TCSS 422: OPERATING SYSTEMS

Limited Direct Execution, Introduction to CPU Scheduling

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

FEEDBACK 1/14

- fork() – will a modification to the data of a forked process trigger Copy-and-Write to duplicate all process data?, or only that which was modified… ?
  - Only modified data – saves memory and time
  - How can this be tested?
- Example: exec.c revisited - example using wait()
  - Recompiled exec.c
  - Unable to reproduce bug (parent existing before child)
  - Old version was compiled with previous Linux kernel, suspect changes may have caused different behavior
  - Check out: waitpid() and capturing return code from wait()

FEEDBACK - 2

- Why **exec**, instead of just calling another function like in Java?
  - Fork(), exec(), and wait() allow ***fine grained*** control over processes
  - Goal: remotely invoke another executable program, from an existing C program
  - Can fork() a new processes to delegate to run an external program
  - Can redirect input, output, stderr of processes
- Exec.. is still a little complex
  - 6 versions: execl(), execlp(), execle(), execv(), execvp(), execvpe()
  - "p" – duplicates the PATH of the shell
  - "e" – allows the environment (variables) to be passed in as an array of pointers to NULL-terminated strings, list is NULL terminated
  - "v" – provide cmd & args as array (***vector***)
  - "l" – provide cmd & args as (***list***) of individual params sent to the function **(see execl.c example)**

exec()

- Supports running an external program
- 6 types: execl(), execlp(), execle(), execv(), execvp(), execvpe()
- execl(), execlp(), execle(): const char *arg ***(example: execl.c)***

  Provide cmd and args as individual params to the function
  Each arg is a pointer to a null-terminated string
  ODD: pass a variable number of args: (arg0, arg1, .. argn)
- Execv(), execvp(), execvpe() ***(example: exec.c)***
  Provide cmd and args as an Array of pointers to strings

  Strings are null-terminated
  First argument is name of command being executed
  Fixed number of args passed in

FEEDBACK - 3

- What is the (too much / too little) control tradeoff for Operating System designs?
  - Who (what) has too much/little control?
  - What are consequences of too much control?
  - … consequences of too little control?
- What is direct vs. limited direct execution with respect to operating systems?

  - **Direct** – code runs directly on HW, OS can not intervene, processes can "run away"…
  - **Limited direct** – code runs on HW, but can not directly access protected resources (memory, devices), OS can intervene and *preempt* a process

FEEDBACK - 4

- How do we identify system processes?
- All kernel processes (threads) are children of process (pid==2)
- **"ps" command to display processes owned by the kernel:**

```
ps --ppid 2 -p 2 -o
uname,pid,ppid,pcpu,pmem,vsz,rssize,start,time,cmd |
more
```

- **"ps" command to display non-kernel processes:**

```
ps --ppid 2 -p 2 -o
uname,pid,ppid,pcpu,pmem,vsz,rssize,start,time,cmd -
-deselect | more
```

- What is significant regarding the output?

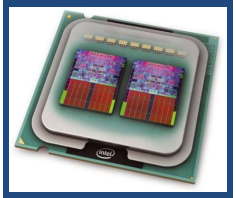
FEEDBACK – 5

- Hard time following along. Overwhelming amount of information. Can't seem to find guideline/direction
  - AFTER CLASS - :
  - (1) Review Slides
  - (2) Read chapters (in this case Ch. 5 & 6)
  - (3) While reviewing/reading, ***make list*** of confusing topics
  - (4) ***Formulate questions*** about these topics
  - THEN
  - (1) Visit professor after class, during office hours, or make an appointment --or--
  - (2) Email list of questions to professor

OBJECTIVES

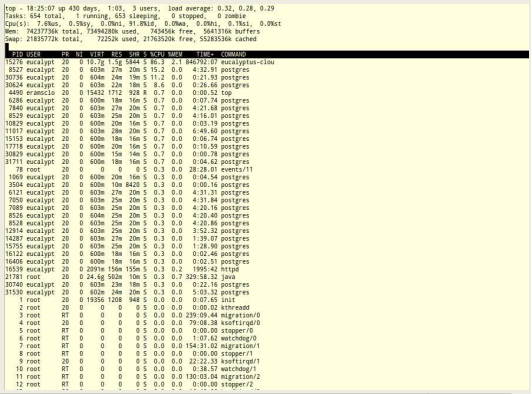
- Active Reading Quiz 1
- Assignment 0 / Linux Tutorial
- C Tutorial
- Chapter 6 – Limited Direct Execution
- **CPU Scheduling:**
- Chapter 7 – Scheduling Introduction
- Chapter 8 – Multi-level Feedback Queue (MLFQ)

