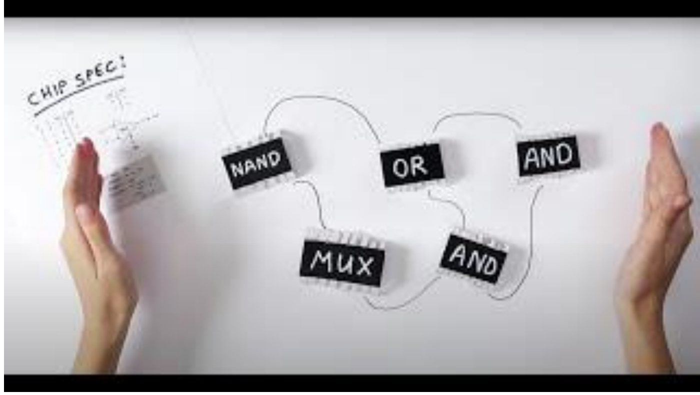


CSc 360
Operating Systems
Processes

Wenjun Yang

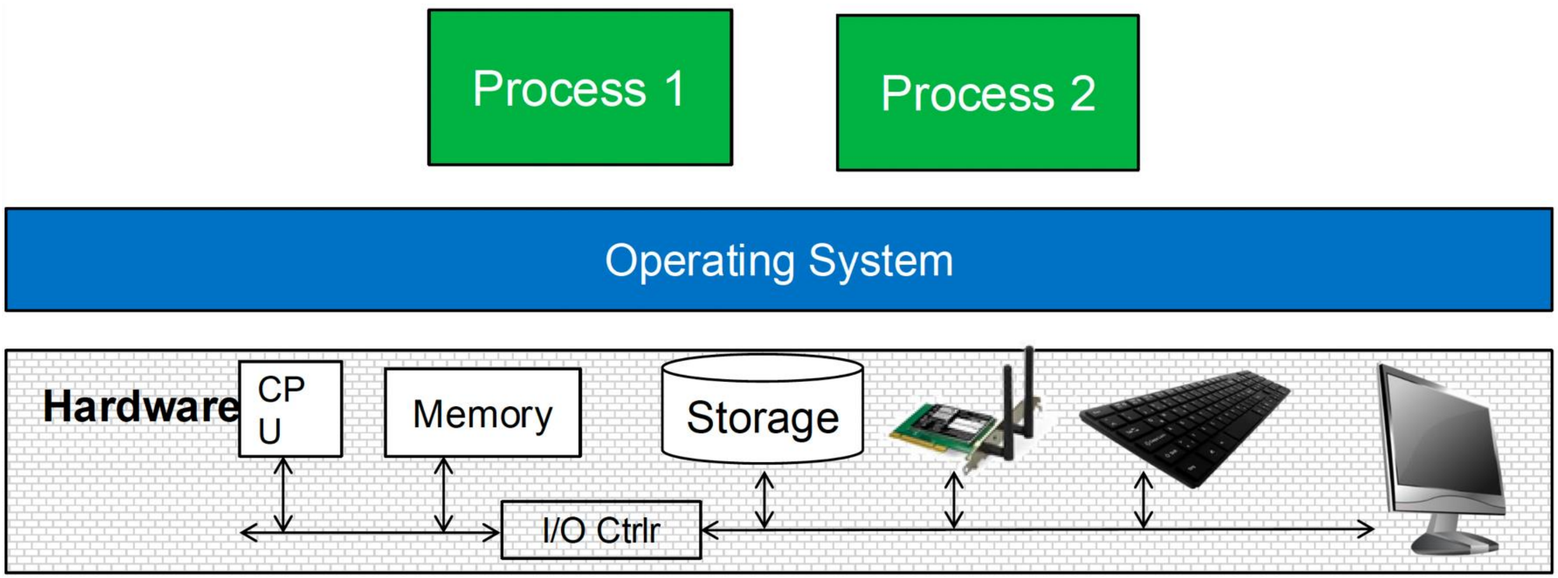
Fall 2025

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

Recall: Topic Breakdown (Course Objective)

