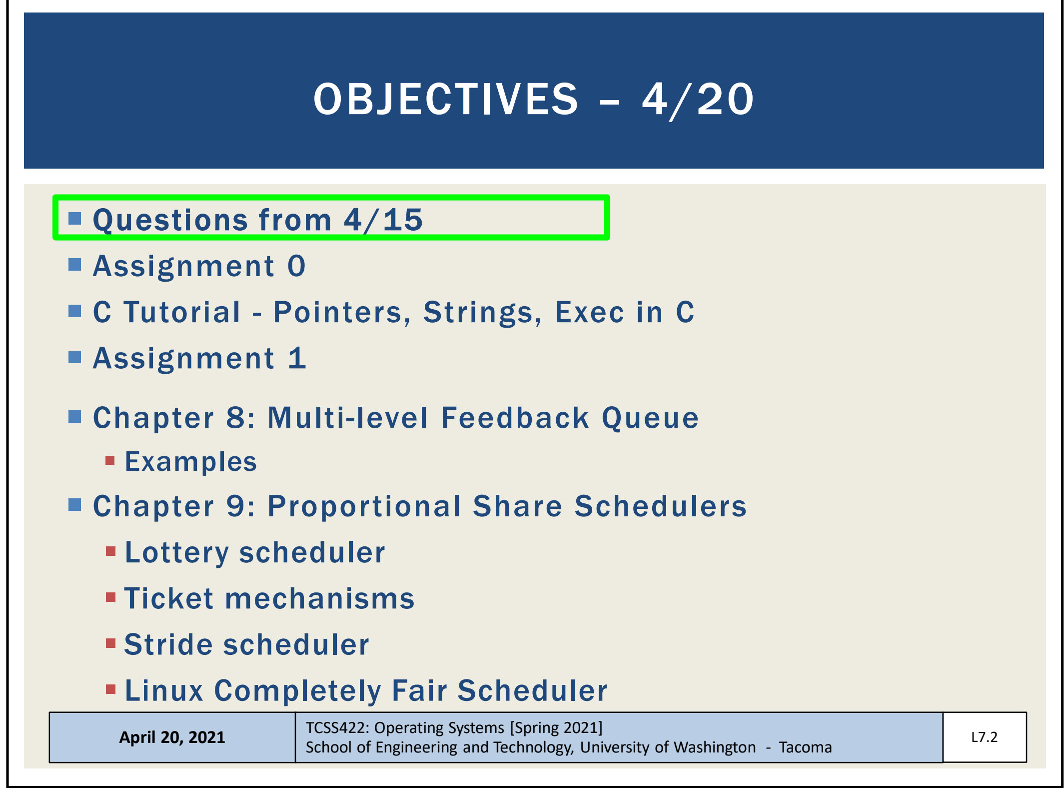
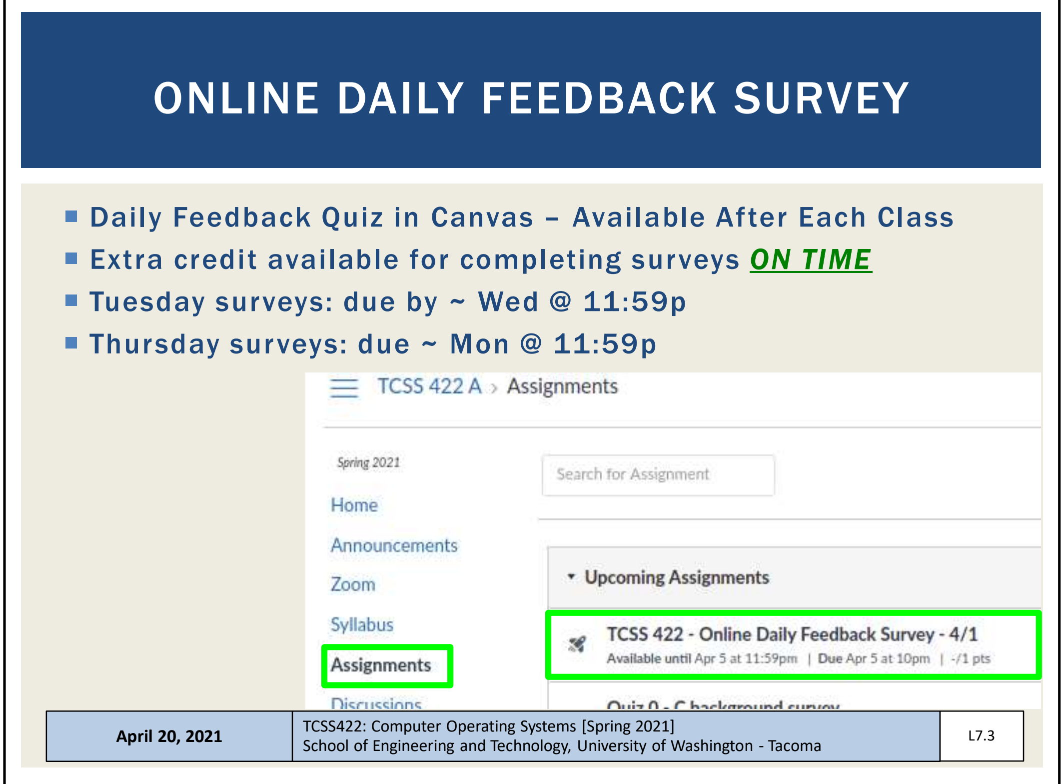
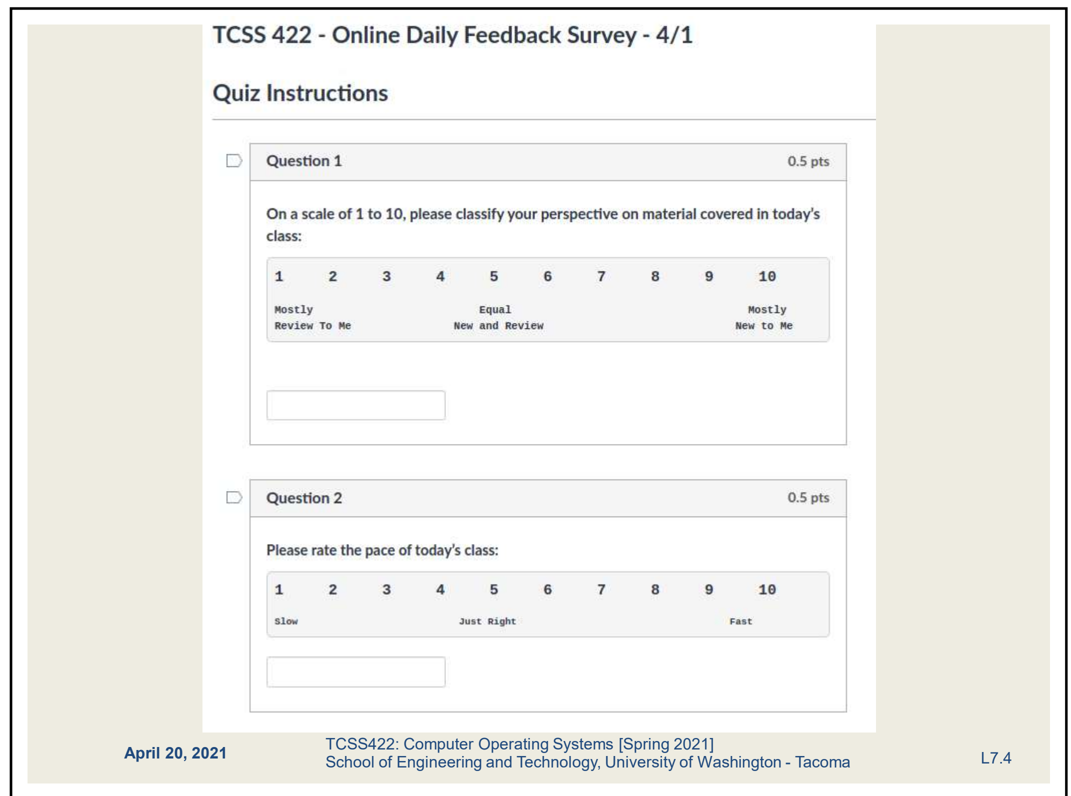
TCSS 422: OPERATING SYSTEMS
MLFQ,
Proportional Share
Schedulers
Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma


OBJECTIVES – 4/20
Questions from 4/15
Assignment 0
C Tutorial - Pointers, Strings, Exec in C
Assignment 1
Chapter 8: Multi-level Feedback Queue
Examples
Chapter 9: Proportional Share Schedulers
Lottery scheduler
Ticket mechanisms
Stride scheduler
Linux Completely Fair Scheduler
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.2


ONLINE DAILY FEEDBACK SURVEY
Daily Feedback Quiz in Canvas – Available After Each Class
Extra credit available for completing surveys ON TIME
Tuesday surveys: due by ~ Wed @ 11:59p
Thursday surveys: due ~ Mon @ 11:59p
TCSS 422 A > Assignments
Spring 2021
Search for Assignment
Home
Announcements
Zoom
Syllabus
Assignments
Upcoming Assignments
TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts
April 20, 2021
TCSS422: Computer Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.3


TCSS 422 - Online Daily Feedback Survey - 4/1
Quiz Instructions
Question 1
0.5 pts
On a scale of 1 to 10, please classify your perspective on material covered in today's class:
1 2 3 4 5 6 7 8 9 10
Mostly Review To Me
Equal New and Review
Mostly New to Me
Question 2
0.5 pts
Please rate the pace of today's class:
1 2 3 4 5 6 7 8 9 10
Slow
Just Right
Fast
April 20, 2021
TCSS422: Computer Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (58 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 7.07 (↓ - previous 7.27)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.48 (↓ - previous 5.52)

FEEDBACK

- ***What is an example of a "batch job"?***
  - Refers to "jobs that can run without end user interaction that can be scheduled to run as resources permit."
  - Run disconnected from any user interface
  - Produce output to log files, or submit results to a ddtabase

- **Examples:**
- Bulk database updates, automated transaction processing, data processing tasks/workflows:
  - Extract, transform, load (ETL) workflow is common in populating data warehouses, and is inherently a batch process
- Bulk operations on digital images: resizing, conversion, watermarking, editing/filtering
- File conversion from one format to another, convert proprietary and legacy files to standard formats: e.g. CSV, JSON, etc.
- "Batch Processing" on https://en.wikipedia.org/wiki/Batch_processing

FEEDBACK - 2

- ***How does the OS decide which queue a job should initially be in?***
  - For the MLFQ scheduler, a newly arriving job is always placed into the HIGH PRIORITY queue, which is the top-most queue
- ***Is it common practice for programs to be written to do their best to game systems for selfishly optimized execution time?***
  - This would be highly attractive to developers leveraging cloud computing platforms
  - If a cloud provider (e.g. AWS or Azure) had known scheduling vulnerabilities, then many users would be interested in gaming the system
  - Similar to cooperative operating systems, we simply can not trust the program (or programmer) to willingly share resources equitably

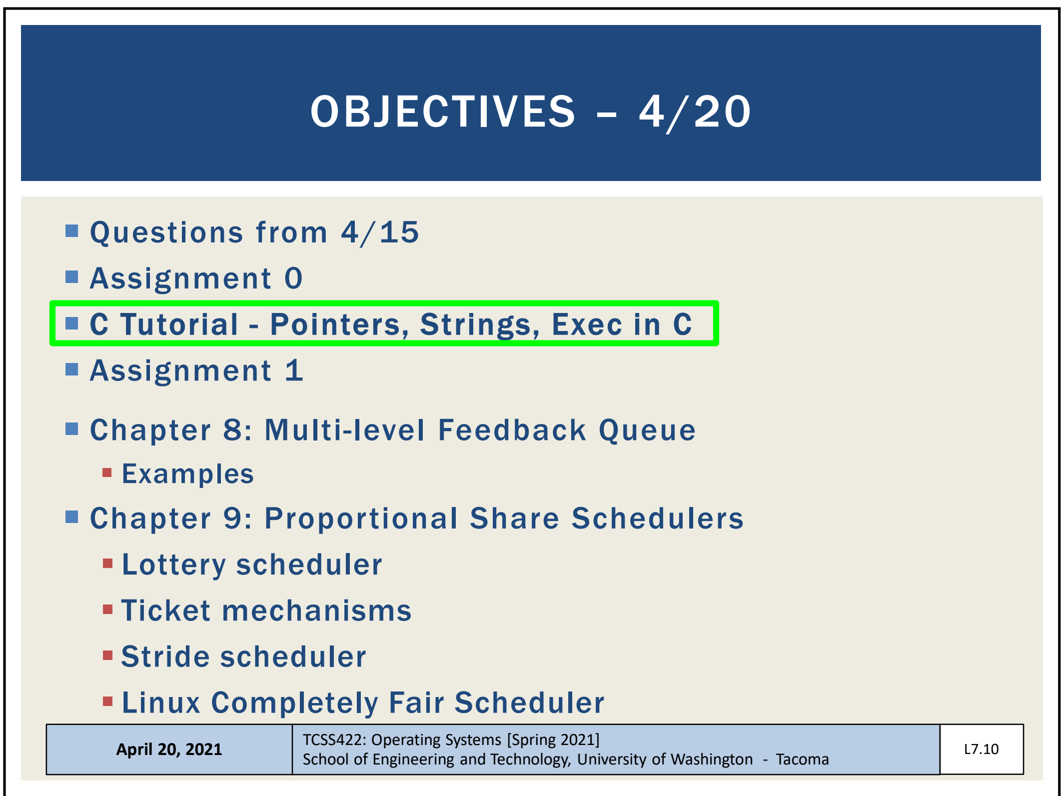
FEEDBACK - 3

- ***The priority boost was explained as, "Reset all jobs to the topmost queue after some time interval S". How long of an interval is S ? Can we change it, or adjust the interval?***
  - The interval "S" is often a multiplier of the scheduler's time slice
  - Time slice is often 10 ms
  - A common priority boost interval (S) is 5-10x the time slice (e.g. 50 to 100ms)
  - The priority boost interval change be adjusted
  - A dynamic scheduler may try to adapt the priority boost interval
  - Dynamic schedulers DO adjust the time slice of a process
- ***How are users actively being a part of the process of distributing tickets in things like lottery or stride scheduling?***
  - Covered in chapter 9…