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



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

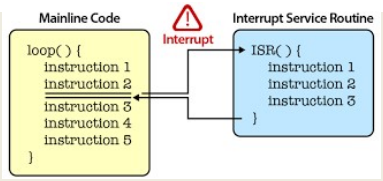
CPU MODES

- **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

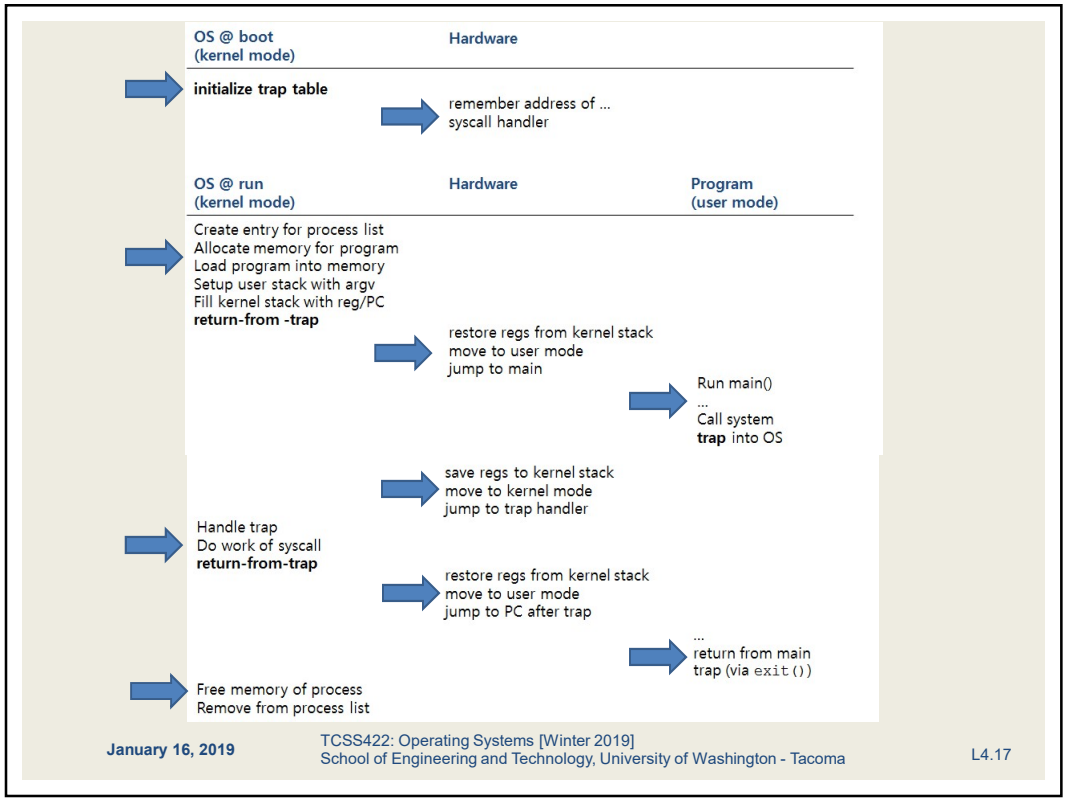
TRAPS: SYSTEM CALLS, EXCEPTIONS, INTERRUPTS

