Reading Reference:

Textbook: Chapter 3

Reference Text (Stevens) Chapter 7,8

W4.1 – UNIX PROCESS ELEMENTS

Aakash Tyagi
CSCE 313 Spring 2017

Key Learnings from Week 3

- Shell Basics
- Spawning Processes
  - Call to `fork`
    - One call, two returns (parent gets child id, child gets 0)
- Replacing Program Executed by Process
  - Call to `execv` (or variant)
    - One call, (normally) no return

Recap Example 1: Fork and Exec


```
int main(int argc, char* argv[] ) {
 int pid = fork();
       if ( pid == 0 ) {
      execvp("date", argv);
}
/* parent sleeps for 2sec to let child go first*/
  wait( 2 );
  printf( "Finished the parent process\n"
         " - the child won't get here--you will only
see this once\n" );
  return 0;
}
```

- What would the child print?
- What would the parent print?
- What would happen if the if (pid == 0) check was removed?

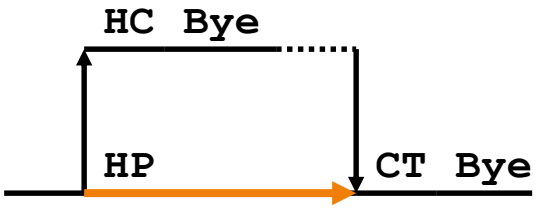
`wait`: Synchronizing With Children

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is `pid` of child process that terminated
  - If `child_status != NULL`, then integer it points to will be set to indicate why child terminated

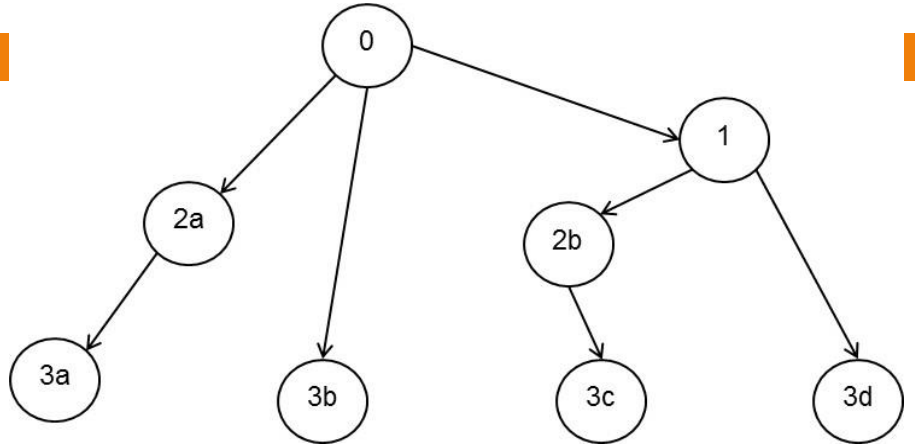
wait: Synchronizing With Children


```
Void wait_demo() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit(0);
}
```



Discussion- Example 2



How would you create the process tree. The numbers inside the circles simply represent the order of process creation

Discussion – Example 3

- How many lines will be printed? Will the output always be the same?

```
File Edit Options Buffers Tools C Help
/* forkdemo2.c - shows how child processes pick up at the return
 *               from fork() and can execute any code they like,
 *               even fork().  Predict number of lines of output.
 */

#include <stdio.h>

main()
{
  printf("my pid is %d\n", getpid() );
  fork();
  fork();
  fork();
  printf("my pid is %d\n", getpid() );
}
```

Discussion: Example 4

What is the output of this program?
Will it remain the same?

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

int main (int argc, char *argv())
{
        int childpid;
        int count1=0, count2=0;
        printf("Before it forks\n");
        childpid=fork();
        if(childpid == 0)
        {
                printf("This is a child process\n");
                while(count1 < 2)
        {
                printf("Child Process: %d\n", count1);
                sleep(5);
                count1++;
        } } else {
                printf("This is the parent process\n");
                while(count2 < 3)
                {
                        printf("Parent Process: %d\n", count2);
                        sleep(2);
                        count2++;
                }
        }
        return 0;
}
```

Discussion – Example 5

4. [12 POINTS] In the following piece of code, what will the output be at Line (X)?