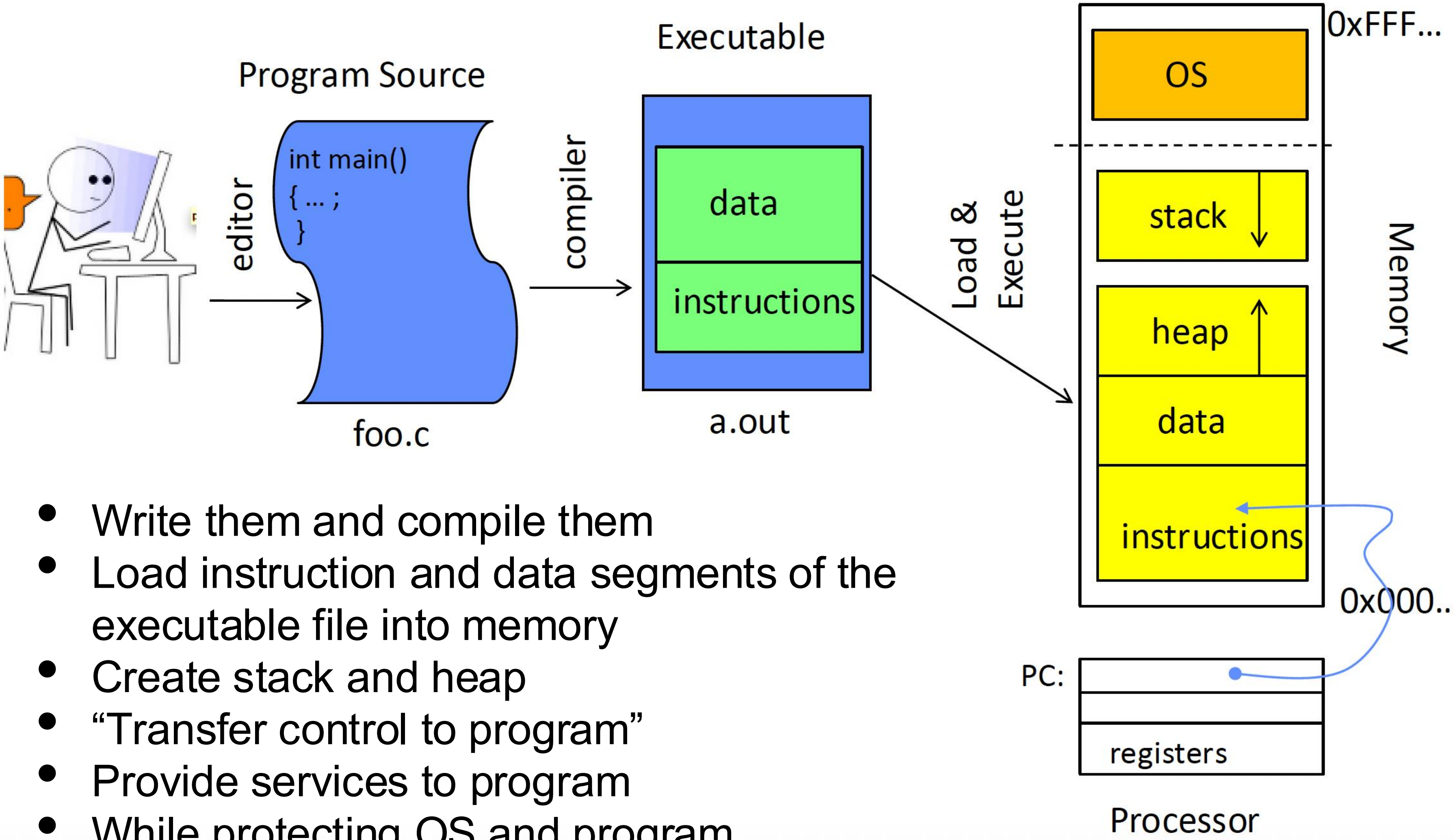
- virtualizing the CPU **(process)**: process, thread, scheduling, synch
- virtualizing memory (**memory)**: memory management, virtual memory
- **storage**: file systems, I/O systems

Process: an active instance of a program



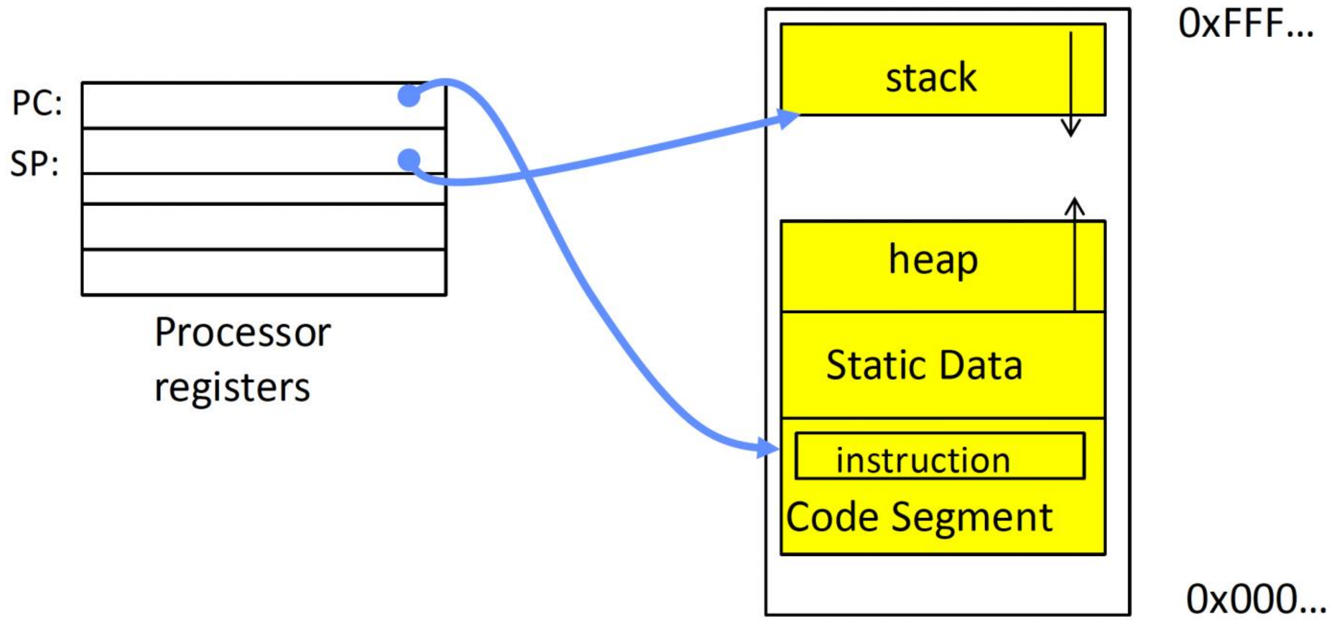
- OS virtualizes hardware to processes (applications) to provide the illusion that each process uses its own machine; also provides illusion of infinite memory and processor