OBJECTIVES – 4/20

- Questions from 4/15
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 8: Multi-level Feedback Queue
  - Examples
- Chapter 9: Proportional Share Schedulers
  - Lottery scheduler
  - Ticket mechanisms
  - Stride scheduler
  - Linux Completely Fair Scheduler

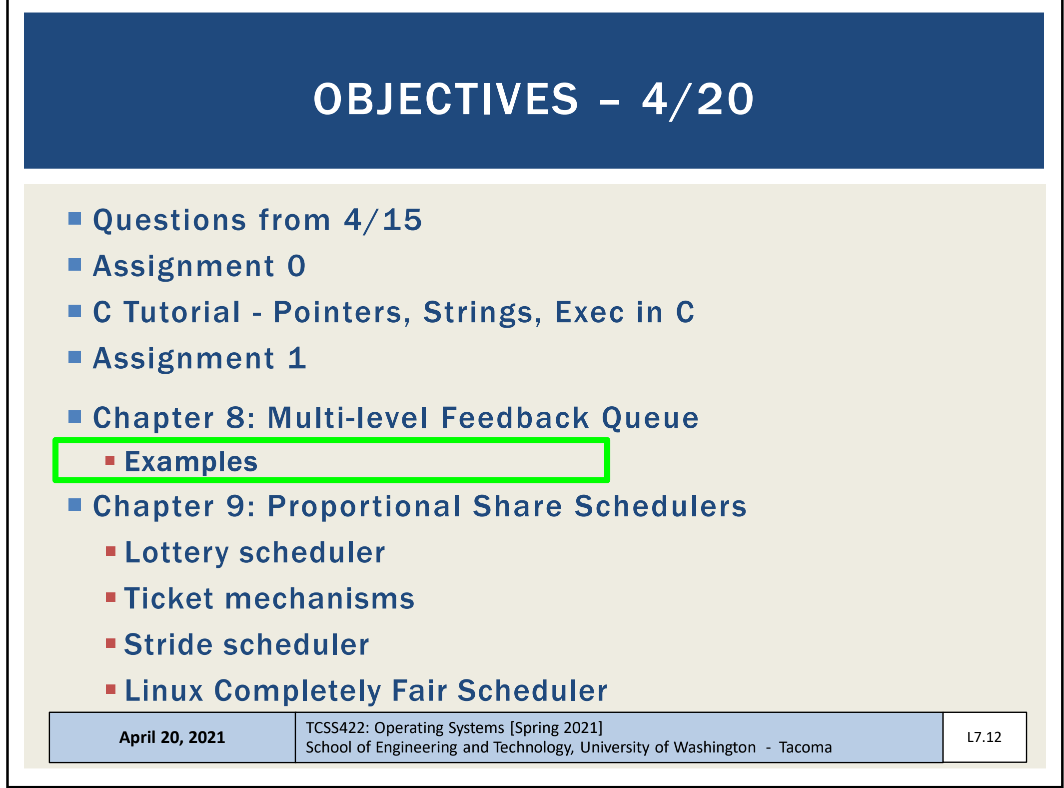
April 20, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L7.9

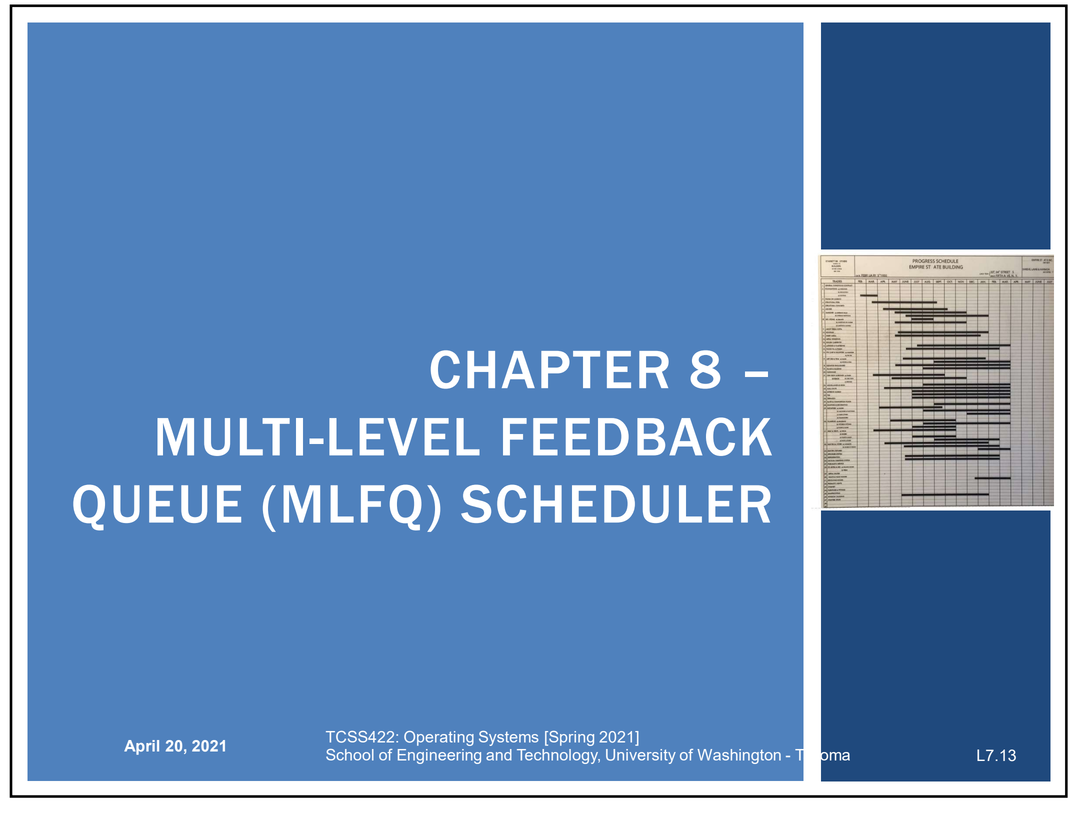


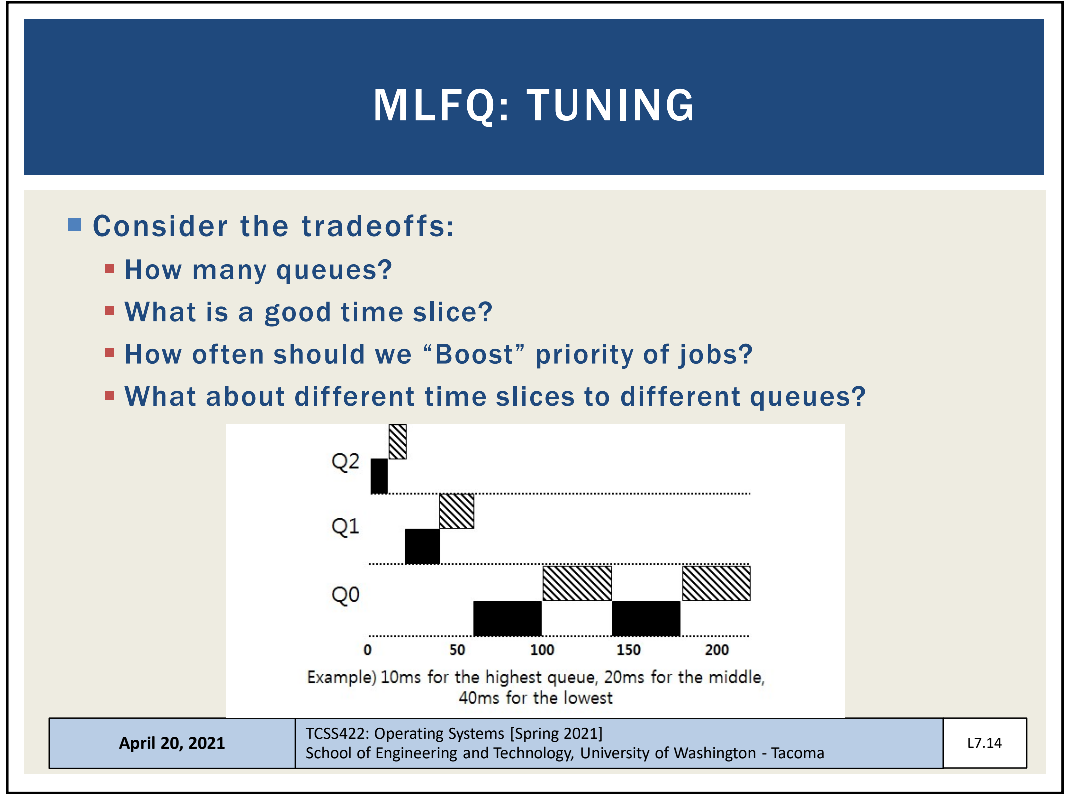
OBJECTIVES – 4/20

- Questions from 4/15
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- **Assignment 1**
- Chapter 8: Multi-level Feedback Queue
  - Examples
- Chapter 9: Proportional Share Schedulers
  - Lottery scheduler
  - Ticket mechanisms
  - Stride scheduler
  - Linux Completely Fair Scheduler

April 20, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L7.11




CHAPTER 8 –
MULTI-LEVEL FEEDBACK
QUEUE (MLFQ) SCHEDULER
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.13


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?
Q2
Q1
Q0
0
50
100
150
200
Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.14

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.

OBJECTIVES – 4/15

- Questions from 4/13
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Chapter 7: Scheduling Introduction
  - RR scheduler
- Chapter 8: Multi-level Feedback Queue
  - MLFQ Scheduler
  - Job Starvation
  - Gaming the Scheduler
  - Examples
- Chapter 9: Proportional Share Schedulers

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.

HIGH |
MED |
LOW |
0

EXAMPLE

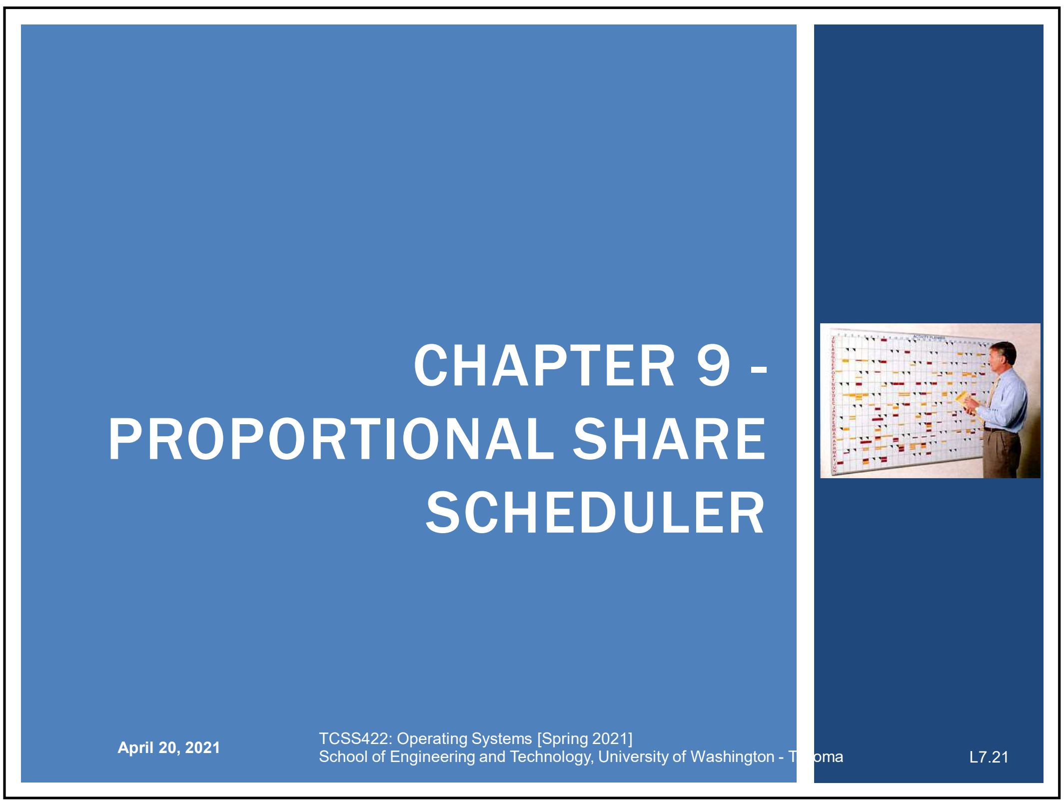
- Question:
- Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level to guarantee that a single long-running (and potentially starving) job gets at least 5% of the CPU?

