Design and Engineering of Computer Systems

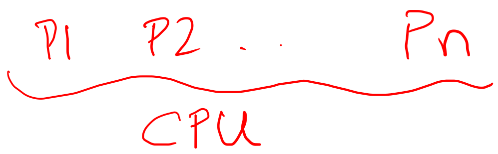
Lecture 6: Processes

Mythili Vutukuru

IIT Bombay

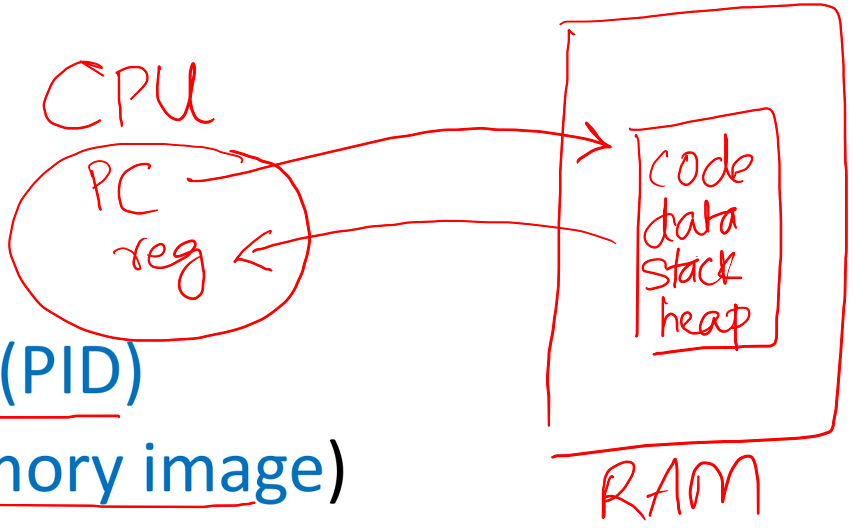
Introduction to processes

- Process = running program (code+data)
  - Different runs of program will be different processes
- Main job of the OS = run multiple processes concurrently on underlying hardware
  - OS virtualizes CPU across multiple processes



- Key tasks of OS in process management:
  - Process lifecycle management: creation, execution, termination of processes
  - Scheduling policy: decides which process runs when on CPU
  - Context switching: mechanism to switch between processes
  - Implementing process-related system call API to user processes
  - Handling interrupts and other events

What defines a process?

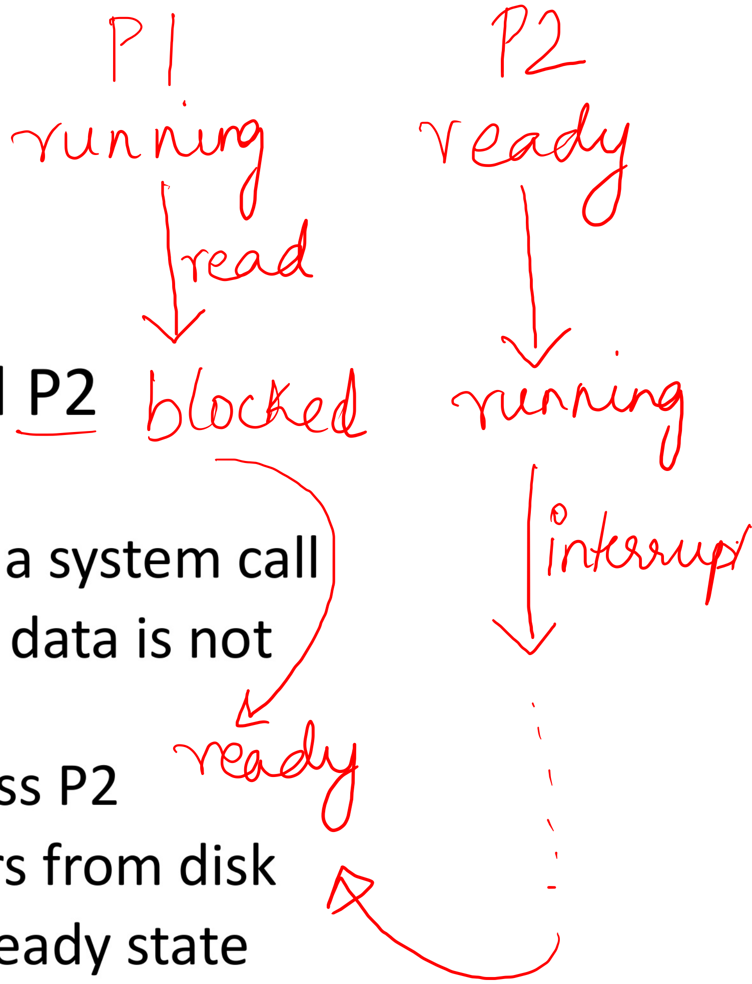


- Every process has a unique process identifier (PID)
- Process occupies some memory in RAM (memory image)
  - Code+data from executable
  - Stack, heap for runtime memory use, and other components
- The execution context of the process (values of CPU registers)
  - PC has address of instruction of process, some registers have process data
  - Process context is in CPU registers when process is running on CPU
  - Context saved in memory when process is paused, restored when run again
- Ongoing communication with I/O devices
  - Information is maintained about files that are open, ongoing network connections, other active connections to I/O devices

