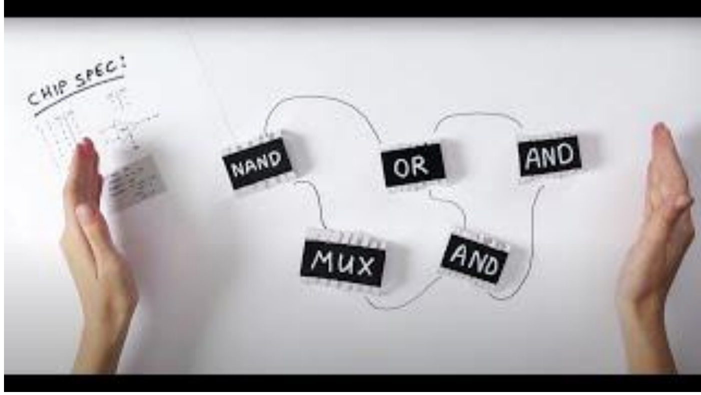


# CSc 360
# **Operating Systems**
## Processes

Jiaping Pan

**Fall 2023**

https://www.nand2tetris.org/ recommended by Erfan. thanks!

# P1 is already out!

- Due Monday, Oct 2, thru Brightspace
  - please make sure to code your own assignment
  - if you use any helper function, do cite it clearly
  - don't waste your time/money to copy&paste
    - we have access to github and old files too
- Please attend tutorials to follow its schedule
- A simple shell interpreter (SSI), able to
  - execute external programs (e.g., ls -l)
  - change directories (i.e., internal commands)
  - execute programs in background

# Too challenging?

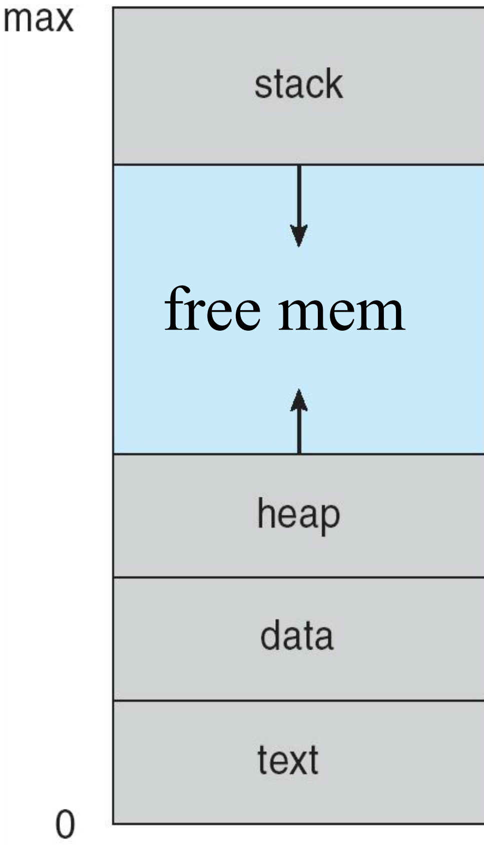
- We are here to help you
  - follow the suggested approach
    - discussed in the first lecture---it's effective!
  - attend lectures and tutorials: both!
  - get started earlier!
  - P1 discussion thread on Teams
    - get help and help others: TA standby in a day
  - *CSC consultant clinic/office: ECS 2nd floor*
  - office hours (tutorial and lecture instructors)

# Responsible use of computers

- Through this course, we will know better about operating systems, how they work, and tricks and tips
- You can practice these things and skills on your own computers **with a VM**
- Do not attempt to trick or compromise computers ***also used by others***
- See UVic policies: IT Policy 6030

* fork() bomb?