```
int value = 5;

int main() {

    pid_t pid = fork();

    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    } else if (pid > 0) { /* parent process */
        wait(NULL); /* wait for child to terminate */
        cout << "PARENT: value = " << value << endl; /* LINE (X) */
        return 0;
    }
}
```

The output at Line (X) is (enter the value):

PARENT: value = _____

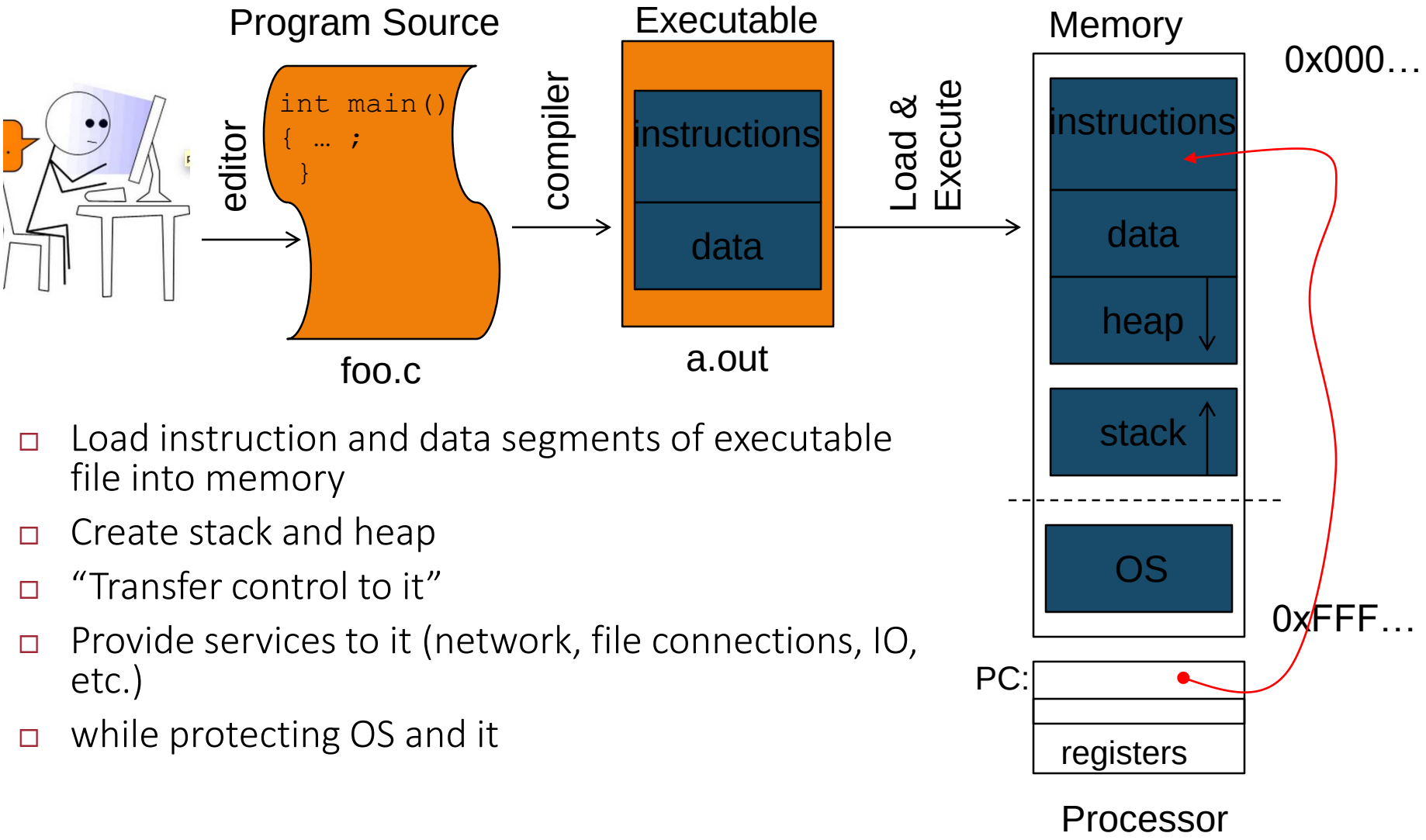
Questions to ponder about processes – what have we learned so far?

- ✓ How is a process created?
  - How is a process deleted?
  - Is there a user process and kernel process
  - Where do we keep information about a process
- ✓ Does a process have to run through completion from start to finish or can it be interrupted?
  - Do processes have priorities?
  - What are the relationships between multiple processes in a system?