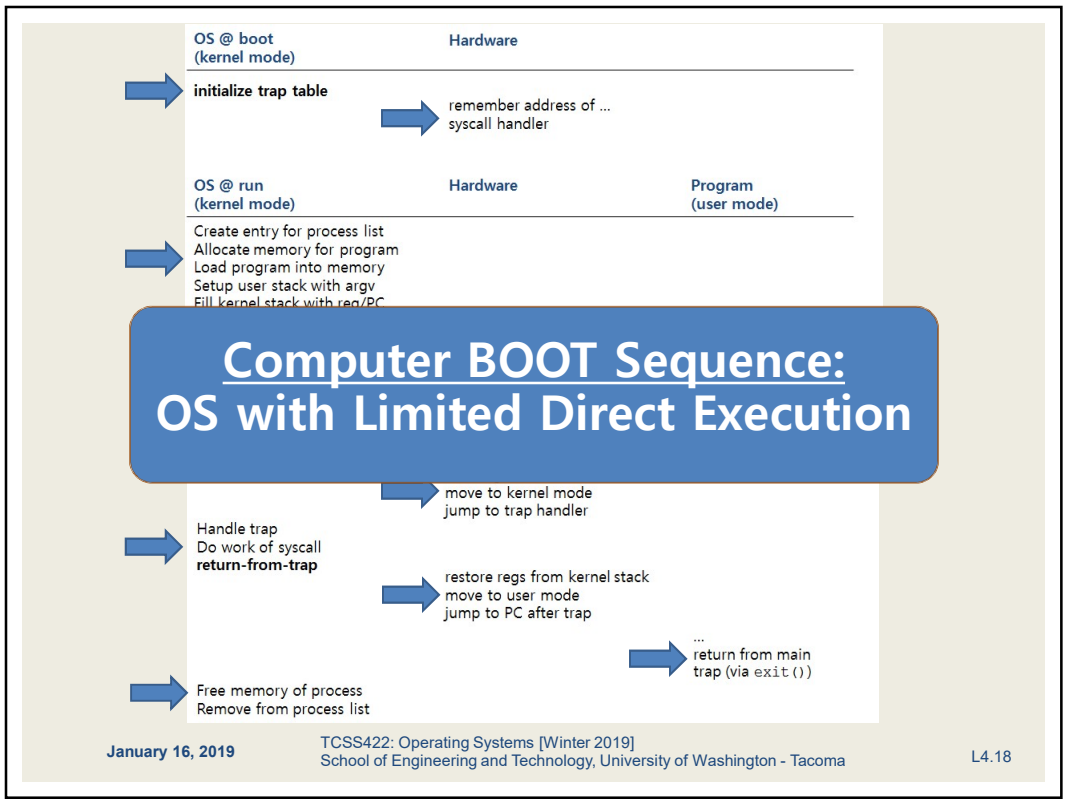


- Trap: any transfer to kernel mode
- Three kinds of traps
  - **System call:** (planned) user → kernel
    - SYSCALL for I/O, etc.
  - **Exception:** (error) user → kernel
    - Div by zero, page fault, page protection error
  - **Interrupt:** (event) user → kernel
    - Non-maskable vs. maskable
    - 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


OS @ boot (kernel mode)
Hardware
initialize trap table
remember address of ... syscall handler
OS @ run (kernel mode)
Hardware
Program (user mode)
Create entry for process list
Allocate memory for program
Load program into memory
Setup user stack with argv
Fill kernel stack with reg/PC
return-from -trap
restore regs from kernel stack
move to user mode
jump to main
Run main()
...
Call system trap into OS
save regs to kernel stack
move to kernel mode
jump to trap handler
Handle trap
Do work of syscall
return-from-trap
restore regs from kernel stack
move to user mode
jump to PC after trap
...
return from main
trap (via exit())
Free memory of process
Remove from process list
January 16, 2019
TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma
L4.17


OS @ boot (kernel mode)
Hardware
initialize trap table
remember address of ... syscall handler
OS @ run (kernel mode)
Hardware
Program (user mode)
Create entry for process list
Allocate memory for program
Load program into memory
Setup user stack with argv
Computer BOOT Sequence:
OS with Limited Direct Execution
move to kernel mode
jump to trap handler
Handle trap
Do work of syscall
return-from-trap
restore regs from kernel stack
move to user mode
jump to PC after trap
...
return from main
trap (via exit())
Free memory of process
Remove from process list
January 16, 2019
TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma
L4.18

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?

MULTITASKING

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

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

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

QUESTION: MULTITASKING

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

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?

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)

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

- (PollEV) What is a good interval for the timer interrupt?

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 timer interrupt?

