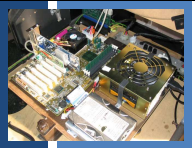
# TCSS 422: OPERATING SYSTEMS



**Limited Direct Execution, Introduction to Scheduling, Multilevel Feedback Queue**

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma



## OBJECTIVES

- Feedback from 1/10
- Homework 0 Questions
- Chapter 6: Limited Direct Execution
  - Virtualizing the CPU
- Ch. 7
  - Scheduling Introduction
  - Scheduling Metrics
  - Scheduling Methods
- Ch. 8
  - Multi-level feedback queue (MLFQ)



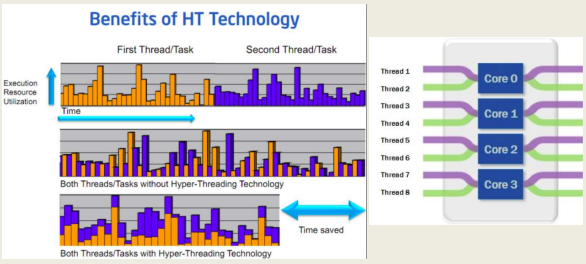
## TCSS422 - VIRTUAL MACHINES

- Stephen Rondeau has created TCSS422 Ubuntu VMs
- Instructions on how to connect and use VMs:
- http://css.insttech.washington.edu/~lab/Support/HowtoUse/UsingVCLQS.html



## SELECTED FEEDBACK FROM 1/10

- **What is hyperthreading in a CPU?**






## FEEDBACK - 2

- **How did the example program (execv) run "wc" when "wc" wasn't in the current working directory?**
- It is in the system's path variable:
- **Check your path variable:**

```
echo $PATH
```

- **Add to your path**

```
export PATH=$PATH:/home/mydir:/.
```

- **Check which command will be used**

```
whereis wc
```



## FEEDBACK - 3

- **How can a program run with exec, return back to the process that called it, when it finishes execution?**
- Fork(), then exec(), and have the parent wait()
- **Is it only exec() that prevents the remaining lines of code from being processed?**
- Yes… the process has been handed off to another executable, which then exits…

## FEEDBACK - 4

- **Can we go over the exec() examples again**
- **What is the difference between the execl's: *execl(), execlp(), execle()* and the execv's: *execv(), execvp(), execvpe()***
- **Execl's**
- Send a NULL terminated list of strings instead of an array
- Variants (execlp, execle, execl) are for different path settings
- **** New example execl() ****
- **Execv's**
- Parameterize exec using an array
- Variance (execv, execvp, execvpe) for different path settings



## FEEDBACK - 5

- **Can you spend a little time going over bash commands**
- Goal of assignment 0 is to engage students in using the internet to research how to accomplish tasks in Linux
- Develop and practice skills to seek answers to Linux and system-oriented questions



## SLIDES AND EXAMPLES

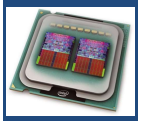
**Source Code Examples**

Source code for examples from class are posted **[HERE]**.

- **Examples posted online at:**
- **http://faculty.washington.edu/wlloyd/courses/tcss422/examples/**
- **Previous slides online at:**
- **http://faculty.washington.edu/wlloyd/teaching.html**
- *Slides from previous TCSS422 sections*



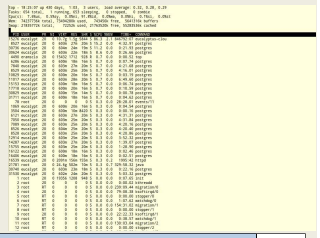
## CH. 6: LIMITED DIRECT EXECUTION





## VIRTUALIZING THE CPU

- How does the CPU support running so many jobs simultaneously?
- **Time Sharing**
- Tradeoffs:
  - Performance
    - Excessive overhead
  - Control
    - Fairness
    - Security
- Both HW and OS support is used





## COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

- What if programs could directly control the CPU / system?

| OS | Program |
|---|---|
| 1. Create entry for process list<br>2. Allocate memory for program<br>3. Load program into memory<br>4. Set up stack with `argc` / `argv`<br>5. Clear registers<br>6. Execute call `main()` | |
| | 7. Run `main()`<br>8. Execute `return` from `main()` |
| 9. Free memory of process<br>10. Remove from process list | |

## COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

- What if programs could directly control the CPU / system?

| OS | Program |
|---|---|
| 1. Create entry for process list<br>2. Allocate memory for | |
| 5. Clear registers<br>6. Execute call `main()` | 7. Run `main()`<br>8. Execute `return` from `main()` |
| 9. Free memory of process<br>10. Remove from process list | |

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



## DIRECT EXECUTION - 2

- **With direct execution:**

  How does the OS stop a program from running, and switch to another to support **time sharing**?

