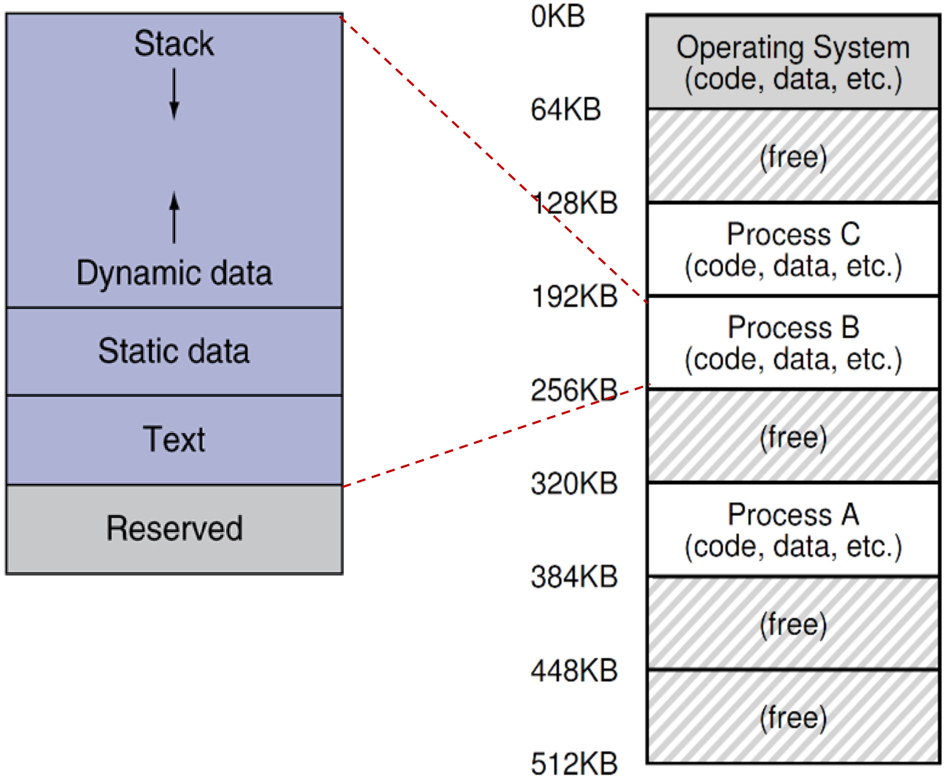
EEEC7376: Operating Systems Interface and Implementation

CPU Virtualization

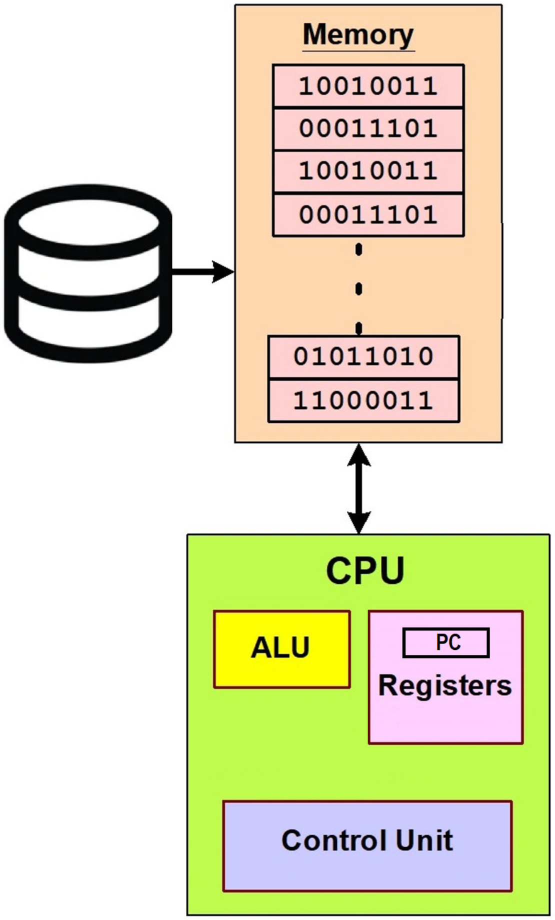
Program vs Process

- The **program** itself is a lifeless thing (a passive entity): it just sits there on the disk as a bunch of bytes representing instructions. It is the OS that takes these bytes and gets them running, transforming the program into a process.
- So, a **process** is a running program.



The Processes

- When you run a program (by clicking on it or using a command line), the **OS loader** creates a process for the program by loading it from the disk to the memory address space of the process and starts executing the first instruction of the program (by pointing the processor **program counter** to the address of this instruction).
- A typical computer may be seemingly running tens or even hundreds of processes at the same time.
- The OS promotes the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few).



Time Sharing and Scheduling

- **Time sharing** is the **mechanism** used by an OS to share a CPU. It allows the CPU to be used for a little while by one process, and then a little while by another, and so forth.

- A **context switch** gives the OS the ability to stop running one process and start running another on a given CPU.

- A **scheduling policy** allows the OS to make the decision of which process to start running and which one to stop.

A Process Machine State

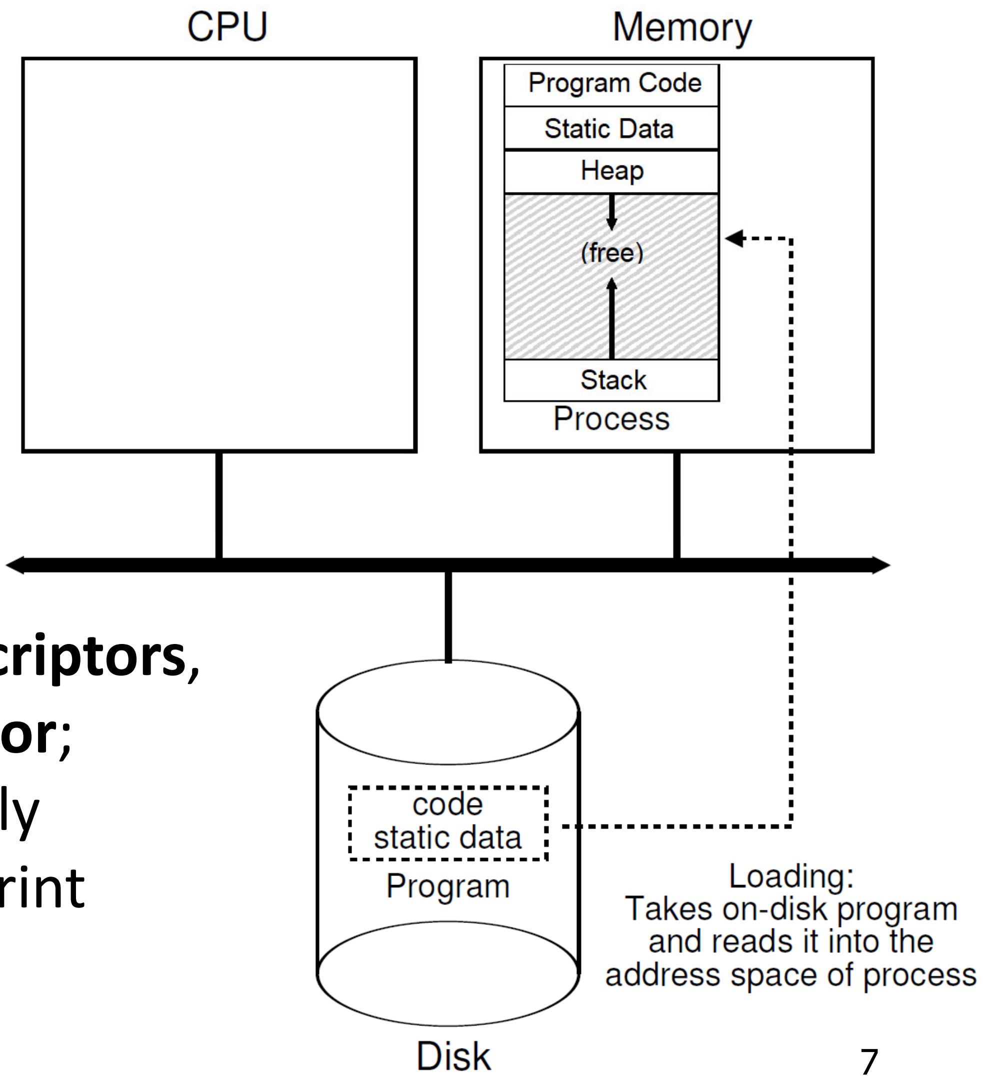
- A process **machine state** represents the components of the machine (the computer) a process can read, or update and they include:
  - The **address space** in the memory where the instructions and data of the process lie.
  - The processors **registers** that are currently used by the process instructions. In particular, the following registers:
    - **PC** (program counter) that contains the address of the instruction of the process to be executed next.
    - **SP** (stack pointer) and **FP** (frame pointer) that are used to manage the stack for function parameters, local variables, and return addresses
  - The **I/O information** that might include information about the files the process currently has open.

Process Related APIs

- Any OS must include APIs (Application Programming Interfaces) to perform different tasks with the processes such as:
- **Create**: The OS must include a method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program.
- **Destroy**: The OS also provides an interface to destroy processes forcefully as a user may wish to kill a process.
- **Wait**: Sometimes it is useful to wait for a process to stop running; thus, some kind of waiting interface is often provided.
- **Suspend**: To stop a process from running, for a while, and then resume it.
- **Status**: There are usually interfaces to get some status information about a process, such as how long it has run for, or what **state** it is in.