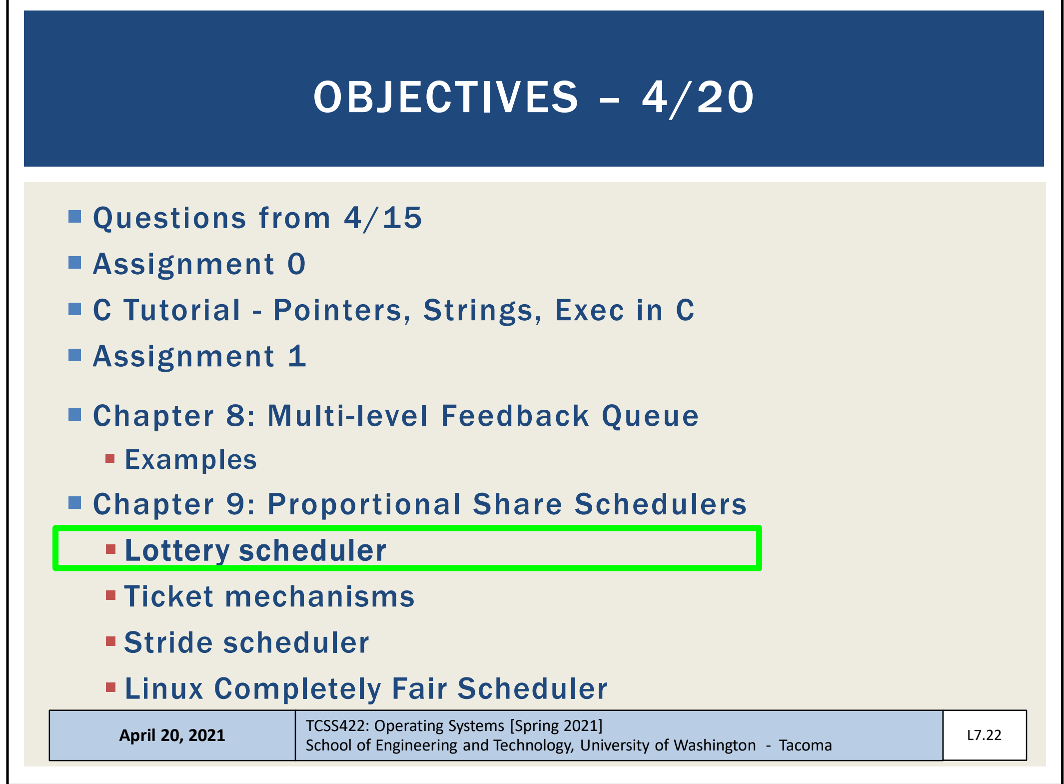
- Some combination of n short jobs runs for a total of 10 ms per cycle without relinquishing the CPU
  - E.g. 2 jobs = 5 ms ea; 3 jobs = 3.33 ms ea, 10 jobs = 1 ms ea
  - n jobs always uses full time quantum (10 ms)
  - Batch jobs starts, runs for full quantum of 10ms
  - All other jobs run and context switch totaling the quantum per cycle
  - If 10ms is 5% of the CPU, when must the priority boost be ???
  - ANSWER → *Priority boost should occur every 200ms*

OBJECTIVES – 4/20

- Questions from 4/15
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1

- Chapter 8: Multi-level Feedback Queue
  - Examples
- Chapter 9: Proportional Share Schedulers
  - Lottery scheduler
  - Ticket mechanisms
  - Stride scheduler
  - Linux Completely Fair Scheduler


CHAPTER 9 -
PROPORTIONAL SHARE
SCHEDULER
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.21

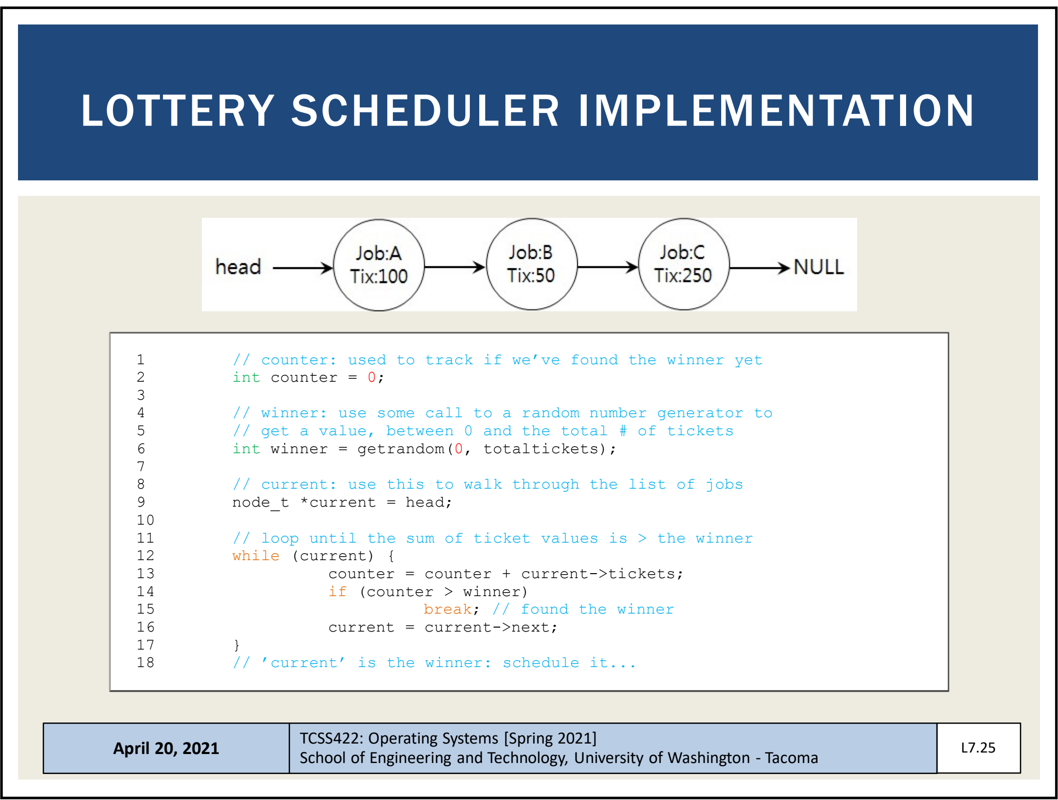

OBJECTIVES – 4/20
Questions from 4/15
Assignment 0
C Tutorial - Pointers, Strings, Exec in C
Assignment 1
Chapter 8: Multi-level Feedback Queue
Examples
Chapter 9: Proportional Share Schedulers
Lottery scheduler
Ticket mechanisms
Stride scheduler
Linux Completely Fair Scheduler
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.22

PROPORTIONAL SHARE SCHEDULER

- Also called fair-share scheduler
  or lottery scheduler
  - Guarantees each job receives some percentage of CPU time based on share of "tickets"
  - Each job receives an allotment of tickets
  - % of tickets corresponds to potential share of a resource
  - Can conceptually schedule any resource this way
    - CPU, disk I/O, memory

LOTTERY SCHEDULER

- Simple implementation
  - Just need a random number generator
    - Picks the winning ticket
  - Maintain a data structure of jobs and tickets (list)
  - Traverse list to find the owner of the ticket
  - Consider sorting the list for speed



OBJECTIVES – 4/20

- Questions from 4/15
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 8: Multi-level Feedback Queue
  - Examples
- Chapter 9: Proportional Share Schedulers
  - Lottery scheduler
  - **Ticket mechanisms**
  - Stride scheduler
  - Linux Completely Fair Scheduler

TICKET MECHANISMS

- Ticket currency / exchange
  - User allocates tickets in any desired way
  - OS converts user currency into global currency
- Example:
  - There are 200 global tickets assigned by the OS

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

TICKET MECHANISMS - 2

- Ticket transfer
  - Temporarily hand off tickets to another process
- Ticket inflation
  - Process can temporarily raise or lower the number of tickets it owns
  - If a process needs more CPU time, it can boost tickets.

LOTTERY SCHEDULING

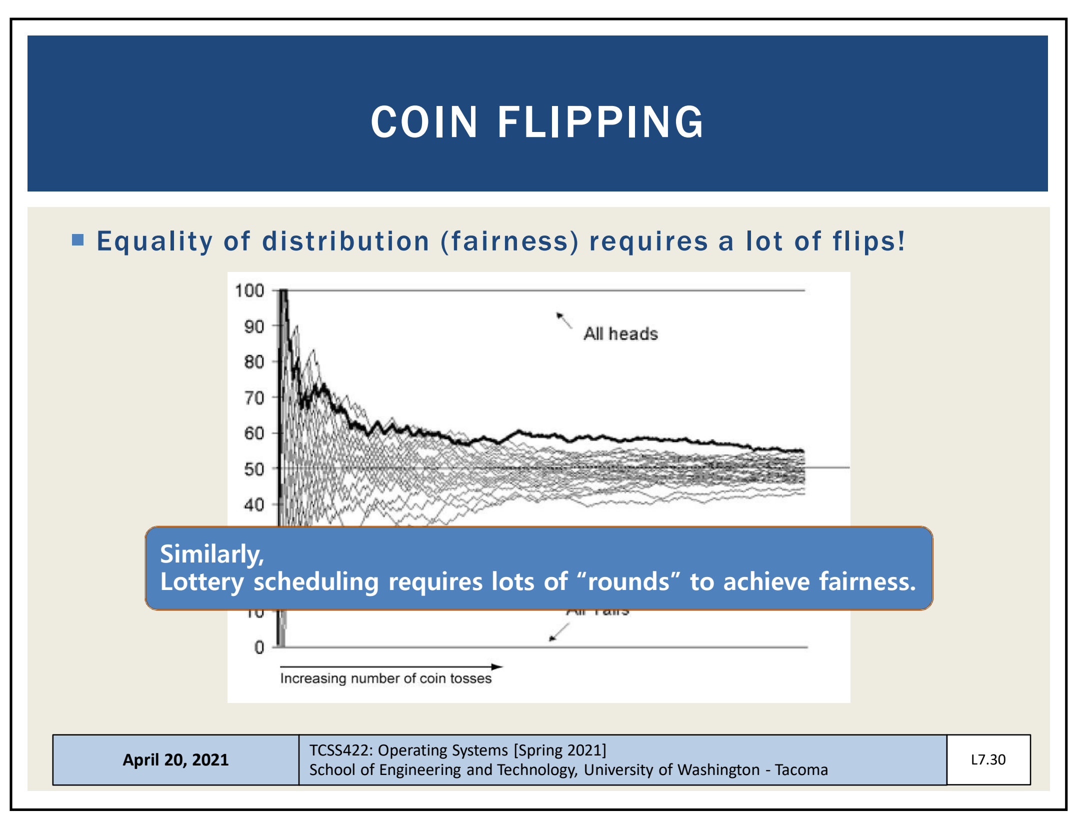
- Scheduler picks a winning ticket
  - Load the job with the winning ticket and run it
- Example:
  - Given 100 tickets in the pool
    - Job A has 75 tickets: 0 - 74
    - Job B has 25 tickets: 75 – 99

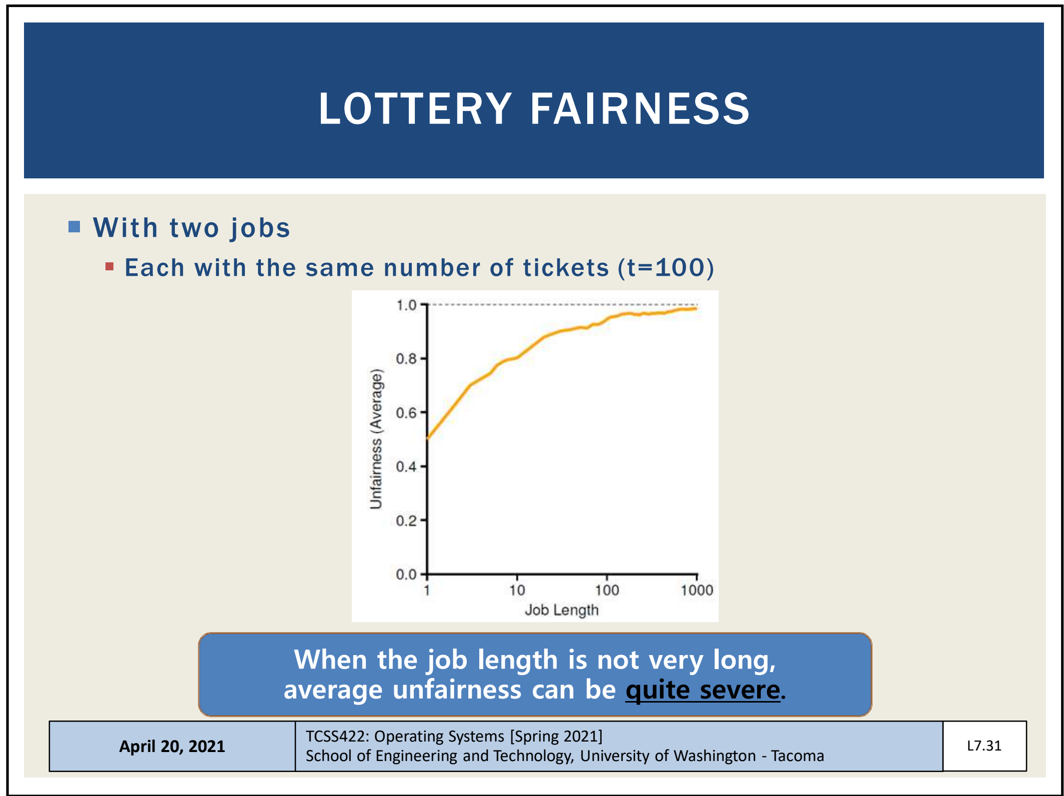
Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Scheduled job:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

- But what do we know about probability of a coin flip?

COIN FLIPPING

- Equality of distribution (fairness) requires a lot of flips!

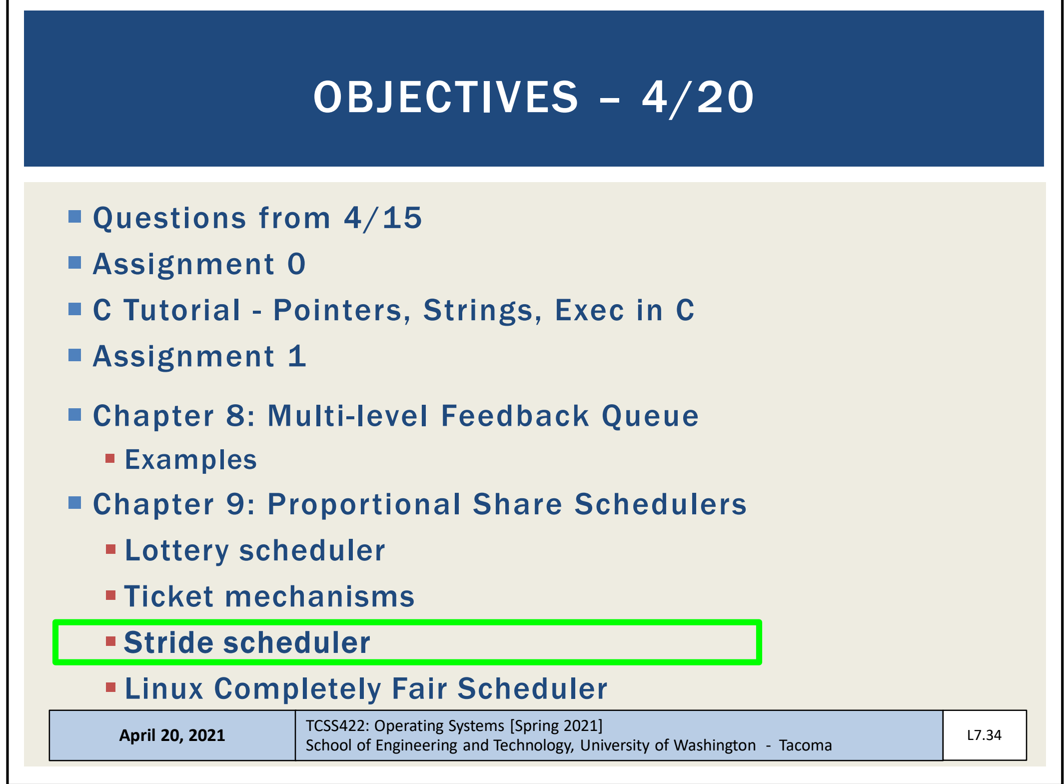




LOTTERY SCHEDULING CHALLENGES

- What is the best approach to assign tickets to jobs?
  - Typical approach is to assume users know best
  - Users are provided with tickets, which they allocate as desired
- How should the OS automatically distribute tickets upon job arrival?
  - What do we know about incoming jobs a priori ?
  - Ticket assignment is really an open problem…

