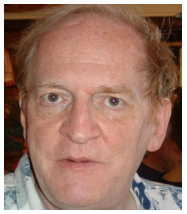
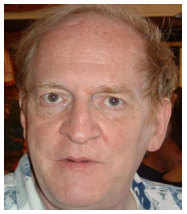
COSC 3360/6310
Wednesday, February 10

Announcements



- Please try to start your next quizzes on time. I am most attentive to issues during the first 15 to 20 minutes of the quiz.
- I plan to have all 120 quizzes graded by next Monday.
- I have extra explanations for the first assignment.

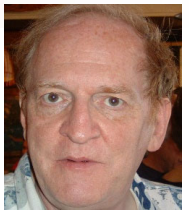
How to load jobs into main memory



- Must be done
  - At beginning of the simulation
  - Each time a job terminates
- Two conditions
  - Less than MPL jobs in main memory

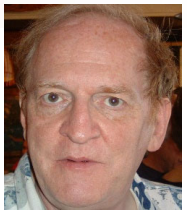
  and
  - There still are jobs ready to be fetched

Fetchjobs routine



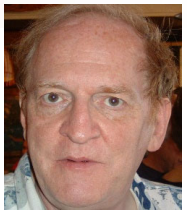
```
fetchjobs(seqNo) {
     while ((njobsinram < mpl)&&(nextjobtofetch < jobcount)) {
          njobsinram ++;
          seqno =  nextjobtofetch;
          nextjobtofetch++;
          pop first job step from job seqno
          split into (request, duration)
          if (request == "CORE")
               process core request for job jobID[seqno]
          else
               printf("PANIC: FIRST STEP IS NOT A CORE REQUEST");
} // fetchjobs
```

Sequence numbers and job IDs



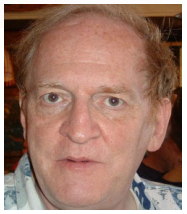
- JobIDs are not consecutive
  - Cannot use them to locate the next step of a specific job
- My solution
  - Use job sequence number inside your program
  - Keep track of job IDs in a jobID array indexed by job sequence numbers

Overview (I)



- ***Input module***
  - Read in all input data
    - Store them in a jobList
  - Fetch up to MPL jobs (**`fetchjobs`**)
- ***Main loop***
  - Pops next event from event list
    - CORE completion
    - DISK completion
    - SPOOLER completion

Overview (II)

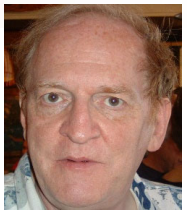


- ***Starting a job***
  - Handled by **`fetchjobs`** routine
- ***Handling job termination***
  - Try to fetch a new job (**`fetchjobs`**)
- ***Detecting the end of the simulation***
  - Event list will be ***empty***

Chapter II
Processes

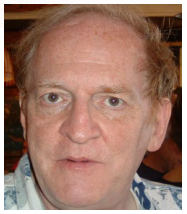
Jehan-François Pâris
jfparis@uh.edu

Chapter Overview



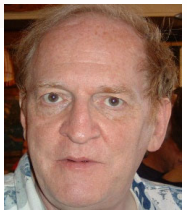
- Processes
- States of a process
- Operations on processes
  - **`fork()`**, **`exec()`**, **`kill()`**, **`signal()`**
- Threads and lightweight processes

PROCESSES



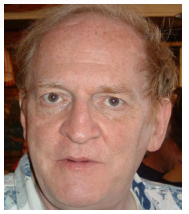
- A process is a ***program*** executing a given ***sequential computation*.**
  - An ***active entity*** unlike a program
  - *Think of the difference between a recipe in a cookbook and the activity of a cook preparing a dish according to the recipe!*

Processes and programs (I)



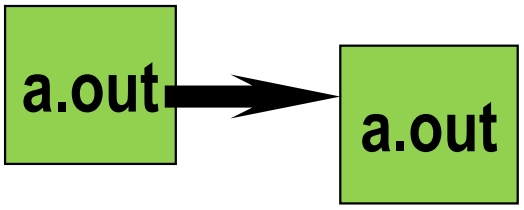
- Can have one program and many processes
  - When several users execute the same program (text editor, compiler, and so forth) at the same time, each execution of the program constitutes a ***separate process***
  - A program that ***forks*** another sequential computation gives birth to a new process.

