



Advanced Operating Systems: Three Easy Pieces

1. Virtualization:

1.1 The CPU

Outline

- **Process and thread:**
 - Abstraction
 - Interlude (intervening)
- **Ensuring Processes cannot harm each other** – System Calls
- **Scheduling:** separation of policy and mechanism
- **Multiprocessor** scheduling
- **Multi-level Feedback Queue:** MLFQ

- 
- **The abstraction**
 - **Interlude**

Process and Threads



The Abstraction: The Process

How to provide the illusion of many CPUs?

■ CPU virtualizing

- ❑ The OS can promote the illusion that many virtual CPUs exist.
- ❑ **Time sharing**: Running one process, then stopping it and running another
 - The potential cost (context switch) is **performance**.

A Process

A process is a **running program / program in execution.**

■ A process Comprises of:

❑ Memory (address space)

- Instructions
- Data section

❑ Registers

- Program counter
- Stack pointer

❑ Data structures to help the OS manage the process

Process API

- **These APIs are available on any modern OS:**
 - ❑ **Create**
 - Create a new process to run a program
 - ❑ **Destroy**
 - Halt a runaway process
 - ❑ **Wait**
 - Wait for a process to stop running
 - ❑ **Miscellaneous Control**
 - Some kind of method to suspend a process and then resume it
 - ❑ **Status**
 - Get some status info about a process

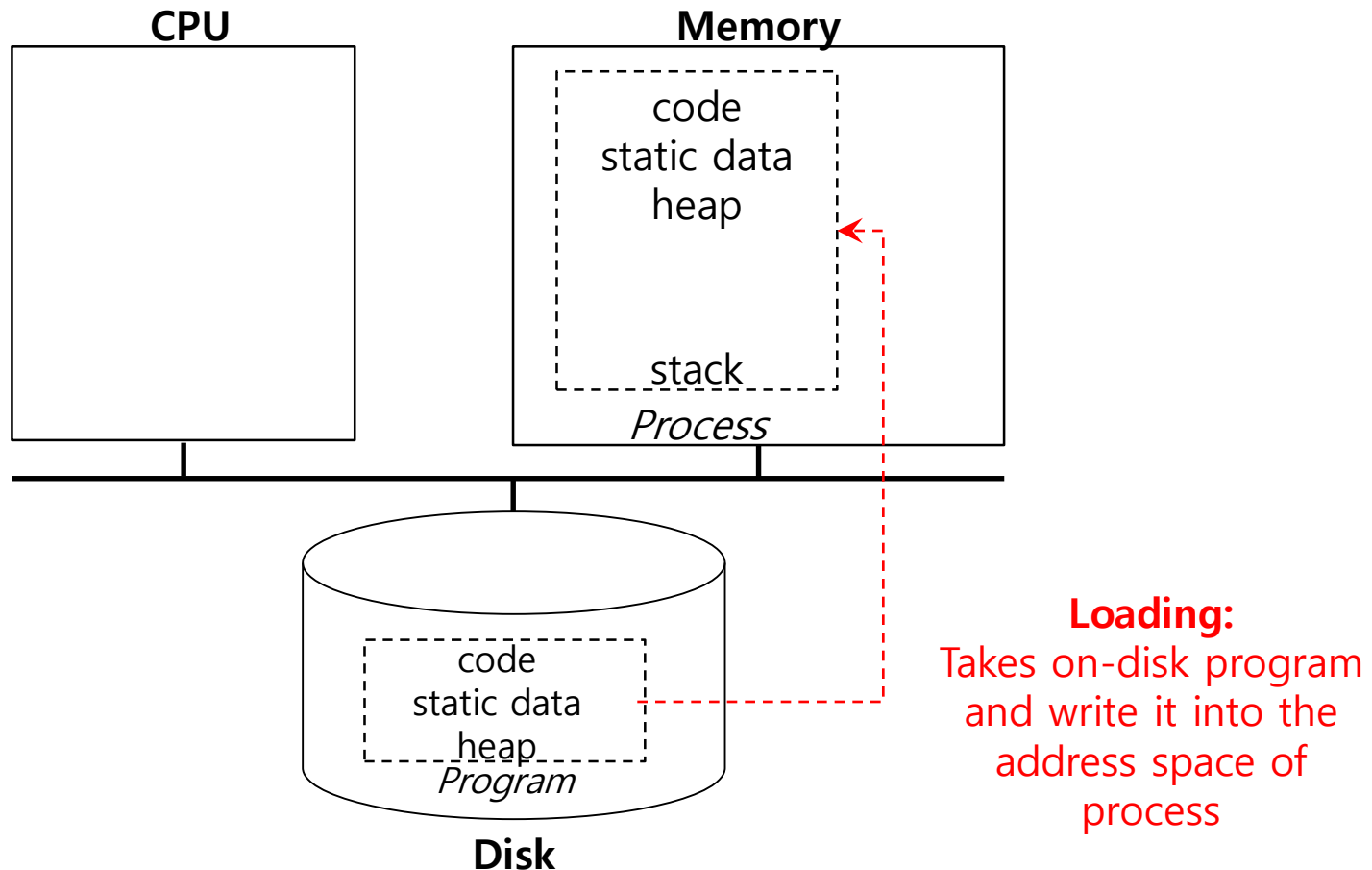
Process Creation

1. **Load** a program code into memory, into the address space of the process:
 - ❑ Programs initially reside on disk in *executable format*.
 - ❑ OS performs the loading process *lazily*.
 - Loading pieces of code or data (dynamic loader) only as they are needed during program execution.
2. The program's run-time **stack** is allocated.
 - ❑ Use the stack for *local variables*, *function parameters*, and *return address*.
 - ❑ Initialize the stack with arguments → as an example, `argc` and the `argv` array of `main()` function

Process Creation (Cont.)

3. The program's **heap** is created.
 - ❑ Used for explicitly requested dynamically allocated memory.
 - ❑ Program request such space by calling malloc() and free it by calling free().
4. The OS do some other initialization tasks:
 - ❑ input/output (I/O) setup
 - Each process by default has three open file descriptors, i.e., Standard input, output and error
5. **Start the program** running at the entry point, namely main().
 - ❑ The OS *transfers control* of the CPU to the newly-created process.

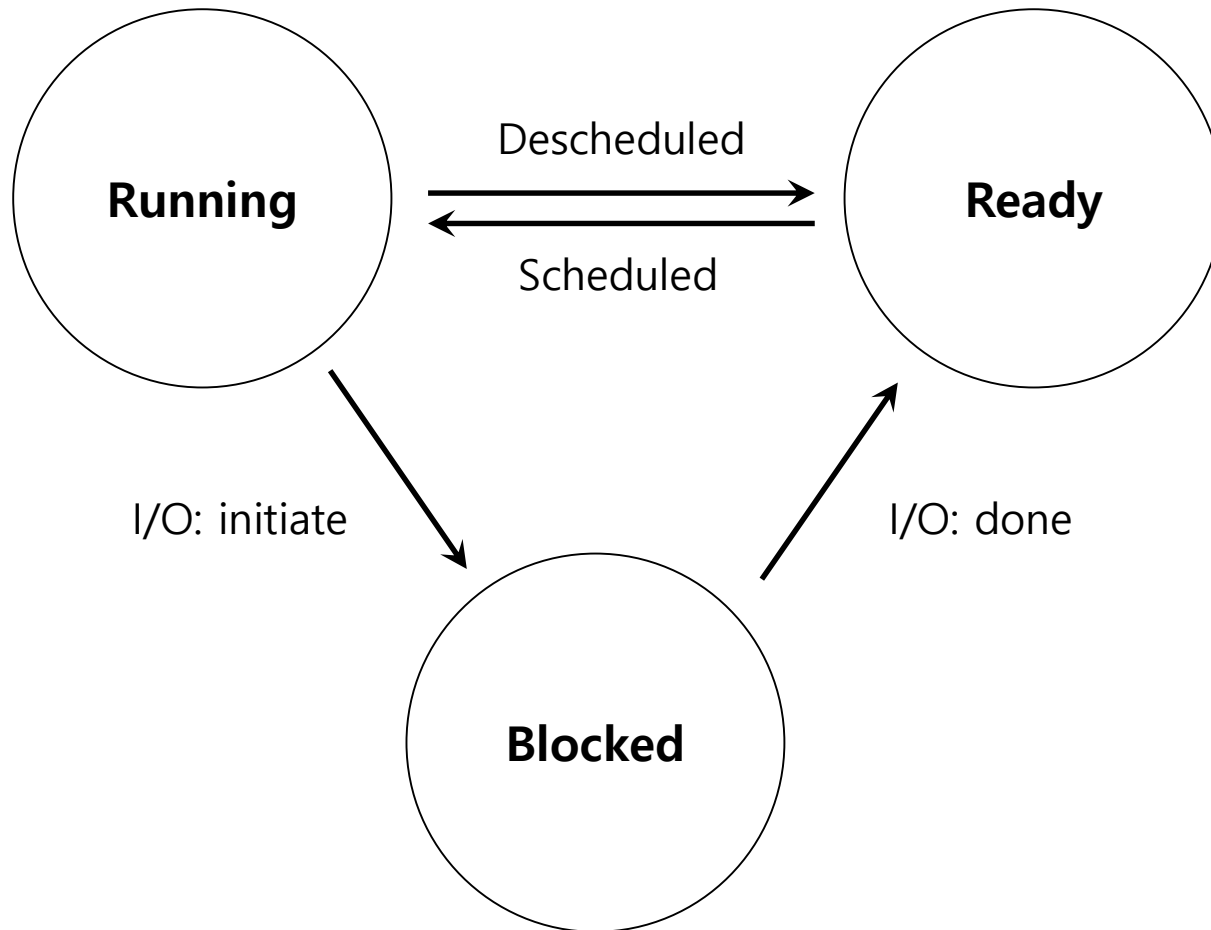
Loading: From Program To Process



Process States

- **A process can be one of three states:**
 - **Running**
 - A process is running on a processor.
 - **Ready**
 - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
 - **Blocked**
 - A process has performed some kind of I/O operation.
 - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition



Data structures

- The OS has **some key data structures** that track various relevant pieces of information.
 - **Process list/queue**
 - Ready processes queue
 - Blocked processes queue
 - Current running process
 - **Register context**
- **PCB(Process Control Block)**
 - A C-structure that contains information **about each process** such as pid, ppid, p_signal, address of u_area, etc.