WE WILL RETURN AT 5:10PM
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - coma
L7.33


OBJECTIVES – 4/20
Questions from 4/15
Assignment 0
C Tutorial - Pointers, Strings, Exec in C
Assignment 1
Chapter 8: Multi-level Feedback Queue
Examples
Chapter 9: Proportional Share Schedulers
Lottery scheduler
Ticket mechanisms
Stride scheduler
Linux Completely Fair Scheduler
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.34

STRIDE SCHEDULER

- Addresses statistical probability issues with lottery scheduling

- Instead of guessing a random number to select a job, simply count…

STRIDE SCHEDULER - 2

- Jobs have a "stride" value
  - A stride value describes the counter pace when the job should give up the CPU
  - Stride value is inverse in proportion to the job's number of tickets (more tickets = smaller stride)

- Total system tickets = 10,000
  - Job A has 100 tickets → A_{stride} = 10000/100 = 100 stride
  - Job B has 50 tickets → B_{stride} = 10000/50 = 200 stride
  - Job C has 250 tickets → C_{stride} = 10000/250 = 40 stride

- Stride scheduler tracks "pass" values for each job (A, B, C)

STRIDE SCHEDULER - 3

- Basic algorithm:
  1. Stride scheduler picks job with the lowest pass value
  2. Scheduler increments job's pass value by its stride and starts running
  3. Stride scheduler increments a counter
  4. When counter exceeds pass value of current job, pick a new job (go to 1)

- KEY: When the counter reaches a job's "PASS" value, the scheduler passes on to the next job…

STRIDE SCHEDULER - EXAMPLE

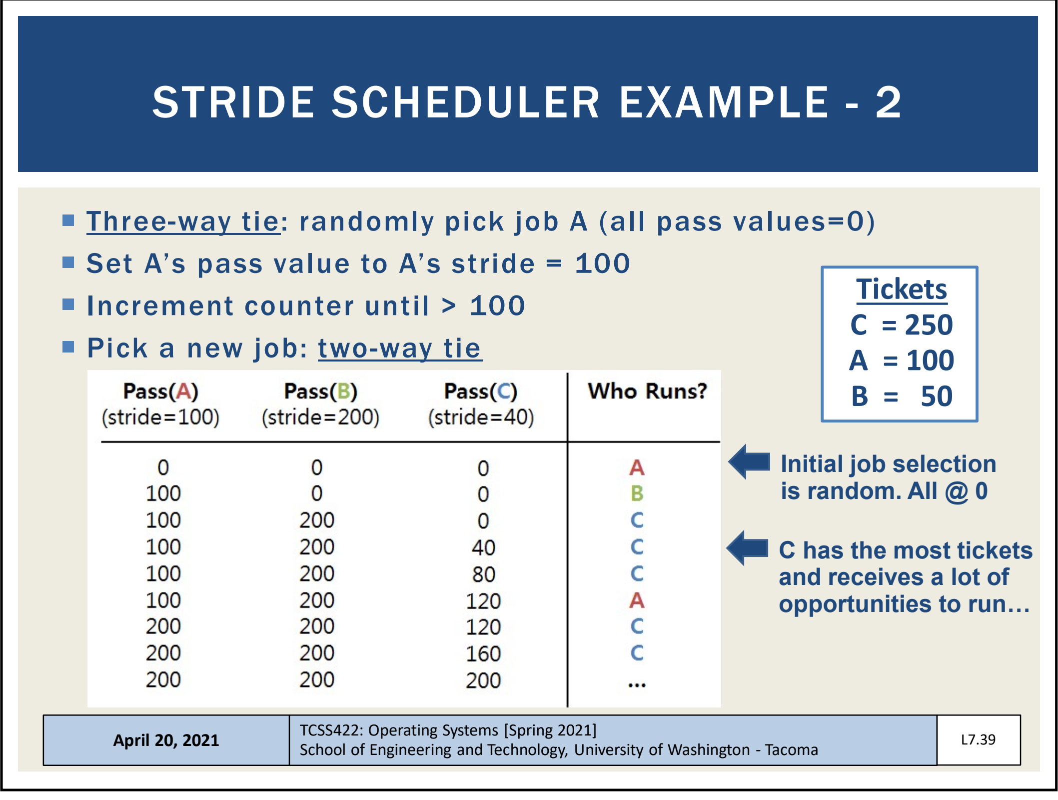
- Stride values
  - Tickets = priority to select job
  - Stride is inverse to tickets
  - Lower stride = more chances to run (higher priority)

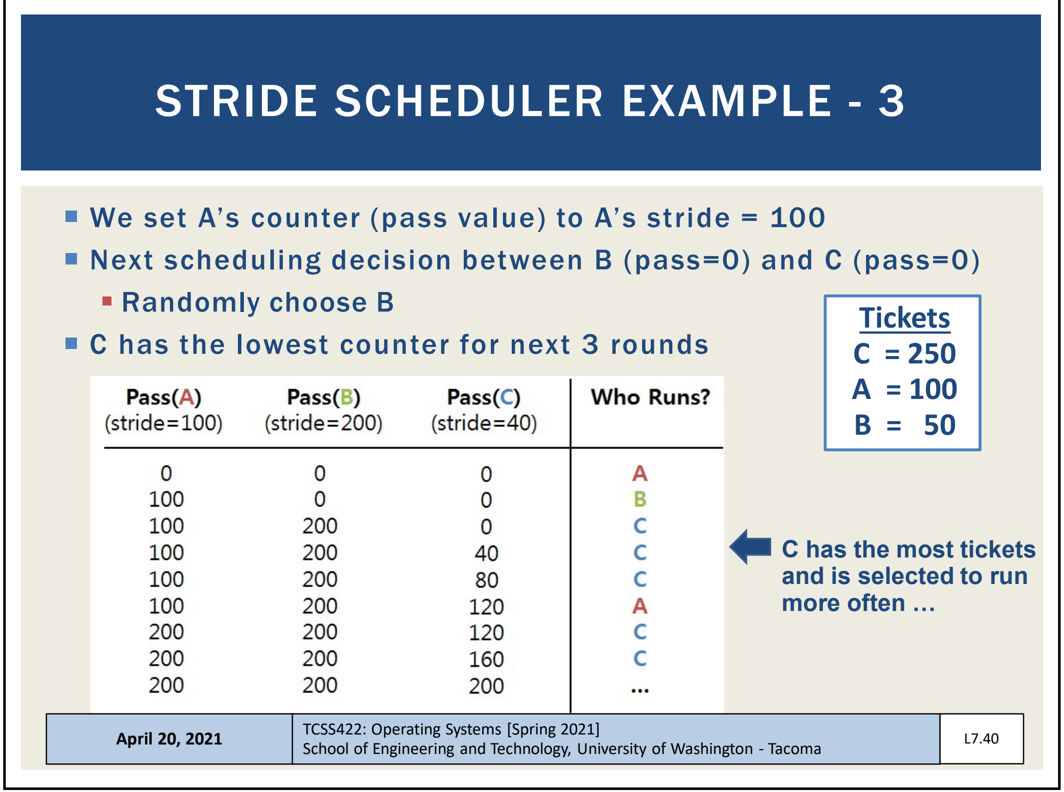
Priority

C stride = 40

A stride = 100

B stride = 200


STRIDE SCHEDULER EXAMPLE - 2
Three-way tie: randomly pick job A (all pass values=0)
Set A's pass value to A's stride = 100
Increment counter until > 100
Pick a new job: two-way tie
Tickets
C = 250
A = 100
B = 50
Pass(A) (stride=100) | Pass(B) (stride=200) | Pass(C) (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 …
Initial job selection is random. All @ 0
C has the most tickets and receives a lot of opportunities to run…
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.39


STRIDE SCHEDULER EXAMPLE - 3
We set A's counter (pass value) to A's stride = 100
Next scheduling decision between B (pass=0) and C (pass=0)
Randomly choose B
C has the lowest counter for next 3 rounds
Tickets
C = 250
A = 100
B = 50
Pass(A) (stride=100) | Pass(B) (stride=200) | Pass(C) (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 …
C has the most tickets and is selected to run more often …
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.40

STRIDE SCHEDULER EXAMPLE - 4

- Job counters support determining which job to run next
- Over time jobs are scheduled to run based on their priority represented as their share of tickets…
- Tickets are analogous to job priority

Tickets
C = 250
A = 100
B = 50

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (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	…

OBJECTIVES – 4/20

- Questions from 4/15
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 8: Multi-level Feedback Queue
  - Examples
- Chapter 9: Proportional Share Schedulers
  - Lottery scheduler
  - Ticket mechanisms
  - Stride scheduler
  - Linux Completely Fair Scheduler

LINUX: COMPLETELY FAIR SCHEDULER (CFS)

- Large Google datacenter study:
  "Profiling a Warehouse-scale Computer" (Kanev et al.)
- Monitored 20,000 servers over 3 years
- Found 20% of CPU time spent in the Linux kernel
- 5% of CPU time spent in the CPU scheduler!
- Study highlights importance for high performance OS kernels and CPU schedulers !

Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

See: https://dl.acm.org/doi/pdf/10.1145/2749469.2750392

LINUX: COMPLETELY FAIR SCHEDULER (CFS)

- Loosely based on the stride scheduler
- CFS models system as a Perfect Multi-Tasking System
  - In perfect system every process of the same priority (class) receive exactly $1/n^{th}$ of the CPU time
- Each scheduling class has a runqueue
  - Groups process of same class
  - In class, scheduler picks task w/ lowest `vruntime` to run
  - Time slice varies based on how many jobs in shared runqueue
  - Minimum time slice prevents too many context switches (e.g. 3 ms)

COMPLETELY FAIR SCHEDULER - 2

- Every thread/process has a scheduling class (policy):
- Normal classes: SCHED_OTHER (TS), SCHED_IDLE, SCHED_BATCH
  - TS = Time Sharing
- Real-time classes: SCHED_FIFO (FF), SCHED_RR (RR)

- How to show scheduling class and priority:
- `#class`
  `ps -elfc`

- `#priority (nice value)`
  `ps ax -o pid,ni,cls,pri,cmd`

COMPLETELY FAIR SCHEDULER - 3

- Linux ≥ 2.6.23: Completely Fair Scheduler (CFS)
- Linux < 2.6.23: O(1) scheduler

- Linux maintains simple counter (**vruntime**) to track how long each thread/process has run
- CFS picks process with lowest **vruntime** to run next

- CFS adjusts timeslice based on # of proc waiting for the CPU
- Kernel parameters that specify CFS behavior:

```
$ sudo sysctl kernel.sched_latency_ns
kernel.sched_latency_ns = 24000000
$ sudo sysctl kernel.sched_min_granularity_ns
kernel.sched_min_granularity_ns = 3000000
$ sudo sysctl kernel.sched_wakeup_granularity_ns
kernel.sched_wakeup_granularity_ns = 4000000
```

COMPLETELY FAIR SCHEDULER - 4

- Sched_min_granularity_ns (3ms)
  - Time slice for a process: busy system (w/ full runqueue)
  - If system has idle capacity, time slice exceed the min as long as difference in vruntime between running process and process with lowest vruntime is less than sched_wakeup_granularity_ns (4ms)
- Scheduling time period is: total cycle time for iterating through a set of processes where each is allowed to run (like round robin)
- Example:
  sched_latency_ns (24ms)
  if (proc in runqueue < sched_latency_ns/sched_min_granularity)
  or
  sched_min_granularity * number of processes in runqueue

Ref: https://www.systutorials.com/sched_min_granularity_ns-sched_latency_ns-cfs-affect-timeslice-processes/