States of a process

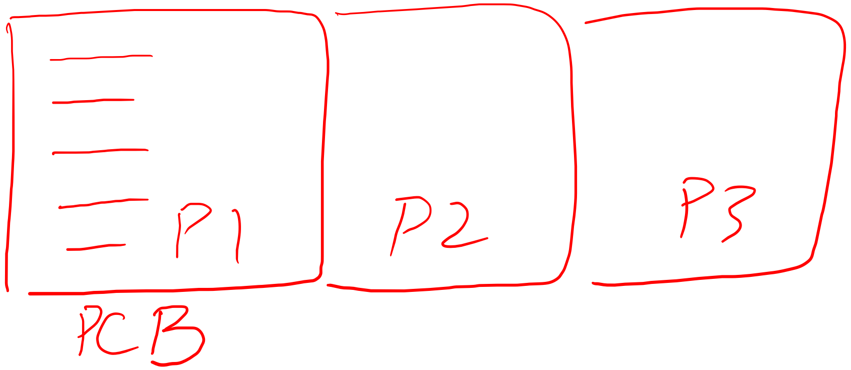
- OS manages multiple active processes at the same time. An active process can be in one of the following situations.
- Running: currently executing on CPU
  - CPU registers contain context of process
- Blocked/suspended/sleeping: process cannot run for some time
  - Example: process has requested data from disk, command issued, but process cannot proceed until the data from disk is available
- Ready/runnable: ready to run but waiting for OS scheduler to switch the process in
  - Many processes can be ready but scheduler can only run one on a CPU core
- Context of blocked and ready processes is saved in memory, so that they can continue to run later on

Example: process state transitions



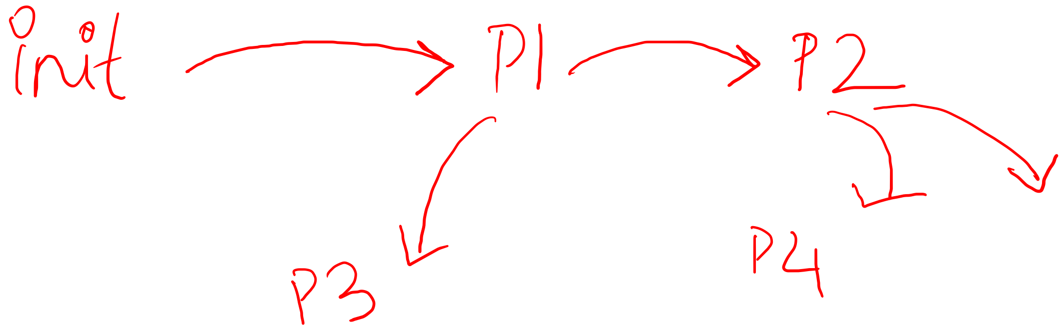
- Consider a system that has two user processes P1 and P2
  - Initially P1 is running, P2 is ready and awaiting its turn
  - P1 opens a file and wants to read some bytes from disk via a system call
  - OS handles the system call and gives command to disk, but data is not available immediately
  - Process P1 is moved to blocked state, OS switches to process P2
  - Process P2 runs for some time, and then an interrupt occurs from disk
  - CPU jumps to OS which handles interrupt, P1 is moved to ready state
  - OS can continue to run P2 again after interrupt and OS scheduler switches to ready process P1 later on after some time

Process control block (PCB)