Example) The xv6 kernel Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

Example) The xv6 kernel Proc Structure (Cont.)

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char    *mem;                // Start of process memory
    uint    sz;                  // Size of process memory
    char    *kstack;             // Bottom of kernel stack
                                   // for this process
    enum    proc_state state;    // Process state
    int     pid;                 // Process ID
    struct  proc *parent;        // Parent process
    void    *chan;               // If non-zero, sleeping on chan
    int     killed;              // If non-zero, have been killed
    struct  file *ofile[NOFILE]; // Open files
    struct  inode *cwd;          // Current directory
    struct  context context;     // Switch here to run process
    struct  trapframe *tf;       // Trap frame for the
    ...                          // current interrupt
};
```



Interlude: Process API

The fork() System Call

■ Create a new process

- The newly-created process has its own copy of the **address space, registers, and PC.**

```
#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 (main)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
    return 0;
}
```

Calling fork() example (Cont.)

Result (Not deterministic)

No guarantee if it is the parent or the child will resume execution first?

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The wait() System Call

- This system call won't return until the child has run and exited.

```
#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());
    } else { // parent goes down this path (main)
        int wc = wait(NULL); // wc = child pid that terminated
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
```

My child pid Pid of the terminated child My (parent) pid

The wait() System Call (Cont.)

Result (Deterministic)

Parent will wait for the child to exit before it resumes execution...

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The exec() System Call

- Run a program that is different from the calling program:

```
#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");                // program: "wc" (word count)
        myargs[1] = strdup("p3.c");              // argument: file to count content
        myargs[2] = NULL;                        // marks end of array
        ...
    }
```

The exec() System Call (Cont.)

(Cont.)

```
...
    execvp(myargs[0], myargs); // runs word count ≡ wc filename
    printf("this shouldn't print out");
} else {
    int wc = wait(NULL);
    // parent goes down this path (main)
    // wc = child pid that exited
    // rc = pid of the child
    // In this cscenario: wc == rc
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
}
return 0;
}
```

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
<#lines    #words    #characters  file name>
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

All of the above with redirection

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.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);
        ...
    }
```

All of the above with redirection (Cont.)

(Cont.)

```
...
// now exec "wc"...
char *myargs[3];
myargs[0] = strdup("wc");           // program: "wc" (word count)
myargs[1] = strdup("p4.c");         // argument: file to count
myargs[2] = NULL;                   // marks end of array
execvp(myargs[0], myargs);          // runs word count
} else {                             // parent goes down this path (main)
    int wc = wait(NULL);
}
return 0;
}
```

Result

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
<#lines    #words    #characters  file name>
prompt>
```


Operating System Roles?

■ What is a resource and its abstraction?

1. **CPU:** process and/or thread
2. **Memory:** address space
3. **Device/Disk:** files

Physical Resource	Abstraction
CPU	Process / Thread
Memory	Address Space
Disk	Files



Ensuring Processes cannot harm each other – System Calls



Mechanism: Limited Direct Execution



How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing.
- **Issues:**
 - ❑ **Performance:** How can we implement virtualization without adding excessive overhead to the system?
 - ❑ **Control:** How can we run processes efficiently while retaining control over the CPU?

Direct Execution

- Just run the program directly on the CPU:

OS	Program
<ol style="list-style-type: none">1. Create entry for process list2. Allocate memory for program3. Load program into memory4. Set up stack with <code>argc / argv</code>5. Clear registers6. Execute call <code>main()</code> <ol style="list-style-type: none">9. Free memory of process10. Remove from process list	<ol style="list-style-type: none">7. Run <code>main()</code>8. Execute <code>return</code> from <code>main()</code>

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as ...
 - ❑ Issuing an I/O request to a disk
 - ❑ Gaining access to more system resources such as CPU or memory
- **Solution:** Using protected control transfer
 - ❑ **User mode:** Applications do not have full access to hardware resources.
 - ❑ **Kernel mode:** The OS has access to the full resources of the machine

System Call

- Allow the kernel to **carefully expose** certain key pieces of functionality (i.e., provide system-level service) to user program, such as:
 - ❑ Accessing the file system
 - ❑ Creating and destroying processes
 - ❑ Communicating with other processes
 - ❑ Allocating more memory

System Call (Cont.)

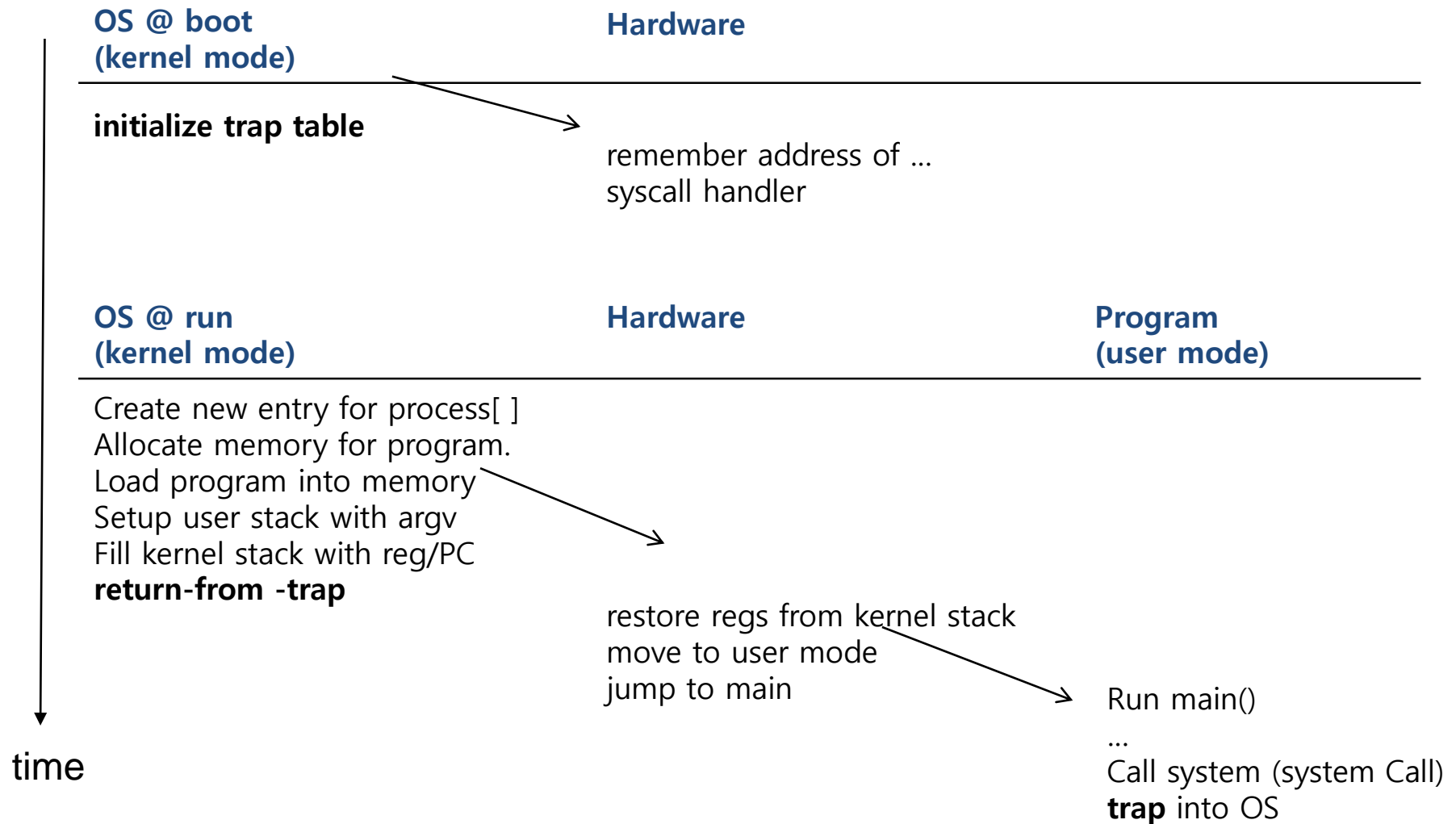
■ **Trap** instruction

- ❑ Jump into the kernel
- ❑ Raise the privilege level to kernel mode

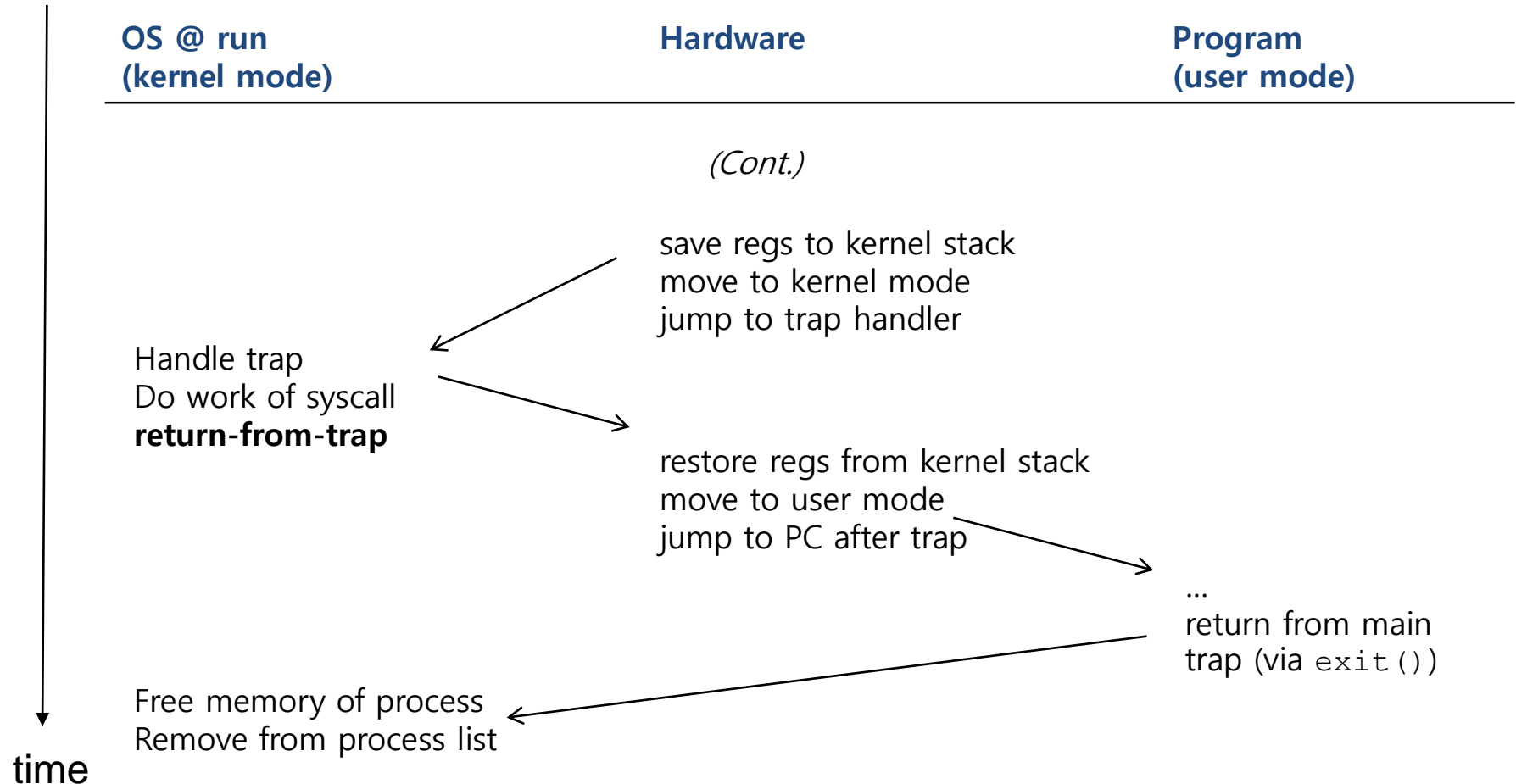
■ **Return-from-trap** instruction

- ❑ Return into the calling user program
- ❑ Reduce the privilege level back to user mode

Limited Direction Execution Protocol: Execute User program



Limited Direction Execution Protocol: Execute System Call



Problem 2: Switching Between Processes

- How can the OS **regain control** of the CPU so that it can switch between *processes*?
 - ❑ **A cooperative Approach:** Wait for system calls
 - ❑ **A Non-Cooperative Approach:** The OS takes control

A cooperative Approach: Wait for system calls

- Processes **periodically give up the CPU** by making **system calls** such as yield:
 - The OS decides to run some other task.
 - Application also transfer control to the OS when they do something illegal.
 - Divide by zero
 - Try to access memory that it shouldn't be able to access
 - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

**A process gets stuck in an infinite loop – not giving up the CPU
→ Reboot the machine!**

A Non-Cooperative Approach: OS Takes Control

■ A timer interrupt

- ❑ During the boot sequence, the OS start the timer.
- ❑ The timer raise an interrupt every so many milliseconds
- ❑ **When the interrupt is raised:**
 - The currently running process is halted.
 - Save enough of the state of the program
 - A pre-configured interrupt handler in the OS runs.

A **timer interrupt** gives OS the ability to run again and schedule the CPU.

Saving and Restoring Context

- Scheduler makes a decision (on return from the clock interrupt):
 - ❑ Whether to continue/resume running the **current process**, or switch to a **different one** (how)?
 - ❑ If the decision is made to switch, the OS executes context switch.

Context Switch

- **A low-level piece of assembly code:**
 - ❑ **Save few register values** for the current process onto its kernel stack
 - General purpose registers
 - PC
 - kernel stack pointer
 - ❑ **Restore few register values** for the soon-to-be-executing process from its kernel stack
 - ❑ **Switch to the kernel stack** for the soon-to-be-executing process

Limited Direction Execution Protocol (Timer interrupt)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A is running ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Limited Direction Execution Protocol (Timer interrupt)

OS @ run
(kernel mode)

Hardware

Program
(user mode)

(Cont.)

Handle the trap
Call switch() routine
 save regs(A) to proc-struct(A)
 restore regs(B) from proc-struct(B)
 switch to k-stack(B)
return-from-trap (into B)

restore regs(B) from k-stack(B)
move to user mode
jump to B's PC

Process B is now running

...

The xv6 Context Switch Code