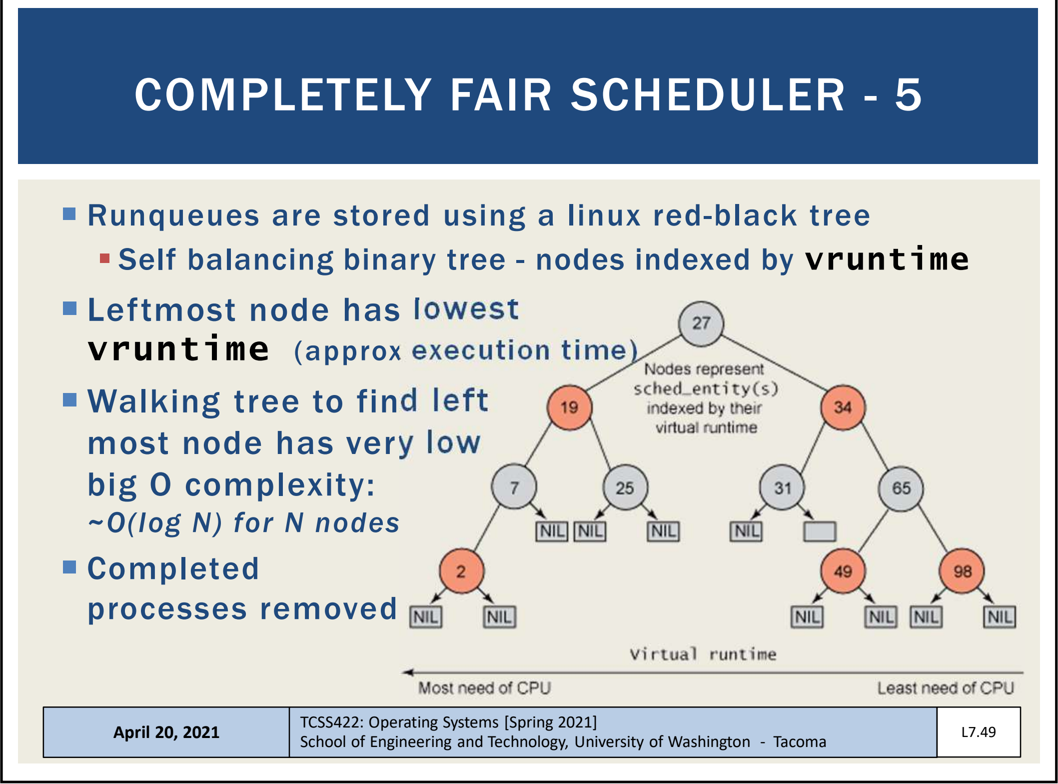
CFS TRADEOFF

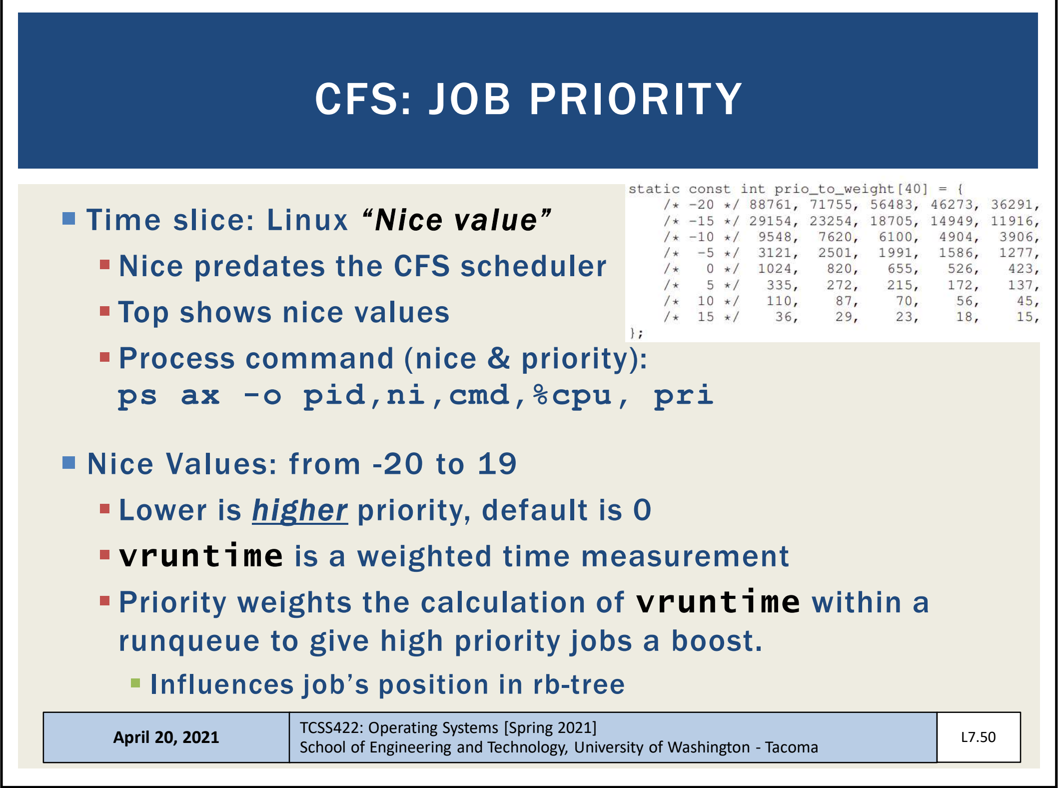
- HIGH sched_min_granularity_ns (timeslice)
  sched_latency_ns
  sched_wakeup_granularity_ns

  reduced context switching → less overhead
  poor near-term fairness

- LOW sched_min_granularity_ns (timeslice)
  sched_latency_ns
  sched_wakreup_granularity_ns

  increased context switching → more overhead
  better near-term fairness


COMPLETELY FAIR SCHEDULER - 5
Runqueues are stored using a linux red-black tree
Self balancing binary tree - nodes indexed by vruntime
Leftmost node has lowest vruntime (approx execution time)
Walking tree to find left most node has very low big O complexity: ~O(log N) for N nodes
Completed processes removed
27
19
34
7
25
31
65
2
49
98
NIL
Nodes represent sched_entity(s) indexed by their virtual runtime
Virtual runtime
Most need of CPU
Least need of CPU
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.49

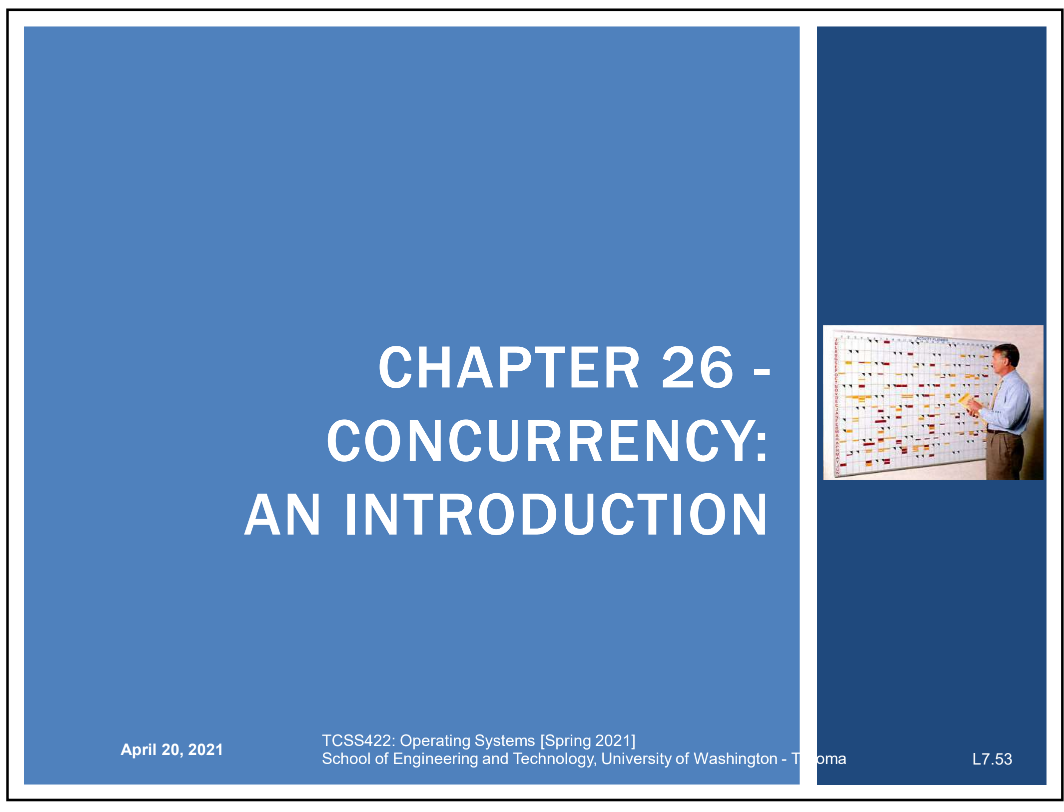

CFS: JOB PRIORITY
Time slice: Linux "Nice value"
Nice predates the CFS scheduler
Top shows nice values
Process command (nice & priority):
ps ax -o pid,ni,cmd,%cpu, pri
static const int prio_to_weight[40] = {
/* -20 */ 88761, 71755, 56483, 46273, 36291,
/* -15 */ 29154, 23254, 18705, 14949, 11916,
/* -10 */ 9548, 7620, 6100, 4904, 3906,
/* -5 */ 3121, 2501, 1991, 1586, 1277,
/* 0 */ 1024, 820, 655, 526, 423,
/* 5 */ 335, 272, 215, 172, 137,
/* 10 */ 110, 87, 70, 56, 45,
/* 15 */ 36, 29, 23, 18, 15,
};
Nice Values: from -20 to 19
Lower is higher priority, default is 0
vruntime is a weighted time measurement
Priority weights the calculation of vruntime within a runqueue to give high priority jobs a boost.
Influences job's position in rb-tree
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.50

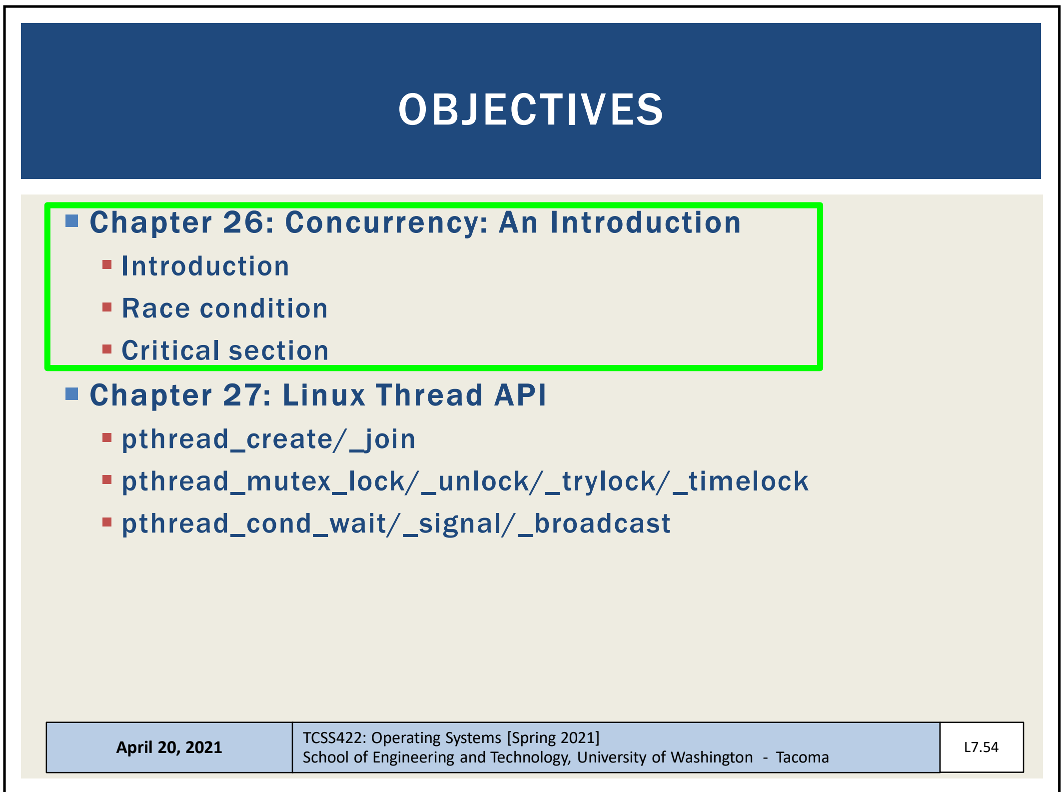
COMPLETELY FAIR SCHEDULER - 6

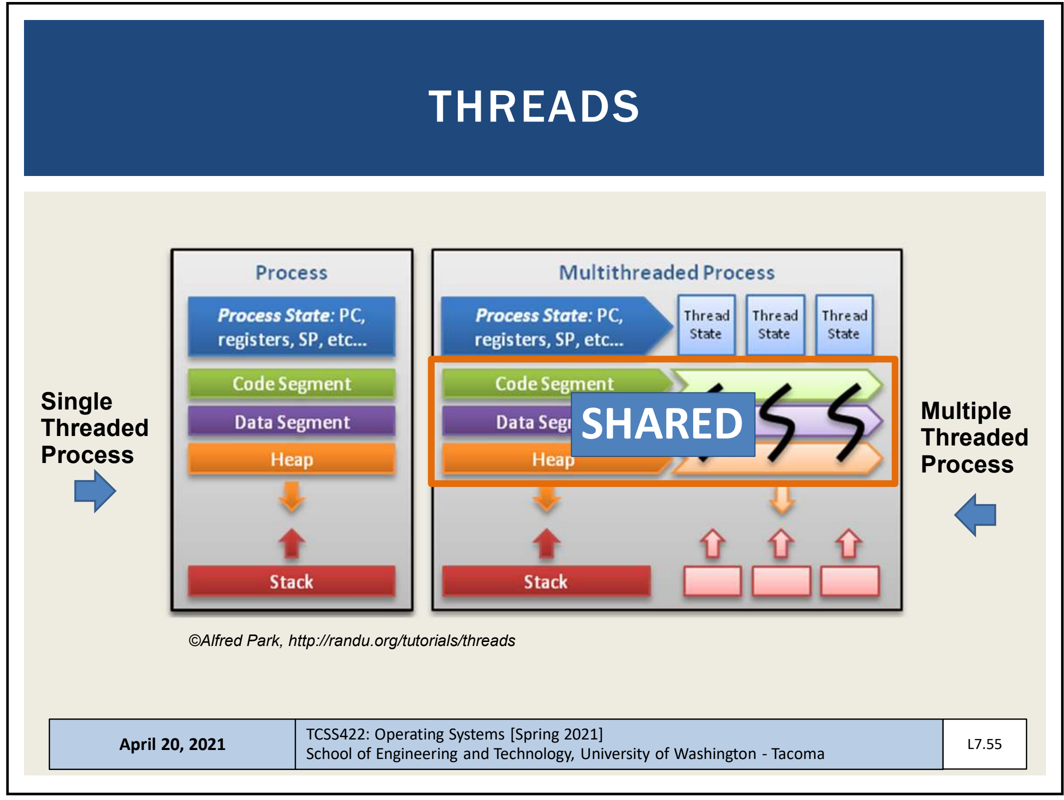
- CFS tracks cumulative job run time in **vruntime** variable
- The task on a given runqueue with the lowest **vruntime** is scheduled next
- **struct sched_entity** contains **vruntime** parameter
  - Describes process execution time in nanoseconds
  - Value is not pure runtime, is weighted based on job priority
  - Perfect scheduler → achieve equal **vruntime** for all processes of same priority
- Sleeping jobs: upon return reset **vruntime** to lowest value in system
  - Jobs with frequent short sleep ***SUFFER !!***
- Key takeaway:
  identifying the next job to schedule is really fast!