  How do programs share disks and perform I/O if they are given direct control? Do they know about each other?

  With direct execution, how can dynamic memory structures such as linked lists grow over time?

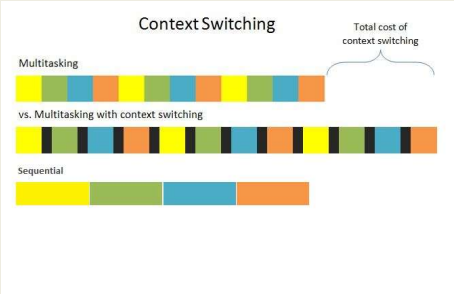


## CONTROL TRADEOFF

- **Too little control:**
  - No security
  - No time sharing
- **Too much control:**
  - Too much OS overhead
  - Poor performance for compute & I/O
  - Complex APIs (system calls), difficult to use



## CONTEXT SWITCHING OVERHEAD






## LIMITED DIRECT EXECUTION

- OS implements LDE to support time/resource sharing
- Limited direct execution means "only limited" processes can execute DIRECTLY on the CPU in ***trusted*** mode
- TRUSTED means the process is trusted, and it can do anything… (e.g. it is a system / kernel level process)
- Enabled by ***protected (safe) control transfer***
- CPU supported context switch
- Provides data isolation



## CPU MODES

- Utilize CPU Privilege Rings (Intel x86)
  - rings 0 (kernel), 1 (VM kernel), 2 (unused), 3 (user)

  access ← no access
- **User mode**:
  Application is running, but w/o direct I/O access
- **Kernel mode**:
  OS kernel is running performing restricted operations

## CPU MODES

- **User mode: ring 3 - untrusted**
  - Some instructions and registers are disabled by the CPU
  - Exception registers
  - HALT instruction
  - MMU instructions
  - OS memory access
  - I/O device access
- **Kernel mode: ring 0 – trusted**
  - All instructions and registers enabled



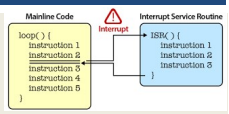
## SYSTEM CALLS

- Implement restricted "OS" operations
- Kernel exposes key functions through an API:
  - Device I/O (e.g. file I/O)
  - Task swapping: context switching between processes
  - Memory management/allocation: malloc()
  - Creating/destroying processes



## TRAPS: SYSTEM CALLS, EXCEPTIONS, INTERRUPTS

- Trap: any transfer to kernel mode
- Three kinds of traps
  - **System call:** (planned) user → kernel
    - SYSCALL for I/O, etc.
  - **Exception:** (error) user → kernel
    - Div by zero, page fault, page protection error
  - **Interrupt:** (event) user → kernel
    - Non-maskable vs. maskable
    - Keyboard event, network packet arrival, timer ticks
    - Memory parity error (ECC), hard drive failure

