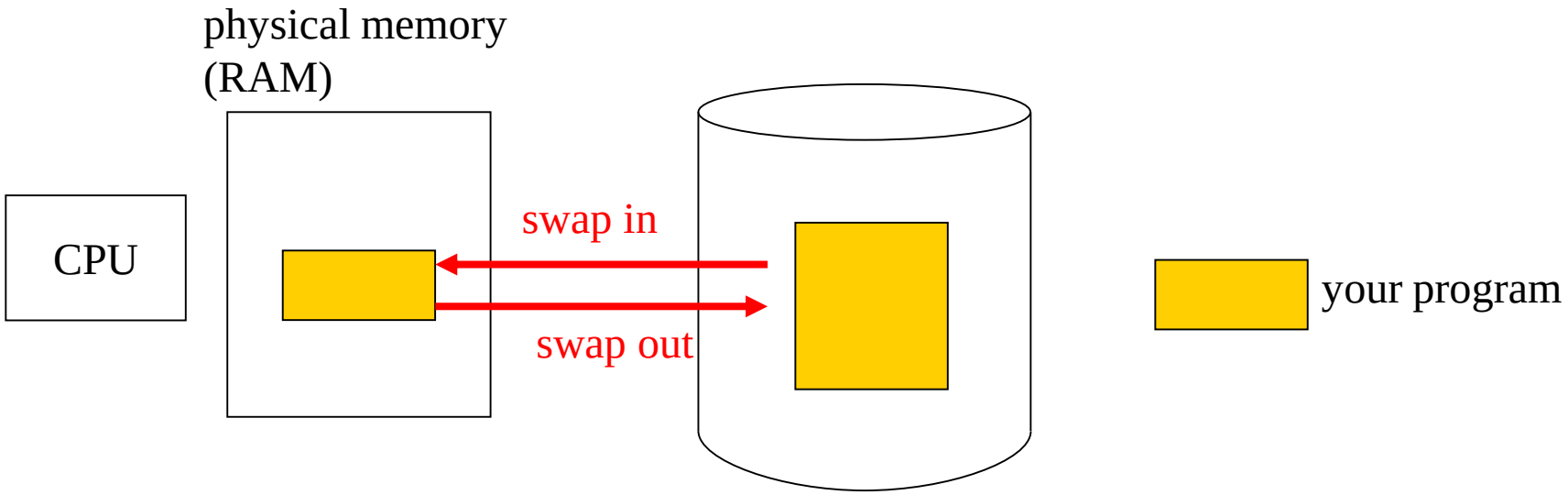
What is virtual memory

- a mechanism provided by OS+CPU
- Purpose:
  - let a program has individual memory space, not interfere with others
  - let you write a program larger than the size of physical memory

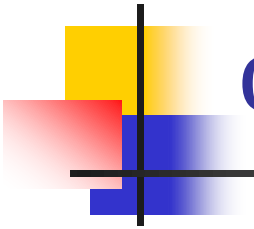


The magic of virtual memory

- use RAM + hard disk to run your program
  - RAM + hard disk > program size
- swap in currently required part
- swap out currently not required part