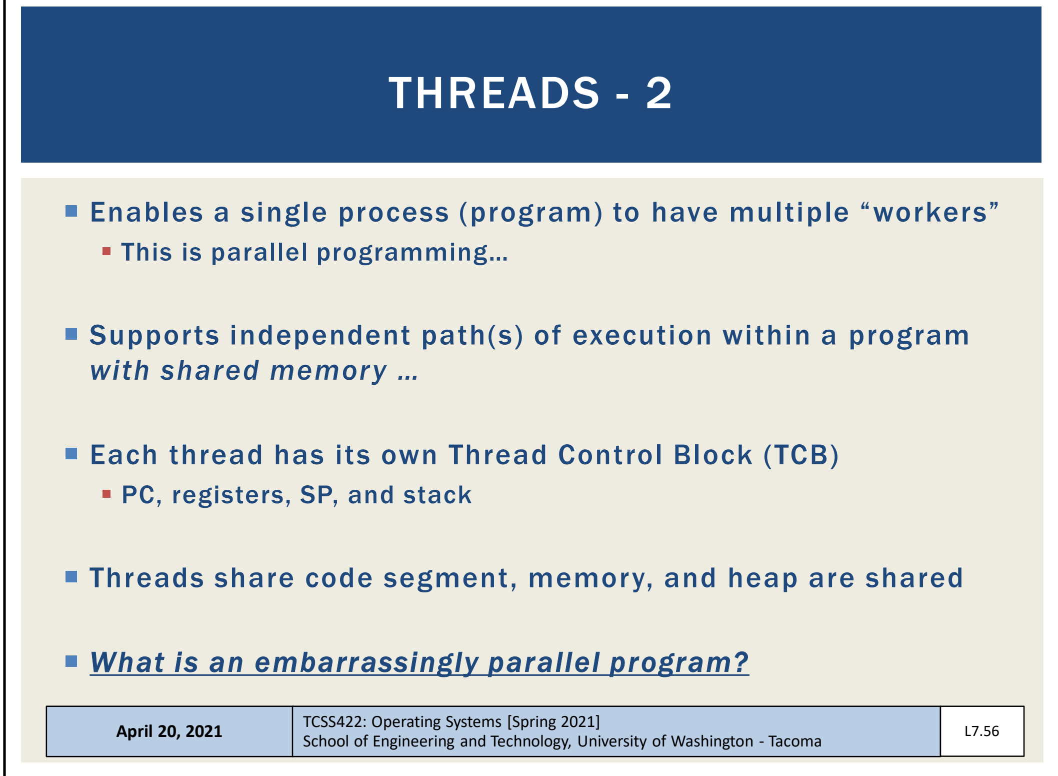
COMPLETELY FAIR SCHEDULER - 7

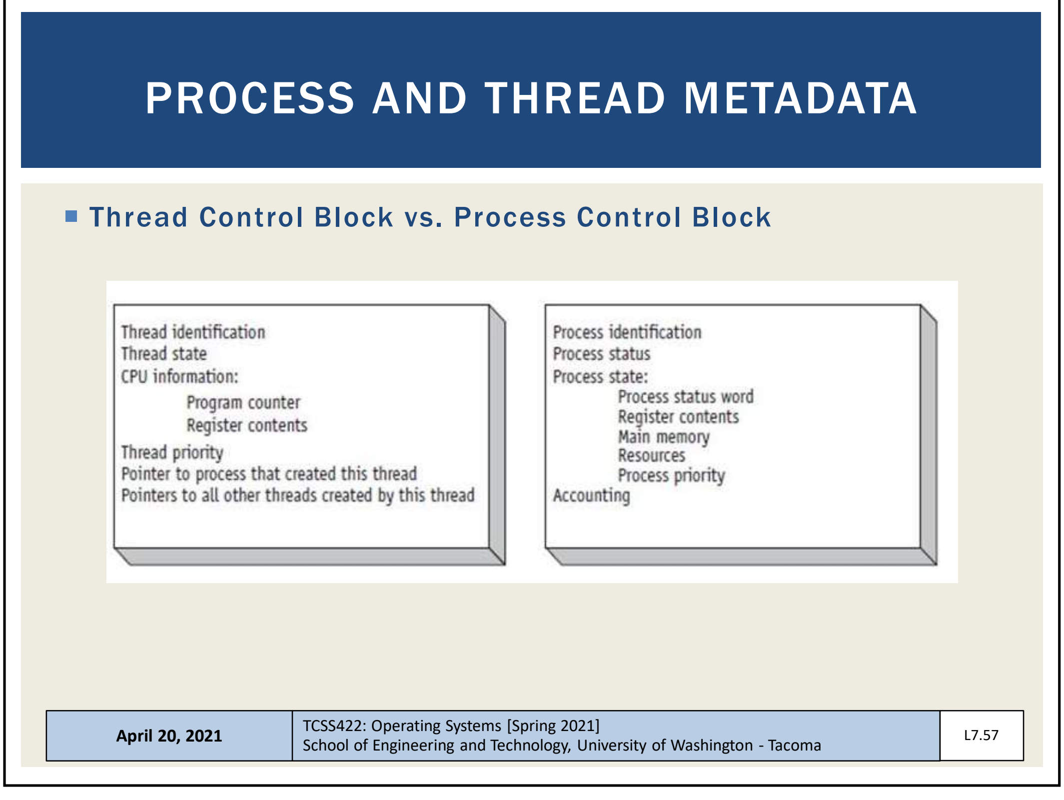
- More information:
- Man page: "man sched" : Describes Linux scheduling API
- http://manpages.ubuntu.com/manpages/bionic/man7/sched.7.html
- https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
- https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- See paper: The Linux Scheduler – a Decade of Wasted Cores
- http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

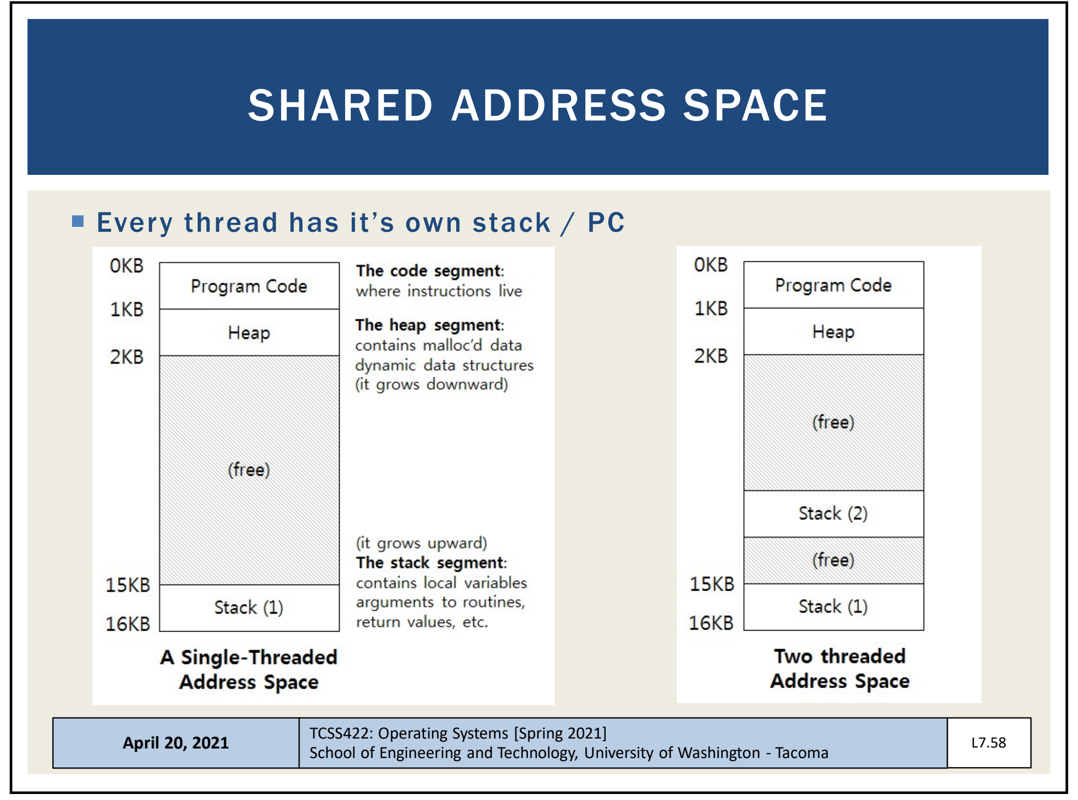

CHAPTER 26 -
CONCURRENCY:
AN INTRODUCTION
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.53

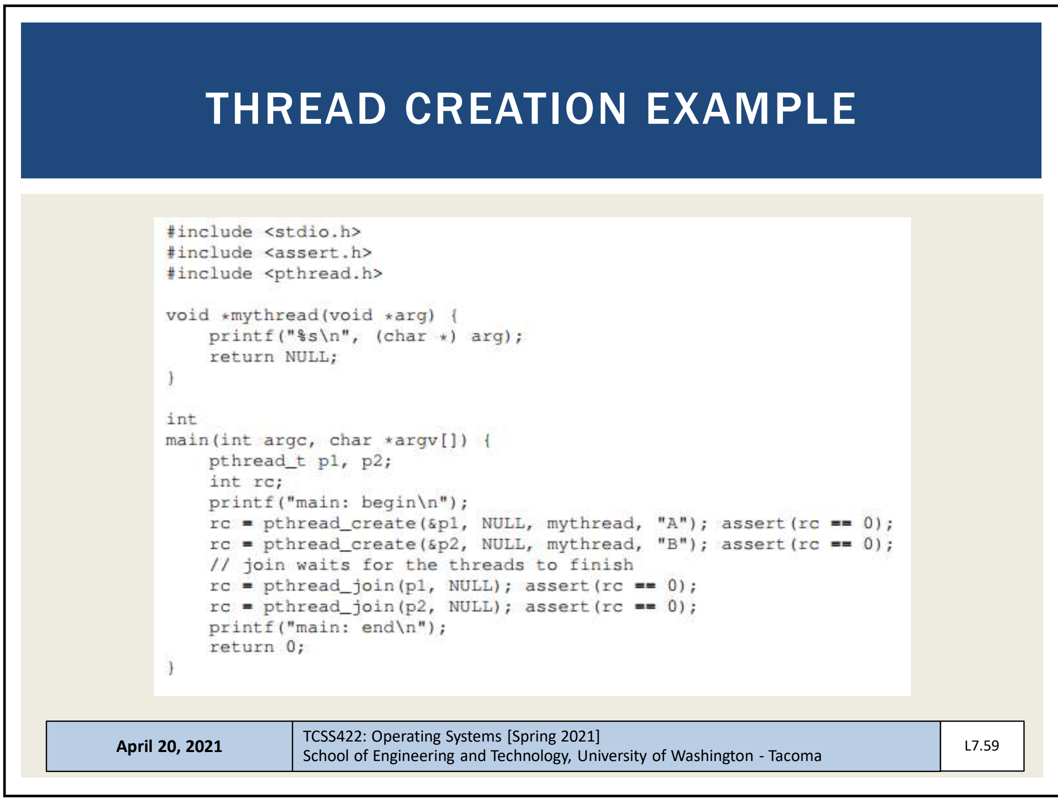

OBJECTIVES
Chapter 26: Concurrency: An Introduction
Introduction
Race condition
Critical section
Chapter 27: Linux Thread API
pthread_create/_join
pthread_mutex_lock/_unlock/_trylock/_timelock
pthread_cond_wait/_signal/_broadcast
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.54


THREADS
Single Threaded Process
Process
Process State: PC, registers, SP, etc...
Code Segment
Data Segment
Heap
Stack
Multithreaded Process
Process State: PC, registers, SP, etc...
Thread State
Thread State
Thread State
Code Segment
Data Segment
Heap
SHARED
Stack
Multiple Threaded Process
©Alfred Park, http://randu.org/tutorials/threads
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.55

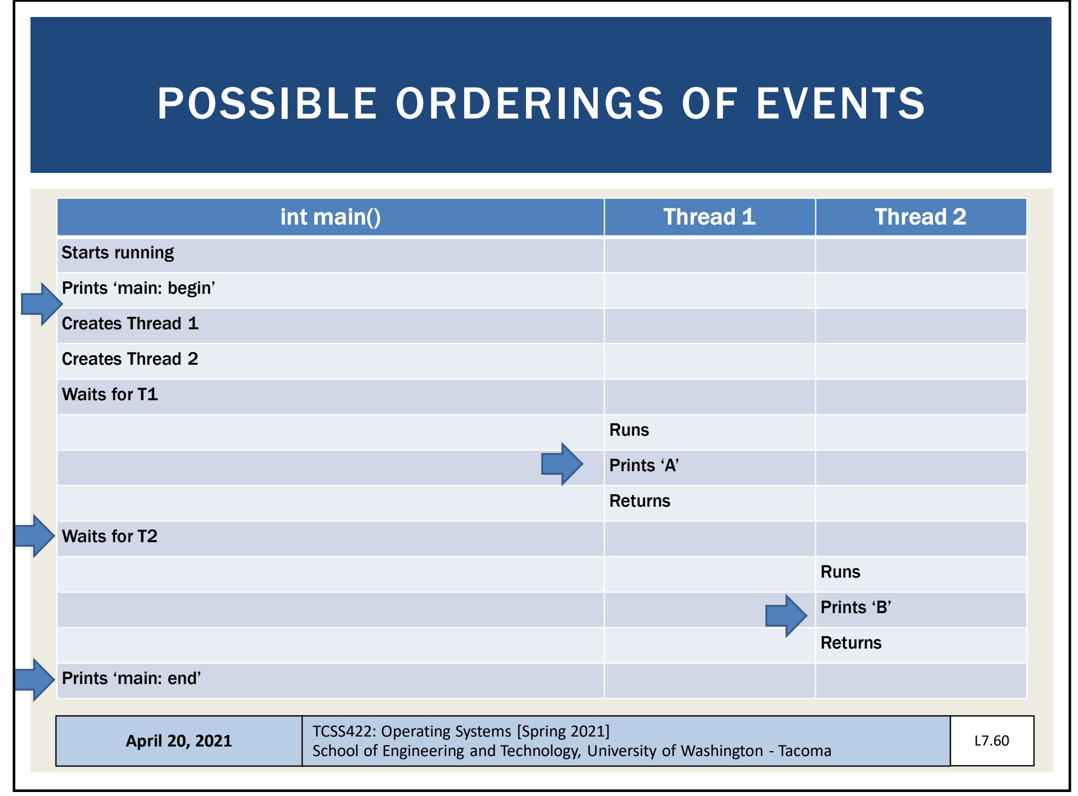

THREADS - 2
Enables a single process (program) to have multiple "workers"
This is parallel programming…
Supports independent path(s) of execution within a program with shared memory …
Each thread has its own Thread Control Block (TCB)
PC, registers, SP, and stack
Threads share code segment, memory, and heap are shared
What is an embarrassingly parallel program?
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.56