# Processes

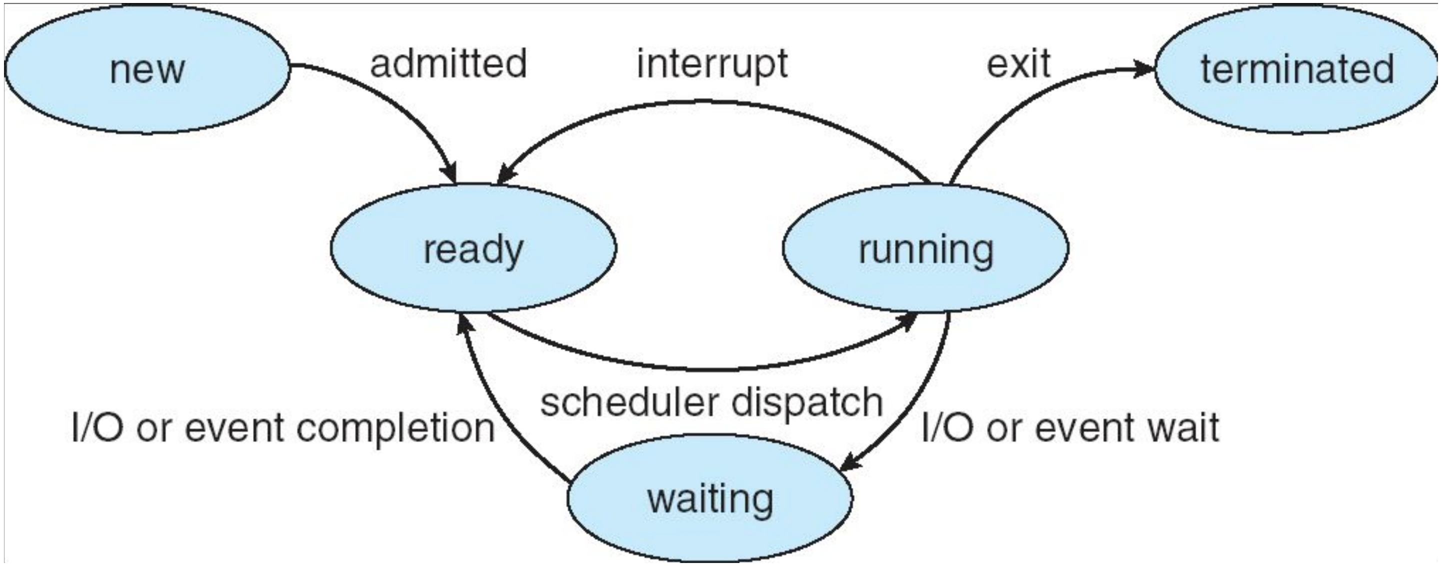


- Process: a program in execution
- Program: passive entity
  - static binary file on storage
    - e.g., gcc -o hello hello.c; ls -l hello
    - -rwxrwxr-x 1 *user group size date/time* hello
- Process: active entity; resource allocated!
    - ./hello
    - text (code); data (static), stack, heap
    - process control block (PCB)

“PCB attacks”?

# Process states

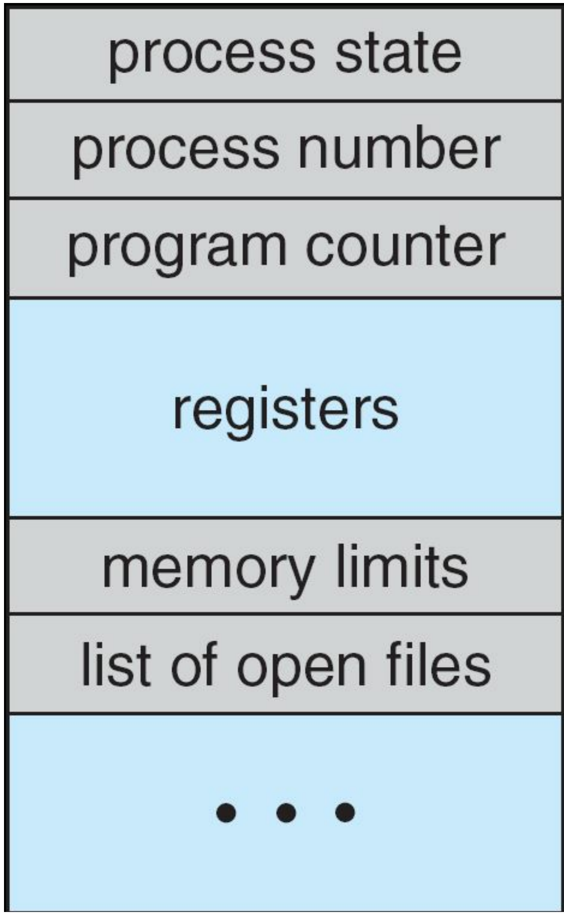
- E.g., one CPU (core)
  - one running process at any time
  - maybe many ready/waiting processes



why interrupt?