  - Can we have multiple processes related to the same program? Would multiple processes of the same program share addresses during execution?
- ✓ How does a program run a program?
- ✓ How does a parent wait for a child to exit?

Rest of the Discussion

- Process Life-Cycle
- What does it take to execute the life-cycle?
- Orphan, Zombie Processes

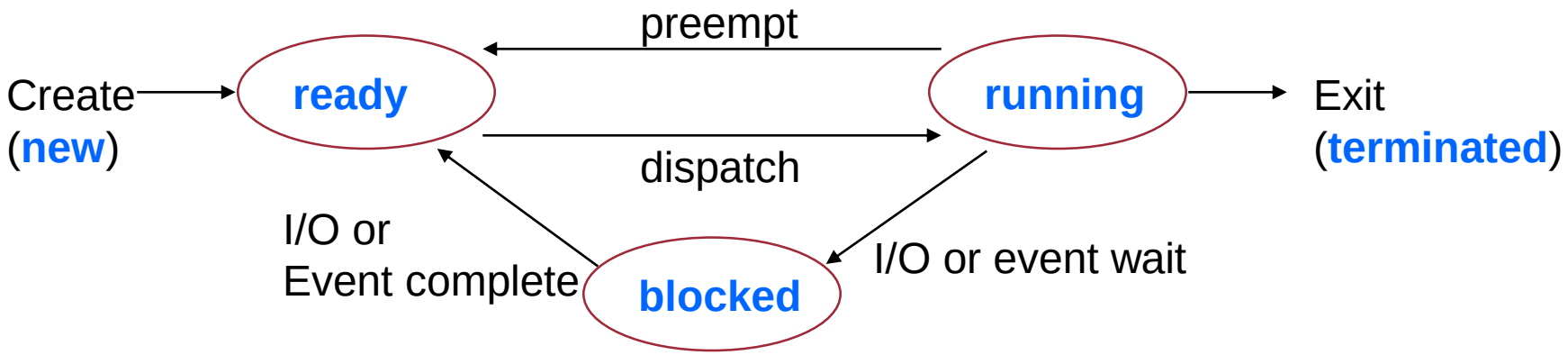
OS Bottom Line: Run Programs



- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to it”
- Provide services to it (network, file connections, IO, etc.)
- while protecting OS and it

States of a Process

- *User view*: A process is executing continuously
- *In reality*: Several processes compete for the CPU and other resources
- A process may be
  - **Running**: it holds the CPU and is executing instructions
  - **Blocked(waiting)**: it is waiting for some I/O event to occur
  - **Ready**: it is waiting to get back on the CPU

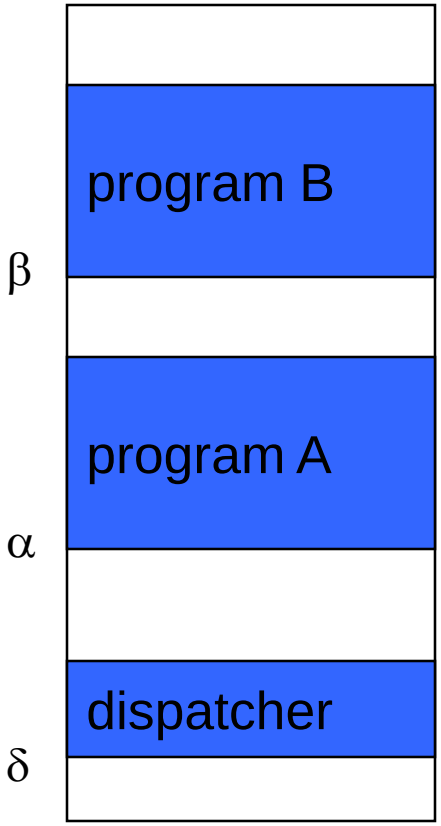


Questions