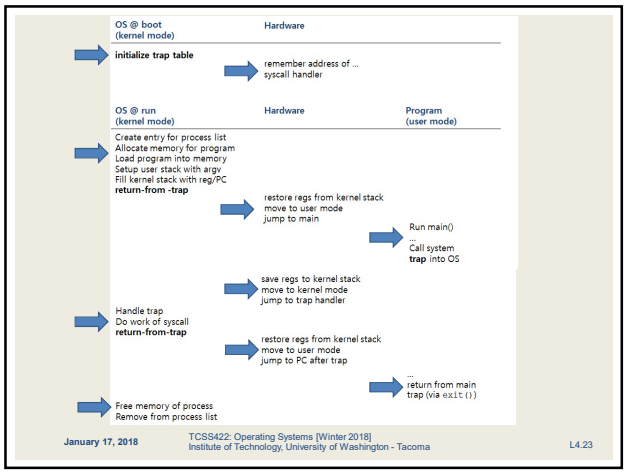




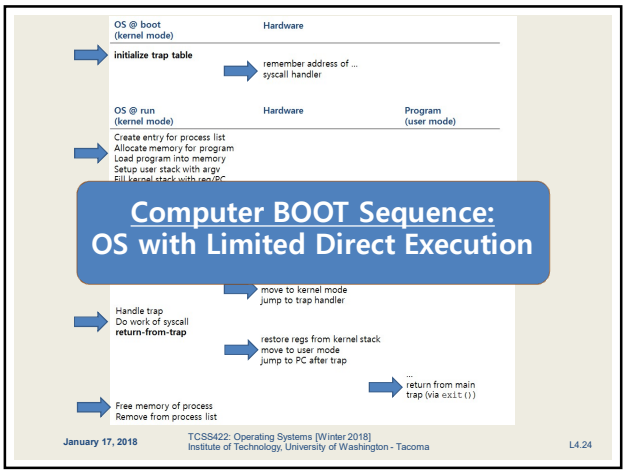

## EXCEPTION TYPES

| Exception type | Synchronous vs. asynchronous | User request vs. coerced | User maskable vs. nonmaskable | Within vs. between instructions | Resume vs. terminate |
|---|---|---|---|---|---|
| I/O device request | Asynchronous | Coerced | Nonmaskable | Between | Resume |
| Invoke operating system | Synchronous | User request | Nonmaskable | Between | Resume |
| Tracing instruction execution | Synchronous | User request | User maskable | Between | Resume |
| Breakpoint | Synchronous | User request | User maskable | Between | Resume |
| Integer arithmetic overflow | Synchronous | Coerced | User maskable | Within | Resume |
| Floating-point arithmetic overflow or underflow | Synchronous | Coerced | User maskable | Within | Resume |
| Page fault | Synchronous | Coerced | Nonmaskable | Within | Resume |
| Misaligned memory accesses | Synchronous | Coerced | User maskable | Within | Resume |
| Memory protection violation | Synchronous | Coerced | Nonmaskable | Within | Resume |
| Using undefined instruction | Synchronous | Coerced | Nonmaskable | Within | Terminate |
| Hardware malfunction | Asynchronous | Coerced | Nonmaskable | Within | Terminate |
| Power failure | Asynchronous | Coerced | Nonmaskable | Within | Terminate |







## MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- Cooperative multitasking (mostly pre 32-bit)
  - < Windows 95, Mac OSX
  - Opportunistic: running programs must give up control
    - User programs must call a special **yield** system call
    - When performing I/O
    - Illegal operations
  - (POLLEV)
    What problems could you for see with this approach?

January 17, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L4.25

## MULTITASKING

- How/when should the OS regain control of the CPU to switch between processes?
- Cooperative multitasking (mostly pre 32-bit)
  - < W
  - Op
    - Illegal operations
  - (POLLEV)
    What problems could you for see with this approach?

A process gets stuck in an infinite loop.
→ **Reboot the machine**

January 17, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L4.26

**W** What problems exist for regaining the control of the CPU with cooperative multitasking OSes?

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app** Total Results

## MULTITASKING - 2

- Preemptive multitasking (32 & 64 bit OSes)
- >= Mac OSX, Windows 95+
- Timer interrupt
  - Raised at some regular interval (in ms)
  - Interrupt handling
    1. Current program is halted
    2. Program states are saved
    3. OS Interrupt handler is run (kernel mode)
- (PollEV) What is a good interval for the timer interrupt?

January 17, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L4.28

## MULTITASKING - 2

- Preemptive multitasking (32 & 64 bit OSes)
- >= Mac OSX, Windows 95+
- Timer
  - Rais
  - Inte
    1. Current program is halted
    2. Program states are saved
    3. OS Interrupt handler is run (kernel mode)
- (PollEV) What is a good interval for the timer interrupt?

A **timer interrupt** gives OS the ability to run again on a CPU.