QUESTION: TIME SLICE

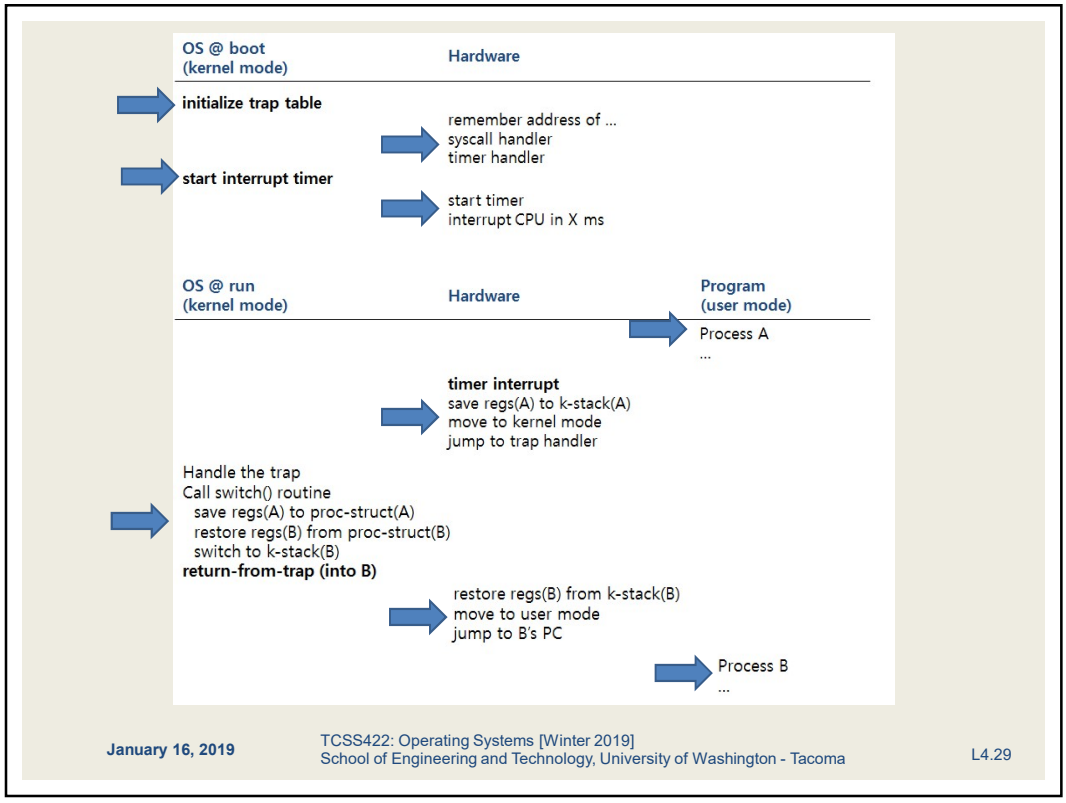
- For an OS that uses a system timer to force arbitrary context switches to share the CPU, what is a good value (in seconds) for the timer interrupt?