- How many processes can be in the running state simultaneously?
- What state do you think a process is in most of the time?
- How many processes can a system support? **

The Execution Trace of Processes

- Two processes and a dispatcher



Traces of processes **A** and **B**

α	β
α+1	β+1
α+2	β+2
α+3	β+3
α+4	β+4
α+5	β+5
α+6	β+6
α+7	β+7
α+8	β+8
α+9	β+9
α+10	β+10
α+11	β+11

Trace of **dispatcher**

δ
δ+1
δ+2
δ+3
δ+4

...
β
β+1
β+2
β+3
β+4
δ
δ+1
δ+2
δ+3
δ+4
α
α+1
α+2
α+3
α+4
δ
δ+1
δ+2
δ+3
δ+4
β+5
β+6
β+7
...

Process Data Structures

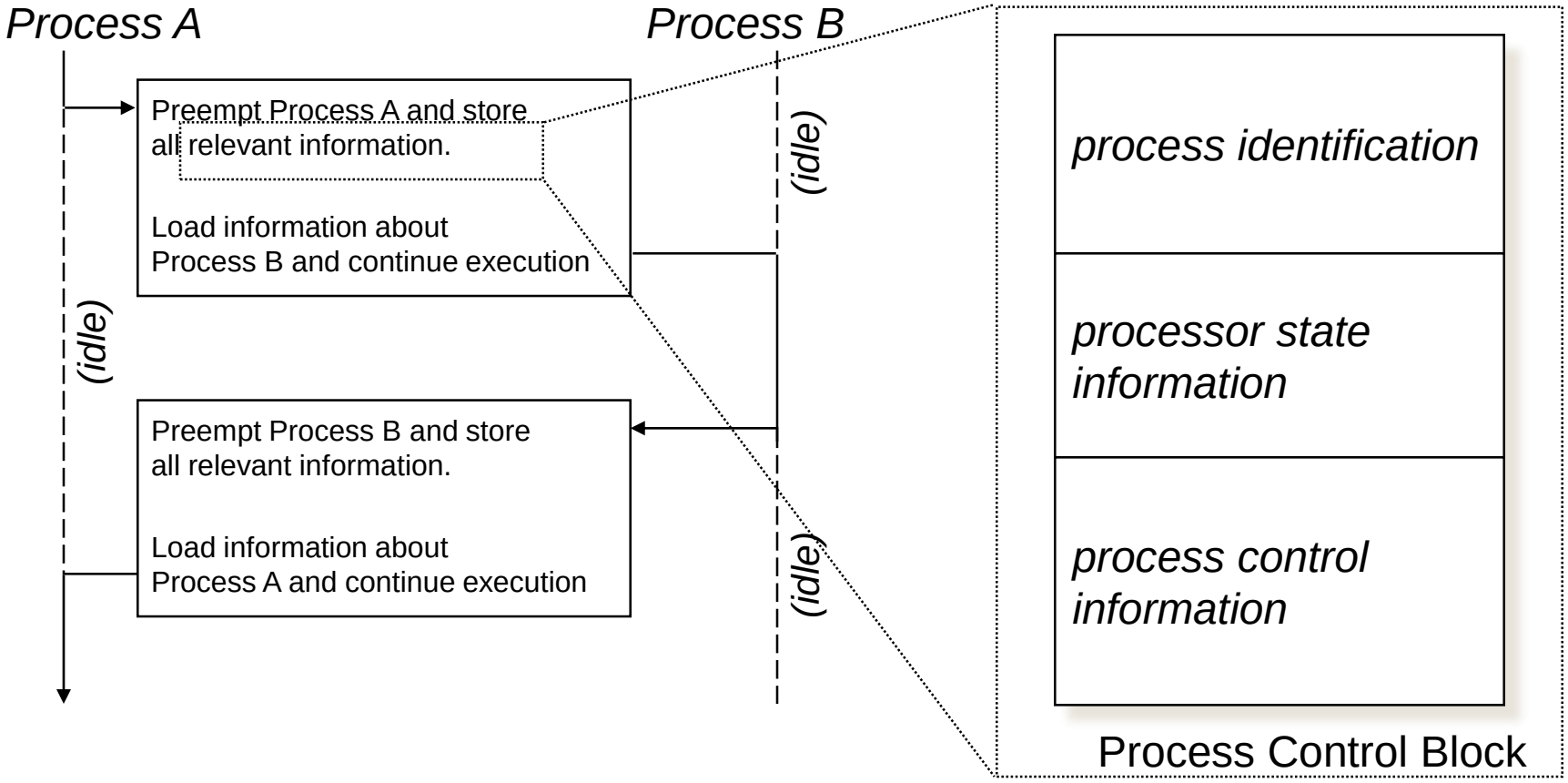
- How does the OS represent a process in the kernel?
  - At any time, there are many processes in the system, each in its particular state
  - The OS data structure representing each process is called the Process Control Block (PCB)
  - The PCB contains all of the info about a process
  - The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
    - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

Process Control Block (PCB)

process identification	process id parent process id user id etc…
processor state information	register set condition codes processor status
process control information	process state scheduling information event (wait-for) memory-mgmt information owned resources (e.g., list of opened files)

Process Context Switch

- Mechanism of a process switch:



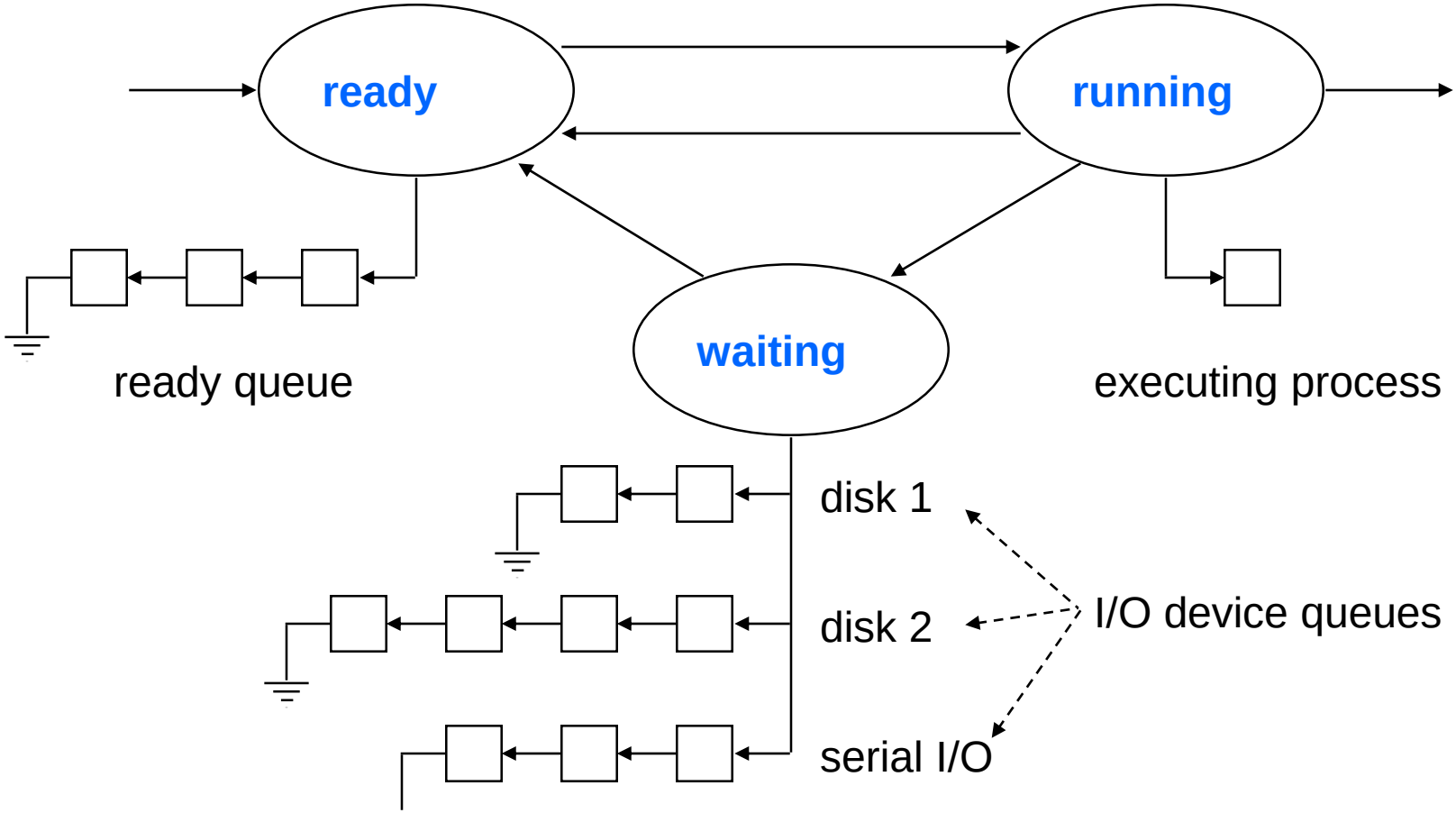
Context Switch: PCB and the Hardware States

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
- The process of changing the CPU hardware state from one process to another is called a context switch
  - This can happen 100 or 1000 or more times a second!

Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

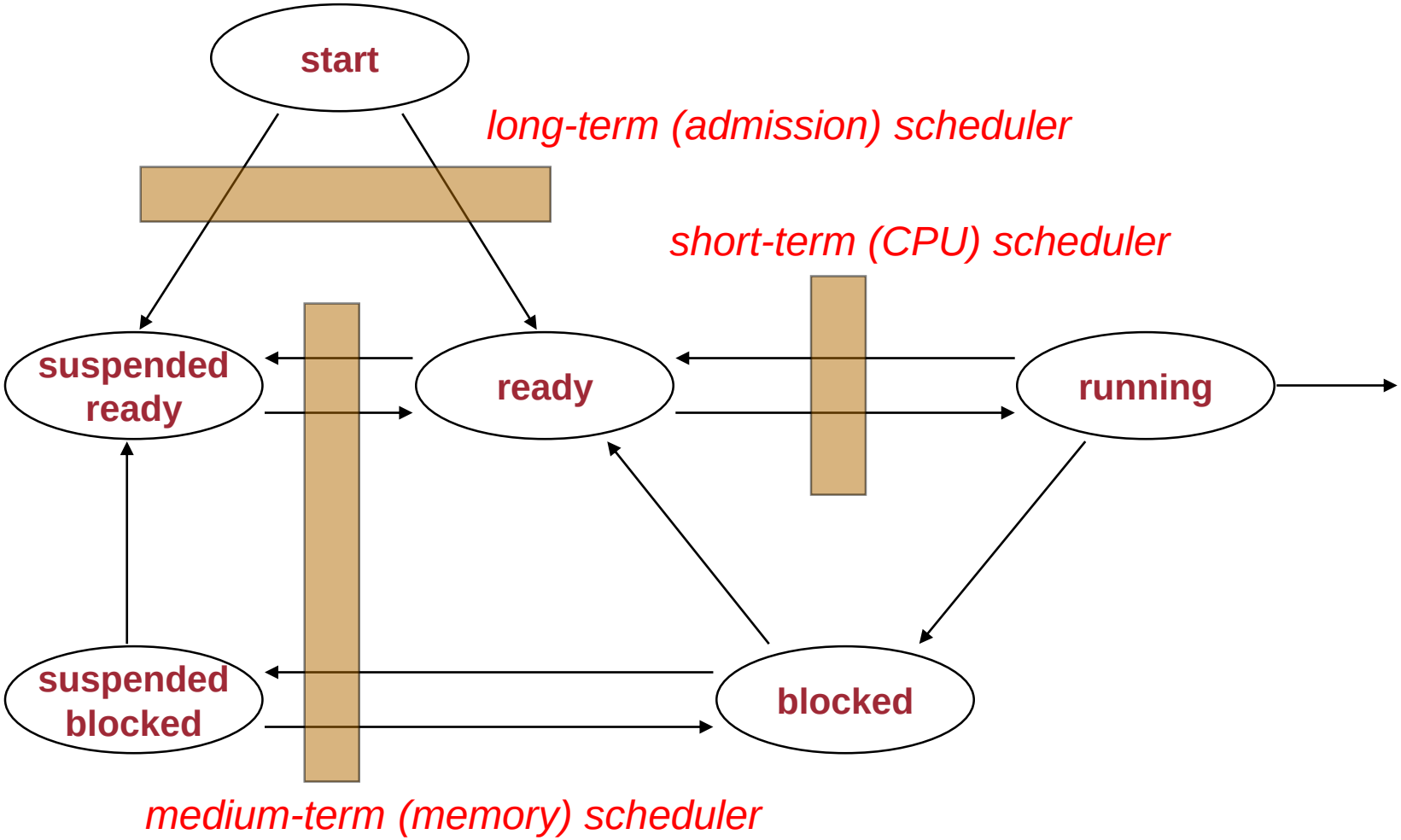
Example for the Use of PCBs: Process Queues



Schedulers

- **Long-term scheduler** (or job scheduler)
  - selects which processes should be brought into the ready queue
  - controls degree of multiprogramming
  - must select a good process mix of I/O-bound and CPU-bound processes
- **Short-term scheduler** (or CPU scheduler)
  - selects which process should be executed next and allocates CPU
  - executes at least every 100ms, therefore must be very fast
- **Medium-term scheduler** (swapper)
  - in some Oss
  - sometimes good to temporarily remove processes from memory (suspended)

Schedulers


start
long-term (admission) scheduler
short-term (CPU) scheduler
suspended ready
ready
running
suspended blocked
blocked
medium-term (memory) scheduler

Zombie Process

- Idea
  - When process terminates, still consumes system resources
    - Various tables maintained by OS (to store exit status)
  - Called a “zombie”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process

Zombie Example