Process Creation

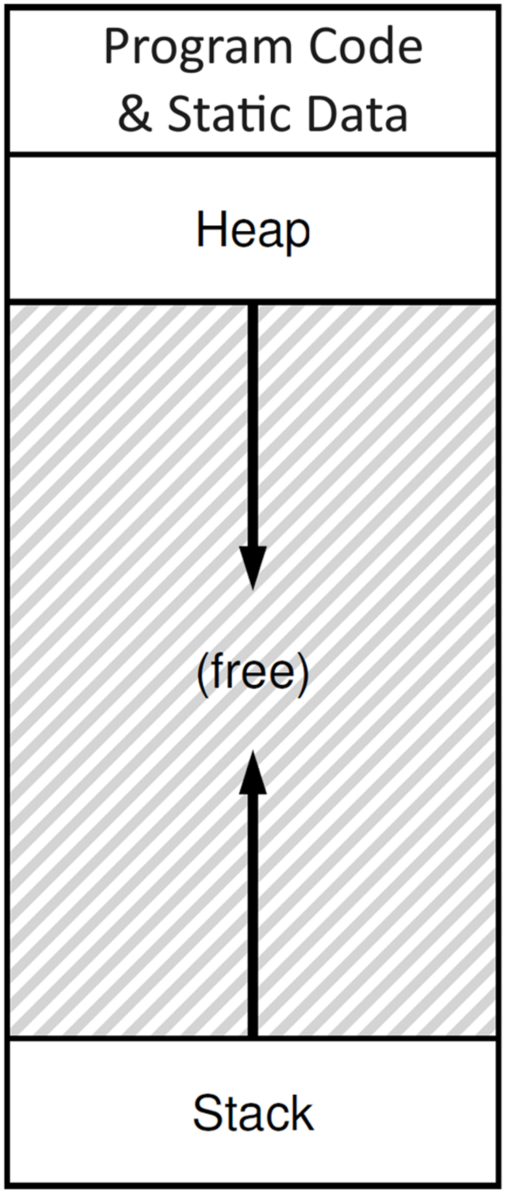
- The first thing that the OS must do to run a program is to load its **code** and any **static data** (e.g., global variables) into the address space of the process.
- Modern OS **loads pieces** of the program code or data only as they are needed during the process execution.
- The OS also allocates memory for the process **Stack** and **Heap**.
- In UNIX systems, each process by default, has **three open file descriptors**, for **standard input**, **output**, and **error**; these descriptors let programs easily read input from the terminal and print output to the screen.



Loading:
Takes on-disk program and reads it into the address space of process

Process Stack and Heap

- *C* programs use the **stack** for local variables, function parameters, and return addresses.
- The OS also initializes the **stack** with `main()` function arguments, i.e., **`argc`** and the **`argv`** array.
- In *C* programs, the **heap** is used for explicitly requested dynamically-allocated data; programs request such space by calling **`malloc()`** and free it explicitly by calling **`free()`**.



Process Execution

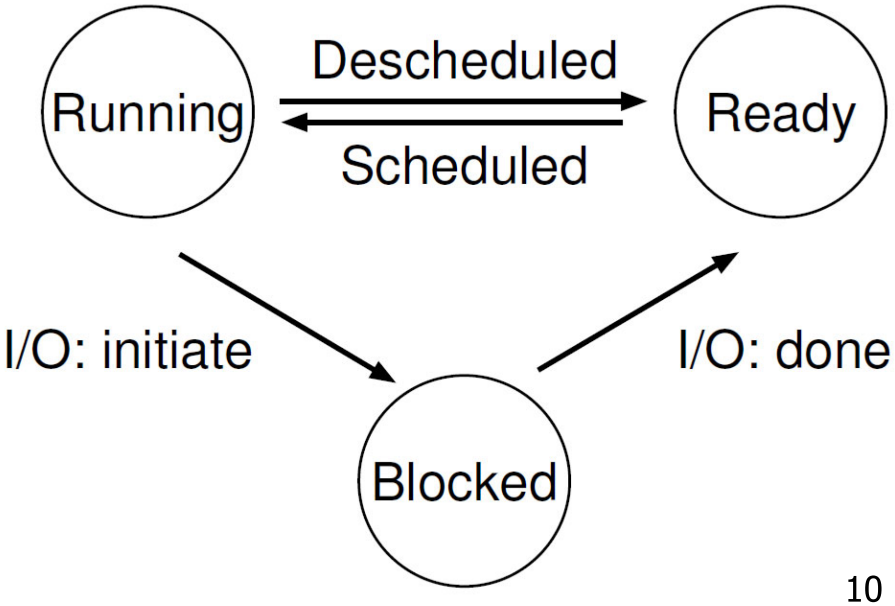
- By loading the code and static data into memory, by creating and initializing the stack/heap, and by doing other work as related to I/O setup, the OS has now (finally) set the stage for the process execution.

- It thus has one last task: to start the program running at the entry point, namely `main()` by transferring control of the CPU (updating its PC register) to the newly-created process, and thus the process begins its execution.

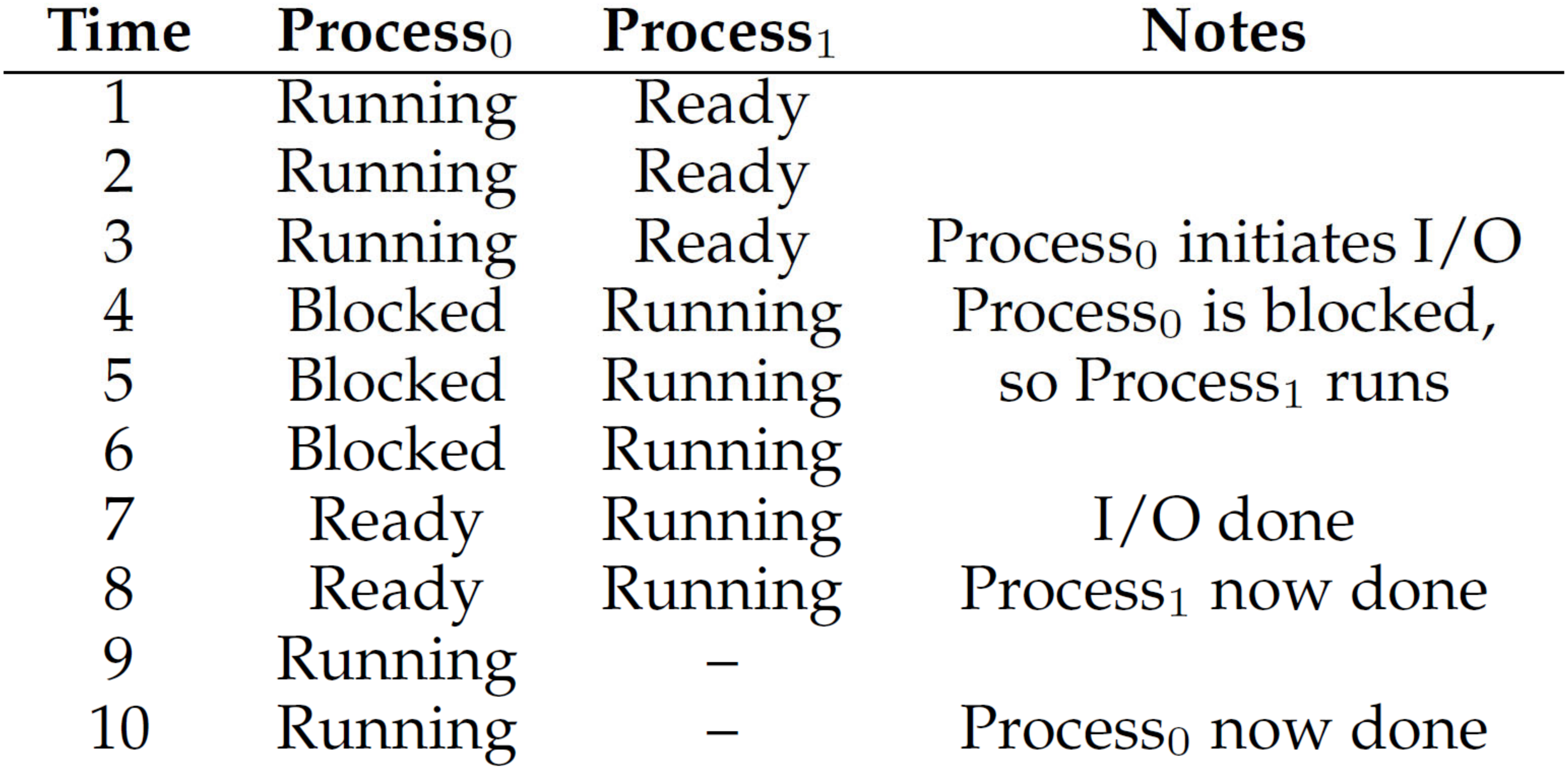
Process States

- At any given time, a process can be in one of three states:
  - **Running**: In the running state, a process is running on a processor. This means the processor is executing the process instructions.
  - **Ready**: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place.

    A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.