```
1 # void swtch(struct context **old, struct context *new);
2 #
3 # Save current register context in old
4 # and then load register context from new.
5 .globl swtch
6 swtch:
7     # Save old registers (Save)
8     movl 4(%esp), %eax           # put old ptr into eax
9     popl 0(%eax)                # save the old IP
10    movl %esp, 4(%eax)          # and stack
11    movl %ebx, 8(%eax)          # and other registers
12    movl %ecx, 12(%eax)
13    movl %edx, 16(%eax)
14    movl %esi, 20(%eax)
15    movl %edi, 24(%eax)
16    movl %ebp, 28(%eax)
17
18    # Load new registers (restore)
19    movl 4(%esp), %eax           # put new ptr into eax
20    movl 28(%eax), %ebp         # restore other registers
21    movl 24(%eax), %edi
22    movl 20(%eax), %esi
23    movl 16(%eax), %edx
24    movl 12(%eax), %ecx
25    movl 8(%eax), %ebx
26    movl 4(%eax), %esp         # stack is switched here
27    pushl 0(%eax)              # return addr put in place
28    ret                         # finally return into new ctxt
```

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- **OS handles these situations:**
 - **Disable interrupts** during interrupt processing
 - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.



Scheduling: separation of Policy and Mechanism



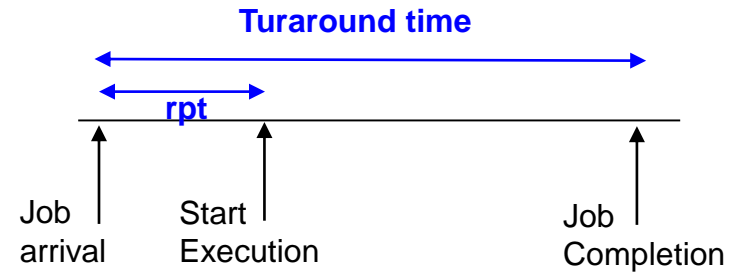
Scheduling: Introduction

Scheduling: Introduction

■ Workload assumptions (for simplicity):

1. Each job runs for the **same amount of time (time slice)**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

Scheduling Metrics



■ Performance metric: Response time

- The time at which **the job start execution** minus the time at which **the job arrived** in the system.

$$T_{\text{response time}} = T_{\text{first run time}} - T_{\text{arrival time}}$$

■ Performance metric: Turnaround time

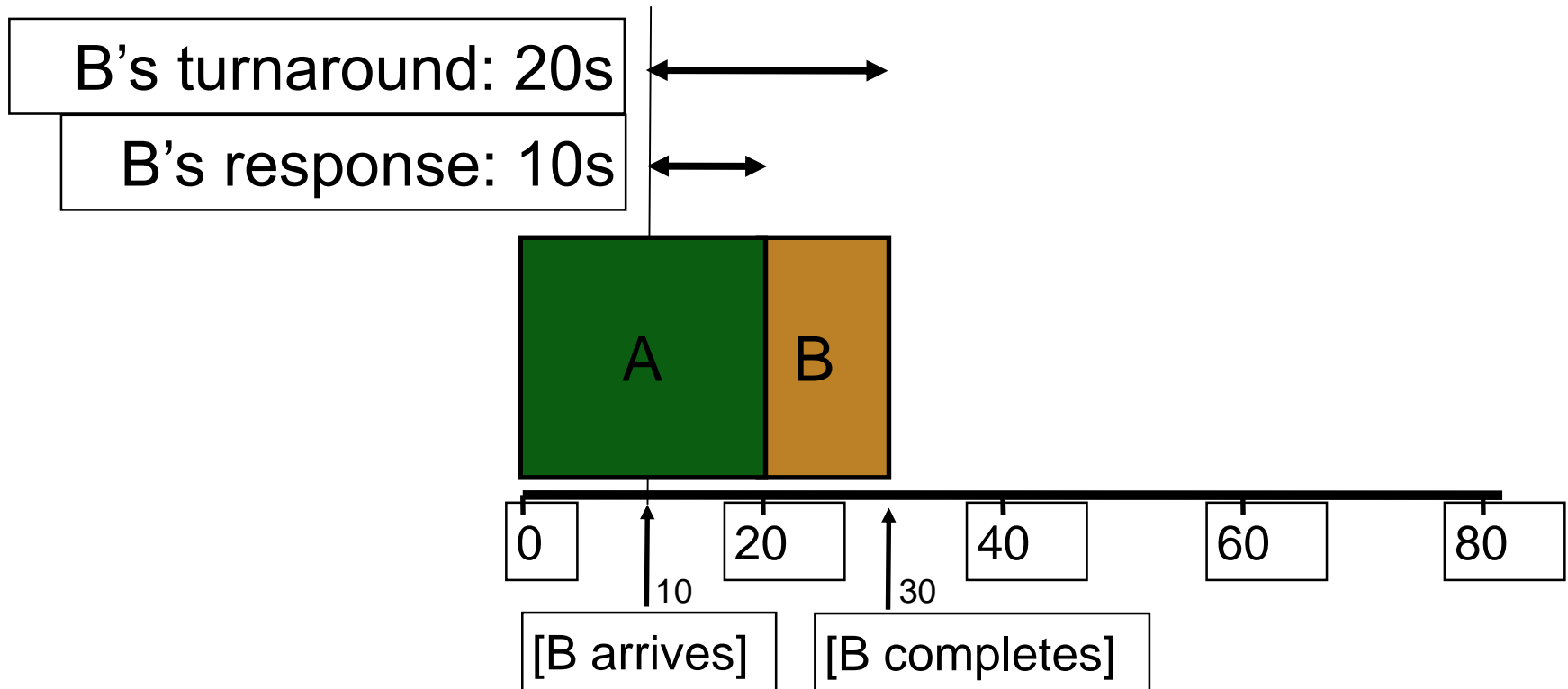
- The time at which **the job completes** minus the time at which **the job arrived** in the system.

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

■ Waiting time metric: Waiting time

$$T_{\text{waiting}} = T_{\text{turnaround}} - T_{\text{service-time}}$$

Scheduling Metrics



■ Another metric is fairness:

- ❑ Performance and fairness are often at odds in scheduling.

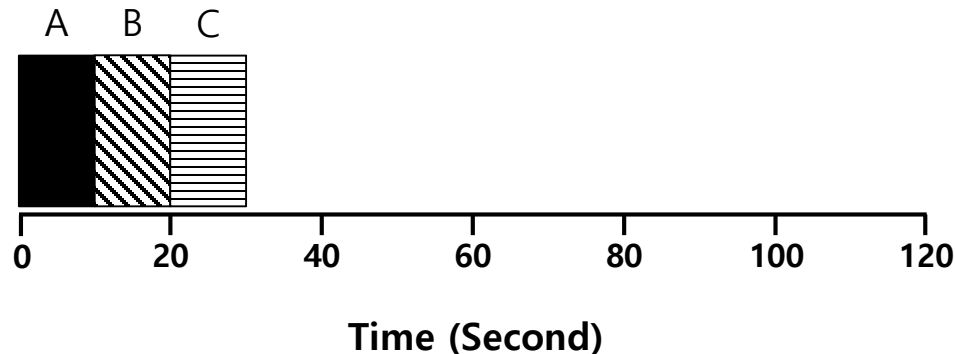
First In, First Out (FIFO)

■ First Come, First Served (FCFS)

- Very simple and easy to implement

■ Example:

- A arrived just before B which arrived just before C.
- Each job runs for 10 seconds.

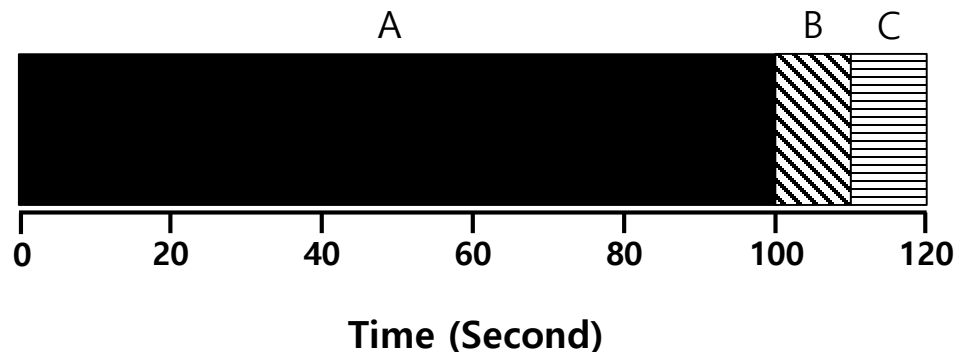


Avg TAT = 20 sec , while service time is 10 sec for A, B and C

$$\text{Average turnaround time} = \frac{10 + 20 + 30}{3} = 20 \text{ sec}$$

Why FIFO is not that great? – Convoy effect

- **Let's relax assumption 1:** Each job **no longer** runs for the same amount of time.
- **Example:**
 - ❑ A arrived just before B which arrived just before C.
 - ❑ A runs for 100 seconds, B and C run for 10 each.

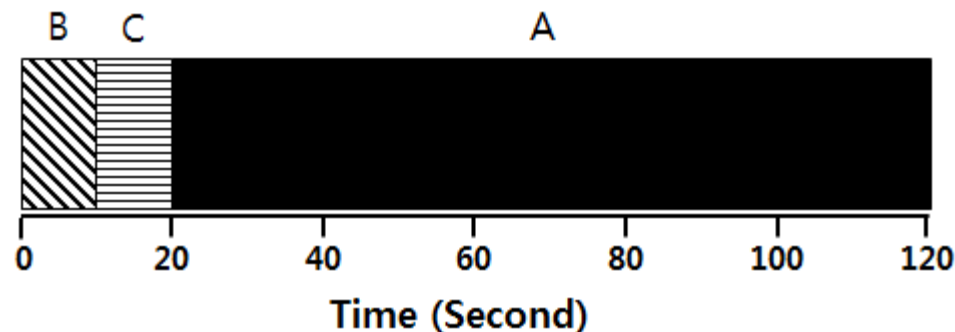


Avg TAT = 110 sec , while service time is 10 sec for B and C

$$\text{Average turnaround time} = \frac{100 + 110 + 120}{3} = 110 \text{ sec}$$

Shortest Job First (SJF)

- Run the shortest job first, then the next shortest, and so on:
 - Non-preemptive scheduler
- **Example:**
 - A arrived just before B which arrived just before C.
 - A runs for 100 seconds, B and C run for 10 each.



Avg TAT = 50 sec, while service time is 10 sec for B and C and 100 sec for A

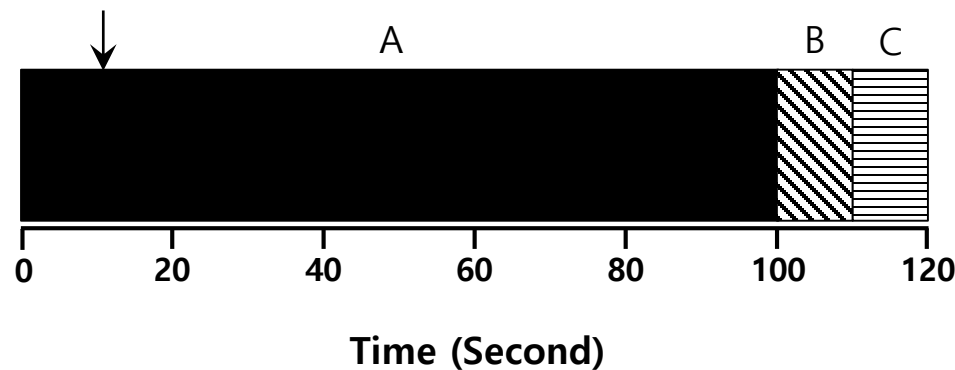
$$\text{Average turnaround time} = \frac{10 + 20 + 120}{3} = 50 \text{ sec}$$

SJF with Late Arrivals from B and C

- **Let's relax assumption 2:** Jobs can arrive at any time.

- **Example:**

- A arrives at $t=0$ and needs to run for 100 seconds.
- B and C arrive at $t=10$ and each need to run for 10 seconds.



Avg TAT = 103 sec, while service time is 10 sec for B and C

$$\text{Average turnaround time} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33 \text{ sec}$$

Shortest Time-to-Completion First (STCF)

- **Add preemption to SJF:**

- ☐ Also known as **Preemptive Shortest Job First (PSJF)**

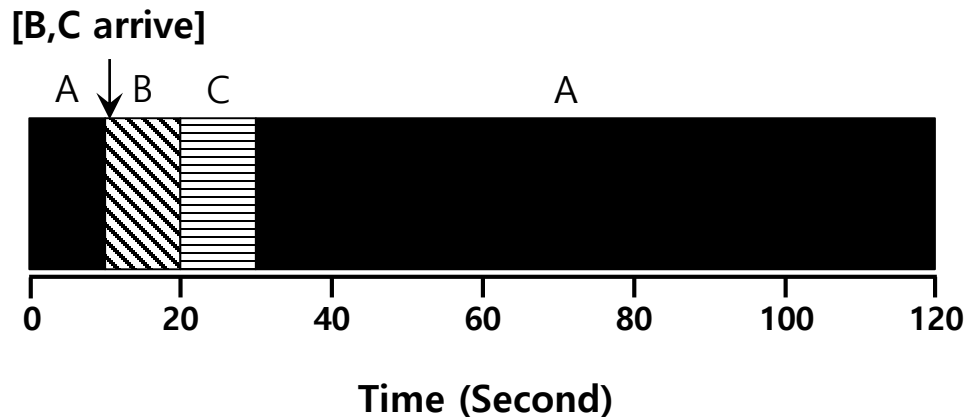
- **A new job enters the system:**

- ☐ Determine between the remaining time of current jobs and the new job
- ☐ Schedule the job which has the least time left

Shortest Time-to-Completion First (STCF)

■ Example:

- A arrives at $t=0$ and needs to run for 100 seconds.
- B and C arrive at $t=10$ and each need to run for 10 seconds



Avg TAT = 50
sec instead of
103 sec while
service time is
10 sec for B and
C

$$\text{Average turnaround time} = \frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50 \text{ sec}$$