January 17, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L4.29

**W** For an OS that uses a system timer to force arbitrary context switches to share the CPU, what is a good value (in seconds) for the time interrupt?

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app** Total Results

## CONTEXT SWITCH

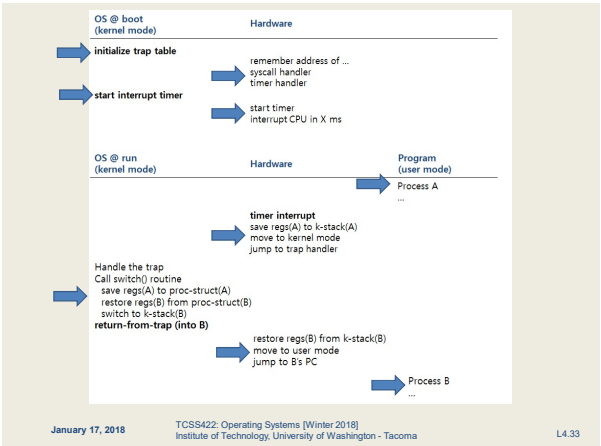
- Preemptive multitasking initiates "trap" into the OS code to determine:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS performs a context switch swapping out the current process for a new one.



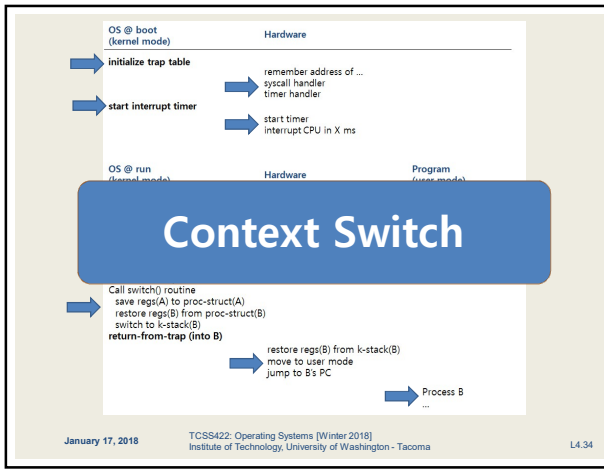
## CONTEXT SWITCH - 2

1. Save register values of the current process to its kernel stack
   - General purpose registers
   - PC: program counter (instruction pointer)
   - kernel stack pointer
2. Restore soon-to-be-executing process from its kernel stack
3. Switch to the kernel stack for the soon-to-be-executing process










## INTERRUPTED INTERRUPTS

- What happens if during an interrupt (trap to kernel mode), another interrupt occurs?
- Linux
  - < 2.6 kernel: non-preemptive kernel
  - >= 2.6 kernel: preemptive kernel



## PREEMPTIVE KERNEL

- Use "locks" as markers of regions of non-preemptibility (non-maskable interrupt)
- Preemption counter (`preempt_count`)
  - begins at zero
  - increments for each lock acquired (not safe to preempt)
  - decrements when locks are released
- Interrupt can be interrupted when `preempt_count=0`
  - It is safe to preempt (maskable interrupt)
  - the interrupt is more important

# CHAPTER 7- SCHEDULING: INTRODUCTION



## SCHEDULING INTRODUCTION

- For simplicity, consider job scheduling with limitations:
  - Each job requires the same CPU time
  - All jobs arrive at the same time
  - All jobs only use the CPU (no I/O)
  - The run-time of each job is known a priori



## SCHEDULING METRICS

- **Metrics**: A standard measure to quantify to what degree a system possesses some property. Metrics provide *repeatable* techniques to quantify and compare systems.
- **Measurements** are the numbers derived from the application of metrics
- Scheduling Metric #1: Turnaround time
- The time at which the job completes minus the time at which the job arrived in the system

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

- How is turnaround time different than execution time?