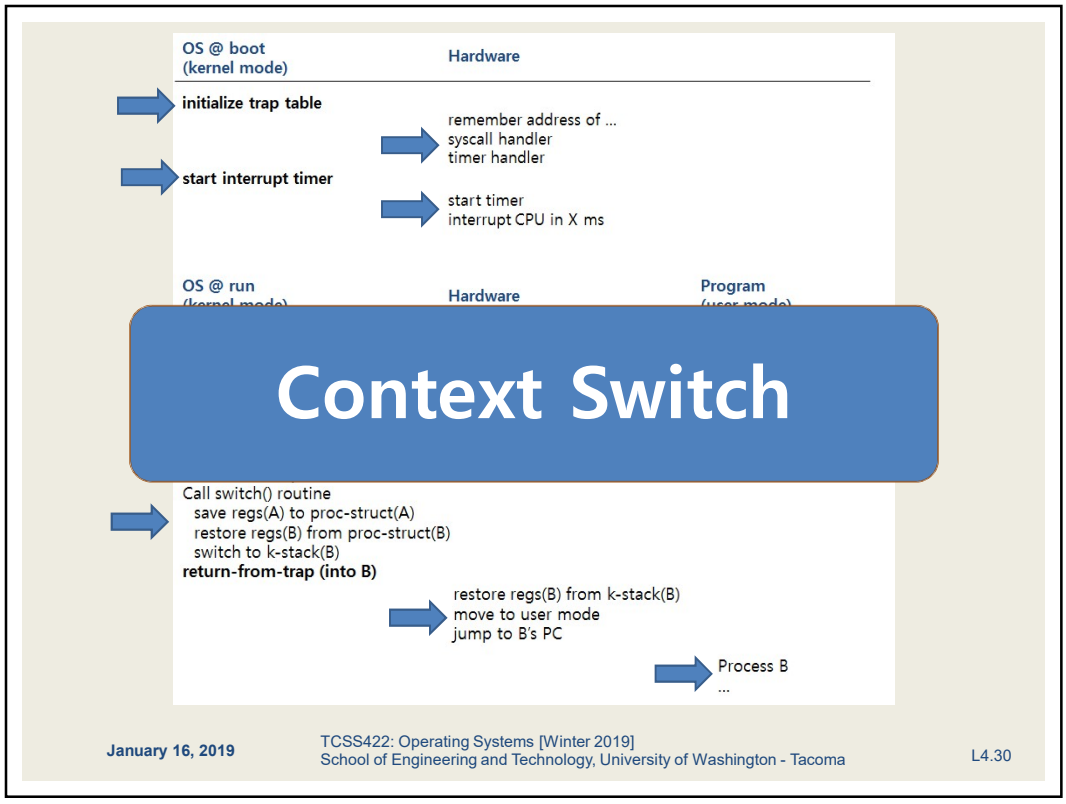
CONTEXT SWITCH

- 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


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
...
timer interrupt
save regs(A) to k-stack(A)
move to kernel mode
jump to trap handler
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
...
January 16, 2019
TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma
L4.29


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
Hardware
Program
Context Switch
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
...
January 16, 2019
TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma
L4.30

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 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

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

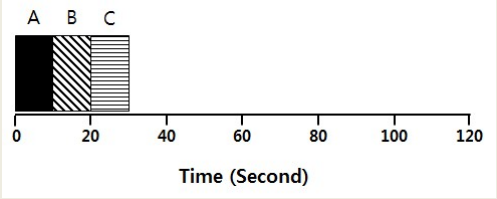
- n processes
- x_i is time share of each process
- worst case = 1/n
- best case = 1

- 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

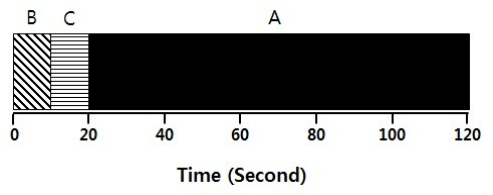
- Consider
  - 3 x 10sec jobs, arrival: A B C, duration 10 sec each



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

SJF: SHORTEST JOB FIRST

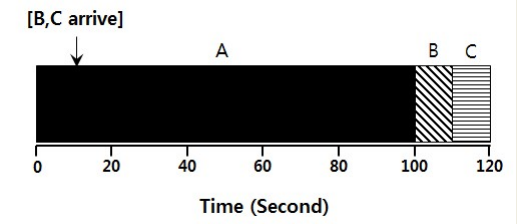
- Given that we know execution times in advance:
  - Run in order of duration, shortest to longest
  - Non preemptive scheduler
  - This is not realistic
  - Arrival: A B C, duration a=100 sec, b/c=10sec



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

SJF: WITH RANDOM ARRIVAL

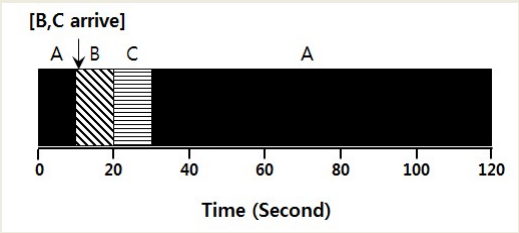
- If jobs arrive at any time: duration a=100s, b/c=10s
- A @ t=0sec, B @ t=10sec, C @ t=10sec



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

STCF - 2

- Consider: duration a=100sec, b/c=10sec
  - 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
  a mu
  timer interrupt
  period.

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

Process	Burst Time
P1	12
P5	5

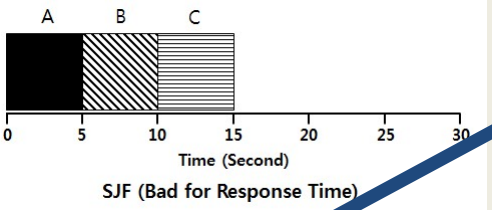
Round Robin scheduling algorithm Gantt chart

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

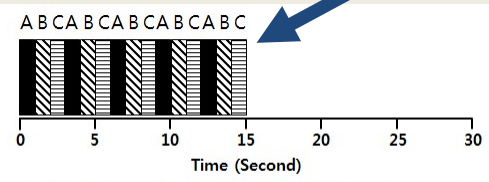
- ABC arrive at time=0, each run for 5 seconds