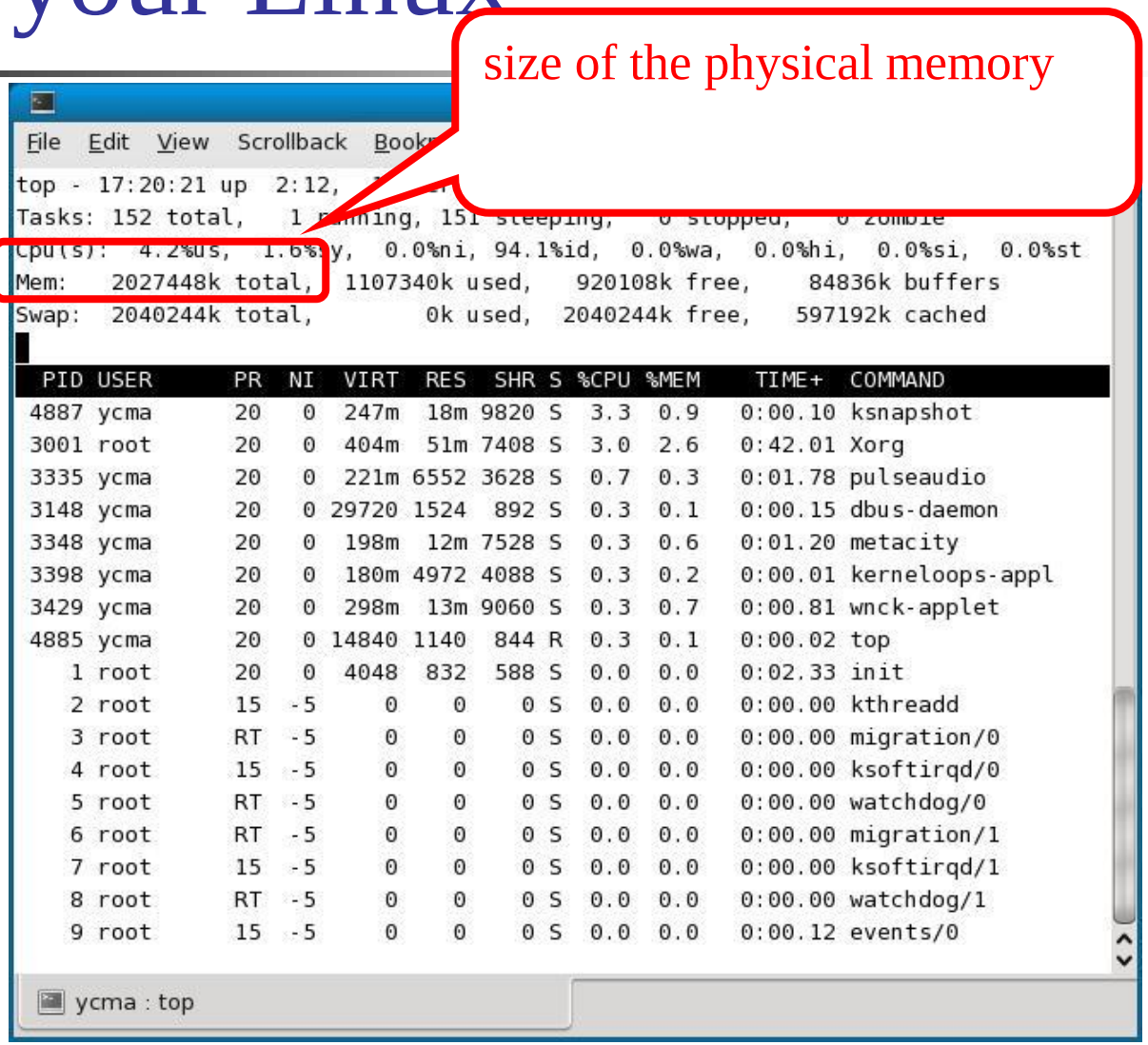


Check the virtual memory setup of your Linux

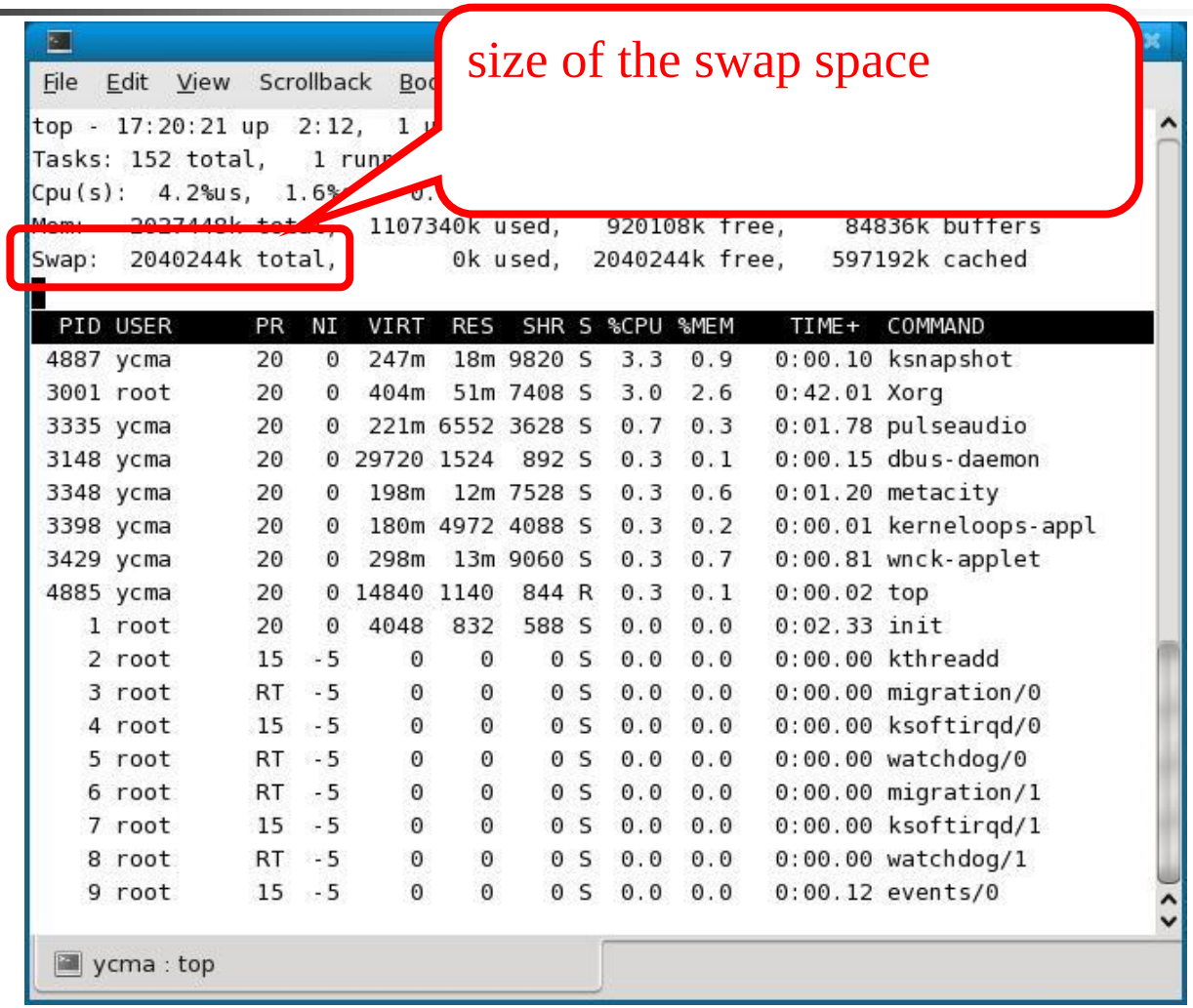


- try command "*top*"