New scheduling metric: Response time

- The time from **when the job arrives** to the **first time it is scheduled to run**.

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

- **STCF** (Shortest To Complete First) and related disciplines are not particularly good for response time.

How can we build a scheduler that is
sensitive to response time?

Round Robin (RR) Scheduling

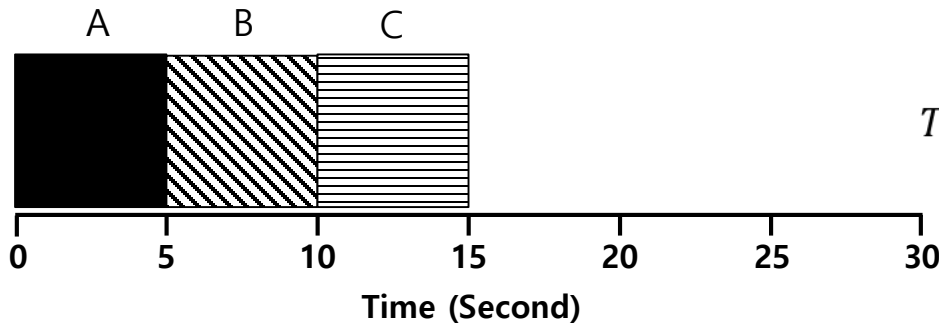
■ Time slicing Scheduling

- ❑ Run a job for a **time slice** and then switch to the next job in the **run queue** until the jobs are finished.
 - Time slice is sometimes called a scheduling quantum.
- ❑ It repeatedly does so until all jobs are finished.
- ❑ The length of a time slice must be *a multiple of the timer-interrupt period*.

RR is fair, but performs poorly on metrics
such as turnaround time

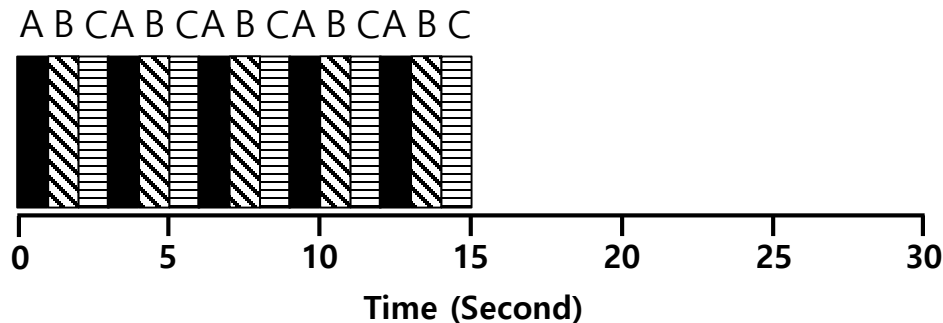
RR Scheduling Example

- A, B and C arrive at the same time.
- They each wish to run for 5 seconds.



$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$

SJF (Bad for Response Time)



$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

RR with a time-slice of 1sec (Good for Response Time and Not for Turn-Around Time)

The length of the time slice is critical

■ The shorter time slice

- ❑ Better response time
- ❑ The cost of context switching will dominate overall performance 😊.

■ The longer time slice

- ❑ Amortize the cost of switching
- ❑ Worse response time

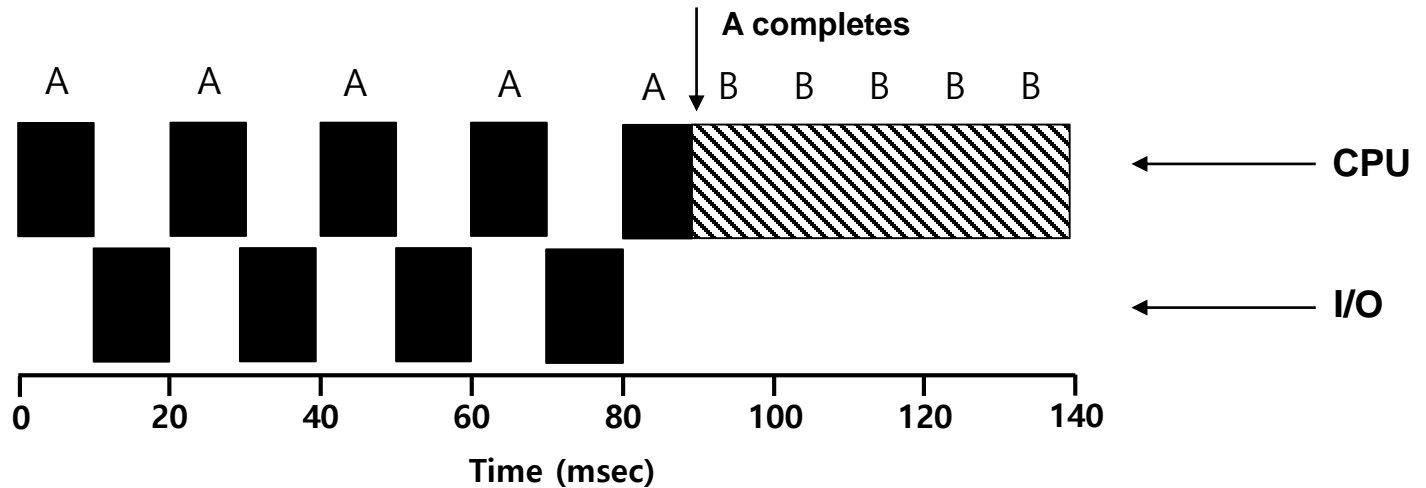
Deciding on the length of the time slice presents
a **trade-off** to a system designer

Incorporating I/O

- **Let's relax assumption 3:** All programs perform I/O
- **Example:**
 - A and B need 50ms of CPU time each.
 - A runs for 10ms and then issues an I/O request then repeat
 - I/Os each take 10ms
 - B simply uses the CPU for 50ms and performs no I/O
 - The scheduler runs A first, then B after

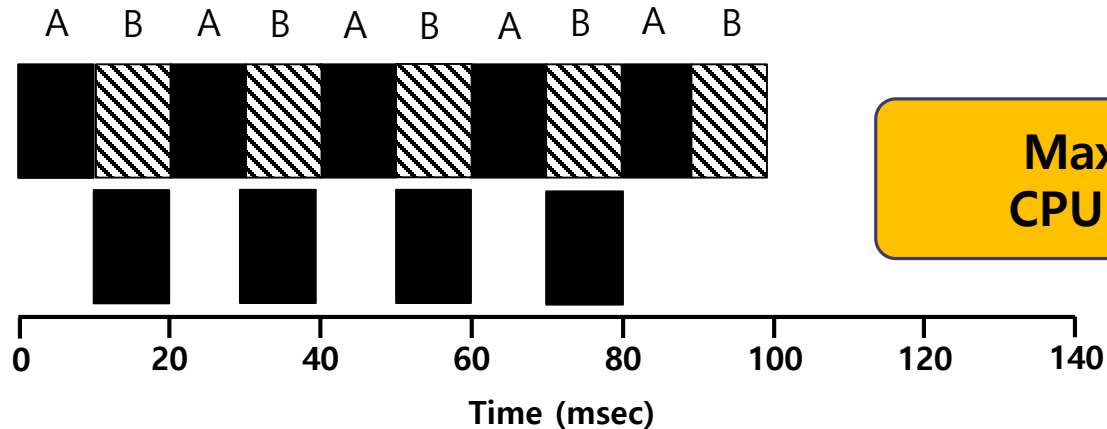
Incorporating I/O (Cont.)

$$\begin{aligned} \text{RPT} &= (0 + 90)/2 \\ &= 45 \\ \text{TAT} &= (90 + 140)/2 \\ &= 115 \end{aligned}$$



Poor Use of Resources

$$\begin{aligned} \text{RPT} &= (0 + 10)/2 \\ &= 5 \\ \text{TAT} &= (90 + 100)/2 \\ &= 95 \end{aligned}$$



Overlap Allows Better Use of Resources

**Maximize the
CPU utilization**

Incorporating I/O (Cont.)

■ When a job initiates an I/O request:

- ❑ The job is blocked waiting for I/O completion.
- ❑ The OS scheduler should schedule another job on the CPU.

■ When the I/O completes:

- ❑ An interrupt is raised.
- ❑ The OS moves the process from blocked back to the ready state.



Scheduling: Proportional Share

Proportional Share Scheduler

■ Fair-share scheduler

- ❑ Guarantee that each job obtain *a certain percentage* of CPU time.
- ❑ Not optimized for turnaround time or response time

Basic Concept

■ Tickets:

- ❑ Represent the share of a resource that a process should receive
- ❑ The percent of tickets represents its share of the system resource in question.

■ Example:

- ❑ **There are two processes, A and B and 100 total tickets:**
 - Process A has 75 tickets → receive 75% of the CPU
 - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- The scheduler picks a winning ticket:
 - Load the state of that *winning process* and run it.

- Example:

- There are 100 tickets
 - Process A has 75 tickets: 0 ~ 74 (out of 100)
 - Process B has 25 tickets: 75 ~ 99 (out of 100)

Generate random number between 0 - 99

Scheduler's winning tickets: 63 85 70 39 76 17 29 41 36 39 10 99 68 83 63

Resulting scheduler: A B A A B A A A A A A B A B A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms

■ Ticket currency:

- ❑ A user allocates tickets among their own jobs in whatever currency they would like.
- ❑ The system converts the currency into the correct global value.
- ❑ **Example:**
 - There are 200 tickets (Global currency)
 - Process type A has 100 tickets & 2 processes (A1, A2)
 - Process type B has 100 tickets & 1 process (B1)

User A $\rightarrow 500$ (A's currency) to A1 $\rightarrow 50$ (global currency)
 $\rightarrow 500$ (A's currency) to A2 $\rightarrow 50$ (global currency)

User B $\rightarrow 100$ (B's currency) to B1 $\rightarrow 100$ (global currency)

Ticket Mechanisms (Cont.)

■ Ticket transfer:

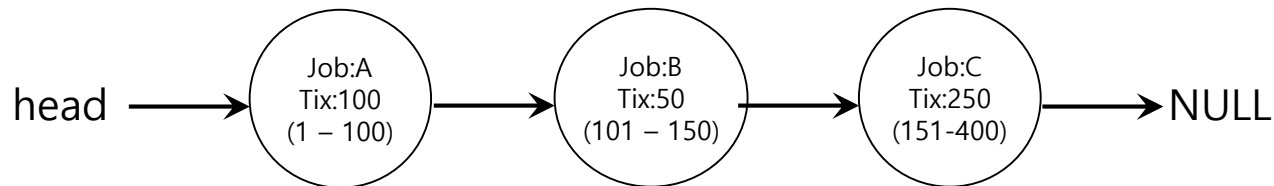
- ❑ A process can temporarily hand off *its tickets* to another process.

■ Ticket inflation:

- ❑ A process can temporarily raise or lower the number of tickets it owns.
- ❑ If any one process needs *more CPU time*, it can boost its tickets.

Implementation

- **Example:** There are three processes, A, B, and C.
 - **Keep the processes in a list:**