# Process control blocks

- PCB: keep track processes
  - state: ready/running, etc
  - CPU
    - PC, registers, priority, etc
  - memory
    - memory control information
  - I/O
    - e.g., list of opened files
  - accounting



process state

process number

program counter

registers

memory limits

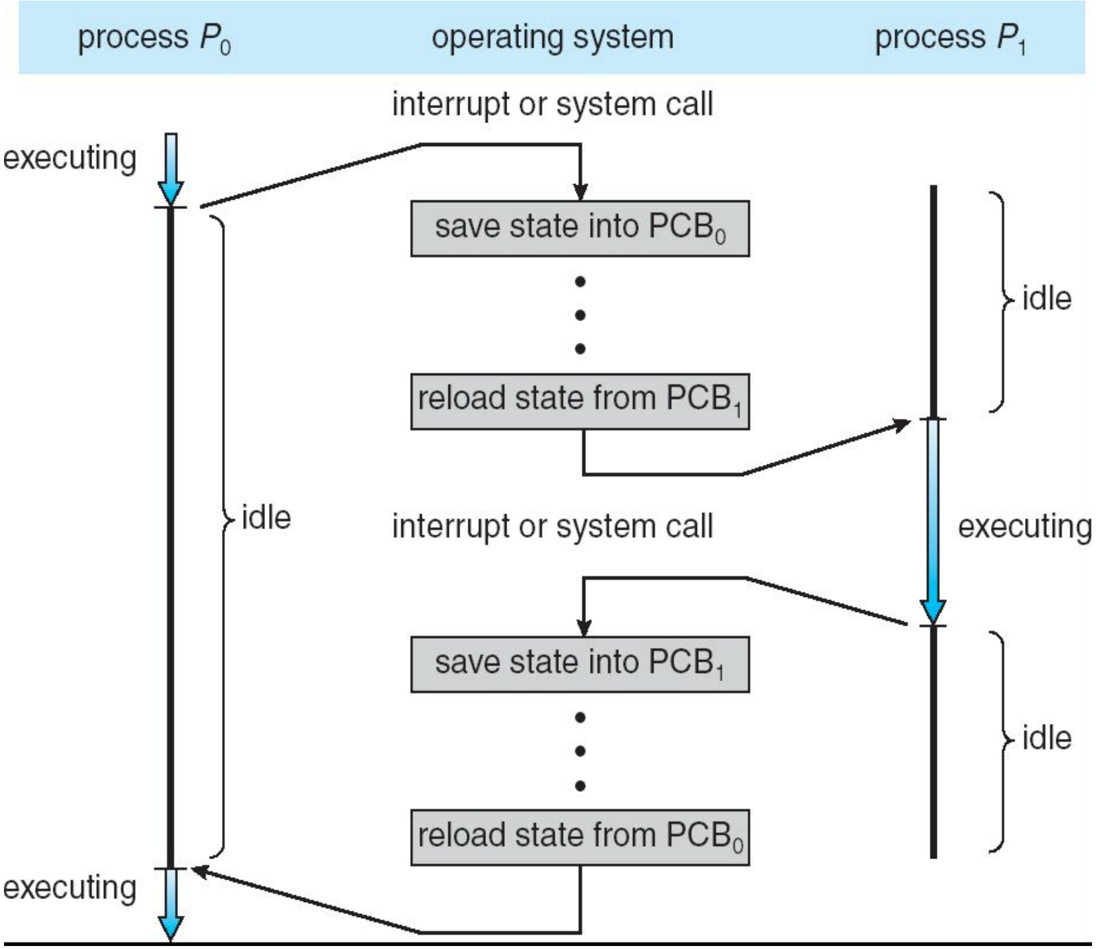
list of open files

• • •

“PCB exhausted!”