Examples

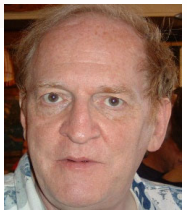


- Several executions of same program

- A program forking a child

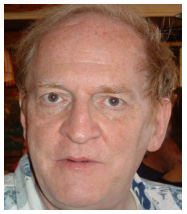


Processes and programs (II)

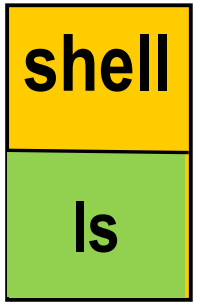


- Can have one process and two—or more—programs
  - A process that performs an **`exec()`** call replaces the program it was executing

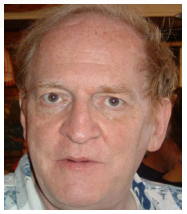
Examples



- One process executing two programs

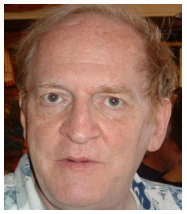


The UNIX shell



- Program that
  - Reads input from the keyboard
  - Creates the process that will execute the command.
  - Wait for the completion of the process it has created unless it was specified otherwise
- *User-level program that you and I could write*

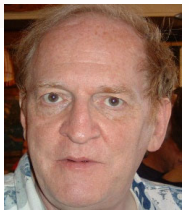
Yes, we can



```
#! /usr/bin/python3
"""  A very very basic shell in Python 3
      Check https://www.python-course.eu/forking.php
 """
import os
 def changeDirectory(argc, argv) :
    if argc ==  2 :
        try :
            os.chdir(argv[1])
        except Exception :
            print("BasicShell: " + argv[0] +
                  ": no such file or directory")
    elif argc == 1 :
        os.chdir(os.environ['HOME'])
    else :
        print(BasicShell: cd: too many arguments")
def vanillaCase(argc, argv) :
    kidpid = os.fork()
    if kidpid == 0 :
        try :
            os.execvp(argv[0], argv)
        except Exception :
            print(argv[0]+": program not found")
    else :
        os.wait()
```

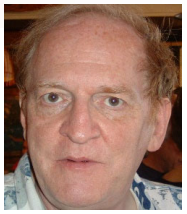
```
while (1) :
    argline = input("BasicShell: ")
    argline.strip()
    argv = argline.split() #  break at spaces
    argc = len(argv)
    if argc == 0 :
        continue
    if argv[0] == 'exit' :
        # Exiting BasicShell
        break
    elif argv[0] == 'cd':
        # Changing current directory
        changeDirectory(argc, argv)
    else :
        vanillaCase(argc, argv)
```

A very basic UNIX shell



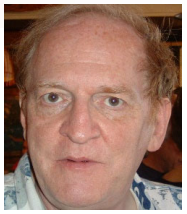
```
for (;;) {
    parse_input_line(arg_vector);
    if built_in_command(arg_vector[0]) {
        do_it(arg_vector);
        continue;
    } // built-in command
    pathname = find_path(arg_vector[0]);
    create_process(pathname, arg_vector);
    if (interactive())
        wait_for_this_child();
} // for loop
```

Notes



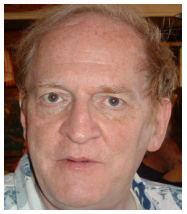
- All functions in italics are templates yet to be written
- Real shells do more:
  - I/O redirection
  - Pipes (as in **ls -alg | more**)
  - Command aliasing,
  - Wildcard characters (as ”*****”)
  - …

Importance of processes



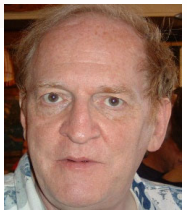
- Processes are the ***basic entities*** managed by the operating system
  - OS provides to each process the illusion it has the whole machine for itself
  - Each process has a dedicated ***address space***

The process address space



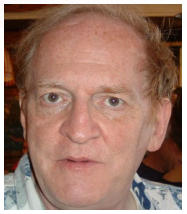
- Set of main memory locations allocated to the process
  - Other processes cannot access them
  - Process cannot access address spaces of other processes
- A process address space is the ***playpen*** or the ***sandbox*** of its owner