## SCHEDULING METRICS - 2

- Scheduling Metric #2: Fairness
  - Jain's fairness index
  - Quantifies if jobs receive a fair share of system resources

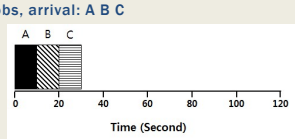
$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} x_i^2}$$

- n processes
- $x_i$ is time share of each process
- worst case = 1/n
- best case = 1
- Consider n=3, worst case = .333, best case=1
- With n=3 and $x_1$=.2, $x_2$=.7, $x_3$=.1, fairness=.62
- With n=3 and $x_1$=.33, $x_2$=.33, $x_3$=.33, fairness=1



## SCHEDULERS

- FIFO: first in, first out
  - Very simple, easy to implement
- Consider
  - 3 x 10sec jobs, arrival: A B C

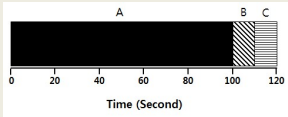



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$



## FIFO: CONVOY EFFECT

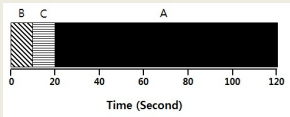
- FIFO with different jobs lengths
- Consider
  - $A_{len}$=100sec, $B_{len}$=10sec, $C_{len}$=10sec




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

## SJF: SHORTEST JOB FIRST

- Given that we know execution times in advance:
  - Run in order of duration, shortest to longest
  - Non preemptive scheduler
  - This is not realistic
  - Arrival: A B C

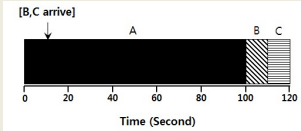



$$Average\ turnaround\ time = \frac{10 + 20 + 120}{3} = 50\ sec$$



## SJF: WITH RANDOM ARRIVAL

- If jobs arrive at any time:
- A @ t=0sec, B @ t=10sec, C @ t=10sec




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



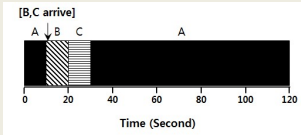
## STCF – SHORTEST TIME TO COMPLETION FIRST

- Add preemption to the Shortest Job First scheduler
  - Also called preemptive shortest job first (PSJF)
- When a new job enters the system:
  - Of all jobs, *Which has the least time left?*
  - PREMPT job execution, and schedule the **new** shortest job
- More realistic, but how do we know execution time in advance?
  - Oracle: All knowing one
  - Only schedule static (fixed size) batch workloads
  - Can we predict execution time?



## STCF - 2

- Consider:
  - $A_{len}$=100 $A_{arrival}$=0
  - $B_{len}$=10, $B_{arrival}$=10, $C_{len}$=10, $C_{arrival}$=10




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



## SCHEDULING METRICS - 3

- Scheduling Metric #3: **Response Time**
- Time from when job arrives until it starts execution

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

- STCF, SJF, FIFO
  - can perform poorly with respect to response time

**What scheduling algorithm(s) can help minimize response time?**



## RR: ROUND ROBIN

- Run each job awhile, then switch to another distributing the CPU evenly (fairly)
- Scheduling Quantum is called a time slice
- Time slice must be a multiple of the timer interrupt period.

| Process | Burst Time |
|---|---|
| P1 | 12 |
| P2 | 8 |
| P3 | 4 |
| P4 | 10 |
| P5 | 5 |

**Round Robin scheduling algorithm Gantt chart**

Scheduling Quantum = 5 seconds

| P1 | P2 | P3 | P4 | P5 | P1 | P2 | P4 | P1 |
|---|---|---|---|---|---|---|---|---|

0 5 10 14 19 24 29 32 37 39

## RR: ROUND ROBIN

- Run each job awhile, then switch to another distributing the CPU evenly (fairly)
- Scheduling Quantum is called a time slice
- Time a mu timer interrupt period.