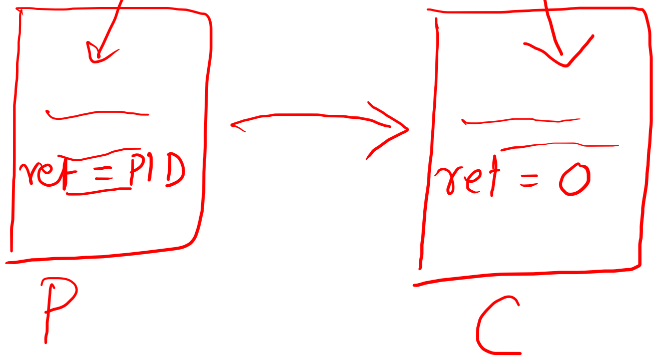
- All information about a process is stored in a data structure called the process control block (PCB)
  - Process identifier (PID)
  - Process state (running, ready, blocked, terminated, ..)
  - Pointers to other related processes (parent, children)
  - Saved CPU context of process when it is not running
  - Information related to memory locations of a process
  - Information related to ongoing I/O communication
  - …
- OS stores PCBs of active processes in a data structure (array, list,..)
  - New PCB added when process created, deleted when process is cleaned up

Process creation: fork

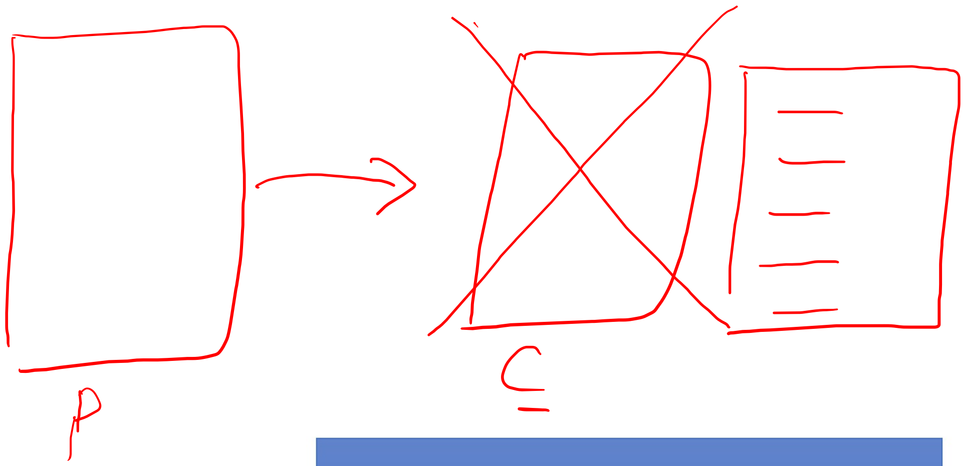


- How are processes created?
  - OS creates the first "init" process in system
  - All other processes are created by "forking" from a parent
- Parent process calls "fork" system call to create (spawn) a new process
  - New child process created with new PID
  - Memory image of parent is copied into that of child
  - Parent and child run different copies of same code
  - Parent and child resume execution in the code after "fork"
  - Child starts executing with a return value of 0 from fork
  - Parent resumes executing with a return value of child PID
  - After fork, parent and child run independently
  - Any changes in parent's data after fork does not impact child

```
…
int ret = fork()
if(ret == 0) {
  print "I am child"
}
else if(ret > 0) {
  print "I am parent"
}
…
```



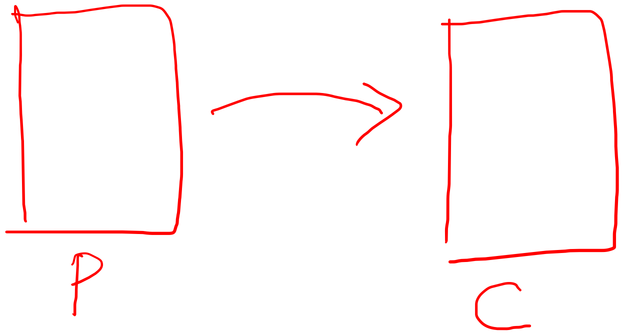
Exec system call



- Isn't it impractical to run the same code in all processes?
  - Sometimes parent creates child to do similar work..
  - .. but other times, child may want to run different code
- Child process uses "exec" system call to get a new "memory image"
  - Allows a process to switch to running different code
  - Exec system call takes another executable as argument
  - Memory image is reinitialized with new executable, new code, data, stack, heap, …
  - Child process does not return to old parent program (unless exec fails)
  - Print statement after exec never prints unless exec fails

```
…
int ret = fork();
if(ret == 0) {
  exec("some_executable")
  print "error: exec failed"
}
else if(ret > 0) {
  print "I am parent"
}
…
```