# Context switching

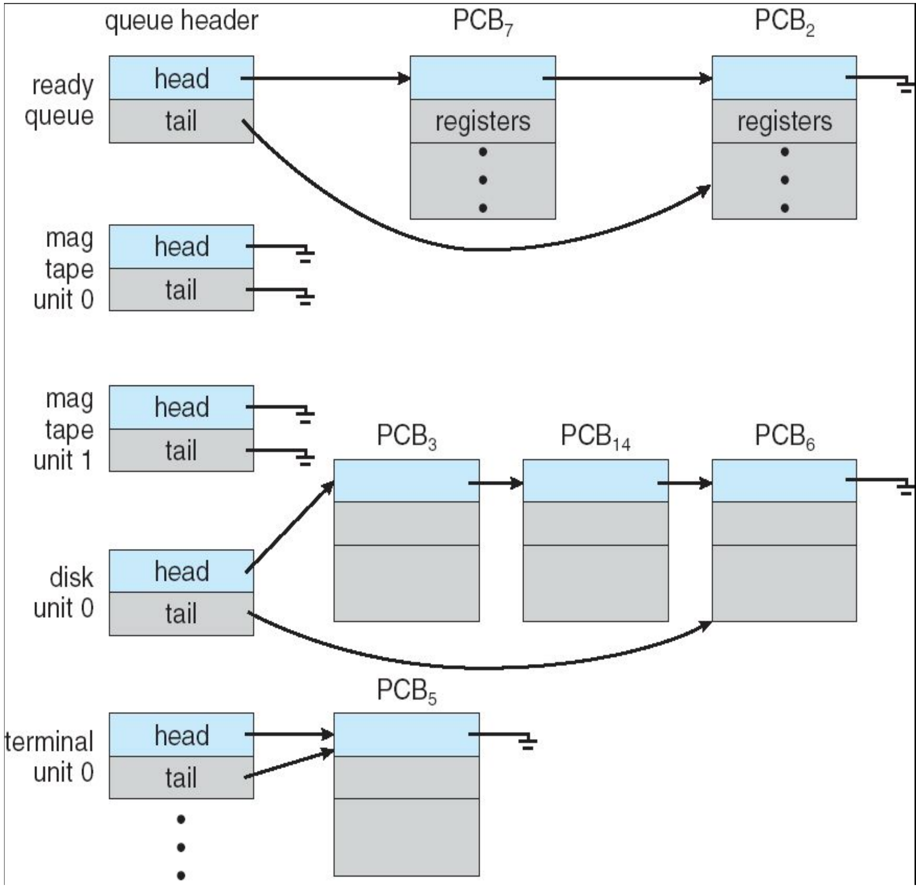
- Context switch
  - save states
  - restore states
- When
  - timer
  - I/O, memory
  - trap
  - waiting sys call