PROCESS AND THREAD METADATA
Thread Control Block vs. Process Control Block
Thread identification
Thread state
CPU information:
Program counter
Register contents
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread
Process identification
Process status
Process state:
Process status word
Register contents
Main memory
Resources
Process priority
Accounting
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.57

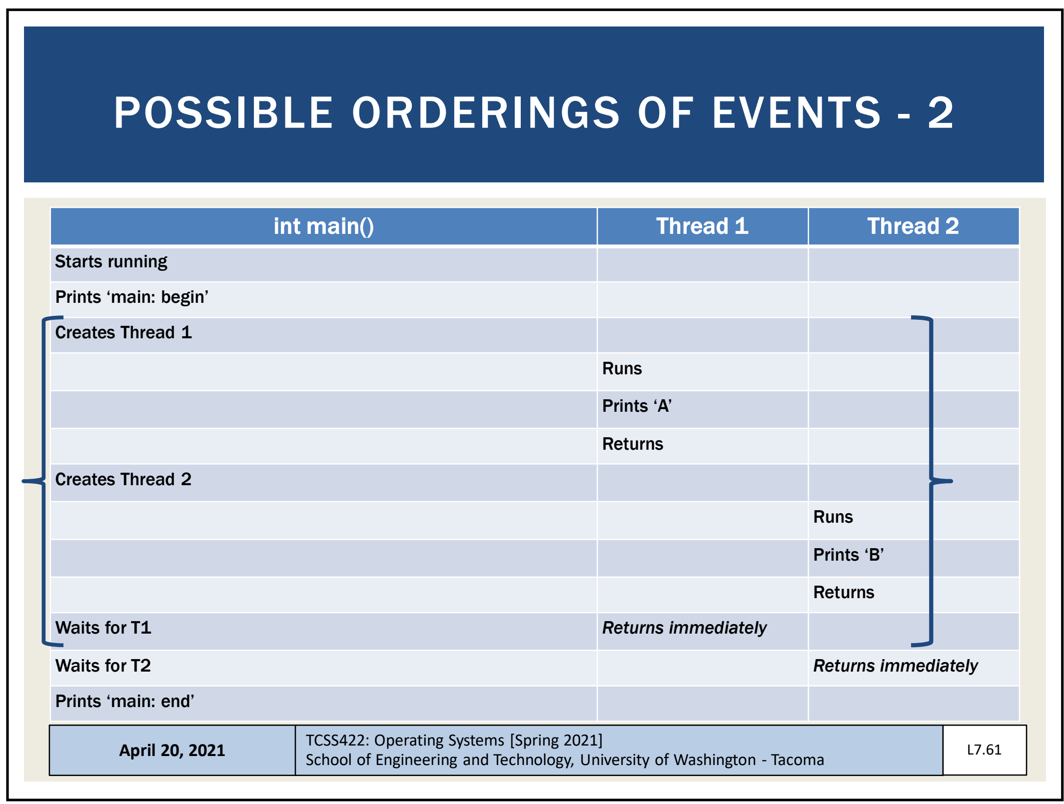

SHARED ADDRESS SPACE
Every thread has it's own stack / PC
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (1)
The code segment: where instructions live
The heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward)
The stack segment: contains local variables arguments to routines, return values, etc.
A Single-Threaded Address Space
Stack (2)
Two threaded Address Space
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.58

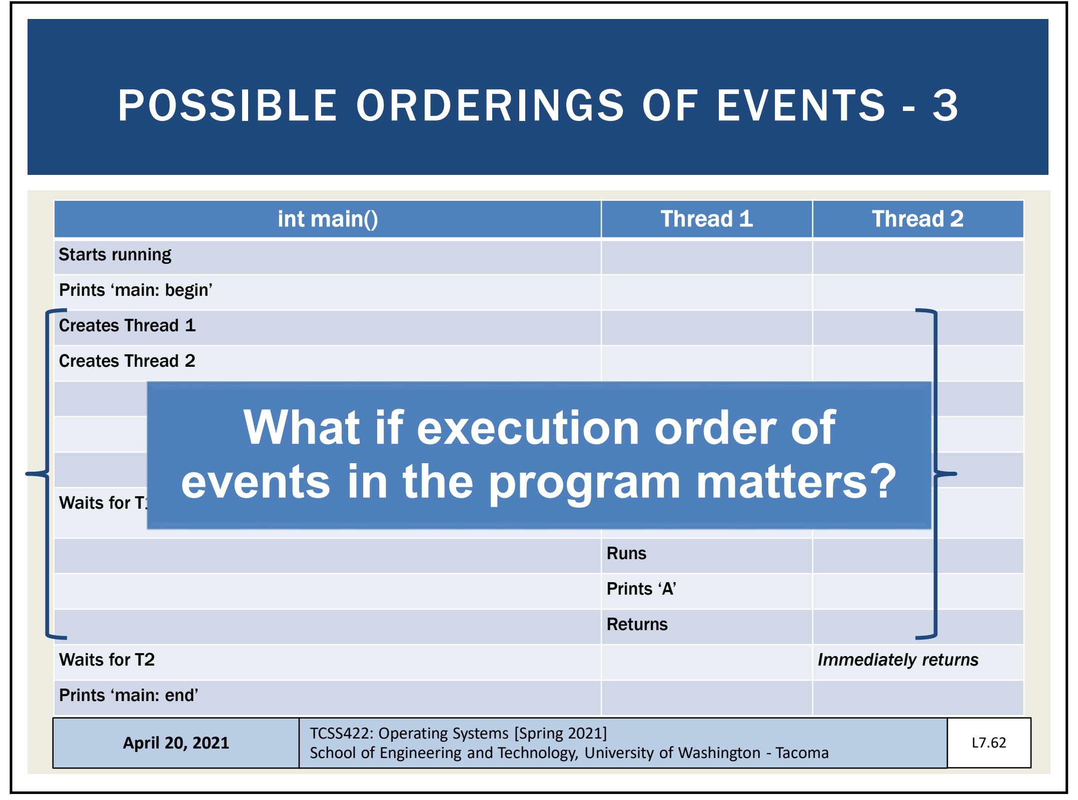



```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```



int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
Waits for T1		
	Runs	
	Prints 'A'	
	Returns	
Waits for T2		
		Runs
		Prints 'B'
		Returns
Prints 'main: end'		


POSSIBLE ORDERINGS OF EVENTS - 2
int main()
Thread 1
Thread 2
Starts running
Prints 'main: begin'
Creates Thread 1
Runs
Prints 'A'
Returns
Creates Thread 2
Runs
Prints 'B'
Returns
Waits for T1
Returns immediately
Waits for T2
Returns immediately
Prints 'main: end'
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.61

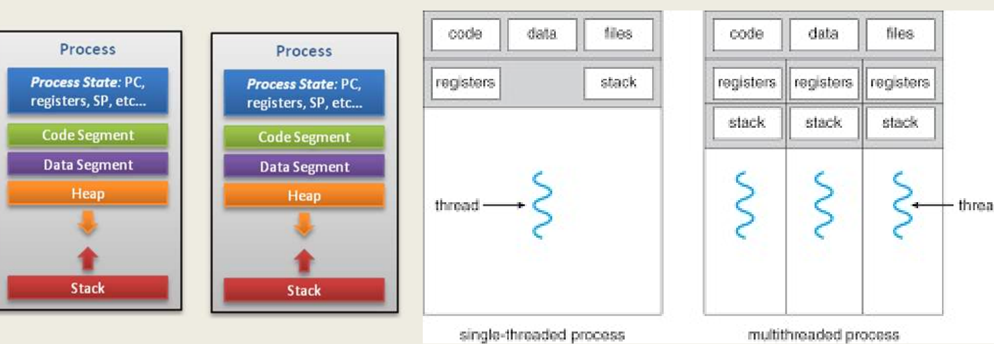

POSSIBLE ORDERINGS OF EVENTS - 3
int main()
Thread 1
Thread 2
Starts running
Prints 'main: begin'
Creates Thread 1
Creates Thread 2
What if execution order of events in the program matters?
Runs
Prints 'A'
Returns
Waits for T2
Immediately returns
Prints 'main: end'
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.62

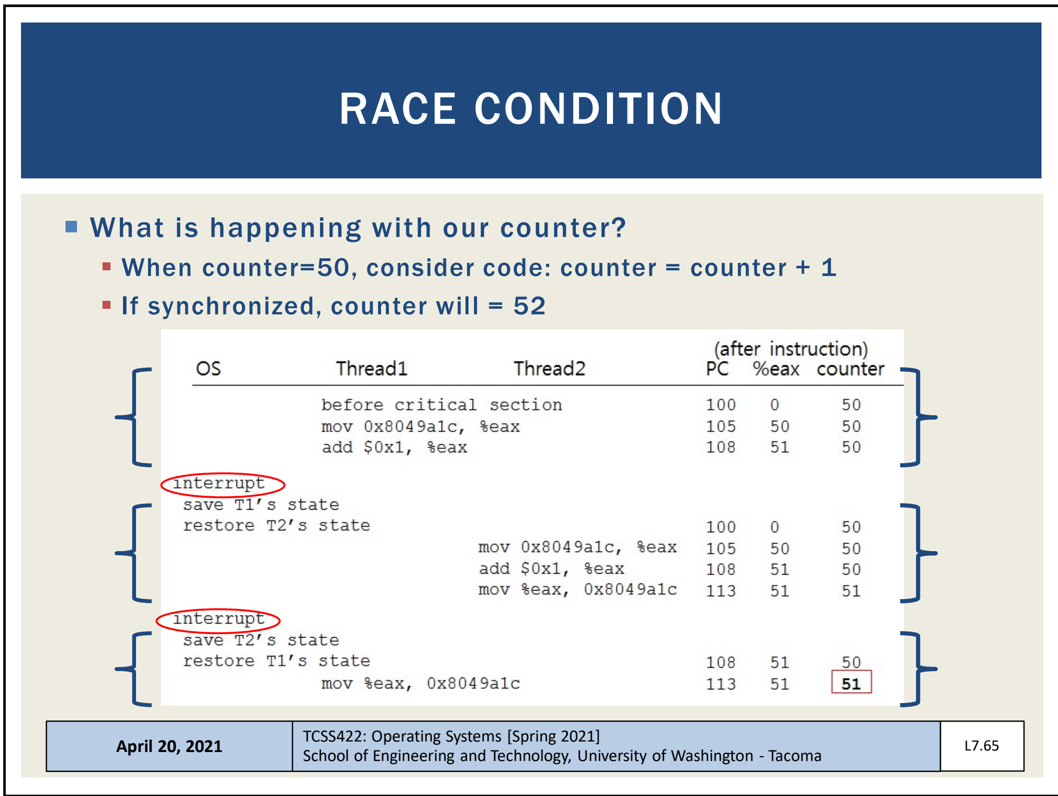
COUNTER EXAMPLE

- Counter example
- A + B : ordering
- Counter: incrementing global variable by two threads
- ***Is the counter example embarrassingly parallel?***
- ***What does the parallel counter program require?***

PROCESSES VS. THREADS

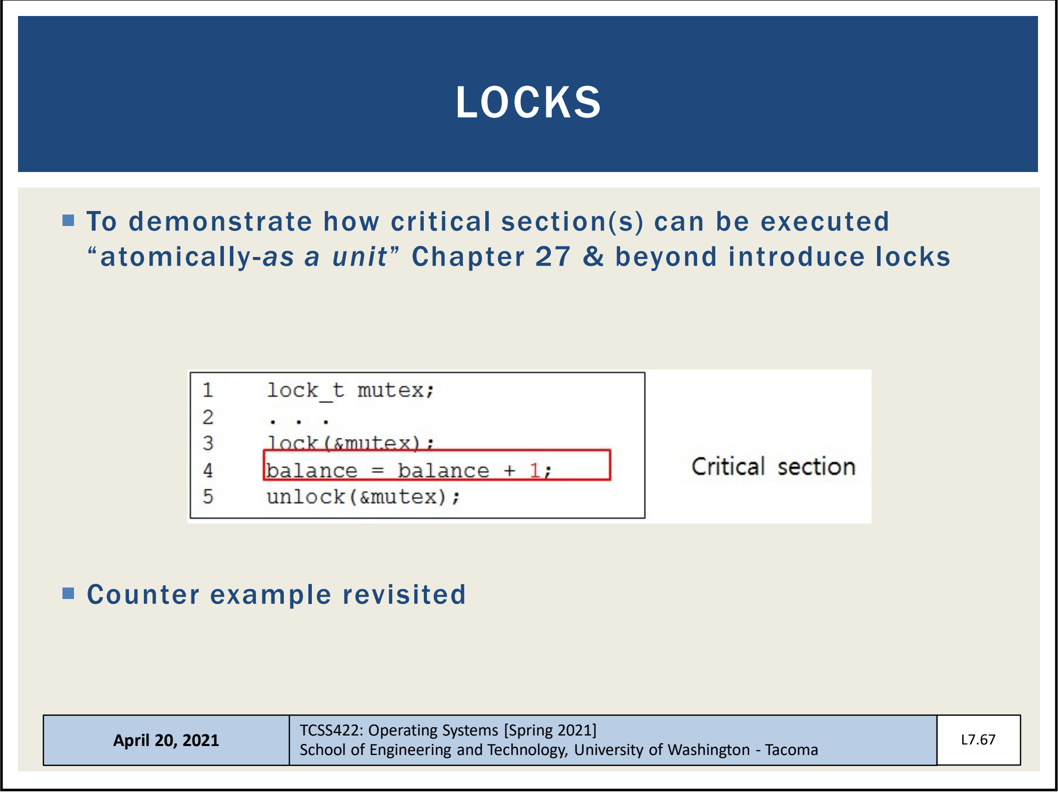
- What's the difference between forks and threads?
  - Forks: duplicate a process
  - Think of ***CLONING*** - There will be two identical processes at the end
  - Threads: no duplication of code/heap, lightweight execution threads

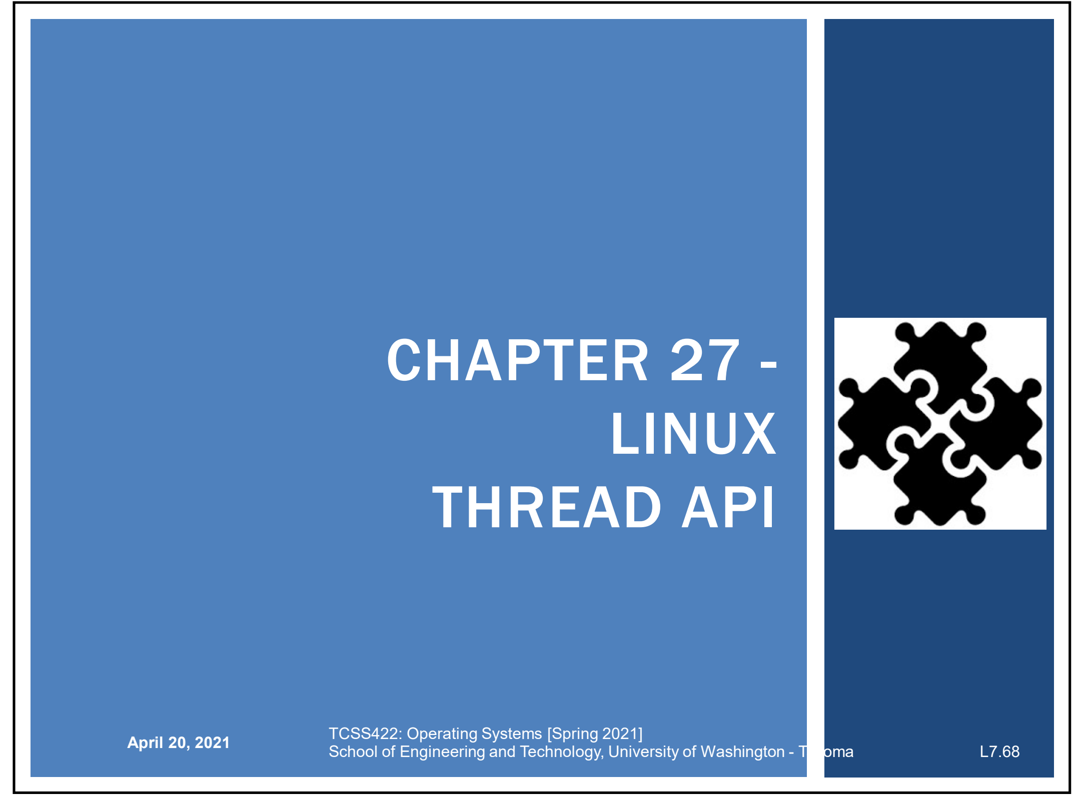




CRITICAL SECTION

- Code that accesses a shared variable must not be ***concurrently*** executed by more than one thread
- Multiple *active* threads inside a ***critical section*** produce a ***race condition***.
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***


LOCKS
To demonstrate how critical section(s) can be executed "atomically-as a unit" Chapter 27 & beyond introduce locks
1 lock_t mutex;
2 . . .
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
Critical section
Counter example revisited
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.67


CHAPTER 27 - LINUX THREAD API
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
L7.68

OBJECTIVES

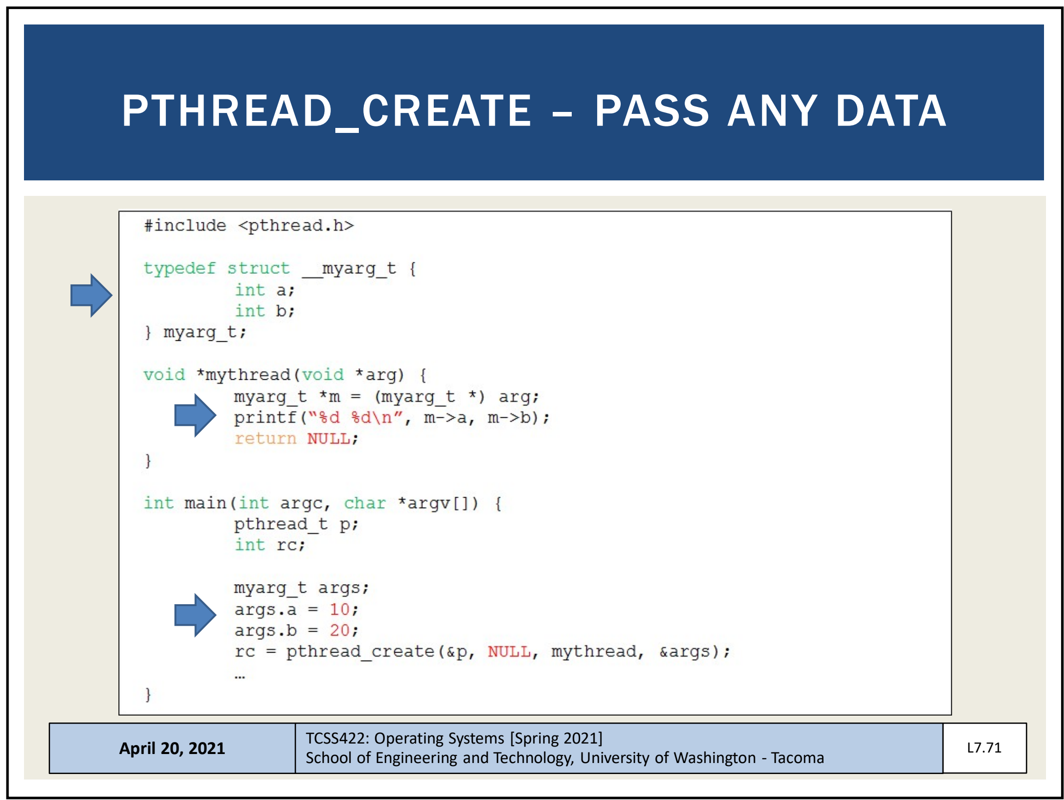
- Chapter 26: Concurrency: An Introduction
  - Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

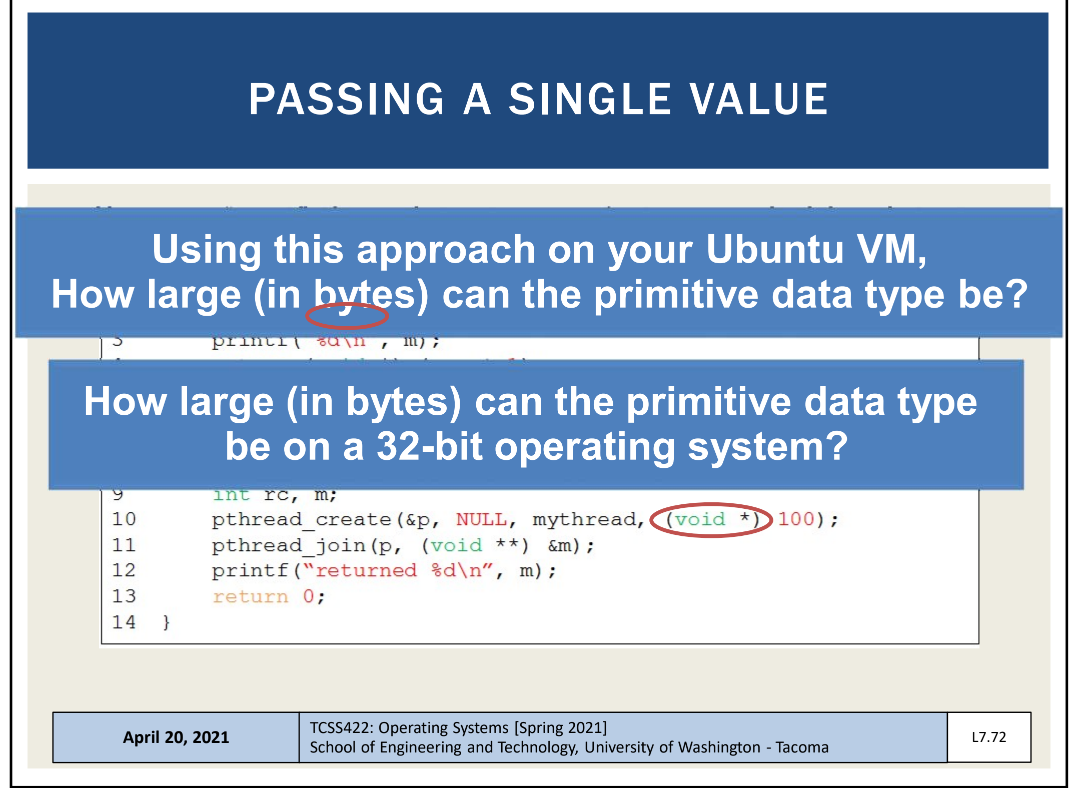
THREAD CREATION

- pthread_create