OS Bottom Line: Run Programs



- Write them and compile them
- Load instruction and data segments of the executable file into memory
- Create stack and heap
- “Transfer control to program”
- Provide services to program
- While protecting OS and program

Address Space



- What's in the code segment? Static data segment?
- What's in the stack segment?
  - How is it allocated? How big is it?
- What's in the heap segment?
  - How is it allocated? How big?

Example

- Stack?
- Heap?
- Data?
- …

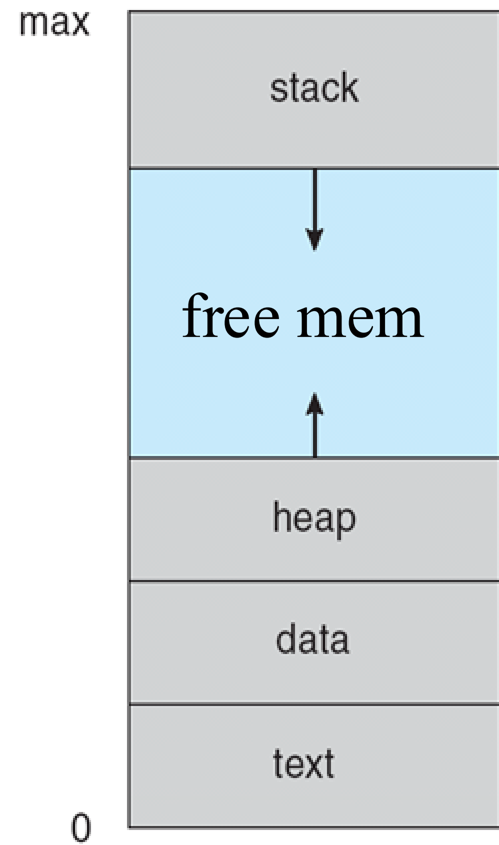
```c
#include <stdio.h>
#include <stdlib.h>

int global_var = 10;

void stack_function() {
    int stack_var_in_func = 30;
    printf("Address of stack_var_in_func: %p\n", (void*)&stack_var_in_func);
}

int main() {
    int stack_var = 20;

    int *heap_var = (int*)malloc(sizeof(int));
    if (heap_var == NULL) {
        fprintf(stderr, "Failed to allocate memory on the heap\n");
        return 1;
    }
    *heap_var = 40; // Storing a value in the heap memory.

    // The programmer is responsible for freeing heap memory.
    free(heap_var);

    return 0;
}
```