when more than two processes

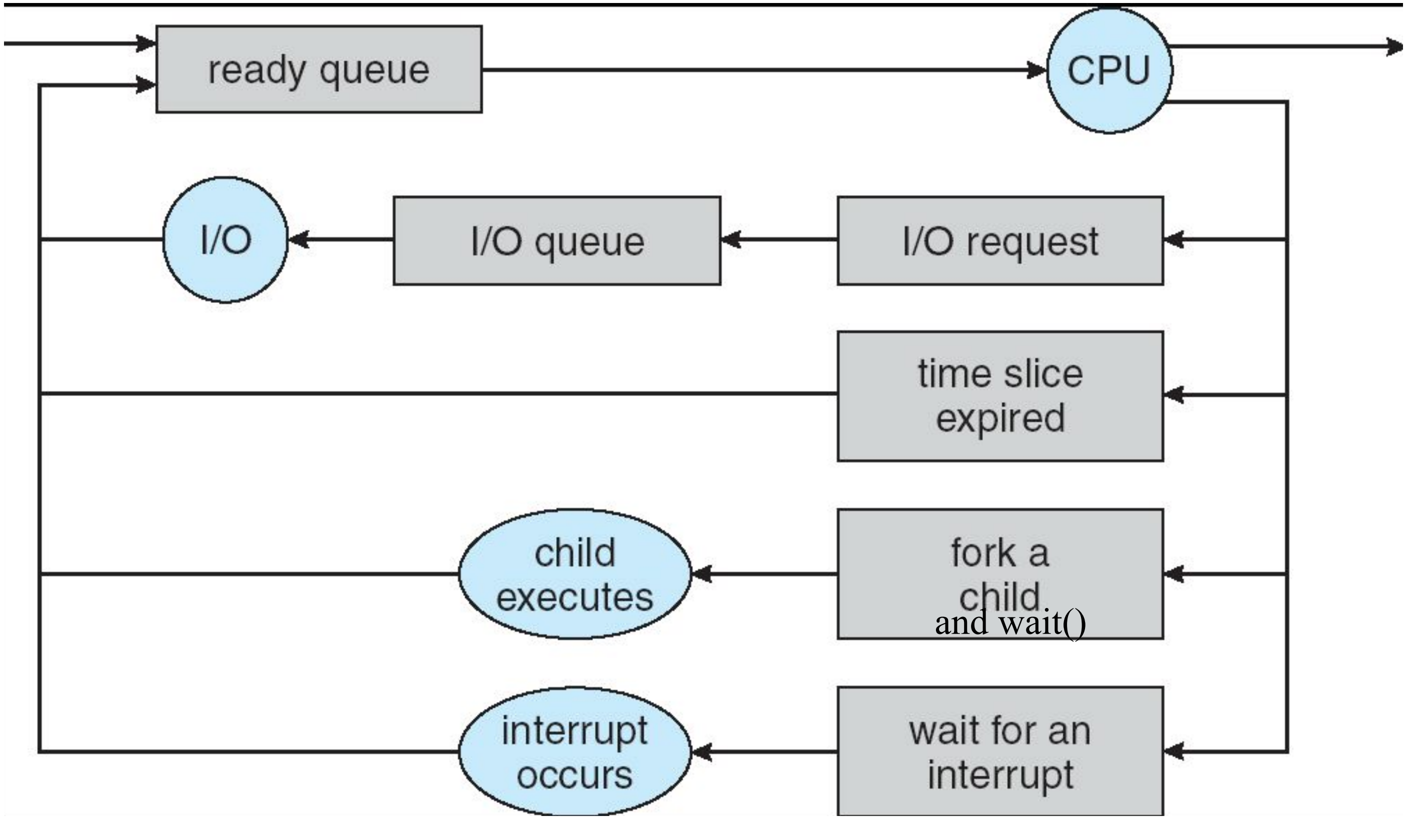
# Process scheduling

- Multiprogramming
  - utilization
- Timesharing
  - interactive
- Scheduling queues
  - linked list
  - ready queue
  - I/O queue



scheduling complexity

# Queuing system



scheduling priority

# Queuing scheduler

- Who’s the next?
- Long-term scheduler
  - – job scheduler (spooling)
  - – get to the ready queue
  - – CPU-intensive vs I/O intensive
- Short-term scheduler
  - – CPU scheduler
  - – frequency vs overhead

where's the long-term scheduler/gatekeeper?

# More on scheduling

- Medium-term scheduler
  - who is NOT the next
    - reduce the degree of multiprogramming
  - swap-in/out
- Scheduling algorithms
  - first-come-first-server, shortest-job-first, priority, round-robin, fair and weighted fair, …
  - more in Chapter 5

HumanOS: we do a lot of scheduling in our daily life too!