- you can setup the size of swap space at install (setup disk partition)

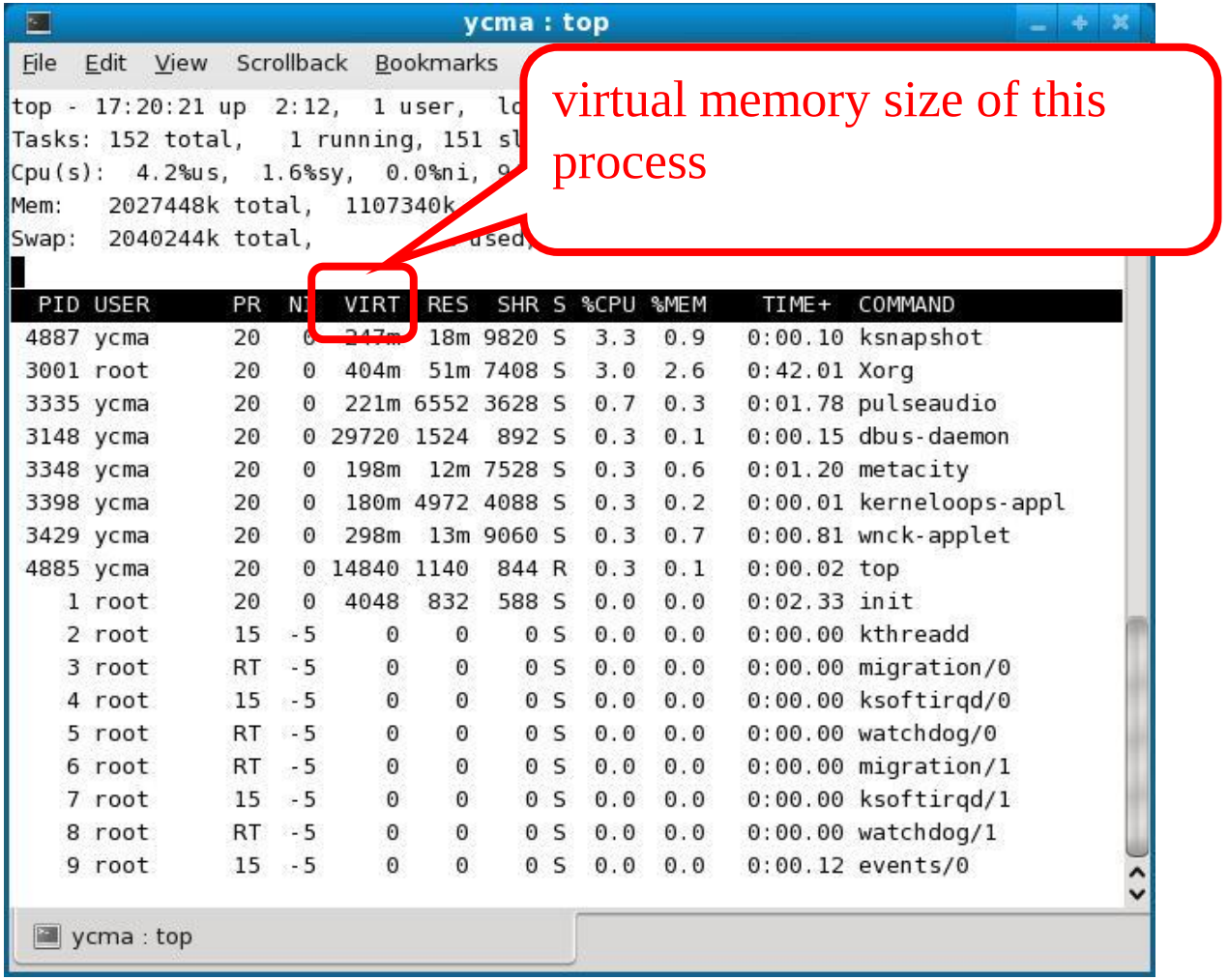
Check the virtual memory setup of your Linux