①

②

③

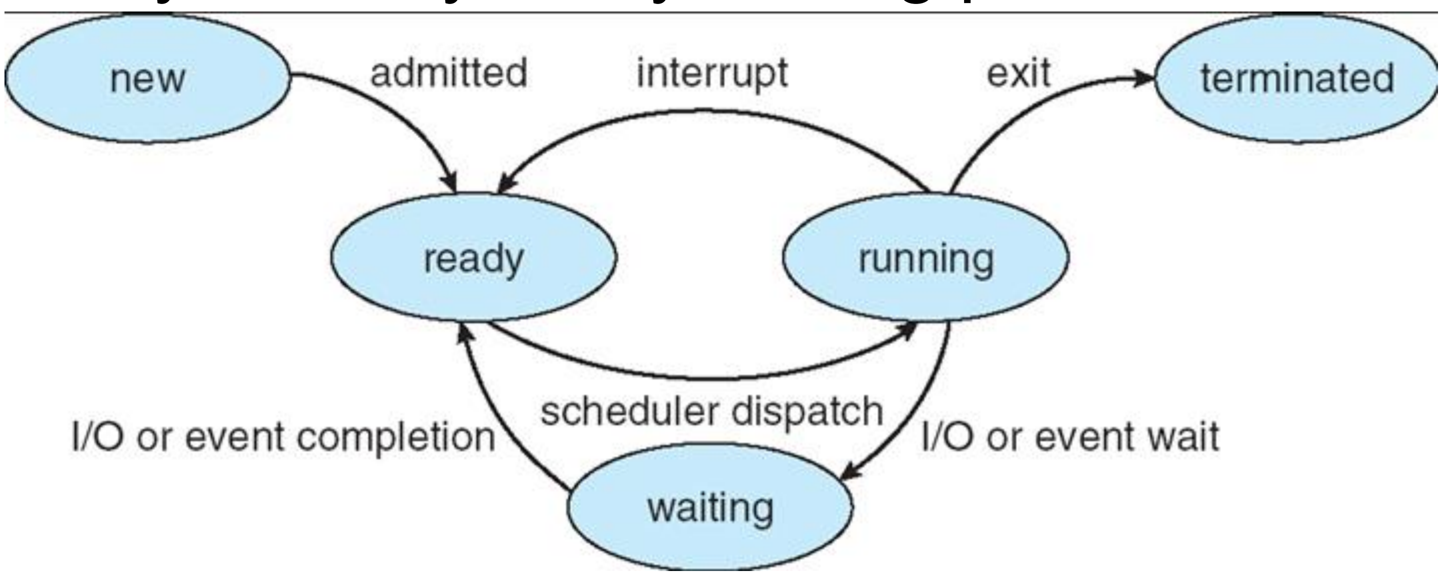
Processes



- Process: a program in execution
- Program: passive entity
  - static binary file on storage
    - e.g., gcc -o hello hello.c
- Process: active entity; resource allocated!
    - ./hello
    - text (code); data (static), stack, heap
    - process control block (PCB)

Process states

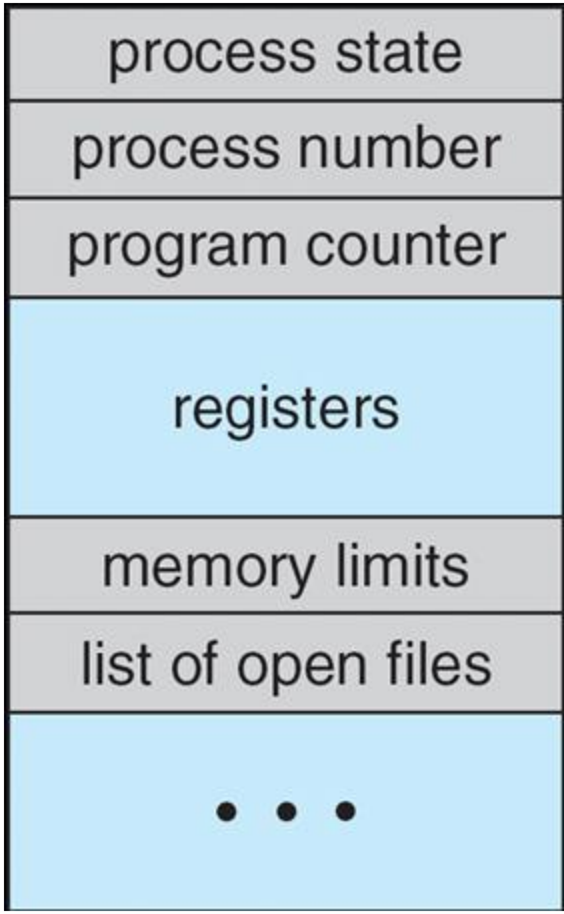
- E.g., one CPU (core)
  – one running process at any time
  – maybe many ready/waiting processes



why interrupt?

Process control blocks

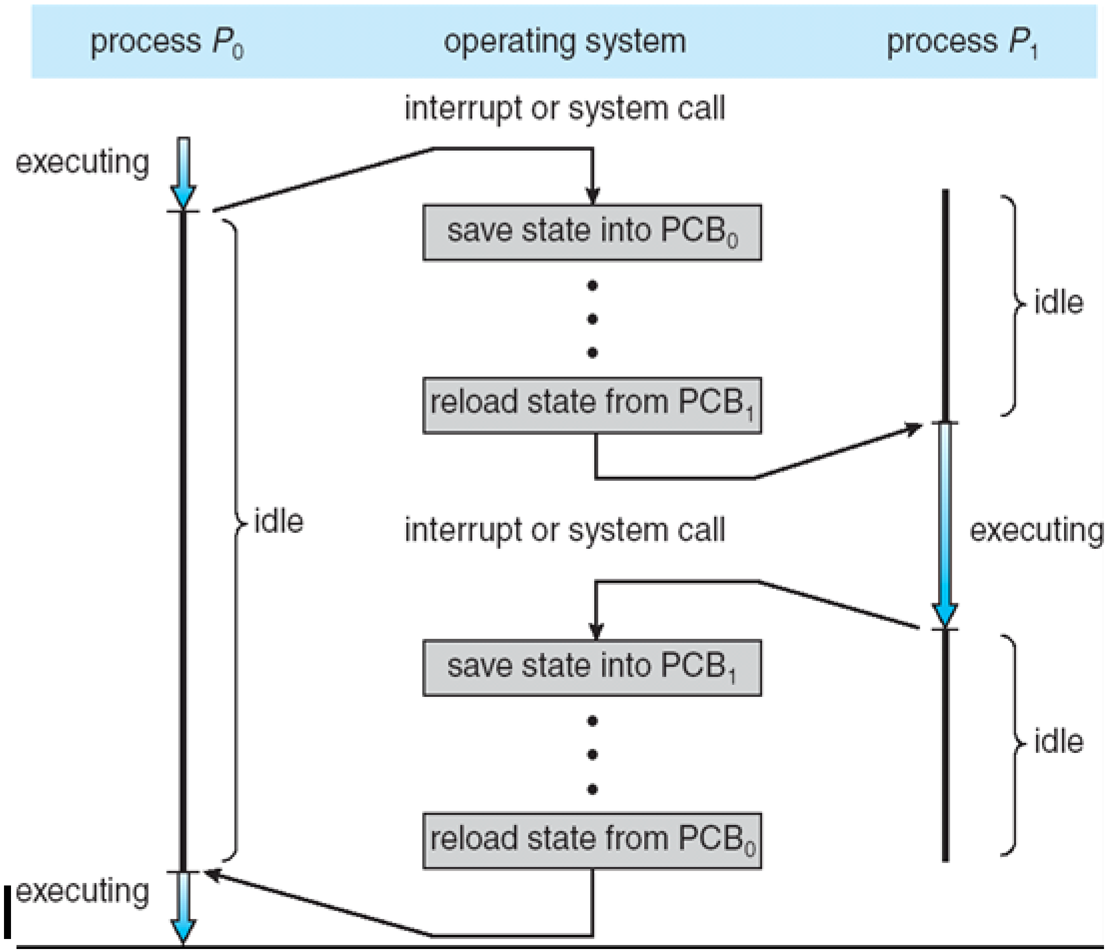
- PCB: keep track processes
  - state: ready/running, etc
  - CPU
    - PC, registers, priority, etc
  - memory
    - memory control information
  - I/O
    - e.g., list of opened files
  - accounting