Example: Tracing Processes States



Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process$_0$ initiates I/O
4	Blocked	Running	Process$_0$ is blocked,
5	Blocked	Running	so Process$_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process$_1$ now done
9	Running	–	
10	Running	–	Process$_0$ now done

The `fork()` API Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/cpu-api
- Run the `p1.c` code several times.
- *Observation*:
  - The `fork()` system call is used to create a new process and its return value to the parent is the **PID** of the created child.
  - The “hello, world” message only got printed out once.
  - The **output is not deterministic** as either the child’s or the parent’s message might appear first.
- *Conclusion*:
  - The process that is created is an (almost) exact copy of the calling process. So, there are two copies of the program p1 running, and both are about to return from the `fork()` system call.
  - The newly-created process (the child) doesn’t start running at `main()`, like you might expect; rather, it just comes into life as if it had called `fork()` itself.
  - Depending on the scheduler decision, either the child or the parent might run first.

fork() Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {  // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {  // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
                getchar();
           }
    return 0;
}
```

For a successful fork, what is the order of the printed messages?

The `wait()` API Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/cpu-api
- Run the `p2.c` code several times.
- *Observation*:
  - The child’s message always appears first.
  - There is a delay in displaying the parent’s message due to de-scheduling of the parent.
- *Conclusion*:
  - The wait() API makes the parent delay its execution until the child finishes executing.
  - Adding a wait() call to the code above makes the output deterministic.

wait() Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");   exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1);
    } else { // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
        getchar();    }
    return 0;
}
```

The execvp() API Experiment

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/cpu-api
- Run the p3.c code several times.
  (**wc** counts the number of lines, words, and bytes in a file)
- *Observation*:
  - The printf() in the child after execvp() **does not run**.
  - The forked child’s **process ID** is the same as the wc “execed” process ID.
- *Conclusion*:
  - fork() is useful if you want to keep running copies of the same program.
  - execvp() allows you to run a different program.
  - a successful call to execvp() never returns in the child.
  - execvp() does not create a new process, rather, it transforms the currently running process (formerly p3 child) into a different running program (wc) with the same PID.
  - Using both fork() and execvp() is essential in building a UNIX shell

execvp() Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  printf("hello world (pid:%d)
          \n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr,
          "fork failed\n");
        exit(1);
    }
else if (rc == 0) { // child (new process)
  printf("hello, I am child (pid:%d)\n",
                (int) getpid());
  char *myargs[3];
  myargs[0] = strdup("wc"); // "wc" (word count)
  myargs[1] = strdup("p3.c"); // file to count
  myargs[2] = NULL; // marks end of array
  execvp(myargs[0], myargs);  // runs word count
  printf("this shouldn't print out\n");
} else {
   // parent goes down this path (original process)
  int wc = wait(NULL);
  printf("hello, I am parent of %d (wc:%d)
         (pid:%d)\n", rc, wc, (int) getpid());
    }
 return 0;
}
```

How can we change the code so that the "this shouldn't print out" message will be printed?

The `execvp()` API

- In the previous program, `p3.c`, the child process calls `execvp()` in order to run the program `wc`, which is the word counting program.

- According to the arguments passed by the child, `wc` runs on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the program source file.

Shell Using `fork()` and `execvp()`

- The shell is just a user program that shows you a prompt and then waits for you to type a command (i.e., the name of an executable program, plus any arguments) into it.
- The shell then figures out where in the file system the executable of the command resides.
- It calls `fork()` to create a new child process to run the command, calls some variant of `execvp()` to run the command, and then waits for the command to complete by calling `wait()`.
- When the child completes, the shell returns from `wait()` and prints out a prompt again, ready for your next command.
- It repeats that in a loop until the typed command is **`exit`**

Output Redirection Experiment (1 of 2)

- The separation of fork() and execvp() allows the shell to do a whole bunch of useful things rather easily. For example:

  ```
  prompt> wc p4.c > p4.output
  ```

  - Here the output of the program wc is redirected into a text file p4.output
- The way the shell accomplishes this task is quite simple:
  - When the child is created, before calling execvp(), the shell closes standard output and opens the file p4.output. By doing so, any output from the soon-to-be-running program wc are sent to the file instead of the screen.

Output Redirection Experiment (2 of 2)

- Code: github.com/remzi-arpacidusseau/ostep-code/tree/master/cpu-api
  Program `p4.c` code does the redirection as described.

- This redirection works due to how the OS manages the file descriptors table. STDOUT_FILENO is the default first output in the table. When the child closes it and opens p4.output, subsequent writes by the child process to the standard output file descriptor (e.g., by `printf()`) will be routed transparently to the newly-opened file instead of the screen.

Output Redirection Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <assert.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
 int rc = fork();
 if (rc < 0) {// fork failed; exit
   fprintf(stderr,
         "fork failed\n");
         exit(1);
   }
else if (rc == 0) {
   // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output",
         O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
         // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");  // "wc" (word count)
    myargs[1] = strdup("p4.c");  // file to count
    myargs[2] = NULL;          // marks end of array
    execvp(myargs[0], myargs);  // runs word count
 } else {
         // parent goes down this path (original process)
         int wc = wait(NULL);
         assert(wc >= 0);
    }
    return 0;
}
```

Pipes

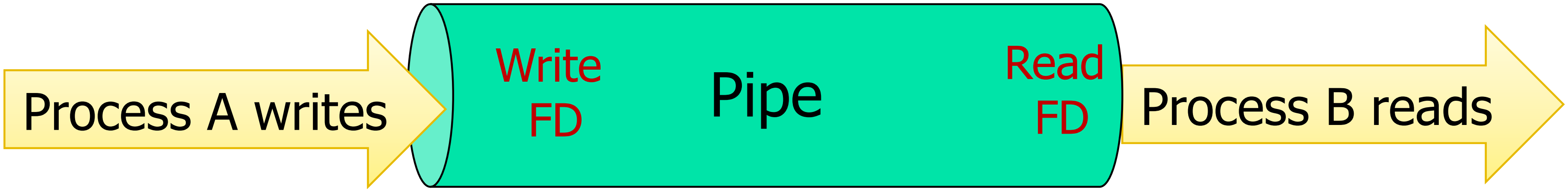
- Shell pipes are used to move data between programs. For example:

```
prompt> ls -l | wc
```

 Performs the same as the following two commands combined (without the need to create the `list.txt` file):

```
prompt> ls -l > list.txt
prompt> wc list.txt
```

- The way the shell accomplishes this task is by creating a pipe that connects the standard output (fd = 1) of “`ls -l`” with the standard input (fd = 0) of “`wc`”.



The `pipe()` System Call

- System call `pipe()` creates a pipe and returns its two associated file descriptors in the array of integers passed to it:

  ```
  int pipe(int fds[]);
  ```

  Parameters :

  fds[0] will have the fd, file descriptor, for the read end of pipe.

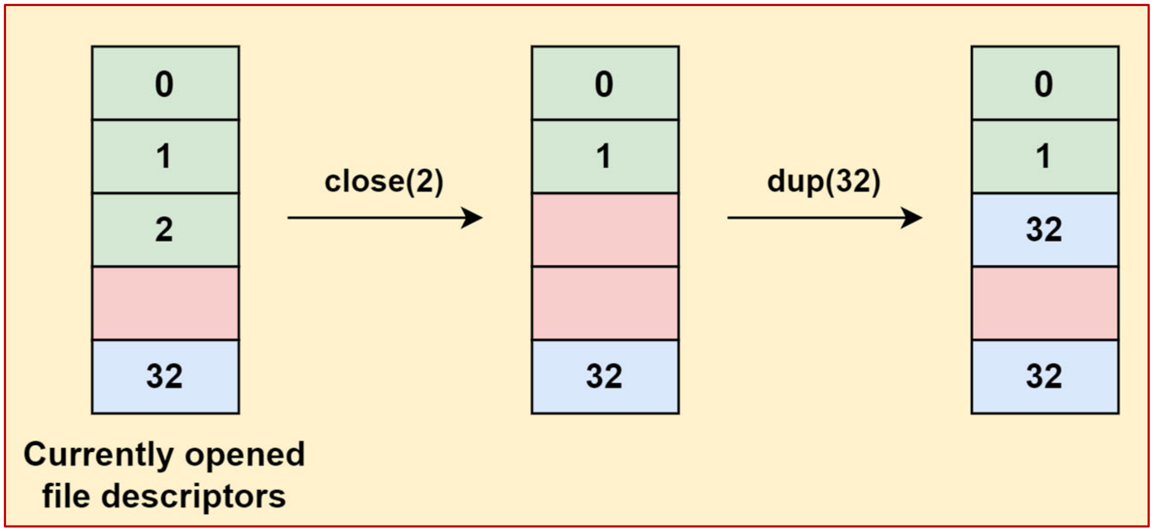
  fds[1] will have the fd for the write end of pipe.

  Returns : 0 on Success or -1 on error.

- After calling pipe(), the parent process needs to fork a child where the above file descriptors will be duplicated.
  - If the communication direction is from the parent to the child, then parent closes fds[0] and writes to fds[1], while the child closes fds[1] and reads from fds[0].
  - If the communication direction is from the child to the parent, then child closes fds[0] and writes to fds[1], while the parent closes fds[1] and reads from fds[0].

The dup( ) Function

- The `int dup(int fileDesciptor)` function, provided by the `unistd.h` library, is used in C programs to replace the lowest-level unused file descriptor with the provided file descriptor.
- The function parameter, `fileDesciptor`, represents the file descriptor that we want to copy.
- Upon successful execution, it returns the new file descriptor to access the file descriptor we just copied. If the function fails, it returns a -1 value that can be used for error handling.