# This lecture so far

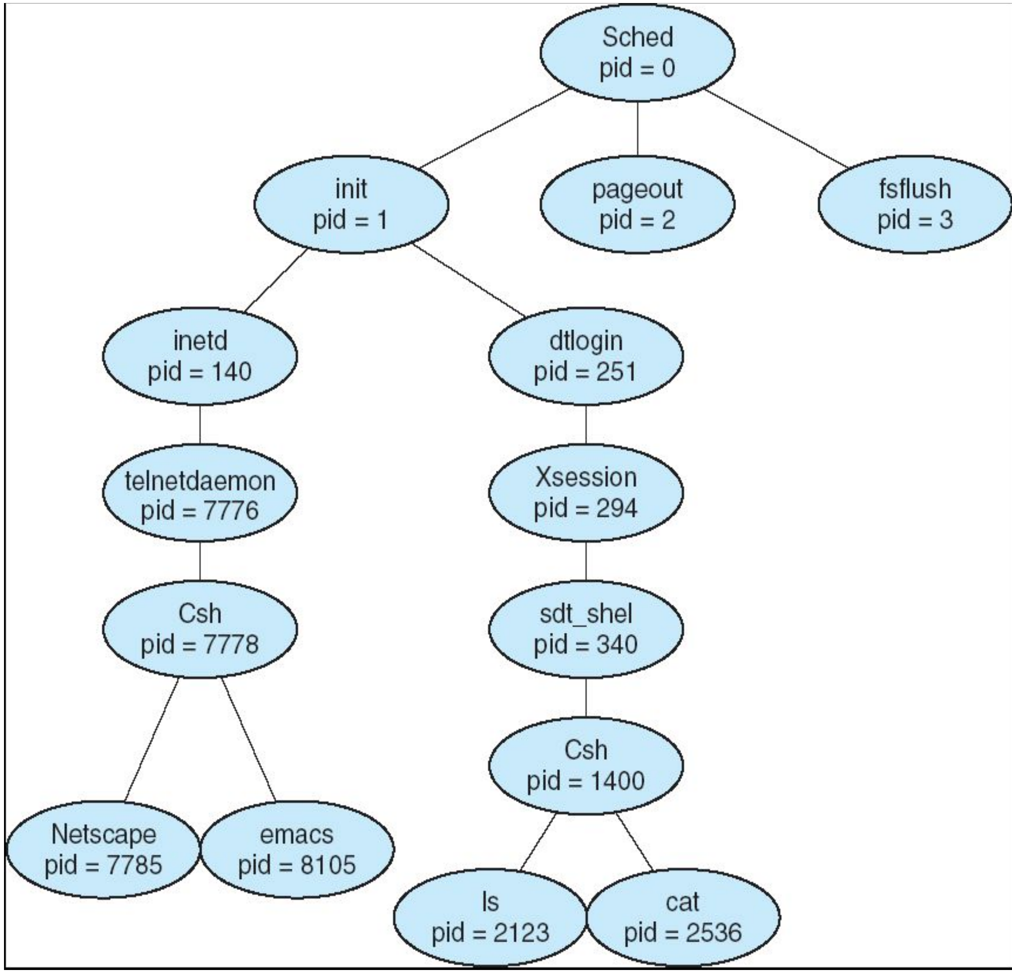
- Process and process scheduling
  - process vs program
  - process control block
    - context switch: what to save/restore
  - process scheduling
- Explore further
  - process status: /bin/ps
  - top CPU processes: /usr/bin/top

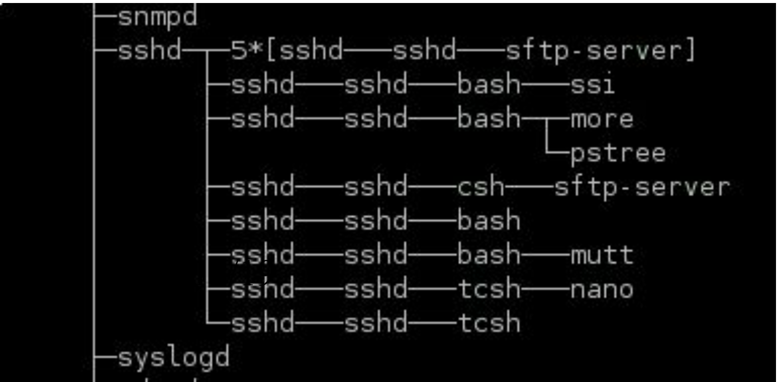
# Process creation

- Creating processes
  - parent process: create child processes
  - child process: created by its parent process
- Process tree
  - recursive parent-child relationship; why tree?
  - /usr/bin/pstree
- Process ID (PID) and Parent PID (PPID)
  - usually nonnegative integer