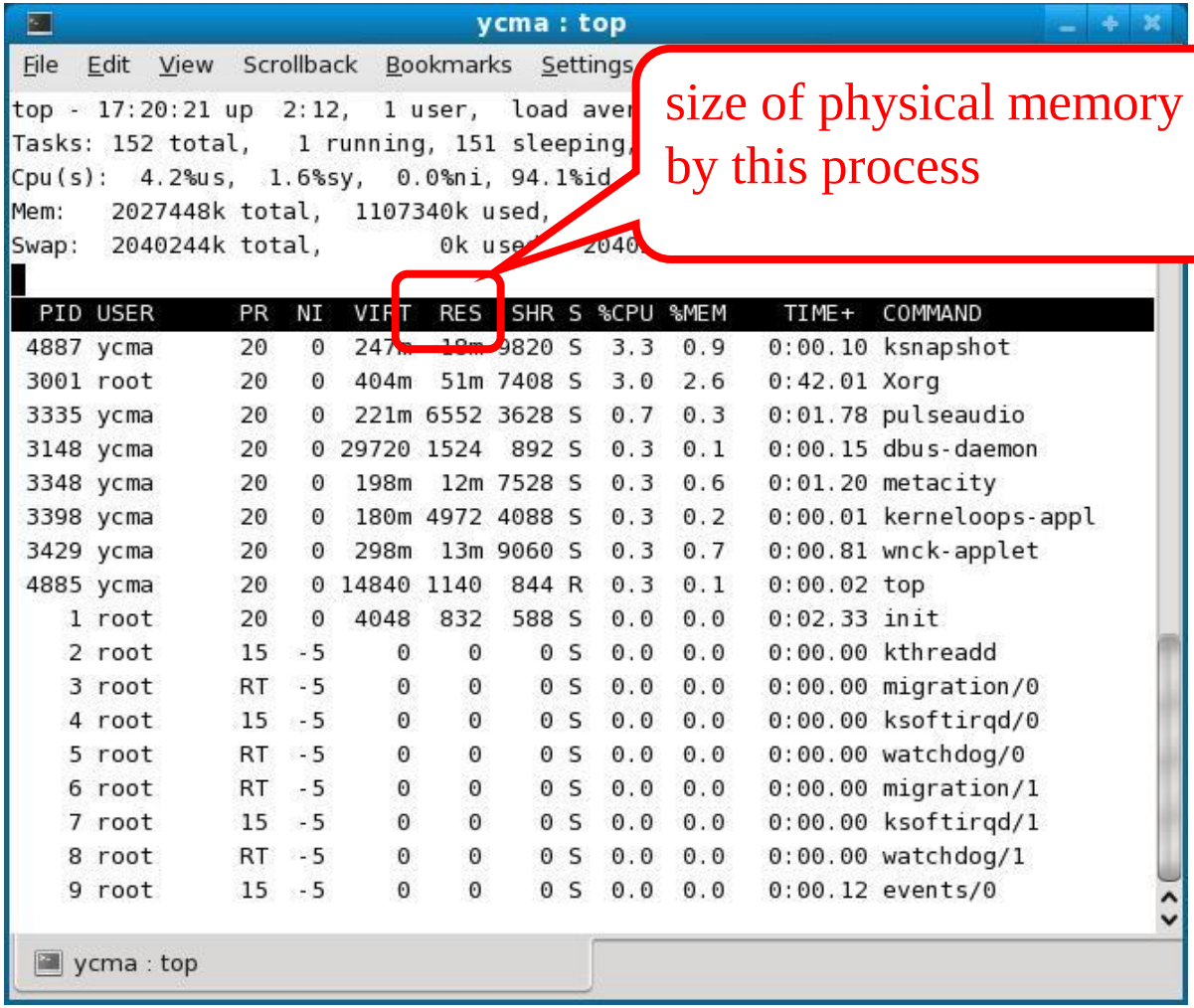
Check the virtual memory setup of your Linux