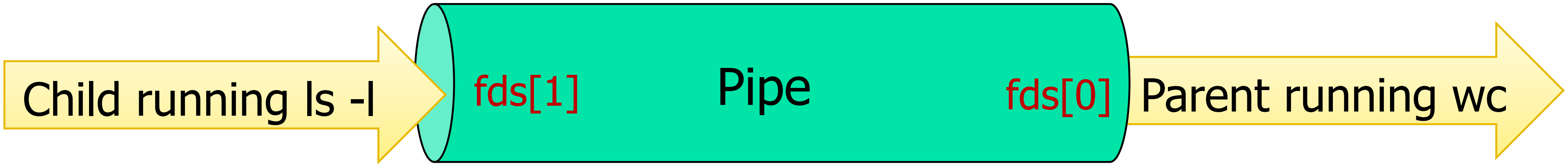
Pipe Example (1 of 3)

- The following code is the implementation of the pipeline command: `prompt> ls -l | wc`

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

int main(int argc, char **argv) {
 // Create pipe
 int fds[2];
 int err = pipe(fds);
 if (err == -1) { perror("pipe");    return 1; }

 // Spawn child
 int ret = fork();
 if (ret < 0)  { perror("fork");     return 1; }
```

Pipe Example (2 of 3)

```
else if (ret == 0) {  //Child implementing ls -l
  close(fds[0]);     // Close read end of pipe
  // Duplicate write end of pipe in standard output so that command ls writes to the pipe
 // instead of writing to the screen.
  close(STDOUT_FILENO);  //STDOUT_FILENO=1 (File number of stdout)
  dup(fds[1]);  //assign fds[1] to the first available file descriptor, which is fd 1
                //   that we have just closed.
  // Child launches command "ls -l"
      char *argv[3];
      argv[0] = "ls";
      argv[1] = "-l";
      argv[2] = NULL;
      execvp(argv[0], argv);

 }
```

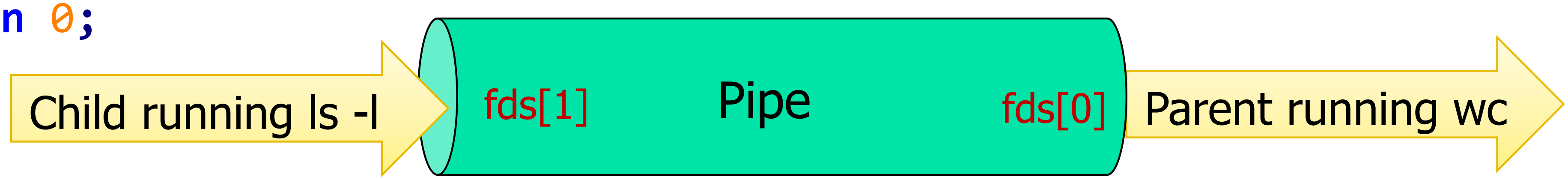


Pipe Example (3 of 3)

```
    else {  //Parent implementing wc
     // Close write end of pipe
        close(fds[1]);
     // Duplicate read end of pipe in standard input so that command wc reads from the pipe
     // instead of waiting to read from the keyboard (as wc reads from the standard input if
     // no FILE argument is given).
        close(STDIN_FILENO);  //STDIN_FILENO=0 (File number of stdin)
        dup(fds[0]);  //assign fds[0] to the first available file descriptor, which is fd 0
                      //    that we have just closed.
        // Parent launches command "wc“ with no file argument
          char *argv[2];
          argv[0] = "wc";
          argv[1] = NULL;
          execvp(argv[0], argv);
        }

  return 0;
}
```

Notes:

- wc waits until it reads from the input end of file character (^d).
- the code of the Child and Parent can be swapped and still the program implements the same pipeline command.



Challenges of Processes Execution

- How can the OS run processes efficiently while retaining control over the CPU?
  - Control over the CPU is needed to switch between processes as part of the time-sharing mechanism.
- What if the process wishes to perform restricted operations, such as issuing an I/O request to a disk?
  - If we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/O operations to the disk; if we did, a process could simply read or write the entire disk and thus all protections would be lost.
- The CPU resolves these challenges by providing two execution modes: **user mode** and **kernel mode**.

CPU Modes of Execution

- In **user mode**, processes can only run a subset of the CPU instructions set, which excludes I/O-related instructions, among others.
- In **kernel mode**, which the operating system kernel runs in, all instructions are available.
- The CPU allows a process in user mode to execute a **system call** instruction.
  - System calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.

Steps of Executing System Calls

1. Execute a special **trap** instruction providing a trap number and its needed arguments. Trap instructions are referred to differently:
   - RISC-V uses **`ecall`**, MIPS uses **`syscall`**, and x86 uses **`int`**.
2. Trap jumps into the kernel and raises privilege level to **kernel mode**.
3. Execute an OS code to perform the required work for the calling process as specified by the trap number and provided arguments.
4. When finished, a special **return-from-trap** instruction, returns into the calling user program while simultaneously reducing the privilege level back to **user mode**. (basically this step moves the user process from the Blocked state to the Ready state).

- The following example from a RISC-V assembly program to print an integer on the screen:

```
li    a7, 1   # Trap value 1 in register a7 to print an integer
li    a0, 5   # The integer argument (value 5) in a0
ecall
```

Trap Numbers

- For a system call, the user process cannot specify an address to jump to (as you would when making a regular function call, for example in RISC using `jal` to call and `jr ra` to return).
  - Doing so would allow user programs to jump anywhere into the kernel which is a **very bad idea**.
- Instead, the user program uses **trap numbers** to make a system call. A level of indirection that serves as a form of **protection**.
- It also allows the operating system to move the **trap handler** routines (the instructions needed to execute the tasks of the system calls) to different places in the memory without the need to modify user programs, which is impossible for existing programs.

Trap Table

- The kernel sets up a **trap table** at boot time. Each entry in the table contains a pair of a **trap number** and the **address** of the corresponding **trap handler**.

- Trap handlers include, not only system calls routines, but also the routines needed to handle other **exceptional events** such as a hard disk interrupt and a keyboard interrupt (i.e., hardware interrupts).

- Locating the trap table in the memory is done using a **privileged** instruction that can execute only in a **kernel mode**.

A Process LDE Protocol Scenario 1 (1 of 2)

- LDE is the **Limited Direct Execution** technique used by the OS to run programs as fast as possible directly on the CPU limited by some control from the OS

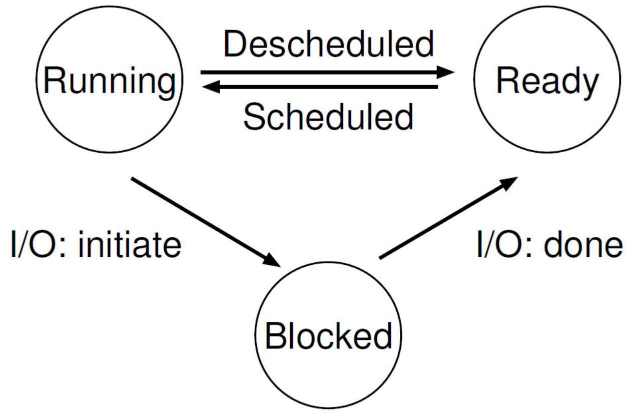
OS (Kernel Mode)	Hardware	Program (user mode)
OS booting: Initialize trap table		
	Remember trap handler address.	
OS running		
		Shell runs. Shell receives a process run request from the user.
Create entry for the process and allocate its memory. Load program into memory. Setup user stack with argv. Fill kernel stack with PC. Return-from-trap.		
	Restore regs (from kernel stack), move to user mode, and jump to user’s main.	

A Process LDE Protocol Scenario 1 (2 of 2)

OS (Kernel Mode)	Hardware	Program (user mode)
		Run main() … Call system call using trap number.
	Save regs (to kernel stack). Move to kernel mode. Jump to trap handler.	
Handle trap by executing its handler instructions. Return-from-trap.		
	Restore regs (from kernel stack), move to user mode, and jump to PC, which is after user trap instruction.	
		Return from main. Run the exit trap.
Free memory of process. Remove from process list.		

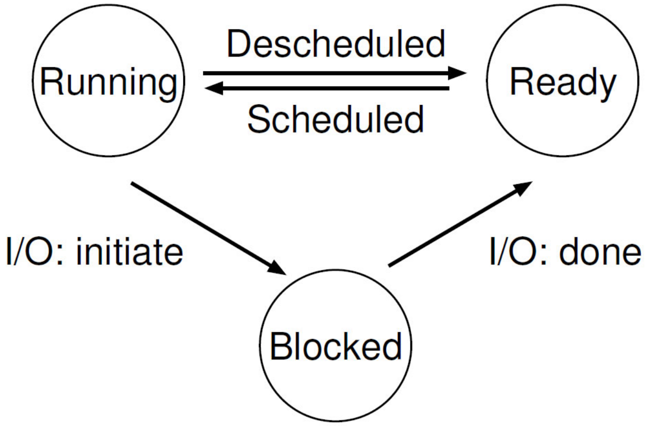
Switching Between Processes

- If a process is running on the CPU, this means the OS is not running. If the OS is not running, how can it do anything at all let alone switching the running process out?!
- Old approach: the OS waits until the running process enters a "trap" and switches it out.
  - Isn't this passive approach less than ideal? What happens, for example, if a process (whether malicious, or just full of bugs) ends up in an infinite loop, and never makes a system call?
- Without some additional help from the hardware, it turns out the OS cannot do much at all when a process refuses to make system calls (or mistakes).



Timer Interrupt

- A timer device can be programmed to raise an interrupt every so many milliseconds.
  - During the boot sequence, the OS must start the timer, which is of course a privileged task.
- When the timer interrupt is raised, the currently running process is halted and brought to the Ready state, and a pre-configured interrupt handler in the OS runs.
- At this point, the OS has regained control of the CPU, and thus can do what it pleases such as starting a different process or resuming the just interrupted process.
- When an interrupt occurs, the hardware is responsible for saving enough of the state of the current running process so that later the process can be resumed correctly.



Context Switch

- After the OS regains control of the CPU from a running process, it needs to decide whether to continue running the same process, or switch to a different one.

- This decision is made by a part of the operating system known as the **scheduler**.

- If the decision is made to switch, the OS then executes a low-level piece of code which we refer to as a **context switch**.