A last word



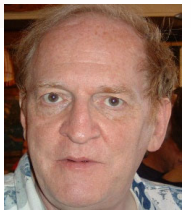
- There are many *quasi-synonyms* for process:
  - Job (very old programmers still use it)
  - Task
  - Program (strongly deprecated)

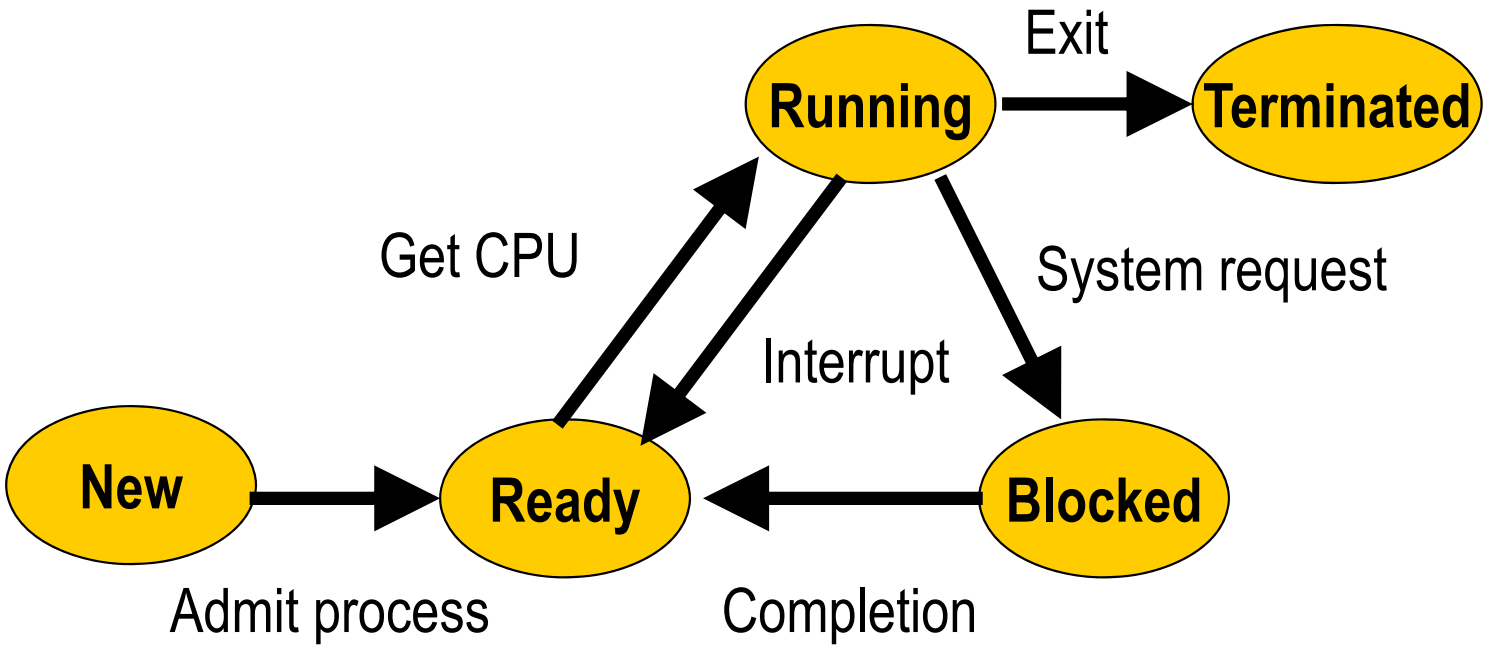
PROCESS STATES



- Processes go repeatedly through several stages during their execution
  - Waiting to get into main memory
  - Waiting for the CPU
  - Running
  - Waiting for the completion of a system call

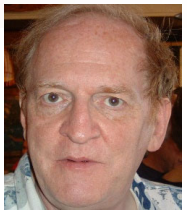
The big diagram





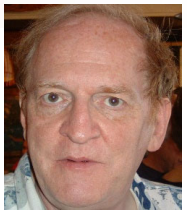
This is fundamental material

Process arrival



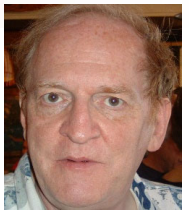
- New process
  - Starts in NEW state
  - Gets allocated a Process Control Block (PCB) and main memory
  - Is put in the READY state waiting for CPU time