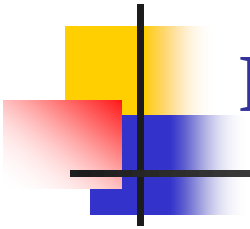
Attributes of a process



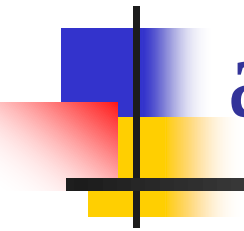
Attributes of a process



Example program for virtual memory

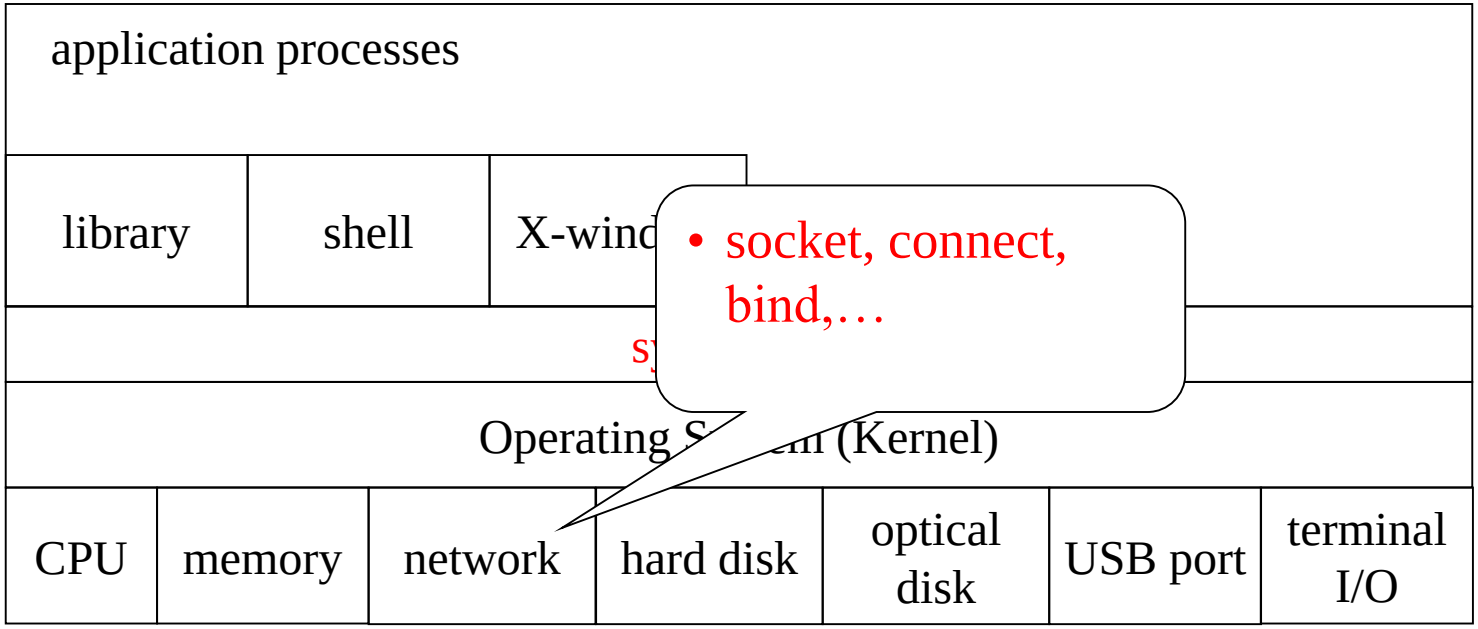


- demo/infinite_loop/vmem.c
  - a program accessing an arry of 2GB size
  - to execute on my laptop with only 1GB RAM



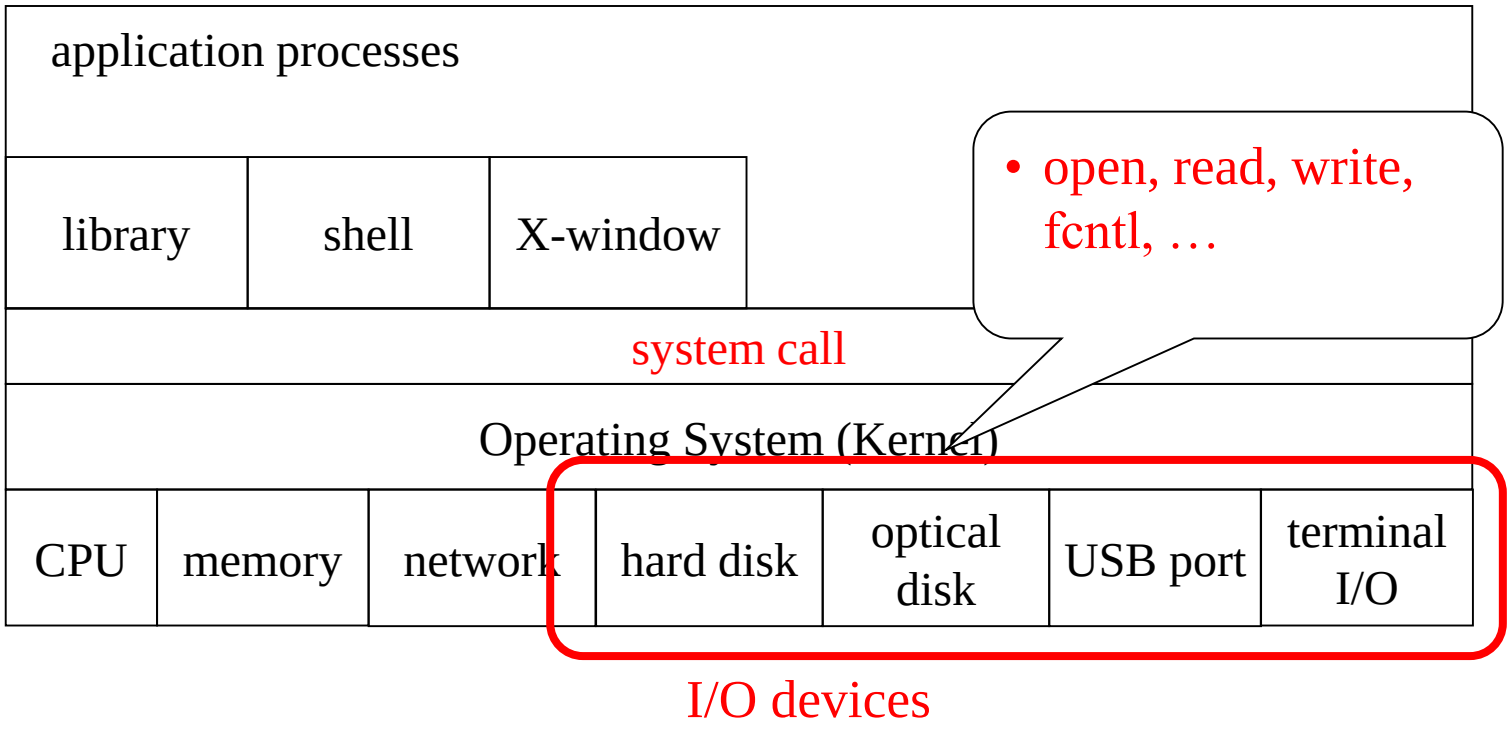
How UNIX manages networking and I/O devices