- During a context switch, the OS saves the registers values for the currently-executing process (onto kernel stack) and restore the registers values for the soon-to-be-executing process (from kernel stack).
  - By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.

A Process LDE Protocol Scenario 2 (1 of 2)

OS (Kernel Mode)	Hardware	Program (user mode)
OS booting: Initialize trap table Start interrupt timer and set it to ***n*** ms.		
	Remember address of trap handler.	
OS running		
		Process ***A*** starts running …
	Timer interrupt. Save regs(***A***)→k-stack(***A***). Move to kernel mode. Jump to trap handler (mainly to run the scheduler code).	

A Process LDE Protocol Scenario 2 (2 of 2)

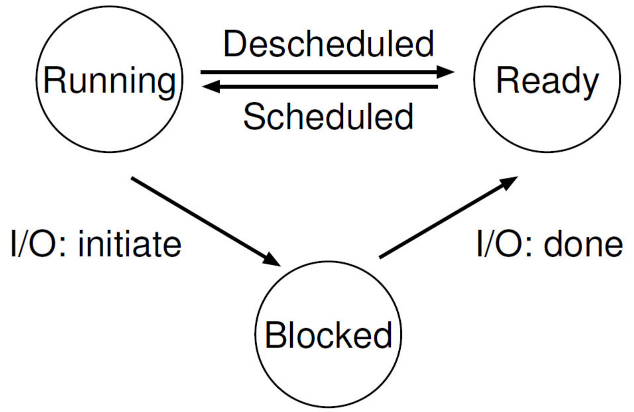
OS (Kernel Mode)	Hardware	Program (user mode)
Handle the trap. Call switch() routine. Save regs(***A***)→proc_t(***A***) Restore regs(***B***)←proc_t(***B***) Switch to k-stack(***B***) Return-from-trap (into ***B***)		
	Restore regs(***B***)←k-stack(***B***). Move to user mode. Jump to B’s PC.	
		Process ***B*** is running. …

Context Switching Overhead

- Context switching requires time to save and restore the registers used by the switched processes.
- Another overhead comes from maintaining the process state that includes the memory caches, virtual/physical memory mapping, and branch predictors.

OS Scheduling

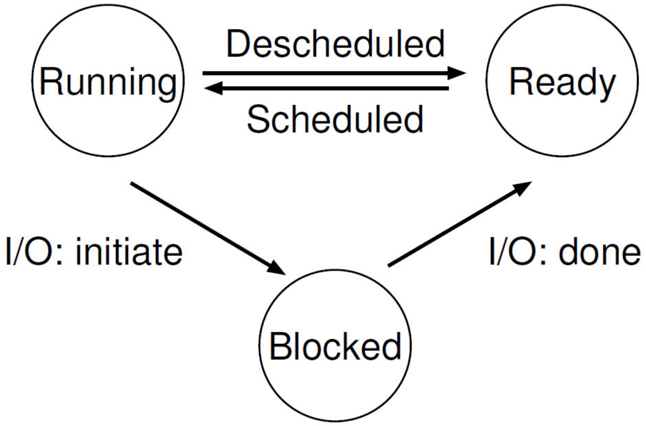
- The OS scheduler is one of the OS kernel programs that schedules which process moves to the "running" state from among all "ready" processes.
- The challenge is to make such scheduling process as "**efficient**" and "**fair**" as possible, subject to varying and often dynamic conditions.
- Efficiency includes minimizing:
  - Processes response time and turn around time,
  - Resources overhead such as context switching.
- Fairness includes avoiding starvation where a process 'never' receives the CPU running time it needs to complete.



Workload Assumptions

- The **workload** refers to the running processes in the system.
- The following are our initial, mostly unrealistic, assumptions about the workload processes, sometimes called **jobs**:
  1. Each job runs for the same amount of time.
  2. All jobs arrive at the same time.
  3. Once started, each job runs to completion.
  4. All jobs only use the CPU (i.e., they do not perform I/O operations).
  5. The run-time of each job is known in advance.

- A **scheduling policy** in the OS will make the decision of which job to move from the "Ready" state to the "Running" state and vice versa.



Scheduling Turnaround Time

- The **turnaround time** of a job is defined as the time at which the job completes minus the time at which the job arrived in the system: $T_{turnaround} = T_{completion} - T_{arrival}$
- It is used as a **performance metric** to compare different scheduling policies.
- Because we have assumed that all jobs arrive at the same time, for now assume $T_{arrival} = 0$ and hence

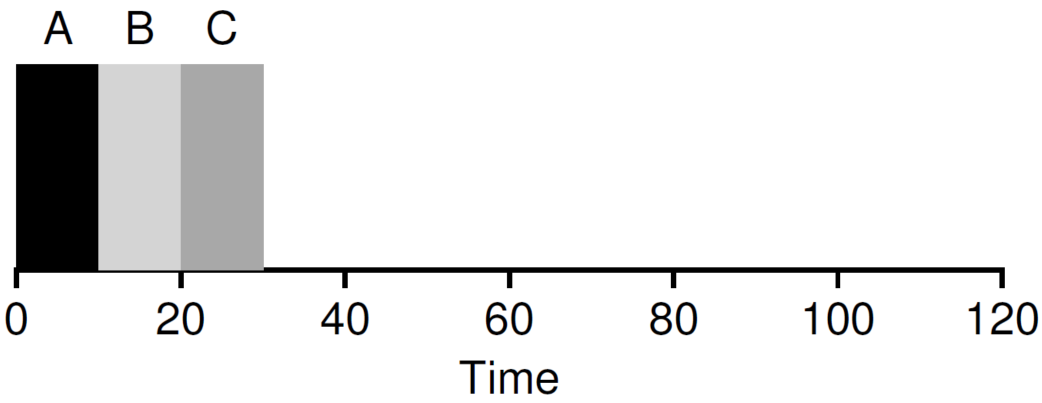
$$T_{turnaround} = T_{completion}$$

- This fact will change as we relax the previous assumptions

First In, First Out (FIFO)

- Sometimes FIFO is referred to as **First Come, First Served** (FCFS).
- *Example*: Assume three jobs arrive in the system, *A*, *B*, and *C*, at roughly the same time ($T_{arrival}$ = 0). Because FIFO has to put some job first, let’s assume that *A* arrived just a hair before *B* which arrived just a hair before *C*. Assume also that each job runs for 10 seconds. What will the **average turnaround time** be for these jobs?
- The average turnaround time for the three jobs is simply:

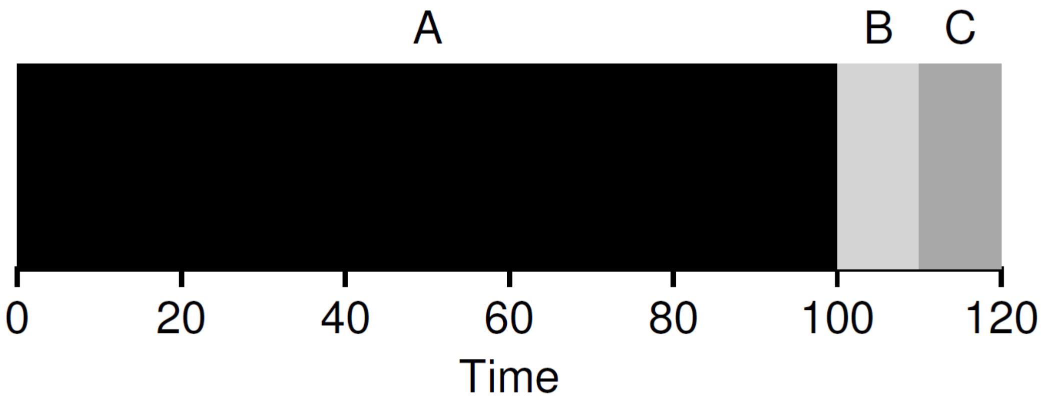
  (10 + 20 + 30)/3 = 20 seconds



FIFO and the Convoy Effect

- Let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time.
- *Example*: Repeat the previous FIFO example but this time *A* runs for 100 seconds while B and C run for 10 each
- The average turnaround time for the three jobs is now:

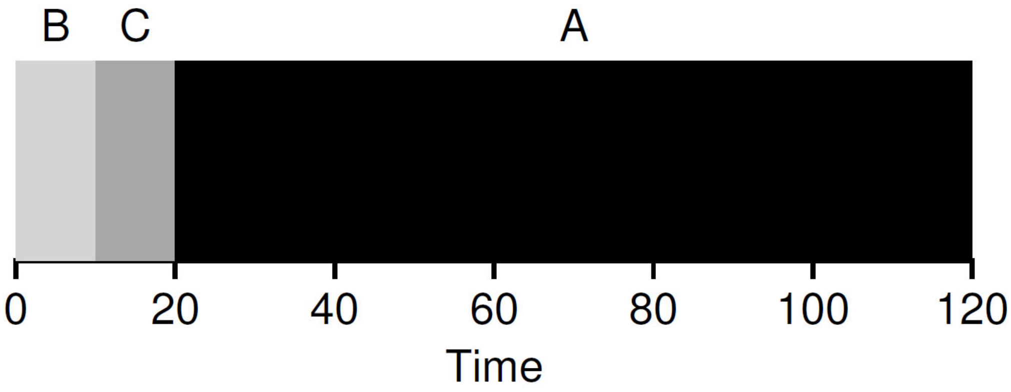
  (100 + 110 + 120)/3 = 110 seconds
- This problem is generally referred to as the **convoy effect**, where number of short jobs are queued behind a long job.



Shortest Job First (SJF)

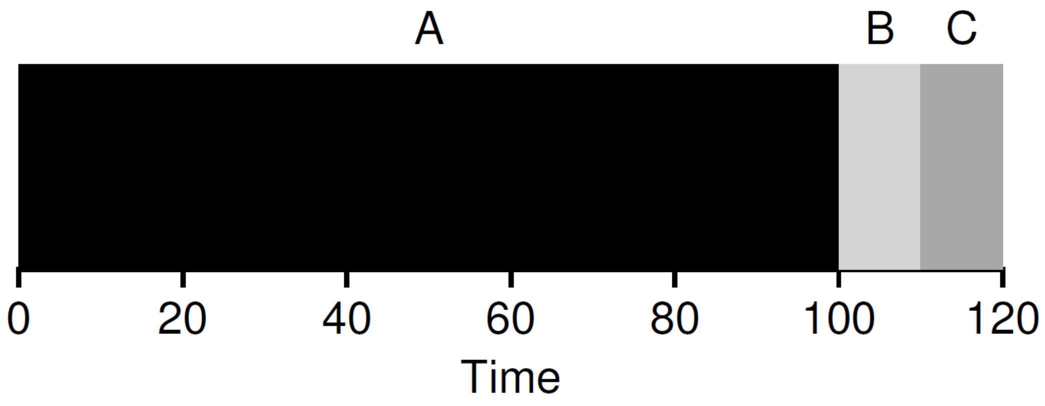
- **SJF** can solve the previous convoy effect problem.
- By applying SJF, the average turnaround time for the three jobs in the previous example is :

  (10 + 20 + 120)/3 = 50 seconds