The ready state



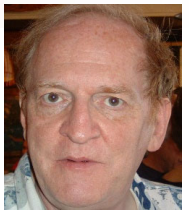
- AKA the ***ready queue***
- Contains all processes waiting for the CPU
- Organized as a ***priority queue***
- Processes leave the priority queue when they get some CPU time
  - Move then to the RUNNING state

The running state (I)



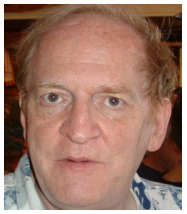
- A process in the running state has exclusive use of the CPU until
  - It ***terminates*** and goes to the ***TERMINATED*** state
  - It does a ***system call*** and goes to the ***BLOCKED*** state
  - It is ***interrupted*** and returns to the ***READY*** state

The running state (II)



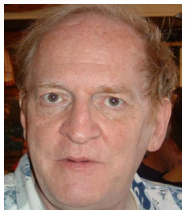
- Processes are forced to relinquish the CPU and return to the ***READY*** state when
  - A ***higher-priority process*** arrives in the ready queue and ***preempts*** the running process
    - *Get out, I'm more urgent than you!*
  - A ***timer interrupt*** indicates that the process has exceeded its time slice of CPU time

The blocked state (I)



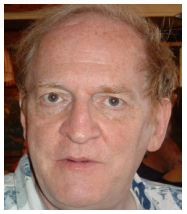
- Contains all processes waiting for the completion of a system request:
  - I/O operation
  - Any other system call
- Process is said to be
  - ***blocked*** (Arpaci-Dusseau & Arpaci-Dusseau)
  - ***waiting***
  - ***sleeping*** (UNIX)

The blocked state (II)



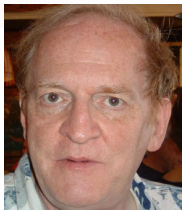
- A system call that does not require callers to wait until its completion is said to be ***non-blocking***
  - Calling processes are immediately returned to the ***READY*** state
- The blocked state is organized as a ***set of queues***
  - One queue per device, OS resource

The process control block (I)



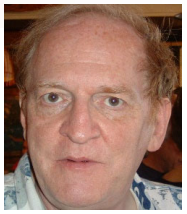
- Contains all the information associated with a specific process:
  - ***Process identification*** (pid), *argument vector*, ...
    - UNIX pids are unique integers
  - ***Process state*** (new, ready, running, …),
  - ***CPU scheduling information***
    - Process priority, processors on which the process can run, ...,

The process control block (II)



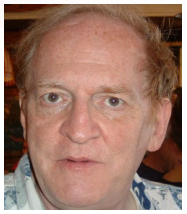
- ***Program counter*** and other CPU registers
  - Including the ***Program Status Word*** (PSW),
- ***Memory management information***
  - Very system specific,
- ***Accounting information***
  - CPU time used, system time used, ...
- ***I/O status information***
  - List of opened files, allocated devices, …

The process table



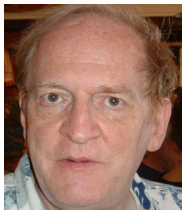
- System-wide table containing
  - ***Process identification*** (pid), *argument vector*, ...
  - ***Process current state***
  - ***Process priority and other CPU scheduling information***
  - A ***pointer*** to the remaining information.