“PCB exhausted!”

Context switching

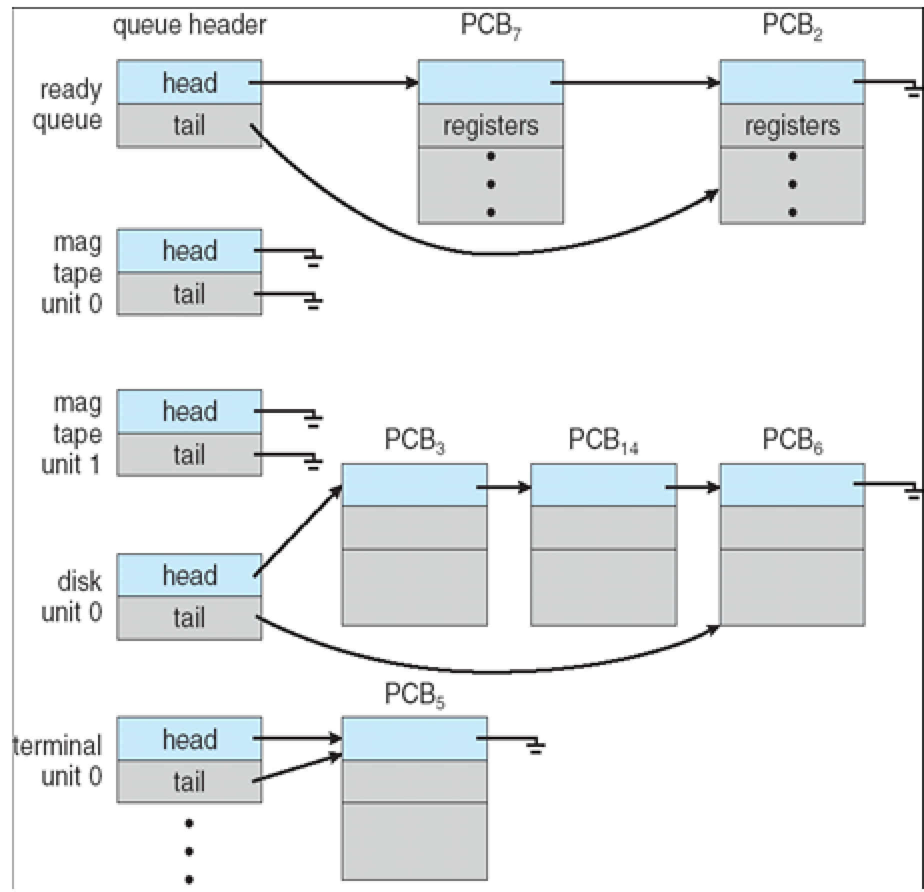
- Context switch
  - save states
  - restore states
- When
  - timer
  - I/O, memory
  - trap
  - waiting sys call