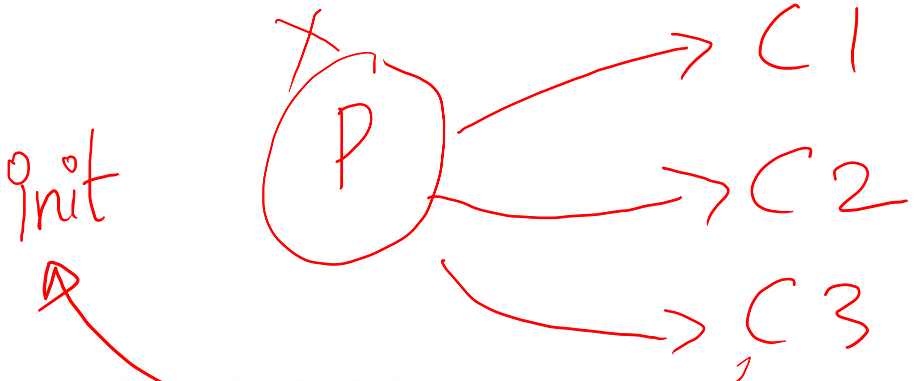
Exit and wait system calls



- When a process finishes execution, it called exit system call to terminate
  - OS switches the process out and never runs it again
  - Exit is automatically called at end of main
  - Process does not disappear, only becomes zombie
- Parent calls "wait" system call to reap (clean up memory of) a zombie child
  - Wait system call blocks parent until child exits
  - After child exit, wait cleans up memory of child and returns
- Exiting child cannot clean up its memory during exit system call due to various reasons relating to how memory is setup
  - Memory has to be cleaned by another process only

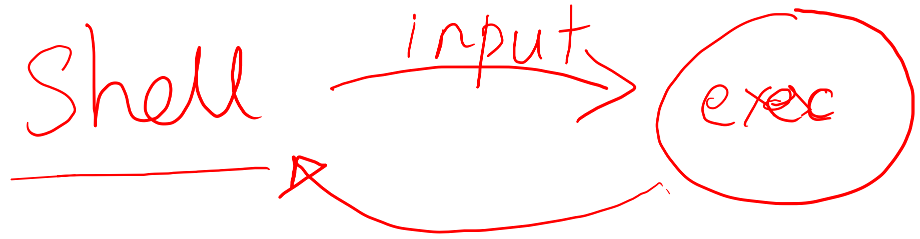
```
…
int ret = fork()
if(ret == 0) {
  print "I am child"
  exit()
}
else if(ret > 0) {
  print "I am parent"
  wait()
}
…
```

More on zombies



- Wait system call "reaps" one dead child at a time
  - Every fork must be followed by call to wait at some point in parent
- What if parent has exited while child is still running?
  - Child will continue to run, becomes orphan
  - Orphans adopted by init process
  - When orphan dies, the zombie is reaped by init
- If parent forks children, but does not bother calling wait for long time, system memory fills up with zombies
  - Common programming error, exhausts system memory

How the shell works



- OS exposes a terminal/shell to run user programs
  - Can be created by first “init” process on boot up
- What happens when you type a command in the shell?
  - Shell runs command, returns back to command prompt again
- How does the shell work?
  - Shell reads input from user
  - Shell process forks a child process
  - Child process runs exec with “echo” program executable as argument (most Linux commands are programs written already for your convenience)
  - Child runs “echo” command, calls exit at end of program
  - Parent shell calls wait, blocks till child terminates, reaps it
  - Once child is done, reads next input command from user
- Think: why doesn’t shell exec command directly?
  - Do we want the shell program code to be rewritten fully?

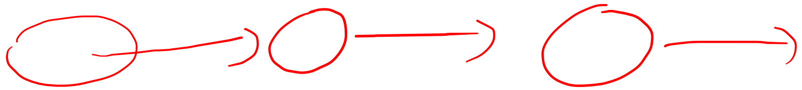
```
$echo hello
hello
$
```

```
do forever {
   input(command)

   int ret = fork()

   if(ret == 0) {
      exec(command)
   }
   else {
      wait()
   }
}
```

PC → instruction

OS scheduler

- OS maintains list of all active processes (PCBs) in a data structure
  - Processes added during fork, removed after clean up in wait
- OS scheduler is special code in the OS that periodically loops over this list and picks processes to run



- Basic outline of scheduler code
  - When invoked, save context of currently running process in its PCB
  - Loop over all ready/runnable processes and identify a process to run next
  - Restore context of new process from PCB and get it to run on CPU
  - Repeat this process as long as system is running
- Note that restoring context of a process resumes its execution
  - PC points to instruction in process code, starts running instruction
  - Other registers are filled with values that existed before process was stopped
  - Process continues execution without realizing it was paused

Summary

- In this lecture:
  - The process abstraction
  - States of a process
  - Process Control Block (PCB)
  - Process system calls: fork, exec, exit, wait
  - How the shell works
- You can use commands like "top" and "ps" on a Linux computer to view all the active processes in your system: how many processes are running on your computer right now?
- Programming exercise: write simple code using fork, exec, wait system calls. Can you write a simple shell?