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

Fast Response Time

High overhead from context switching

Long Time Slice

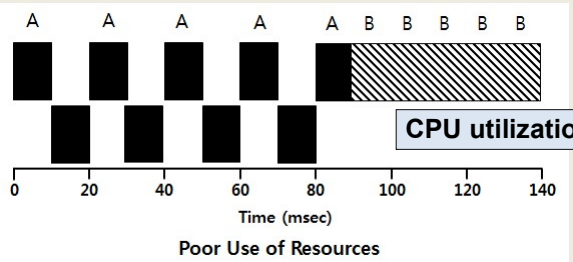
Slow Response Time

Low overhead from context switching

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

SCHEDULING WITH I/O

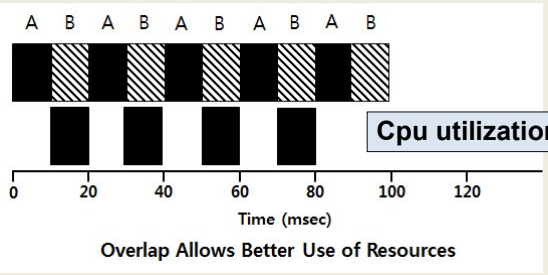
- STCF scheduler
  - A: CPU=50ms, I/O=40ms, 10ms intervals
  - B: CPU=50ms, I/O=0ms
  - Consider A as 10ms subjobs (CPU, then I/O)
- Without considering I/O:



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)



Overlap Allows Better Use of Resources

Which scheduler, thus far, best address fairness and average response time of jobs?

Respond at **PollEv.com/wesleylloyd641**

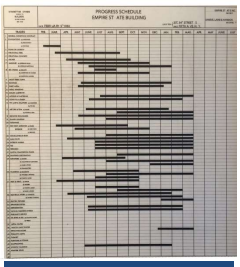
Text **WESLEYLLOYD641** to **22333** once to join, then **1, 2, 3, 4, 5...**

First In - First Out (FIFO)	1
Shortest Job First (SJF)	2
Shortest Time to Completion First (STCF)	3
Round Robin	4
None of the Above	5
All of the Above	6