when more than two processes

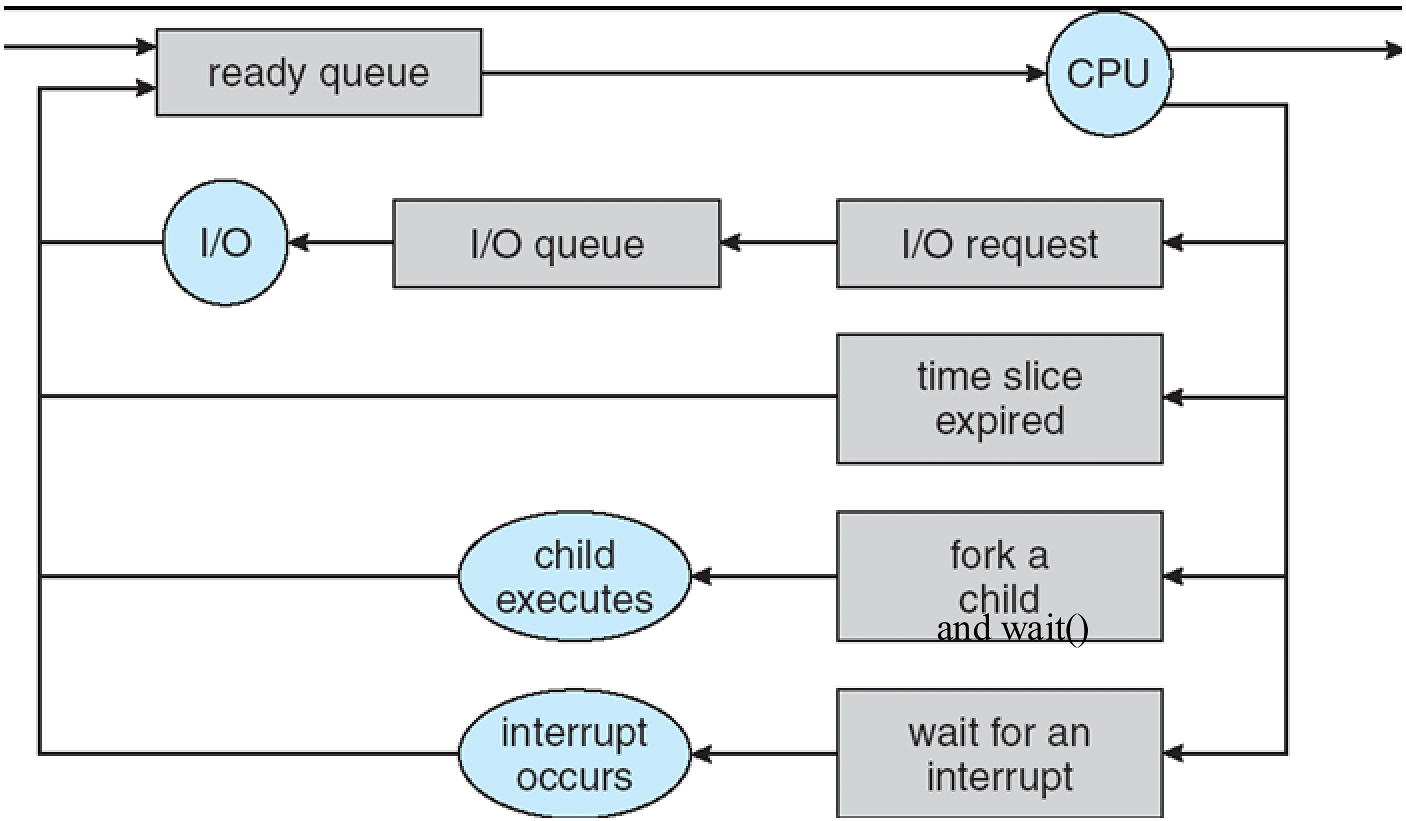
Process scheduling

- 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

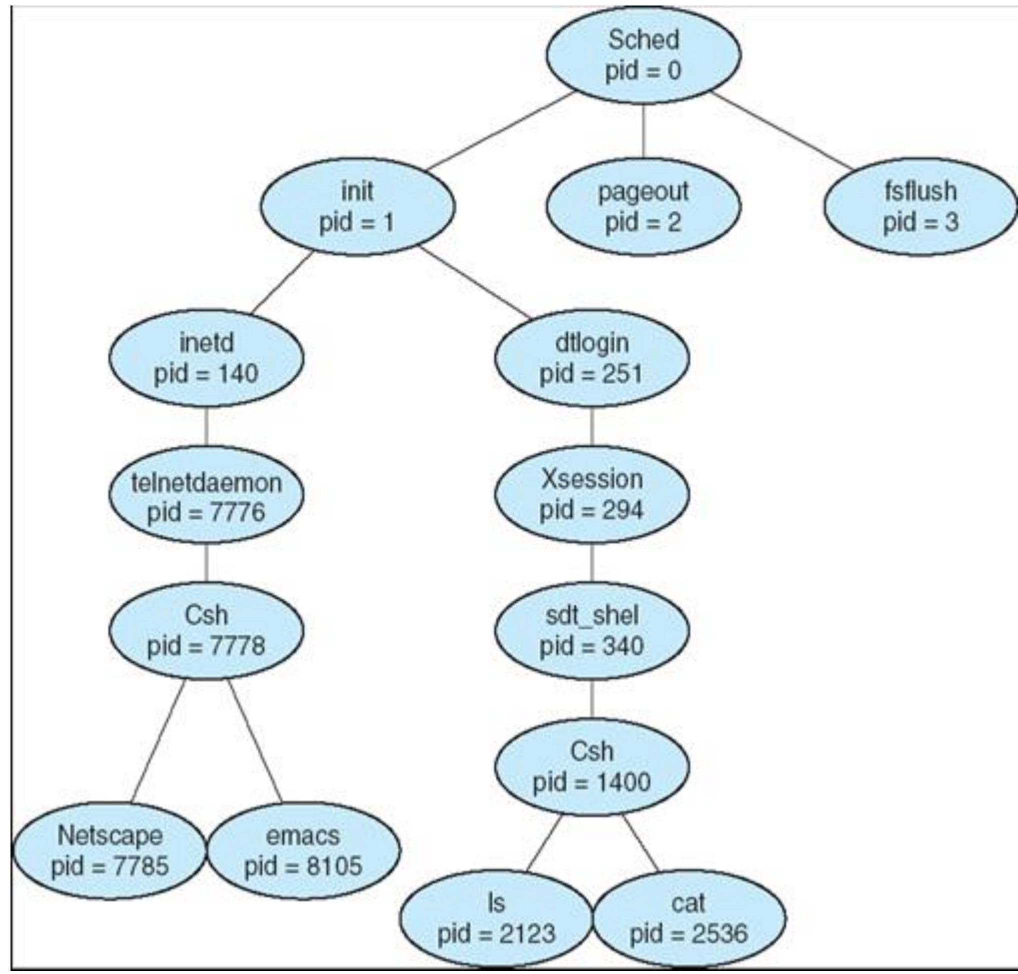
- Process and process scheduling
  - process vs program
  - process control block
    - context switch: what to save/restore
  - process scheduling
- Explore further
  - process status: /bin/ps
  - top CPU processes: /usr/bin/top

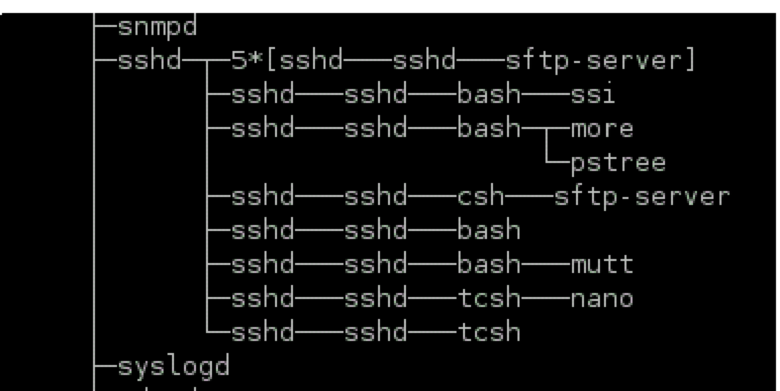
Process creation

- 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





pstree on linux.csc.uvic.ca

Parent vs child processes

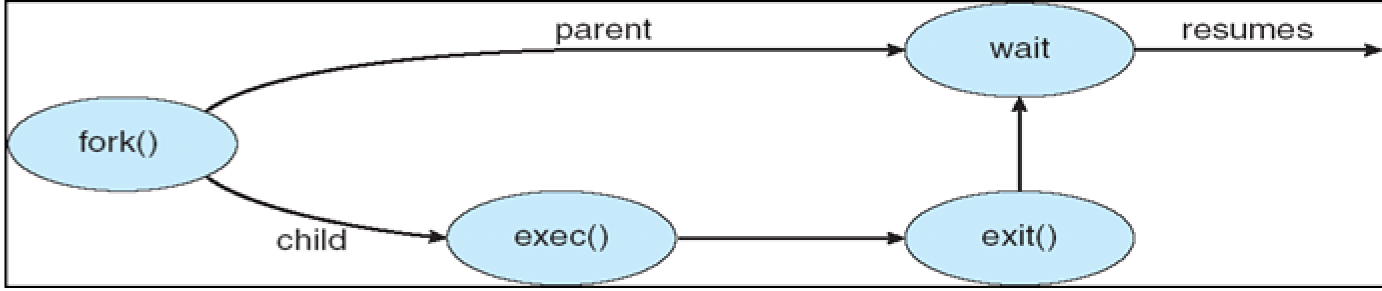
- Process: running program + resources
- Resource sharing: possible approaches
  – all shared, or
  – some shared (e.g., read-only code), or
  – nothing shared*
- Process execution: possible approaches
  – parent waits until child finishes, or
  – parent and child run **concurrently***

* 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()?

Win In-class Bonus (1 point)

- Submit your answer within 10min from the start
  - for the following two questions, 0.4 points each
  - 0.2 points for active participation (as long as you submit, name, V#, A0#)
  - group discussion is allowed
  - no internet searching
- How to use your bonus grades
  - Final = min{100, weighted grades(midterm, assignments) + bonus}
- More bonuses on the way

Here is the code of file `ex1.c`:

Q1

```
# include <stdio.h>
# include <sys/types.h>
# include <unistd.h>

int main()
{
    fork();
    fork();
    fork();
    printf("Hello CSC360! PID = %d\n", getpid());
    return 0;
}
```

After executing the commands `gcc ex1.c` and `./a.out`, how many lines of `"Hello CSC360! PID = ###"` we will have?