- In fact, given our assumptions about jobs all arriving at the same time, SJF is indeed an **optimal** scheduling algorithm.



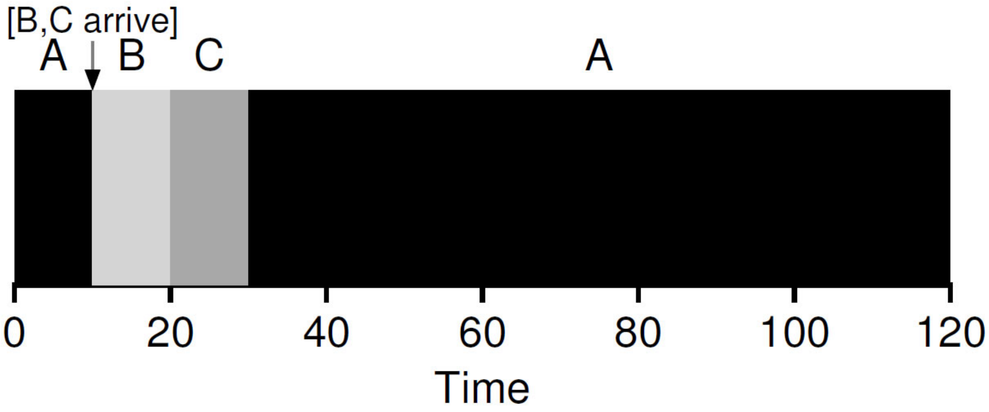
SJF With Different Arrival Times

- Let’s relax assumption 2, and now assume that jobs can arrive at any time instead of all at once.
- *Example*: This time, assume *A* arrives at t = 0 and needs to run for 100 seconds, whereas *B* and *C* arrive at t = 10 and each need to run for 10 seconds.
- The average turnaround time for the three jobs using SJF is now: [100 + (110-10) + (120-10)]/3 = 103.33 seconds
- The problem here is that SJF is a **non-preemptive** scheduler.



Shortest Time-to-Completion First (STCF)

- Let’s relax assumption 3 where jobs can be preempted before their completion.
- STCF scheduler adds **preemption** to SJF: Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.
- *Example*: Applying STCF on the last example will preempt *A* and run *B* and *C* to completion then resume *A*.
- The average turnaround time for the three jobs using STCF is: [(120-0) + (20-10) + (30-10)]/3 = 50 seconds



Scheduling Response Time

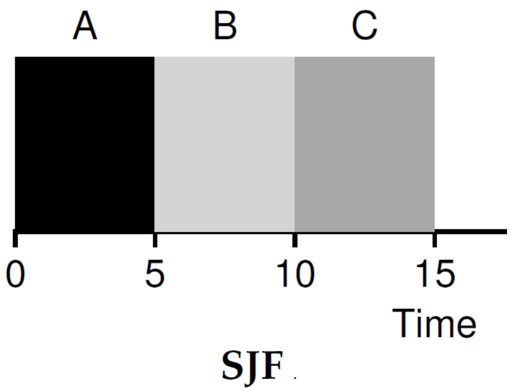
- Another scheduler performance metric is the **response time** of a job, which is defined as the time from when the job arrives in a system to the first time it is scheduled:

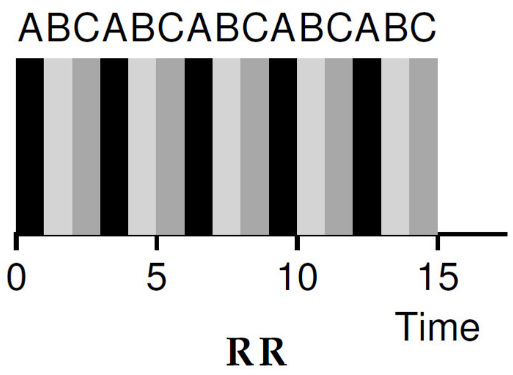
$$T_{response} = T_{firstrun} - T_{arrival}$$

- Response time is important in time-shared computers where users would sit at a terminal and demand interactive performance from the system.
- FIFO, SJF, and STCF schedulers are not particularly good for response time.

Round Robin (RR)

- RR runs a job for a **time slice** (sometimes called a scheduling **quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished.
- Note that the length of a time slice must be a multiple of the **timer-interrupt** period.
- *Example*: Assume three jobs *A*, *B*, and *C* arrive at the same time in the system, and that they each wish to run for 5 seconds.
- SJF average response time = (0 + 5 + 10)/3 = 5 seconds
- Average response time of RR with time-slice of 1 second = (0 +1 + 2)/3 = 1 second



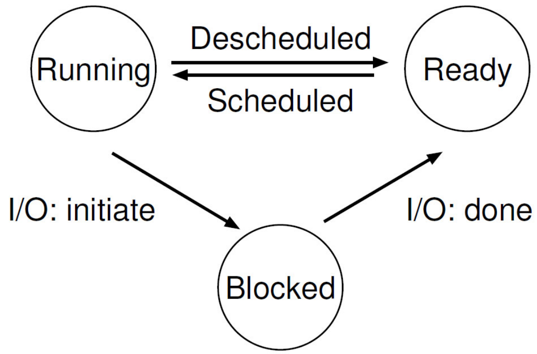


What are the average turnaround times?

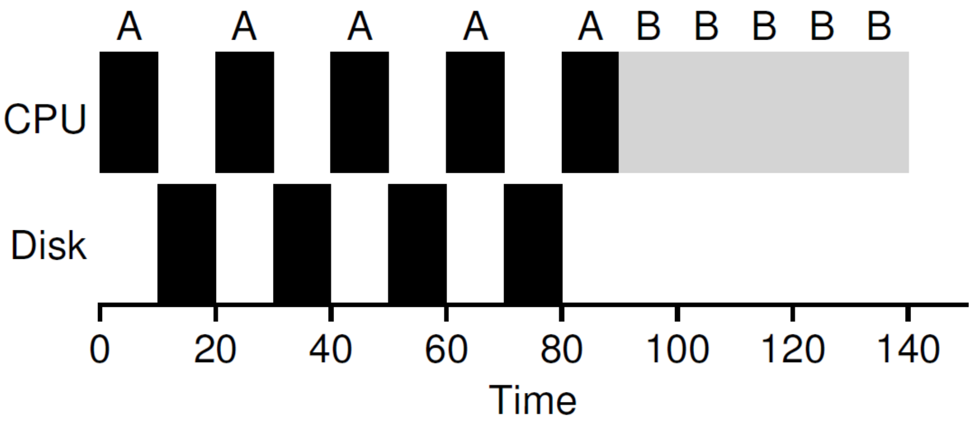
RR Tradeoff

- With RR scheduling, the shorter the time slice, the better the performance of RR under the response-time metric.
- However, making the time slice too short results in the **overhead of context switching** dominating overall performance.
- RR, with a reasonable time slice, is thus an **excellent** scheduler if response time is our main metric.
- RR is indeed one of the **worst** policies if turnaround time is our main metric.
- More generally, any policy (such as RR) that is fair, i.e., that evenly divides the CPU among active processes on a small-time scale, will perform poorly on metrics such as turnaround time.

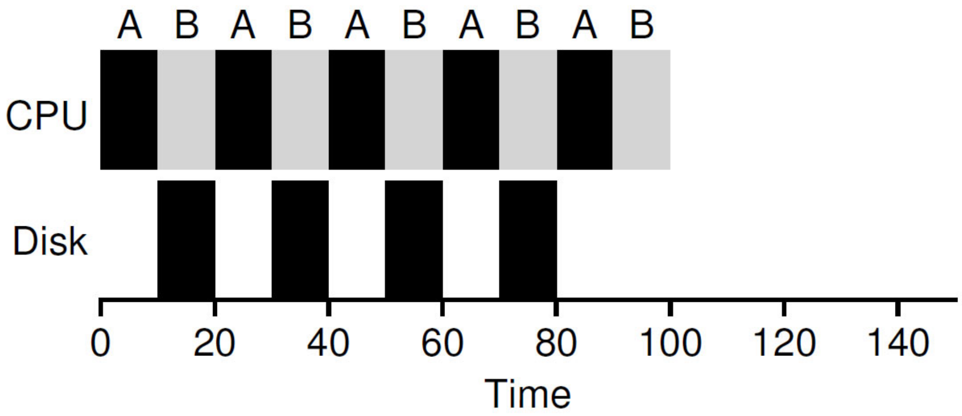
Incorporating I/O



- Let's relax assumption 4 where jobs can perform I/O operations.
- Example: assume we have two jobs, *A* and *B*, which each needs 50 ms of CPU time. However, *A* runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas *B* simply uses the CPU for 50 ms and performs no I/O.
- The following are the two options the scheduler has:



Poor Use Of Resources



Overlap Allows Better Use Of Resources

SJF, STCF, and RR Shortcomings

- Both SJF and STCF can optimize the *turnaround* time, however they need the knowledge of how long a job will run for, which the OS doesn’t generally know in advance.

- RR reduces *response* time but is terrible for *turnaround* time.

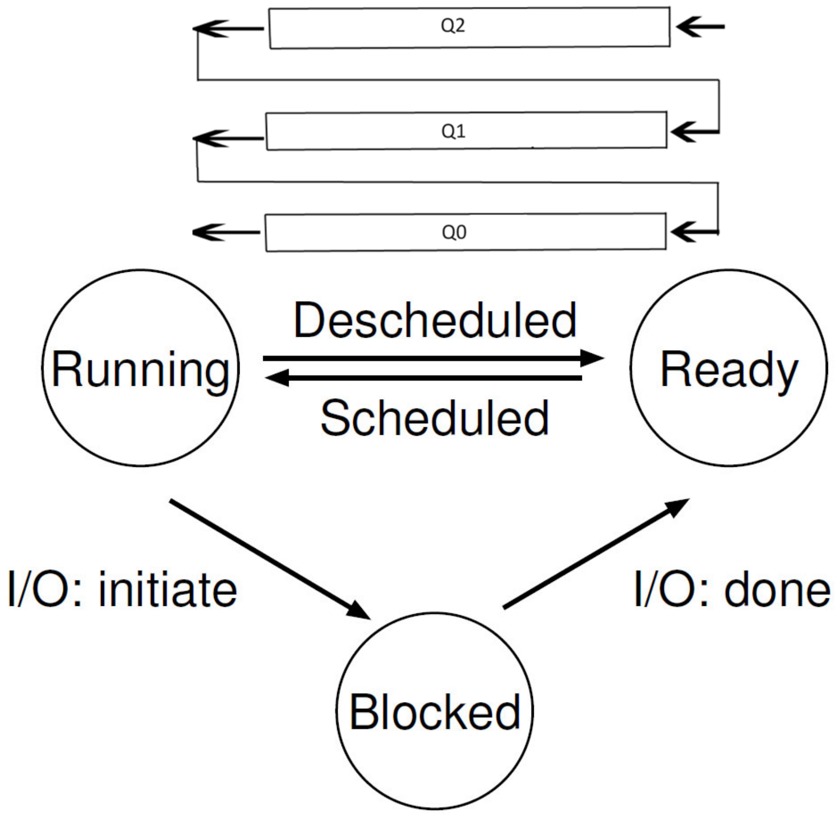
- How can we have a scheduler that learns, as the system runs, the characteristics of the jobs it is running, and thus makes better scheduling decisions?

The MLFQ Scheduler

- The *Multi-Level Feedback Queue* (**MLFQ**) scheduler has a number of **distinct** queues.
- Each “ready” job is assigned to one of these queues.
- Each queue has a different **priority** level.
- MLFQ uses **RR** to run the jobs on the highest priority queue.
- The first two basic rules for MLFQ:

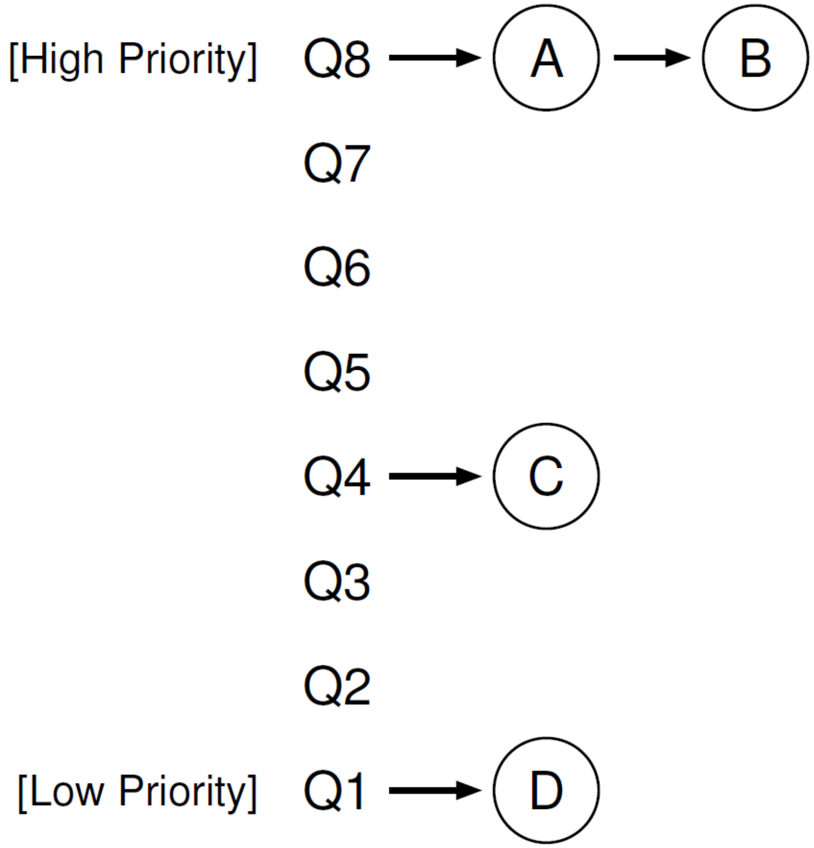
  Rule 1: If Priority(*A*) > Priority(*B*), *A* runs (*B* doesn’t).

  Rule 2: If Priority(*A*) = Priority(*B*), *A* & *B* run in RR.



MLFQ Snapshot