What services UNIX provides

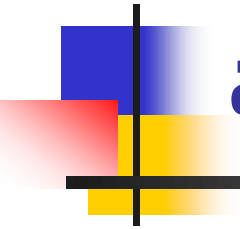


What services UNIX provides

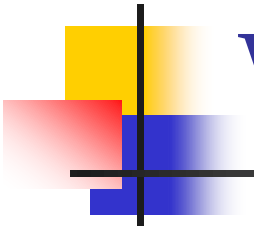
- I/O devices are accessed like files



How the OS provides security and protection

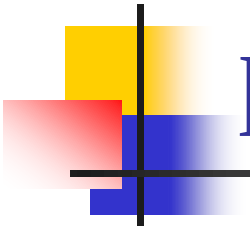


What an OS should provides?



- for general users: ease to use application programs
- for programmers: ease of programming over all hardware resources
- security and users protection
  - access permission for files, I/O devices for multi-users
  - network security

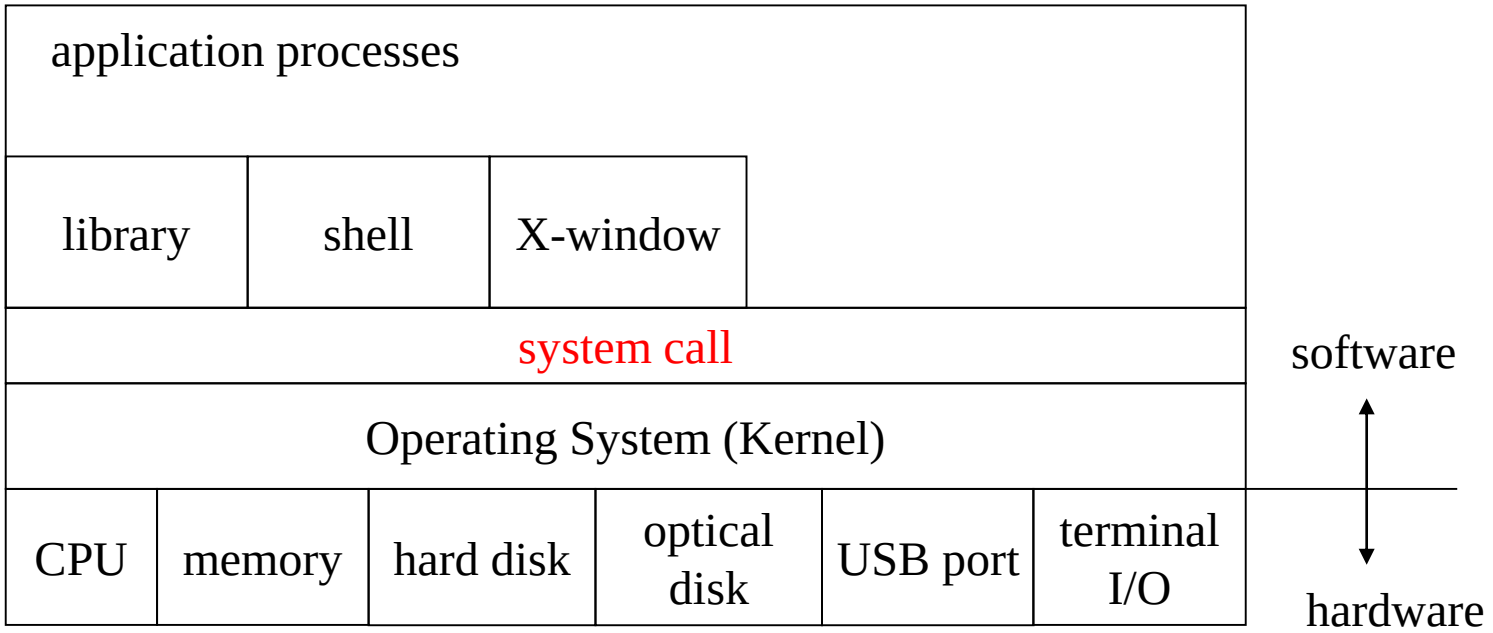
Example of security and protection

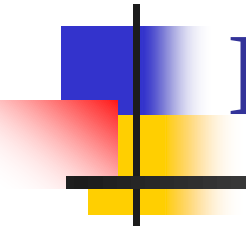


- trying to access other ones file
  - check the file permission

UNIX provides protection through *system call*

- system call: a special function call provided by OS
- all hardware resources can only be accessed through system calls
- the system call checks for access permission