```
1      // counter: used to track if we've found the winner yet
2      int counter = 0;
3
4      // winner: use some call to a random number generator to
5      // get a value, between 0 and the total # of tickets (400)
6      int winner = getrandom(0, totaltickets);
7
8      // current: use this to walk through the list of jobs
9      node_t *current = head;
10
11     // loop until the sum of ticket values is > the winner
12     while (current) {
13         counter = counter + current->tickets;
14         if (counter > winner)
15             break; // found the winner (i.e., current)
16         current = current->next;
17     }
18     // 'current' is the winner: schedule it...
```

Implementation (Cont.)

■ U: unfairness metric

- The time the first job completes divided by the time that the second job completes. ← For equal service time $TAT_1/TAT_2 = 1$

■ Example: Assume two jobs arrive at the same time:

- **There are two jobs, each jobs has runtime 10:**

- First job finishes at time 10
- Second job finishes at time 20

- $U = \frac{10}{20} = 0.5$

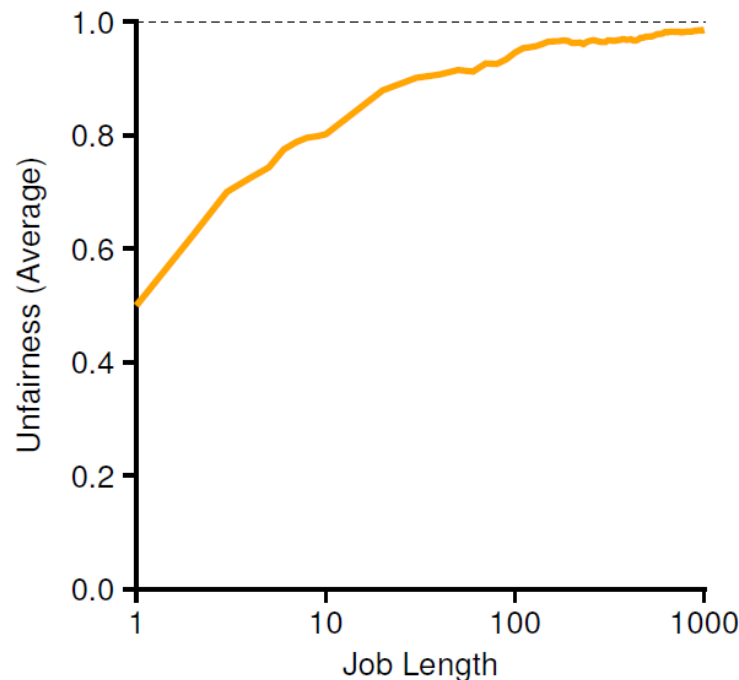
- U will be close to 1 when both jobs finish at nearly the same time.

Unfairness metric is between 0.5 – 1.0

TAT is “Turn Around Time”

Lottery Fairness Study

- There are two jobs:
 - Each jobs has the same number of tickets (100).



When the job length is not very long,
average unfairness can be **quite severe (number/small number)**.

Stride Scheduling

- **Stride** of each process (opposite of # of tickets)
 - $\text{Stride} = (\text{A large number}) / (\text{the number of tickets of the process})$
 - **Example:** Assume a large number = 10,000
 - Process A has 100 tickets \rightarrow stride of A is $(10,000/100) = 100$
 - Process B has 50 tickets \rightarrow stride of B is $(10,000/50) = 200$
- Select process with lowest “**Pass**” value (measure for how long you ran) to run (take it out of the queue); “Pass” is a measure that enables processes with high tickets (i.e., low stride) to run more frequently
- Run the selected process
- When done with the quantum, increment process “Pass” value by its “stride” value and put it back in the queue