```
main ()
{
  if (fork() == 0) {
    /* Child */
    printf("Terminating Child, PID = %d\n", getpid());
    exit(0);
  } else {
    printf("Running Parent, PID = %d\n", getpid());
    while (1)
      ; /* Infinite loop */
  }
}
```

```
:: cc forkdemo3.c

[tyagi]@linux2 ~/csce313/sp15/forkdemo2> (12
:: a.out
Running Parent, PID = 45839
Terminating Child, PID = 45840
^Z
[2]+  Stopped                 a.out

[tyagi]@linux2 ~/csce313/sp15/forkdemo2> (12:23:21 02/10/15)
:: bg
[2]+ a.out &

[tyagi]@linux2 ~/csce313/sp15/forkdemo2> (12:23:24 02/10/15)
:: ps
  PID TTY          TIME CMD
34302 pts/37   00:00:00 bash
45563 pts/37   00:00:00 emacs-nox
45839 pts/37   00:00:48 a.out
45840 pts/37   00:00:00 a.out <defunct>
46047 pts/37   00:00:00 ps

[tyagi]@linux2 ~/csce313/sp15/forkdemo2> (12:23:27 02/10/15)
:: kill -9 45839
[2]-  Killed                  a.out

[tyagi]@linux2 ~/csce313/sp15/forkdemo2> (12:23:43 02/10/15)
:: ps
  PID TTY          TIME CMD
34302 pts/37   00:00:00 bash
45563 pts/37   00:00:00 emacs-nox
46177 pts/37   00:00:00 ps
```

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped

Orphan Process

- Without a parent process
- Adopted immediately by the “Init” process
  - Sometimes orphans are killed with the SIGHUP signal
  - (it can be overruled with the nohup handler)

Summary

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - fork/exec (Unix)

Next Class

- Unix I/O