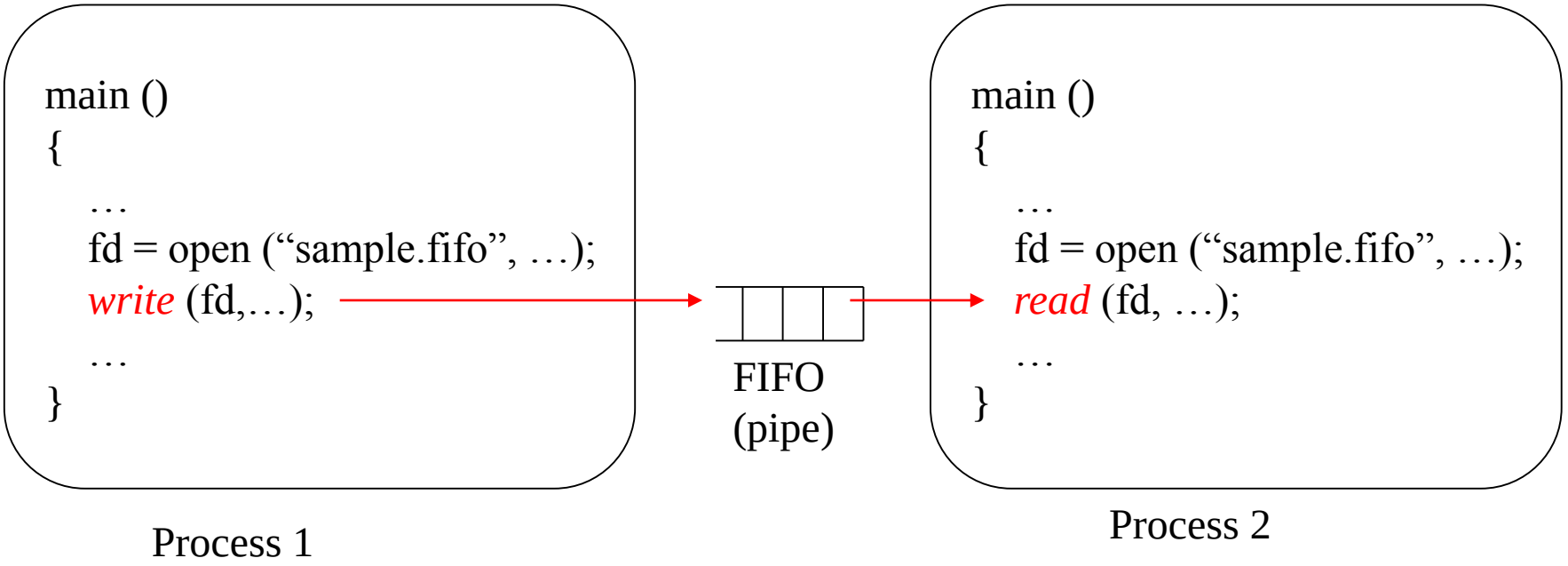


More on process

How to create a new process and execute a program

The IPC (inter-process communication)

Processes may also communicate like accessing files



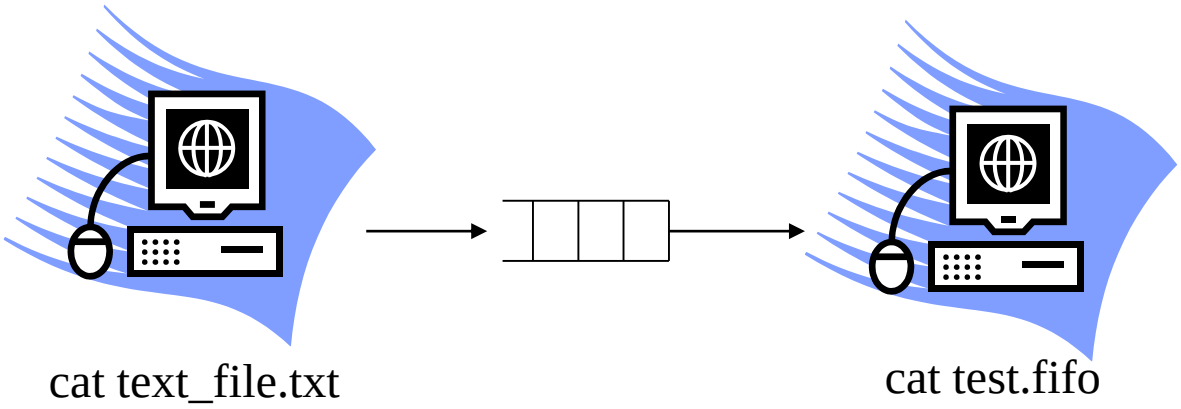
A simple exercise on using FIFO

- Step 1: execute command

  mkfifo test.fifo
- Step 2: open a terminal and execute command

  cat test.fifo
- Step 3: open another terminal and execute command

  cat text_file.txt > test.fifo

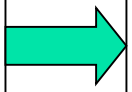


How to create a new process?