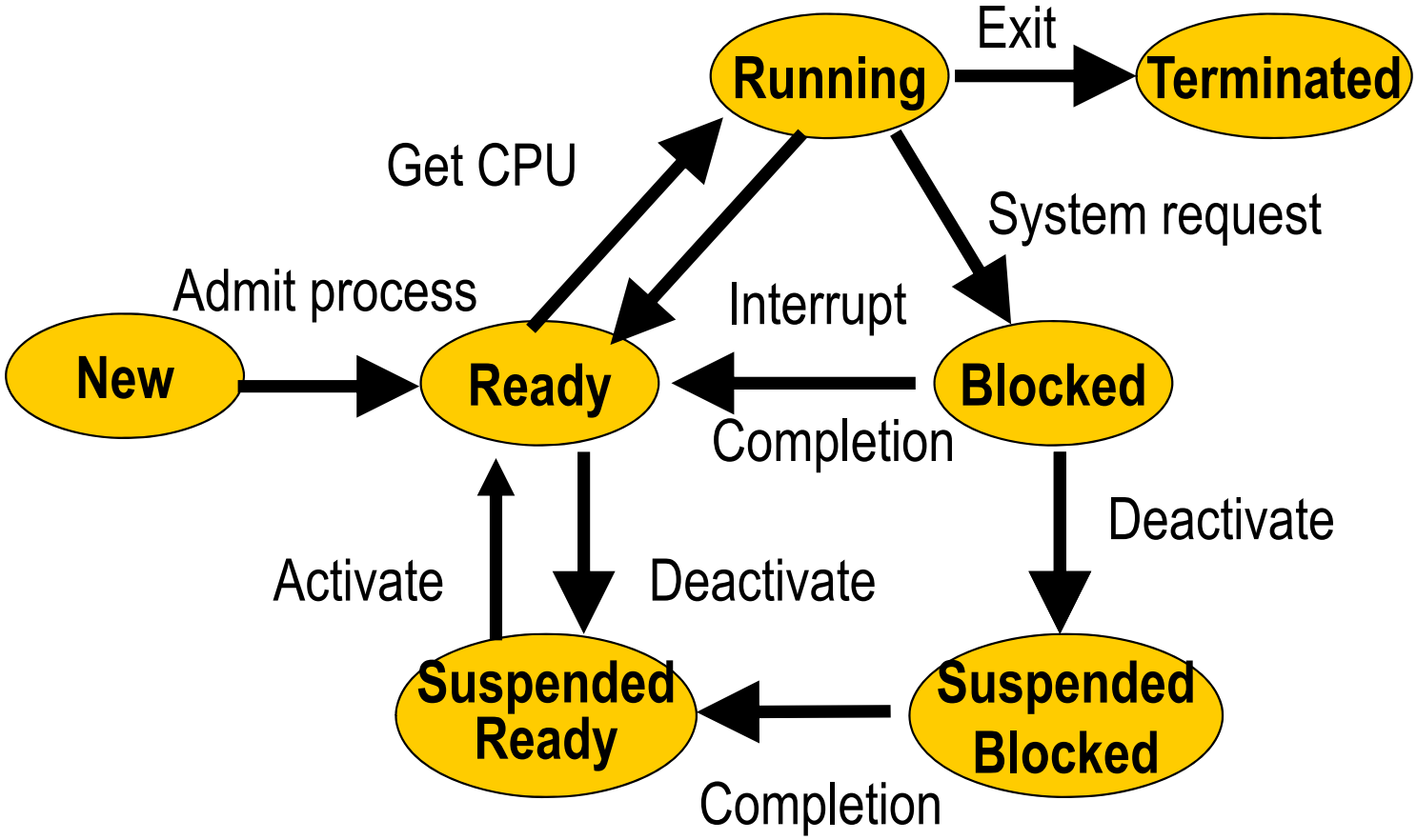
Swapping



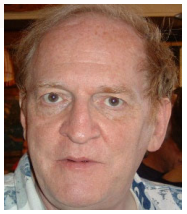
- Whenever the system is very loaded, we might want to expel from main memory or ***swap out***
  - Low priority processes
  - Processes that have been waiting for a long time for an external event
    - *User is out of the office*
- These processes are said to be ***swapped out*** or ***suspended*.**

How it works



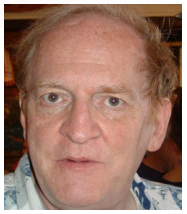

Running
Exit
Terminated
Get CPU
System request
Admit process
Interrupt
New
Ready
Blocked
Completion
Deactivate
Activate
Deactivate
Suspended Ready
Suspended Blocked
Completion

Suspended processes



- Suspended processes
  - Do not reside in main memory
  - Continue to be included in the process table
- Can distinguish between two types of suspended processes:
  - Waiting for the completion of some request (***blocked_suspended***)
  - Ready to run (***ready_suspended***).

A warning



- A system should ***not*** swap out ready processes unless their priority is ***very low***
- Otherwise swapping out ready processes can only be a ***desperate measure***