# Process tree

- sched (0)
  - init (1)
    - all user processes
  - pageout
    - memory
  - fsflush
    - file system





```
├snmpd
├sshd─┬─5*[sshd───sshd───sftp-server]
│     ├─sshd───sshd───bash───ssi
│     ├─sshd───sshd───bash─┬─more
│     │                    └─pstree
│     ├─sshd───sshd───csh───sftp-server
│     ├─sshd───sshd───bash
│     ├─sshd───sshd───bash───mutt
│     ├─sshd───sshd───tcsh───nano
│     └─sshd───sshd───tcsh
├syslogd
```

pstree on linux.csc.uvic.ca

# Parent vs child processes

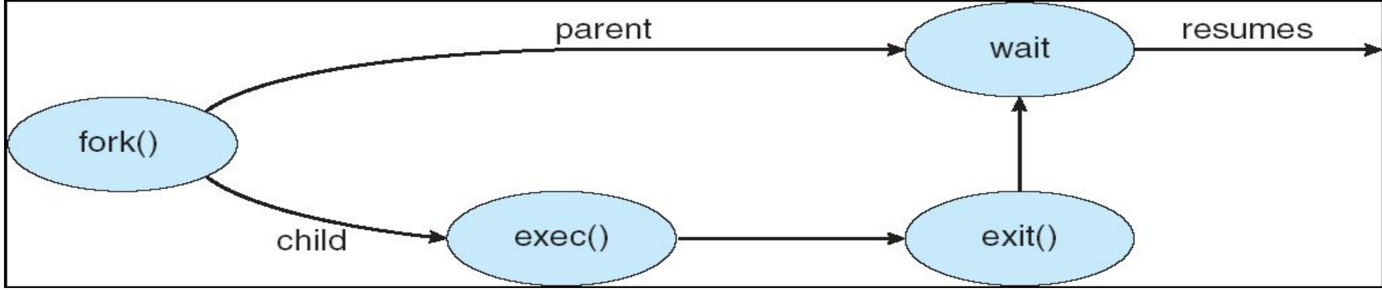
- Process: running program + resources
- Resource sharing: possible approaches
  - all shared, or
  - some shared (e.g., read-only code), or
  - nothing shared*
- Process execution: possible approaches
  - parent waits until child finishes, or
  - parent and child run **concurrently***

* default behaviors on linux

# fork(), exec*(), wait()

- Create a child process: fork()
  - return code < 0: error (in “parent” process)
  - return code = 0: you’re in child process
  - return code > 0: you’re in parent process
    - return code = child’s PID
- Child process: load a new program
  - exec*(): front-end for execve(file, arg, environ)
- Parent process: wait() and waitpid()

* details during tutorials



# Example

```
int main()
{
  Pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
   /*parent will wait for the child to complete*/
     wait (NULL);
     printf ("Child Complete");
     exit(0);
   }
}
```

* what if no wait()?