```
current = remove_min(queue);           // pick client with minimum pass
schedule(current);                     // use resource for quantum
current->pass += current->stride;       // compute next pass using stride
insert(queue, current);                // put back into the queue
```

A pseudo code implementation

Stride Scheduling Example

Stride = 10,000 / tickets

Pass(A) tickets = 100 (stride=100)	Pass(B) Tickets = 50 (stride=200)	Pass(C) Tickets = 250 (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Pass

If new job enters with pass value 0,
It will **monopolize** the CPU!



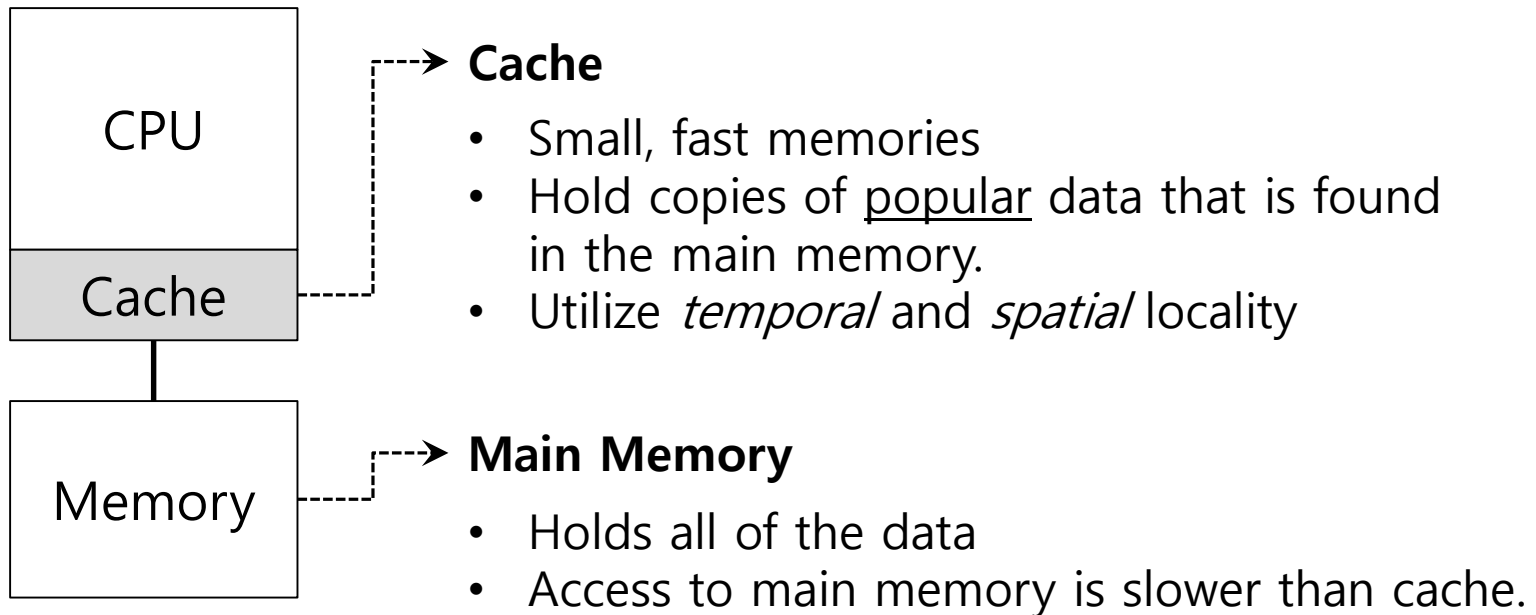
Multiprocessor Scheduling

Multiprocessor Scheduling

- The rise of the **multicore processor** is the source of multiprocessor-scheduling proliferation.
 - **Multicore**: Multiple CPU cores are packed onto a single chip and sharing memory.
- Adding more CPUs does not make that single application run faster → You'll have to rewrite application to run in parallel, using **threads**.

How to schedule jobs on **Multiple CPUs**?

Single CPU with cache

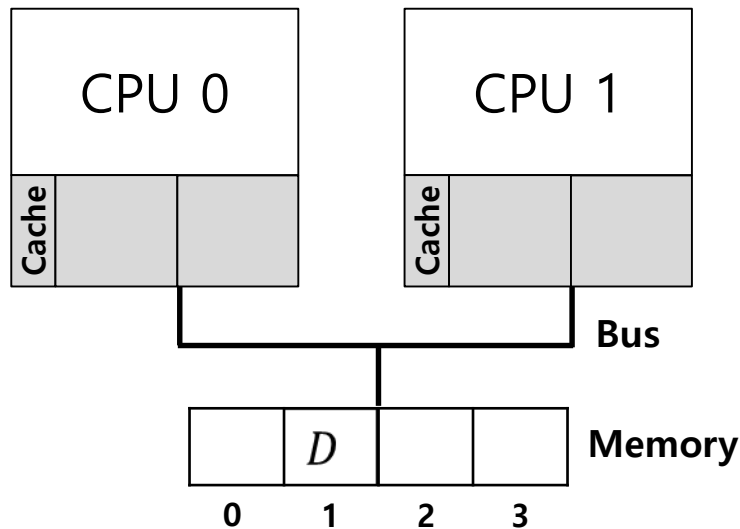


By keeping data in cache, the system can make slow memory
appear to be a fast one

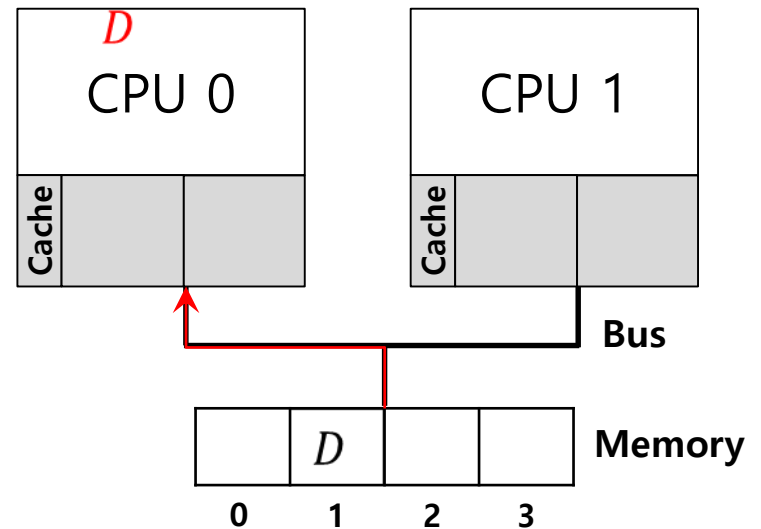
Cache coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

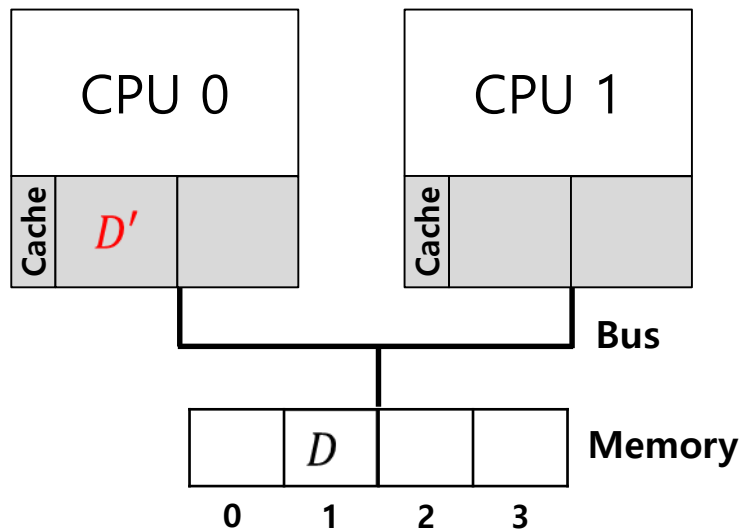


1. CPU0 reads a data at address 1.

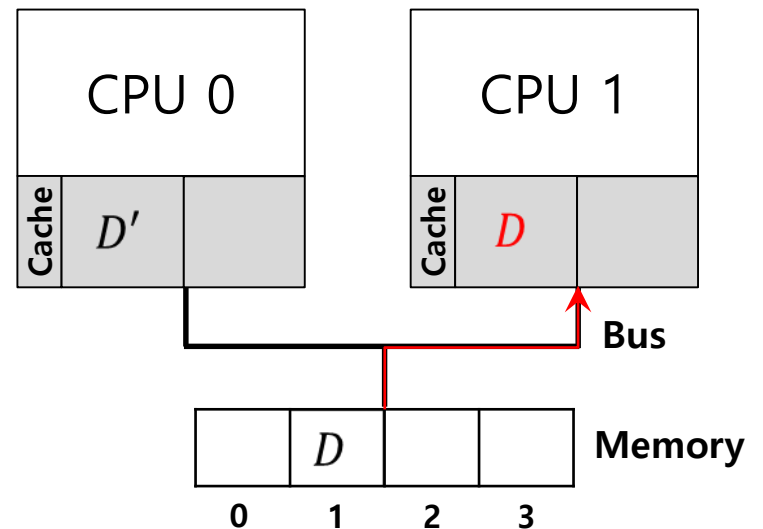


Cache coherence (Cont.)

2. D is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A

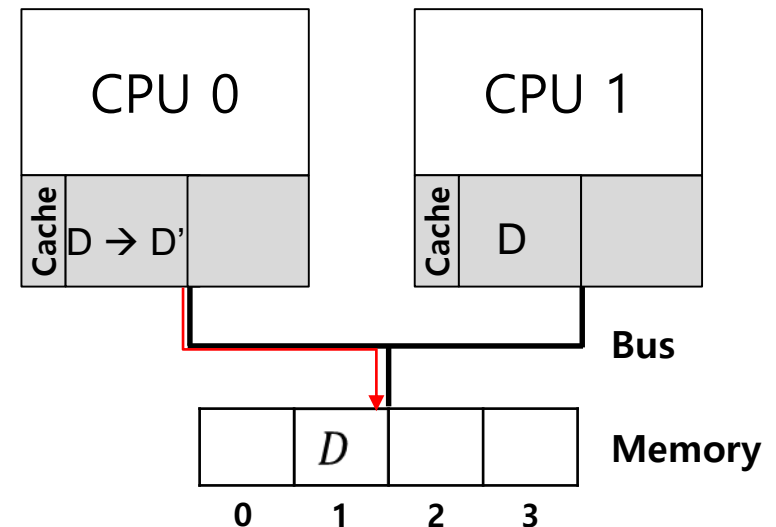


CPU1 gets the **old value D** instead of the correct value D' .

Cache coherence solution

■ Bus snooping:

- Each cache pays attention to memory updates by **observing the bus**.



- When a CPU sees an update for a data item it holds in its cache, it will notice the change and either invalidate its copy or update it.

Don't forget synchronization

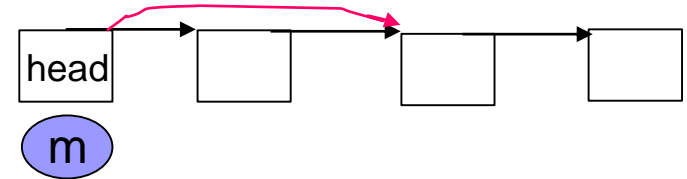
- When accessing shared data across CPUs, **mutual exclusion** primitives should be used to guarantee correctness.