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

Total Results

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



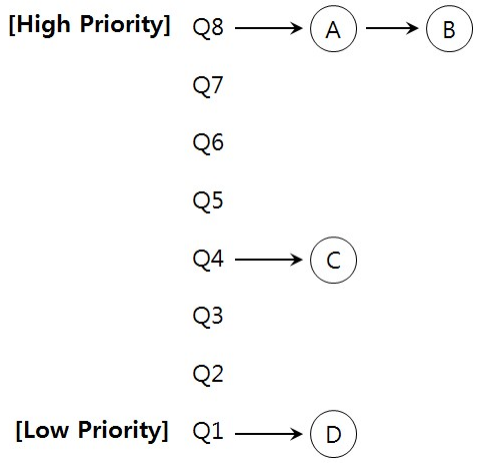
MULTI-LEVEL FEEDBACK QUEUE

- 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



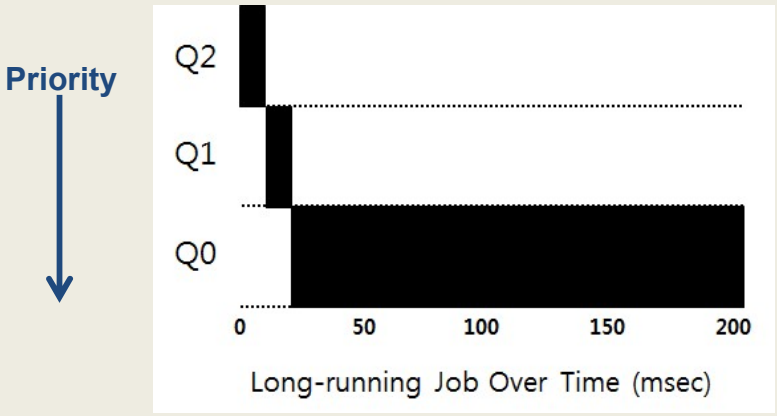
MLFQ: DETERMINING JOB PRIORITY

- 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

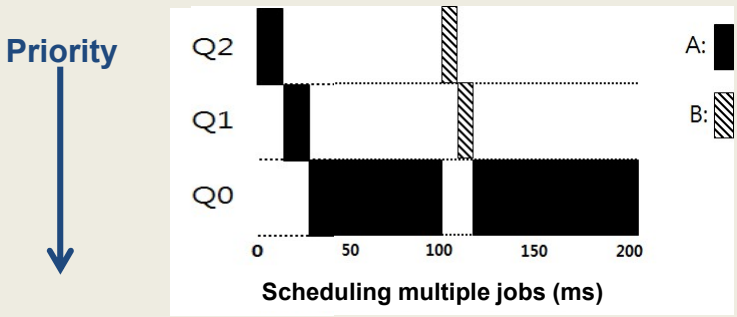
MLFQ: LONG RUNNING JOB

- Three-queue scheduler, time slice=10ms



MLFQ: BATCH AND INTERACTIVE JOBS

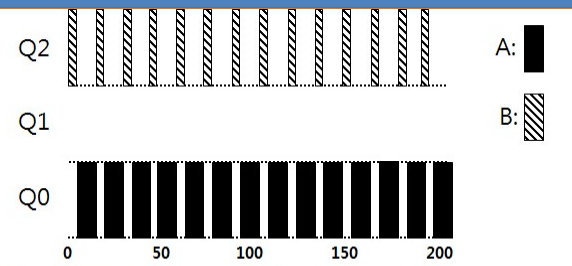
- $A_{arrival\_time}$ =0ms, A_{run_time}=200ms,
- B_{run_time} =20ms, $B_{arrival\_time}$ =100ms



MLFQ: BATCH AND INTERACTIVE - 2

- Continuous interactive job (B) with long running batch job (A)
  - Low response time is good for B
  - A continues to make progress

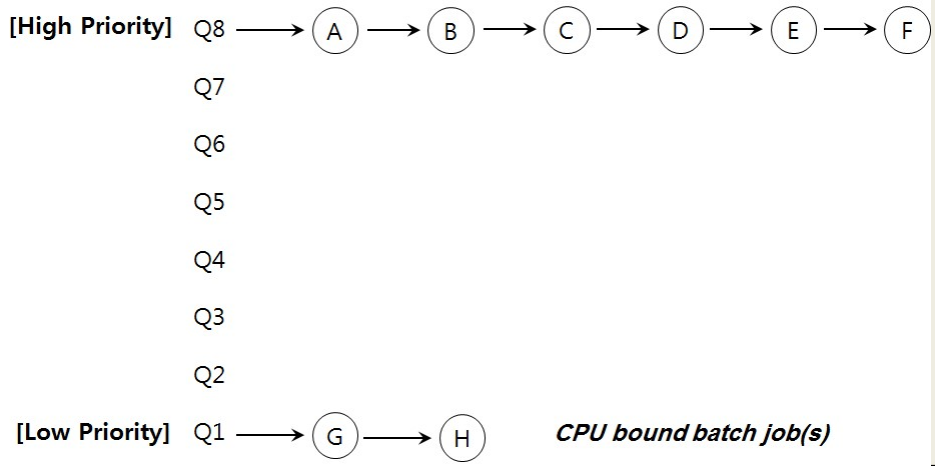
The MLFQ approach keeps interactive job(s) at the highest priority



A Mixed I/O-intensive and CPU-intensive Workload (msec)

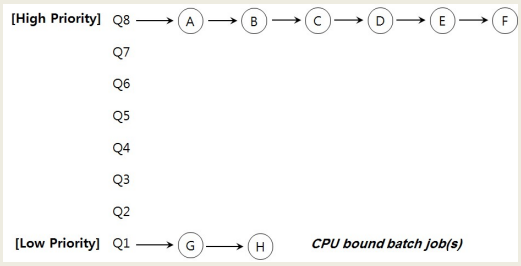
MLFQ: ISSUES

- Starvation



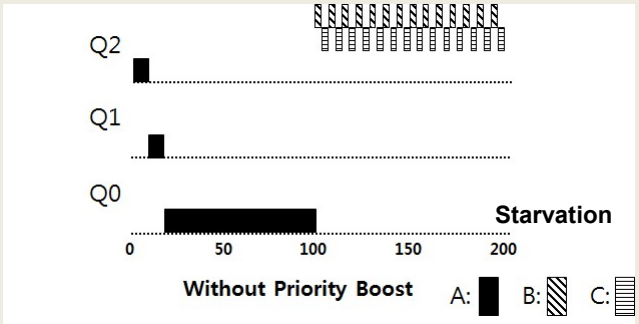
MLFQ: ISSUES - 2

- Gaming the scheduler
  - Issue I/O operation at 99% completion of the time slice
  - Keeps job priority fixed – never lowered
- Job behavioral change
  - CPU/batch process becomes an interactive process