- *fork* and generate a process with identical content

Parent Process

```
pid = fork ();
if (pid==0) {//child process
    execlp (filename, filename, (void*)0);
    printf ("Error if you see this line\n");
}
else { //parent process
    printf ("This is in parent process\n");
}
```



Child Process

```
pid = fork ();
if (pid==0) {//child process
    execlp (filename, filename, (void*)0);
    printf ("Error if you see this line\n");
}
else { //parent process
    printf ("This is in parent process\n");
}
```

identical copy to the parent process

How to create a new process? (cont'd)

- But different PID results in different execution path

Parent Process

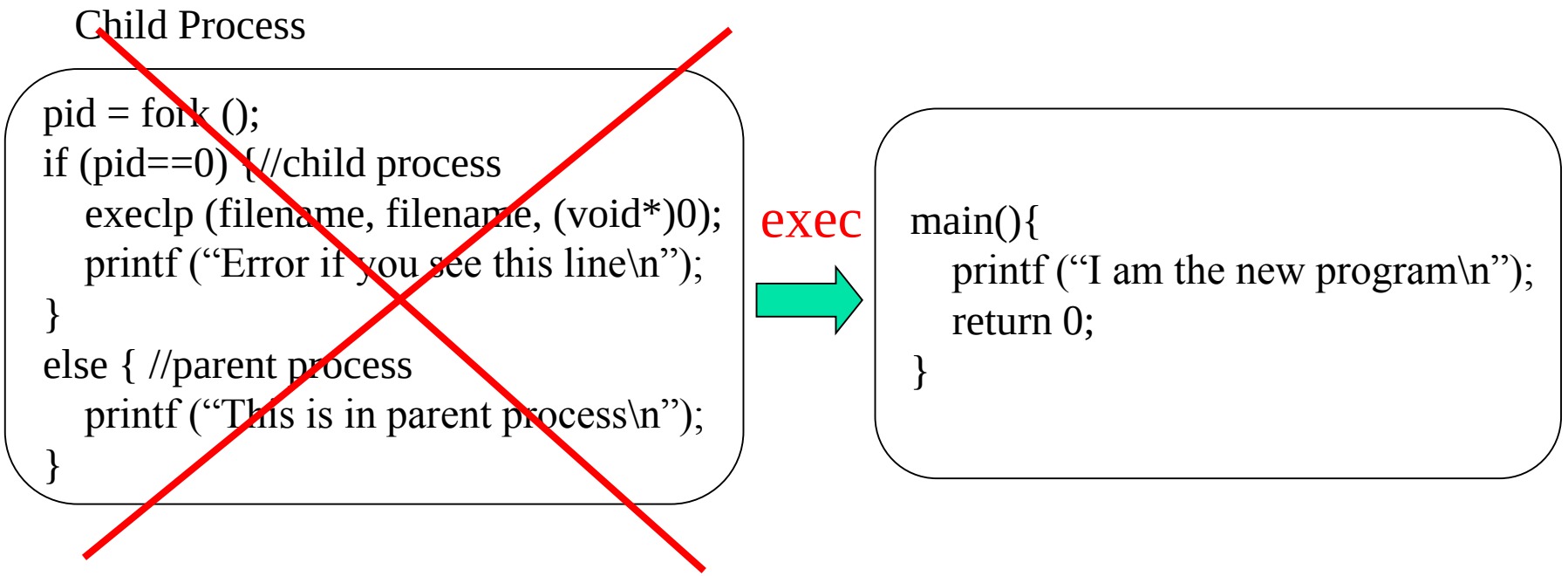
```
pid = fork ();
if (pid==0) {//child process
    execlp (filename, filename, (void*)0);
    printf ("Error if you see this line\n");
}
else { //parent process
    printf ("This is in parent process\n");
}
```

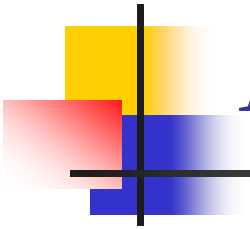
Child Process

```
pid = fork ();
if (pid==0) {//child process
    execlp (filename, filename, (void*)0);
    printf ("Error if you see this line\n");
}
else { //parent process
    printf ("This is in parent process\n");
}
```

How to create a new process? (cont'd)

- The child process executes an *exec*() system call to refill its content

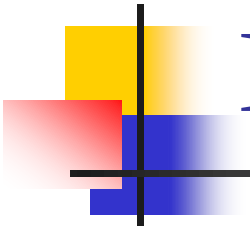




A simple exercise

- What will happen?

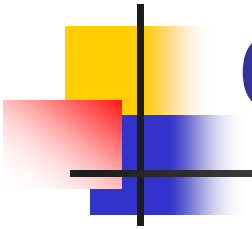
```
while (1) {
    fork ();
}
```



Next Lecture

- Basic file operations
  - Chap. 3 of [Stevens]

- Remark: History of UNIX skipped (Chap. 2)
  - various versions of UNIX and compatibility issues
  - in simple words: see *man* before you use a function/command/system call



Open Problem

- Does an embedded device (e.g. cell phone) need a protected-mode OS?

- Trend on embedded OS (RTOS) research
  - embedded Linux
  - uCOS
  - TinyOS (UC-Berkeley)