Here is the code of file `ex1.c`:

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

int main(int argc, char *argv[])
{
    printf("PID of ex1.c = %d\n", getpid());
    char *args[] = {"Hello", "CSC", "360", NULL};
    execv("./ex2", args);
    printf("Back to ex1.c");
    return 0;
}
```

And here is the code of file `ex2.c`:

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

int main(int argc, char *argv[])
{
    printf("We are in ex2.c\n");
    printf("PID of ex2.c = %d\n", getpid());
    return 0;
}
```

After executing the commands `gcc ex1.c -o ex1`, `gcc ex2.c -o ex2`, and `./ex1`, the output will be:

Q2

A.
```
PID of ex1.c = 5962
We are in ex2.c
PID of ex2.c = 5962
```

B.
```
PID of ex1.c = 5962
We are in ex2.c
PID of ex2.c = 5963
```

C.
```
PID of ex1.c = 5962
Back to ex1.c
```

D.
```
PID of ex1.c = 5962
We are in ex2.c
PID of ex2.c = 5962
Back to ex1.c
```

Process termination

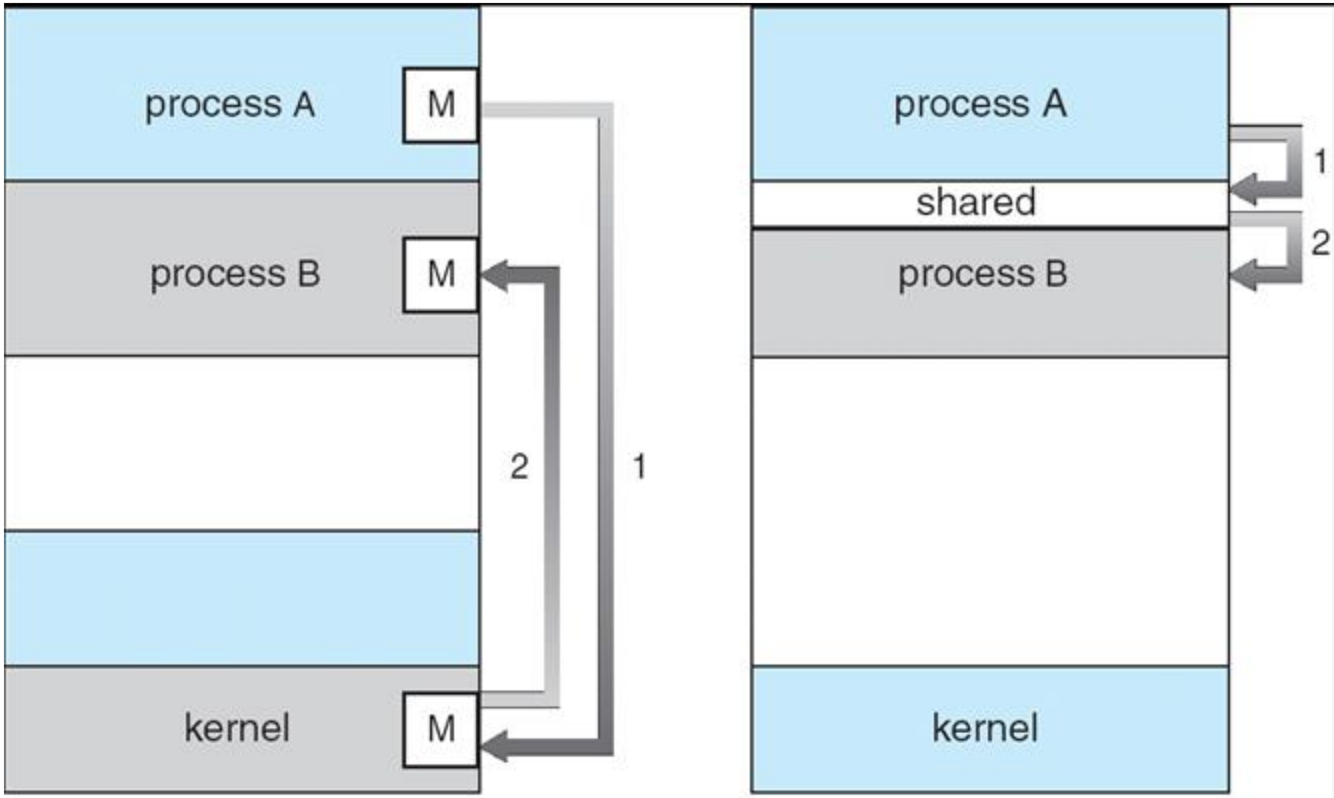
- Terminate itself: exit()
  – report status to parent process
  – release allocated resources
- Terminate child processes: kill(pid, signal)
  – actually send a signal to the child
    - child resource exceeded, child process no longer needed, and so on
  – parent is exiting
    - cascading termination,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



(a) (b)

* 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)