Priority becomes stuck

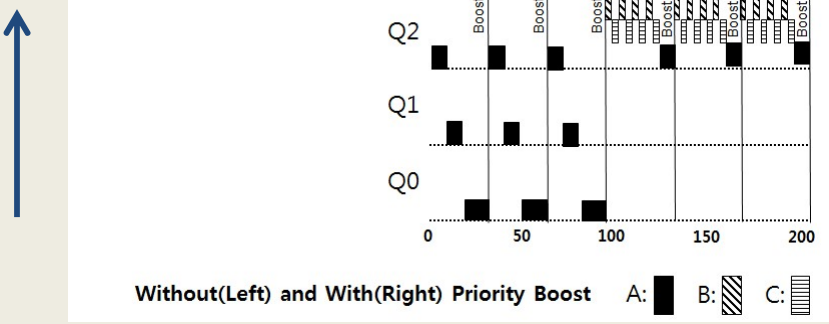
RESPONDING TO BEHAVIOR CHANGE



- Priority Boost
  - Reset all jobs to topmost queue after some time interval S

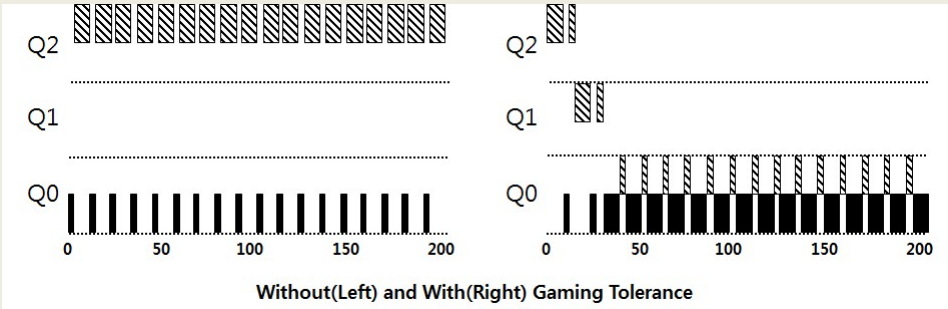
RESPONDING TO BEHAVIOR CHANGE - 2

- With priority boost
  - Prevents starvation



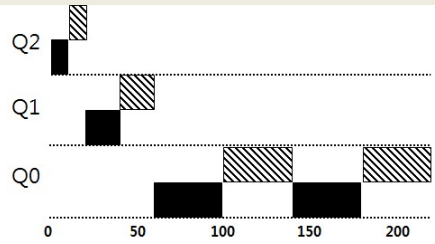
PREVENTING GAMING

- Improved time accounting:
  - Track total job execution time in the queue
  - Each job receives a fixed time allotment
  - When allotment is exhausted, job priority is lowered



MLFQ: TUNING

- Consider the tradeoffs:
  - How many queues?
  - What is a good time slice?
  - How often should we "Boost" priority of jobs?
  - What about different time slices to different queues?



Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest

PRACTICAL EXAMPLE

- Oracle Solaris MLFQ implementation
  - 60 Queues → w/ slowly increasing time slice (high to low priority)
  - Provides sys admins with set of editable table(s)
  - Supports adjusting time slices, boost intervals, priority changes, etc.
- Advice
  - Provide OS with hints about the process
  - Nice command → Linux

MLFQ RULE SUMMARY

- The refined set of MLFQ rules:
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **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 on queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Jackson deploys a 3-level MLFQ scheduler. The time slice is 1 for high priority jobs, 2 for medium priority, and 4 for low priority. This MLFQ scheduler performs a Priority Boost every 6 timer units. When the priority boost fires, the current job is preempted, and the next scheduled job is run in round-robin order.

Job	Arrival Time	Job Length
A	T=0	4
B	T=0	16
C	T=0	8

(11 points) Show a scheduling graph for the MLFQ scheduler for the jobs above.
Draw vertical lines for key events and be sure to label the X-axis times as in the example.
Please draw clearly. An unreadable graph will loose points.

QUESTIONS

COMPUTER BOOT SEQUENCE: OS WITH DIRECT EXECUTION

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

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