# Process termination

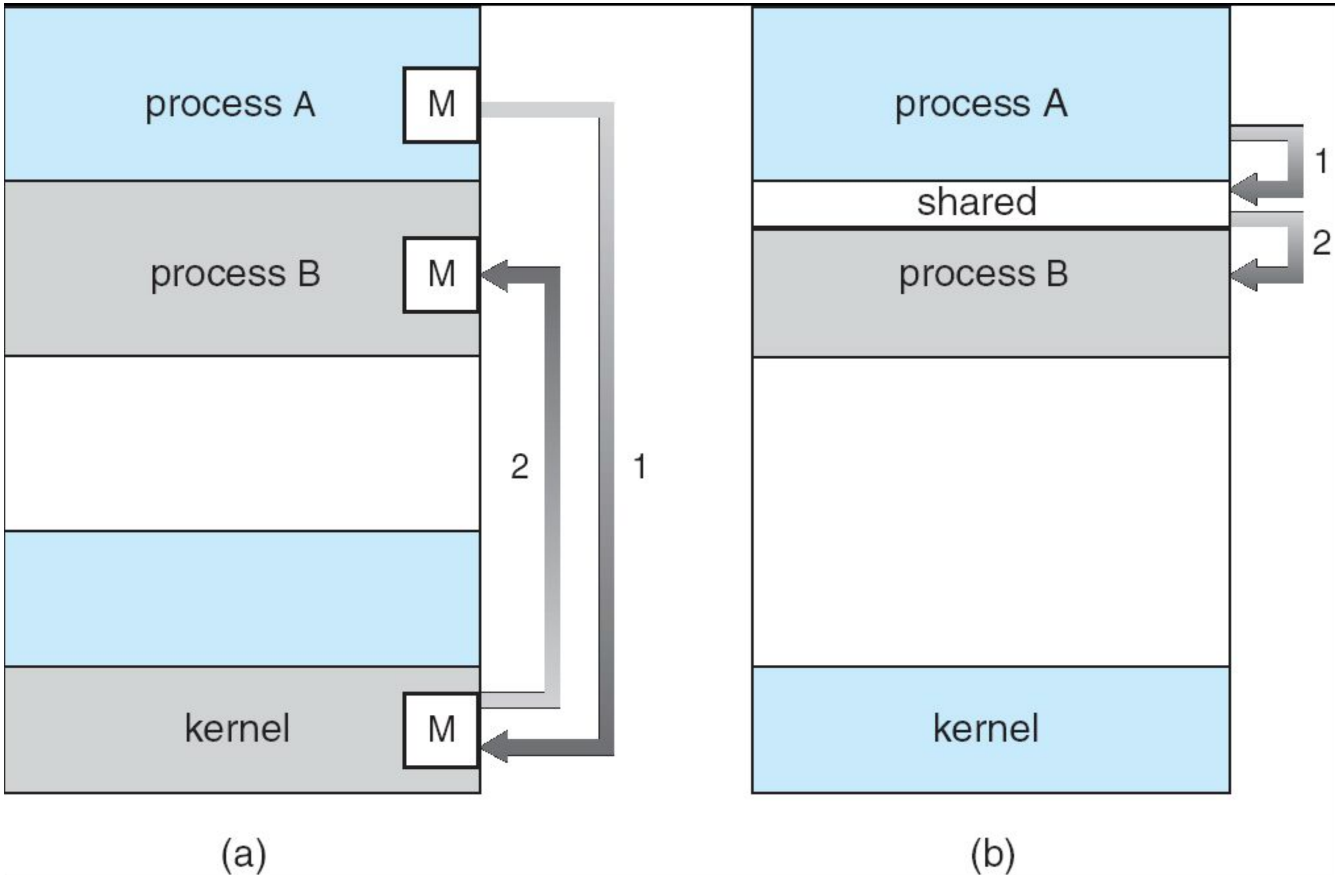
- Terminate itself: exit()
  - report status to parent process
  - release allocated resources
- Terminate child processes: kill(pid, signal)
  - actually send a signal to the child
    - child resource exceeded, child process no longer needed, and so on
  - parent is exiting
    - cascading termination, or find another parent

# Process communication

- Independent process
  - standalone process
- Cooperating process
  - affected by or affecting other processes
    - sharing, parallel, modularity, convenience
- Process communication
  - shared memory
  - message passing

# Message passing vs shared memory

- Overhead
- Protection



* again pros and cons

# The 2nd half of this lecture

- Process operations
  - process creation
    - process tree
  - process termination
  - the need for inter-process communication
- Explore further
  - /bin/ps, /usr/bin/top, /usr/bin/pstree
  - how does a child process find its parent’s PID?

# Next lecture

- Inter-process communication
  - read OSC7 Chapter 3 (or OSC6 Chapter 4)