- In the shown snapshot of the a MLFQ and given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between *A* and *B* because they are the highest priority jobs in the system. Jobs *C* and *D* would never even get to run until *A* and *B* are done!
- How does job priority change over time?



Changing Priority in MLFQ

- Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its ***observed behavior***.
- MLFQ tries to learn about processes as they run, and thus use the *history* of the job to predict its *future* behavior.
  - Usually, the OS workload is a mix of **interactive** jobs that are short-running (and may frequently give up the CPU), and some longer-running "**CPU-bound**" jobs that need a lot of CPU time but where response time isn't important.
- More MLFQ rules:

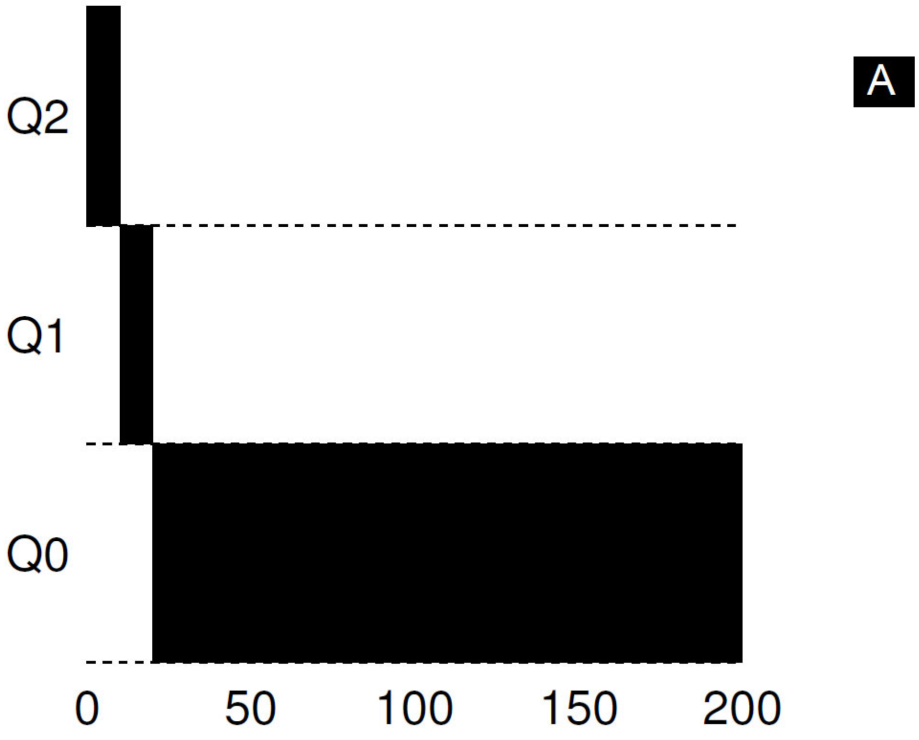
Rule 3: When a job enters the system, it is placed at the highest priority

Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).

Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.

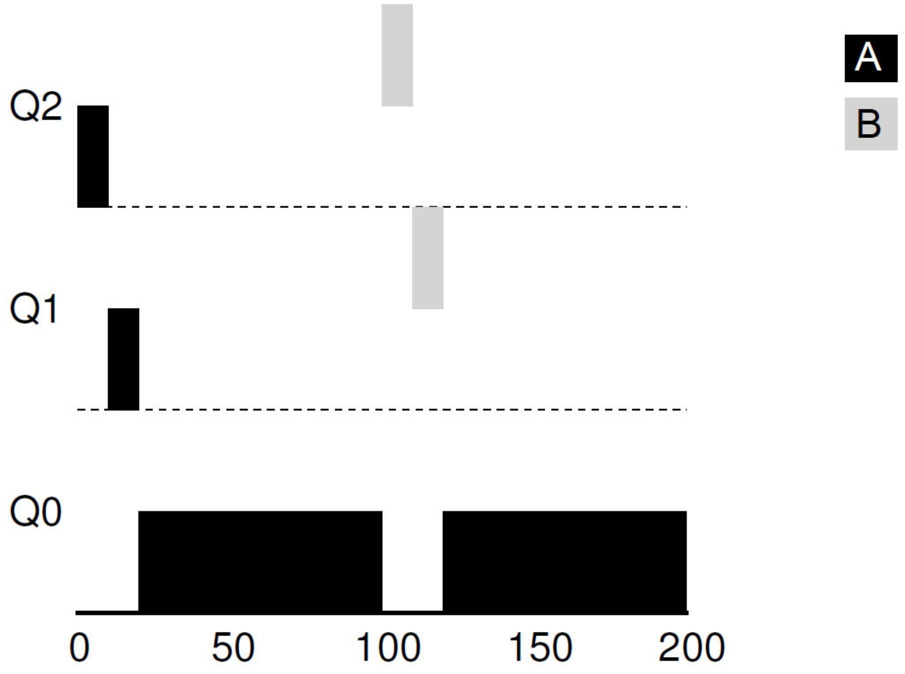
Example: MLFQ Single Batch Job

- Here we have a long running, *batch*, job *A* in a system with a three-queue scheduler.
  - The job enters at the highest priority (*Q*2).
  - After a single time-slice of 10ms, the scheduler reduces the job's priority by one, and thus the job is on *Q*1.
  - After running at *Q*1 for a time slice, the job is finally lowered to the lowest priority in the system (*Q*0), where it remains until it is done.



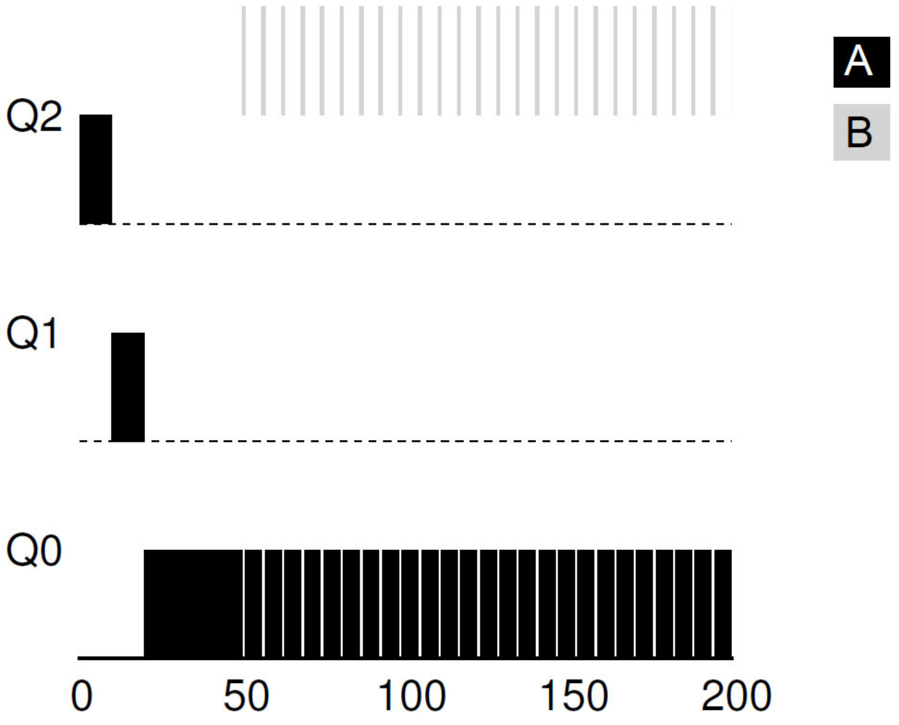
Example: Both Batch and Interactive Jobs

- The two jobs are *A*, which is a long-running CPU-intensive job, and *B*, which is a 20ms short-running interactive job. Assume *A* has been running for some time, and then *B* arrives at time 100ms.
  - *B* completes before reaching the bottom queue, in two time slices; then *A* resumes running (at low priority).
  - Because MLFQ doesn’t know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues and handled as a long-running job.



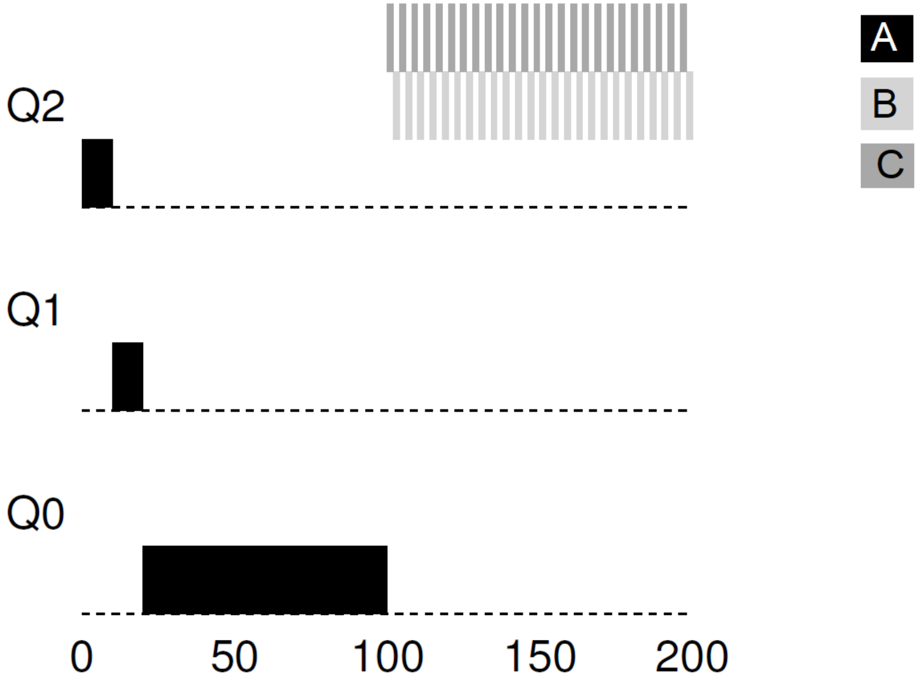
Example: An Interactive Job with I/O

- If job *B* is doing a lot of I/O, it will give up the CPU before its time slice is complete; in such case, we don’t wish to penalize the job and thus simply keep it at the same level (**Rule 4b**).
  - The MLFQ approach keeps *B* at the highest priority because *B* keeps releasing the CPU.
  - MLFQ further achieves its goal of running interactive jobs quickly



Problems: Starvation and Gaming

- If there are "too many" interactive jobs in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time results in the problem of **starvation**.
- **Gaming the scheduler** generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource. A smart programmer could issue an I/O operation right before the time slice is over and this way it remains in the same queue.

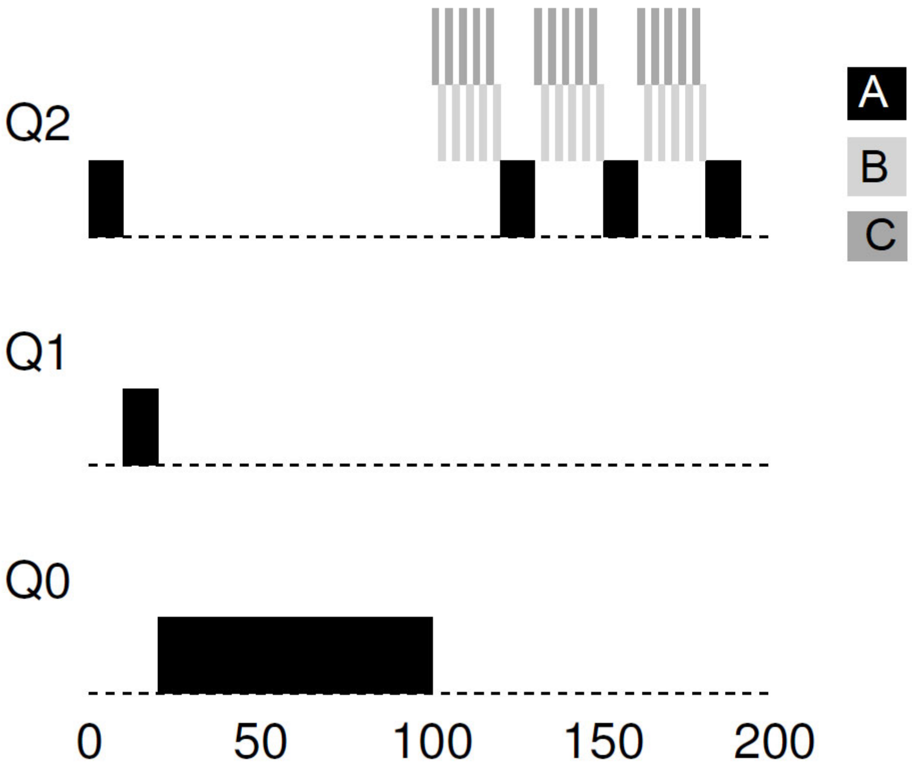


The Priority Boost Rule

- To avoid the starvation problem, MFLQ periodically can boost the priority of all the jobs in system.

Rule 5: After some time period *S*, move all the jobs in the system to the topmost queue

- This rule solves another problem when a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost .



The Better Accounting Rule

- To avoid the gaming problem, MFLQ can perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much of a time slice a process used at a given level, the scheduler should keep track; once a process has used its **allotment**, it is demoted to the next priority queue.
- Current rules 4a and 4b:

  Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).

  Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
- Rewrite these rules, 4a and 4b, to the following single rule:

  Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Tuning MLFQ

- The big question about MLFQ is how to **parameterize** such a scheduler.
  - Number of queues.
  - Time slice (same level allotment time) length
  - The boosting time period *S*.
- Only some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.
- Most MLFQ variants allow for **varying time-slice** length across different queues. The high-priority queues are usually given short time slices. The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well.

Readings and Resources List

- From Arpaci-Dusseau textbook:
  - Section 4.5 Data Structures
  - Section 5.4 Why? Motivating The API
  - Chapter 9 Scheduling: Proportional Share
  - Chapter 10 Multiprocessor Scheduling