```
1      typedef struct __Node_t {
2          int value;
3          struct __Node_t *next;
4      } Node_t;
5
6      int List_Pop() {
7          Node_t *tmp = head;           // remember old head ...
8          int value = head->value;       // ... and its value
9          head = head->next;             // advance head to next pointer
10         free(tmp);                    // free old head
11         return value;                 // return value at head
12     }
```

Simple List Delete Code

Don't forget synchronization (Cont.)

■ Solution



```
1      pthread_mutex_t  m;
2      typedef struct __Node_t {
3          int value;
4          struct __Node_t *next;
5      } Node_t;
6
7      int List_Pop() {
8          lock(&m)
9          Node_t *tmp = head;           // remember old head ...
10         int value = head->value;       // ... and its value
11         head = head->next;             // advance head to next pointer
12         free(tmp);                    // free old head
13         unlock(&m)
14         return value;                 // return value at head
15     }
```

Simple List Delete Code with lock

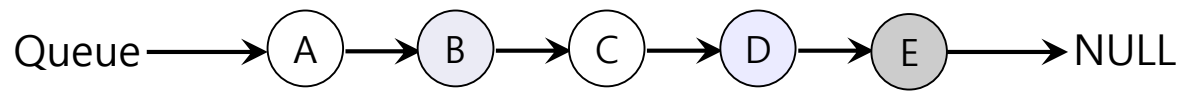
Cache Affinity

- Keep a process on **the same CPU** if at all possible:
 - A process typically builds up a fair bit of state in the cache of a CPU.
 - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU.

A multiprocessor scheduler should consider **cache affinity** when making its scheduling decision.

Single queue Multiprocessor Scheduling (SQMS)

- Put all jobs that need to be scheduled into a **single queue**:
 - Each CPU simply picks the next job from the globally shared queue.
 - **Cons:**
 - Some form of **locking** have to be inserted → **Lack of scalability**
 - **Cache affinity, i.e., lack of**
 - **Example:**

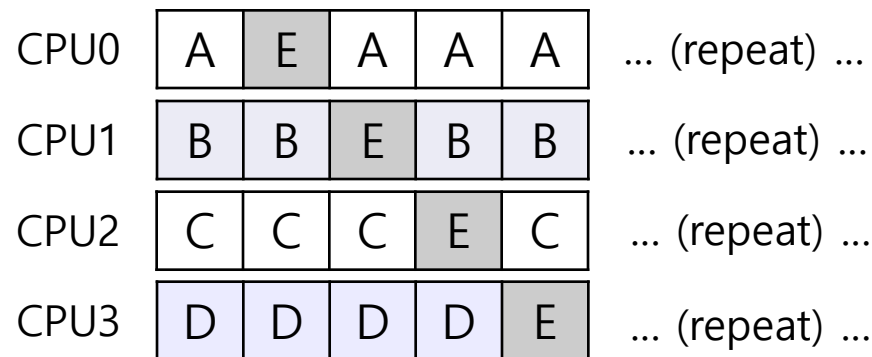
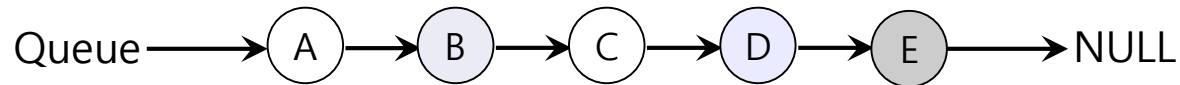


- **Possible job scheduler across CPUs:**

P.S. Process A switches between different CPUs constantly after every quantum

CPU0	A	E	D	C	B	... (repeat) ...
CPU1	B	A	E	D	C	... (repeat) ...
CPU2	C	B	A	E	D	... (repeat) ...
CPU3	D	C	B	A	E	... (repeat) ...

Scheduling Example with Cache affinity



❑ Preserving affinity for most

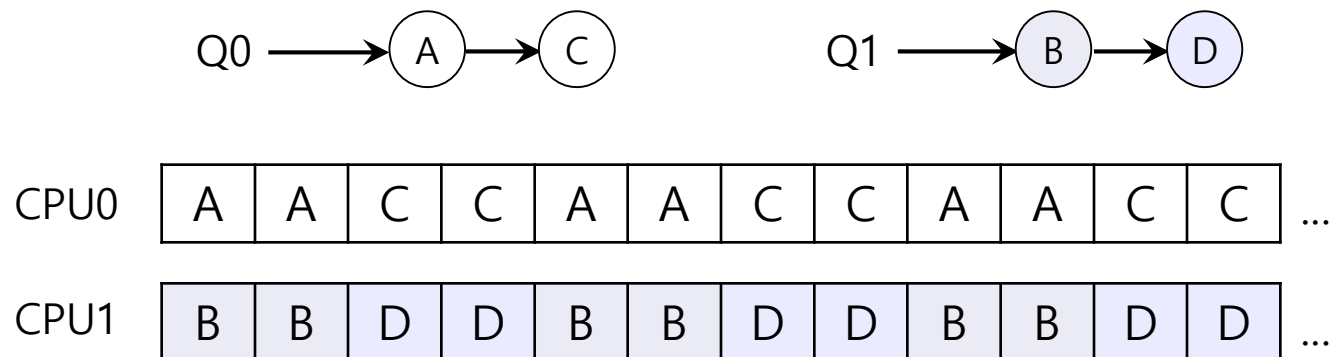
- Jobs A through D are not moved across processors.
 - Only job E Migrating from CPU to CPU.
- ❑ Implementing such a scheme can be **complex**.

Multi-queue Multiprocessor Scheduling (MQMS)

- MQMS consists of **multiple scheduling queues**:
 - Each queue per CPU will follow a particular scheduling discipline.
 - When a job enters the system, it is placed in **exactly one** scheduling queue.
 - Avoid/Minimize the problems of information sharing and synchronization.

MQMS Example

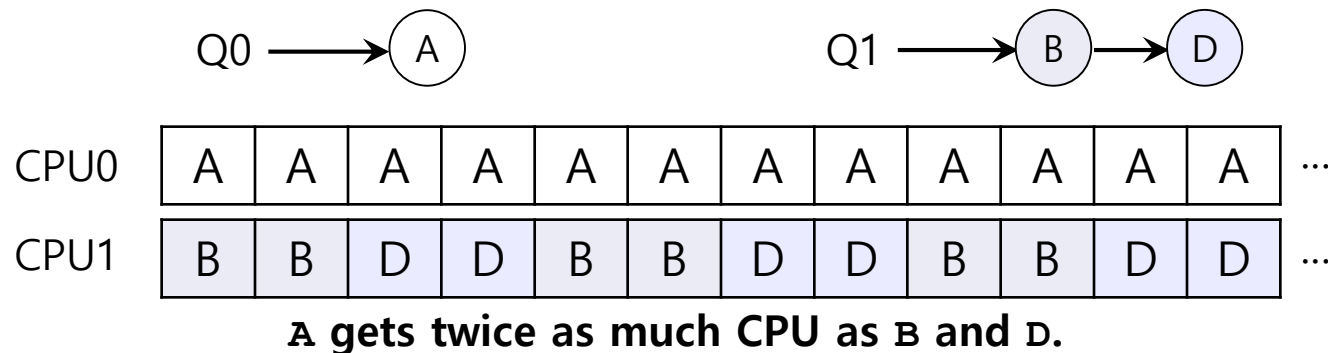