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

| Process | Burst Time |
|---|---|
| P1 | 12 |
| | |
| | |
| | |
| P5 | 5 |

Round Robin scheduling algorithm Gantt chart

Scheduling Quantum = 5 seconds

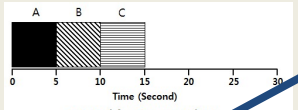
| P1 | P2 | P3 | P4 | P5 | P1 | P2 | P4 | P1 |
|---|---|---|---|---|---|---|---|---|

0 5 10 14 19 24 29 32 37 39



## RR EXAMPLE

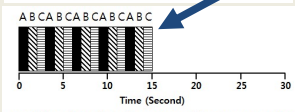
- ABC arrive at time=0, each run for 5 seconds




SJF (Bad for Response Time)

OVERHEAD not considered

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




RR with a time-slice of 1sec (Good for Response Time)

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



## ROUND ROBIN: TRADEOFFS

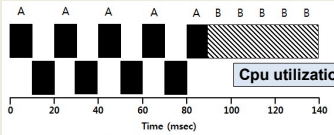
| Short Time Slice | Long Time Slice |
|---|---|
| Fast Response Time | Slow Response Time |
| High overhead from context switching | Low overhead from context switching |

- Time slice impact:
  - Average turnaround time: ts(1,2,3,4,5)=14,14,13,14,10
  - Fairness: round robin is always fair, J=1



## SCHEDULING WITH I/O

- STCF scheduler
  - A: CPU=50ms, I/O=40ms, 10ms intervals
  - B: CPU=50ms, I/O=0ms
  - Consider A as 10ms subjobs (CPU, then I/O)
- Without considering I/O:



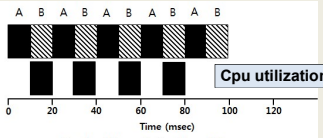

Cpu utilization = 100/140=71%

Poor Use of Resources



## SCHEDULING WITH I/O - 2

- When a job initiates an I/O request
  - A is blocked, waits for I/O to compute, frees CPU
  - STCF scheduler assigns B to CPU
- When I/O completes → raise interrupt
  - Unblock A, STCF goes back to executing A: (10ms sub-job)

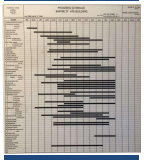



Cpu utilization = 100/100=100%

Overlap Allows Better Use of Resources



## CHAPTER 8 – MULTI-LEVEL FEEDBACK QUEUE (MLFQ) SCHEDULER



## MULTI-LEVEL FEEDBACK QUEUE

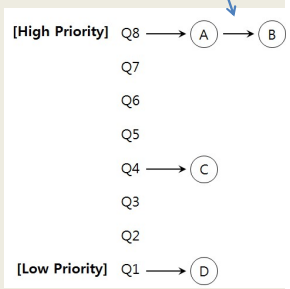
- Objectives:
  - Improve turnaround time:
    *Run shorter jobs first*
  - Minimize response time:
    *Important for interactive jobs (UI)*
- Achieve without a priori knowledge of job length



## MLFQ - 2

Round-Robin within a Queue

- Multiple job queues
- Adjust job priority based on observed behavior
- Interactive Jobs
  - Frequent I/O → keep priority high
  - Interactive jobs require fast response time (GUI/UI)
- Batch Jobs
  - Require long periods of CPU utilization
  - Keep priority low






## MLFQ: DETERMINING JOB PRIORITY

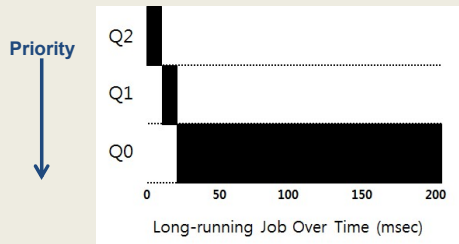
- New arriving jobs are placed into highest priority queue
- If a job uses its entire time slice, priority is reduced (↓)
  - Jobs appears CPU-bound ( "batch" job), not interactive (GUI/UI)
- If a job relinquishes the CPU for I/O priority stays the same

**MLFQ approximates SJF**



## MLFQ: LONG RUNNING JOB

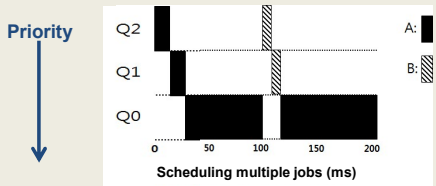
- Three-queue scheduler, time slice=10ms






## MLFQ: BATCH AND INTERACTIVE JOBS

- $A_{arrival\_time}$ =0ms, $A_{run\_time}$=200ms,
- $B_{run\_time}$ =20ms, $B_{arrival\_time}$ =100ms






## QUESTIONS