- With designated processes on every processor and use **round robin** between processes on every CPU, the system might produce a schedule that looks like this:



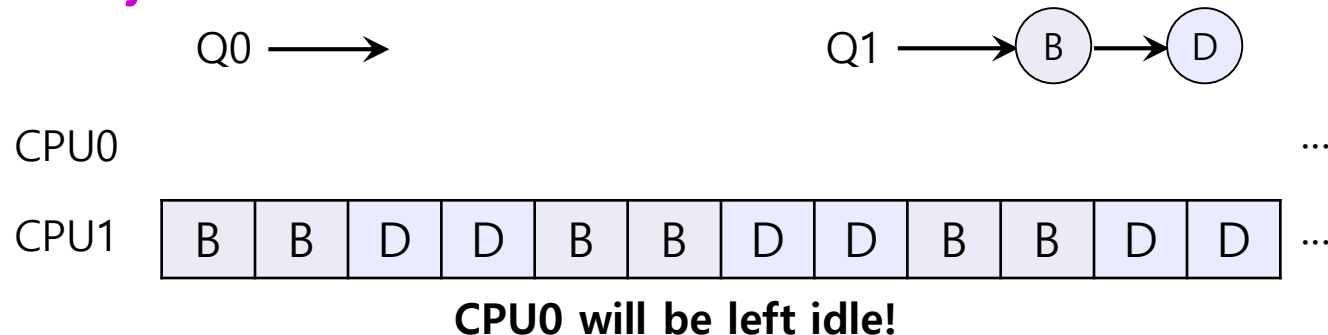
MQMS provides more scalability and cache affinity.

Load Imbalance issue of MQMS

- After job C in Q0 finishes:



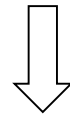
- After job A in Q0 finishes:



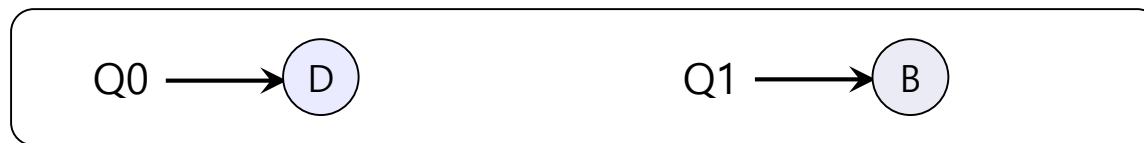
How to deal with load imbalance?

- The answer is to move jobs (**Migration**) – rebalance the workload:

- **Example:**



The OS moves one of B or D to CPU 0



Or



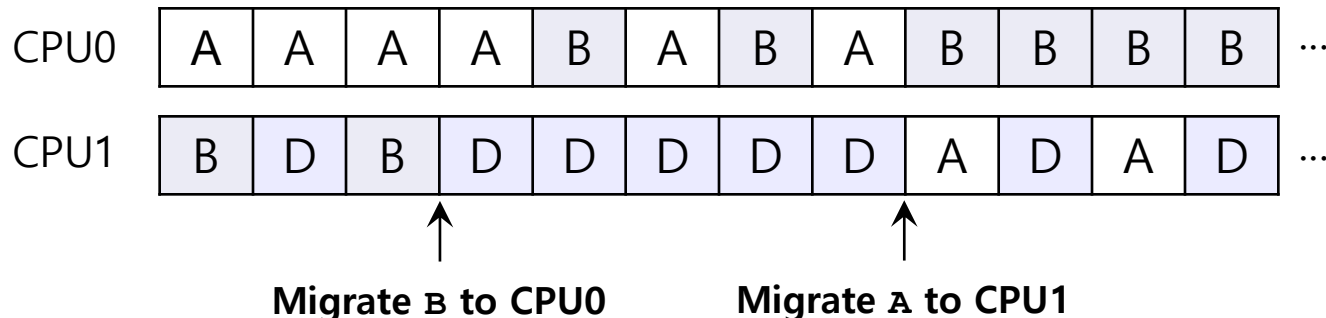
How to deal with load imbalance? (Cont.)

■ A more tricky case:



■ A possible migration pattern:

□ Keep switching jobs:



Work Stealing

■ Move jobs between queues:

□ Implementation:

- A source queue that is low on jobs is picked.
- The source queue occasionally peeks at another target queue.
- If the target queue is more full than the source queue, the source will “**steal**” one or more jobs from the target queue.

□ Cons:

- *High overhead, no CPU affinity, and trouble scaling*

Linux Multiprocessor Schedulers

■ $O(1)$:

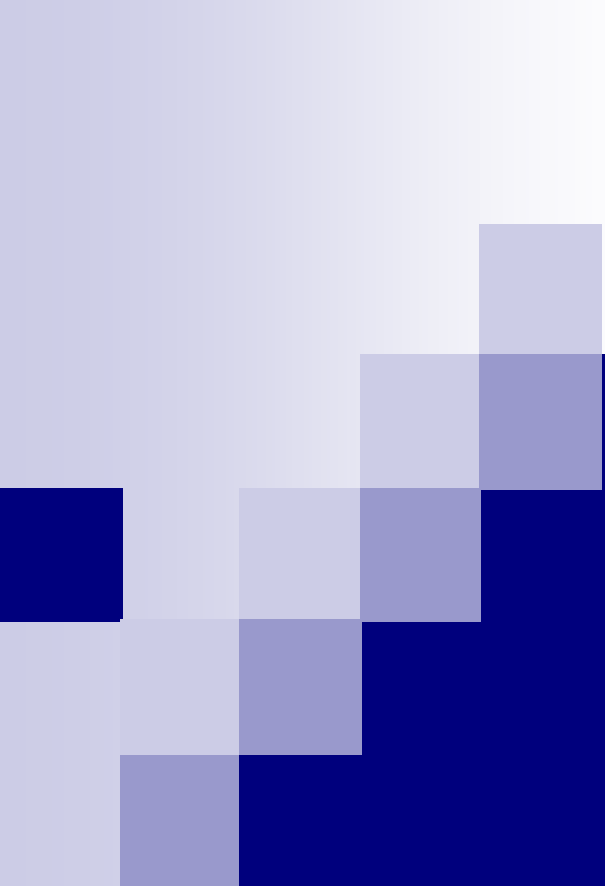
- ❑ A Priority-based scheduler
- ❑ Use Multiple queues
- ❑ Change a process's priority over time
- ❑ Schedule those with highest priority
- ❑ Interactivity is a particular focus

■ Completely Fair Scheduler (CFS – Linux 2.6.23):

- ❑ Deterministic proportional-share approach
- ❑ Multiple queues
- ❑ Goal is to maximize CPU utilization in interactive environment; instead of queue use Red/Black tree to order jobs execution

Linux Multiprocessor Schedulers (Cont.)

- **BF (Brain F) Scheduler (BFS):**
 - ❑ A single queue approach
 - ❑ Proportional-share
 - ❑ Based on **Earliest Eligible Virtual Deadline First (EEVDF)**
 - ❑ BFS has been retired in favor of MuQSS (Multiple Queue Skiplist Scheduler - a re-written implementation of the same concept)



Multi-level Feedback Queue Scheduling – Advanced Scheduling


MLFQ

- **Goal:** general-purpose scheduling
- Must support two job types with distinct goals
 - “**interactive**” programs care about **response time**
 - “**batch**” programs care about **turnaround time**
- **Approach:** multiple levels of round-robin; each level has higher priority than lower levels and preempts them


MLFQ: Priorities

Rule 1: If $\text{priority}(X) > \text{Priority}(Y)$, X runs

Rule 2: If $\text{priority}(X) == \text{Priority}(Y)$, X & Y run in RR

Q3 → 

“Multi-level”

Q2 → 

How to know how to set priority?

Q1

Approach 1: nice

Q0 →  → 

Approach 2: history “feedback”

MLFQ: Rules

Rule 1: If $\text{priority}(X) > \text{Priority}(Y)$, X runs

Rule 2: If $\text{priority}(X) == \text{Priority}(Y)$, X & Y run in RR

More rules:

Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process
(longer time slices result in lower priorities)



END