```
#include <pthread.h>

int
pthread_create(      pthread_t*       thread,
               const pthread_attr_t* attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

- thread: thread struct
- attr: stack size, scheduling priority… (*optional*)
- start_routine: function pointer to thread routine
- arg: argument to pass to thread routine (*optional*)


PTHREAD_CREATE – PASS ANY DATA
#include <pthread.h>
typedef struct __myarg_t {
int a;
int b;
} myarg_t;
void *mythread(void *arg) {
myarg_t *m = (myarg_t *) arg;
printf("%d %d\n", m->a, m->b);
return NULL;
}
int main(int argc, char *argv[]) {
pthread_t p;
int rc;
myarg_t args;
args.a = 10;
args.b = 20;
rc = pthread_create(&p, NULL, mythread, &args);
…
}
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.71


PASSING A SINGLE VALUE
Using this approach on your Ubuntu VM,
How large (in bytes) can the primitive data type be?
How large (in bytes) can the primitive data type be on a 32-bit operating system?
10 pthread_create(&p, NULL, mythread, (void *) 100);
11 pthread_join(p, (void **) &m);
12 printf("returned %d\n", m);
13 return 0;
14 }
April 20, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L7.72

WAITING FOR THREADS TO FINISH

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- thread: which thread?
- value_ptr: pointer to return value
  type is dynamic / agnostic

- Returned values *must* be on the heap
- Thread stacks destroyed upon thread termination (join)
- Pointers to thread stack memory addresses are invalid
  - May appear as gibberish or lead to crash (seg fault)
- Not all threads join – ***What would be Examples ??***

What will this code do?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  struct myarg output;
  output.a = 1;
  output.b = 2;
  return (void *) &output;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_
  pthread_
  printf("
  return 0
}
```

Data on thread stack

```
$ ./pthread_struct
a=10 b=20
Segmentation fault (core dumped)
```

How can this code be fixed?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  input->a = 1;
  input->b = 2;
  return (void *) &input;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_create(&p1, NULL, worker, &args);
  pthread_join(p1, (void *)&ret_args);
  printf("returned %d %d\n", ret_args->a, ret_args->b);
  return 0;
}
```

How about this code?

```
$ ./pthread_struct
a=10 b=20
returned 1 2
```


ADDING CASTS

- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for

- Example: uncasted capture in pthread_join

```
pthread_int.c: In function 'main':
pthread_int.c:34:20: warning: passing argument 2 of 'pthread_join'
from incompatible pointer type [-Wincompatible-pointer-types]
   pthread_join(p1, &p1val);
```

- Example: uncasted return

```
In file included from pthread_int.c:3:0:
/usr/include/pthread.h:250:12: note: expected 'void **' but argument
is of type 'int **'
 extern int pthread_join (pthread_t __th, void **__thread_return);
```

ADDING CASTS - 2

- pthread_join

```
int * p1val;
int * p2val;
pthread_join(p1, (void *)&p1val);
pthread_join(p2, (void *)&p2val);
```

- return from thread function

```
int * counterval = malloc(sizeof(int));
*counterval = counter;
return (void *) counterval;
```


LOCKS

- pthread_mutex_t data type
- /usr/include/bits/pthread_types.h

```
// Global Address Space
static volatile int counter = 0;
pthread_mutex_t lock;

void *worker(void *arg)
{
  int i;
  for (i=0;i<10000000;i++)  {
    int rc = pthread_mutex_lock(&lock);
    assert(rc==0);
    counter = counter + 1;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

LOCKS - 2

- Ensure critical sections are executed atomically-*as a unit*
  - Provides implementation of "***Mutual Exclusion***"
- API

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Example w/o initialization & error checking

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

  - Blocks forever until lock can be obtained
  - Enters critical section once lock is obtained
  - Releases lock

LOCK INITIALIZATION

- Assigning the constant

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- API call:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

- Initializes mutex with attributes specified by 2nd argument
- If NULL, then default attributes are used
- Upon initialization, the mutex is initialized and unlocked

LOCKS - 3

- Error checking wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- What if lock can't be obtained?

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- trylock – returns immediately (fails) if lock is unavailable
- timelock – tries to obtain a lock for a specified duration

CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- pthread_cont_t datatype
- pthread_cond_wait()
  - Puts thread to "sleep" (waits)   (THREAD is BLOCKED)
  - Threads added to >**FIFO queue**<, lock is released
  - Waits ***(listens)*** for a "signal"   (NON-BUSY WAITING, no polling)
  - When signal occurs, interrupt fires, wakes up first thread, (THREAD is RUNNING), lock is provided to thread

CONDITIONS AND SIGNALS - 2

```
int pthread_cond_signal(pthread_cond_t * cond);
int pthread_cond_broadcast(pthread_cond_t * cond);
```

- `pthread_cond_signal()`
  - Called to send a "signal" to wake-up first thread in **FIFO "wait" queue**
  - The goal is to unblock a thread to respond to the signal
- `pthread_cond_broadcast()`
  - Unblocks ***all*** threads in **FIFO "wait" queue**, currently blocked on the specified condition variable
  - Broadcast is used when all threads should wake-up for the signal
- Which thread is unblocked first?
  - Determined by OS scheduler (based on priority)
  - Thread(s) awoken based on placement order in **FIFO wait queue**
  - When awoken threads acquire lock as in `pthread_mutex_lock()`

CONDITIONS AND SIGNALS - 3

- Wait example:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&cond, &lock);
// Perform work that requires lock
a = a + b;
pthread_mutex_unlock(&lock);
```

- wait puts thread to sleep, releases lock
- when awoken, lock reacquired (but then released by this code)
- When initialized, another thread signals

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

State variable set,
Enables other thread(s)
to proceed above.

CONDITION AND SIGNALS - 4

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&cond, &lock);
// Perform work that requires lock
a = a + b;
pthread_mutex_unlock(&lock);
```

- Why do we wait inside a while loop?
- The while ensures upon awakening the condition is rechecked
  - A signal is raised, but the pre-conditions required to proceed may have not been met. **MUST CHECK STATE VARIABLE**
  - Without checking the state variable the thread may proceed to execute when it should not. (e.g. too early)

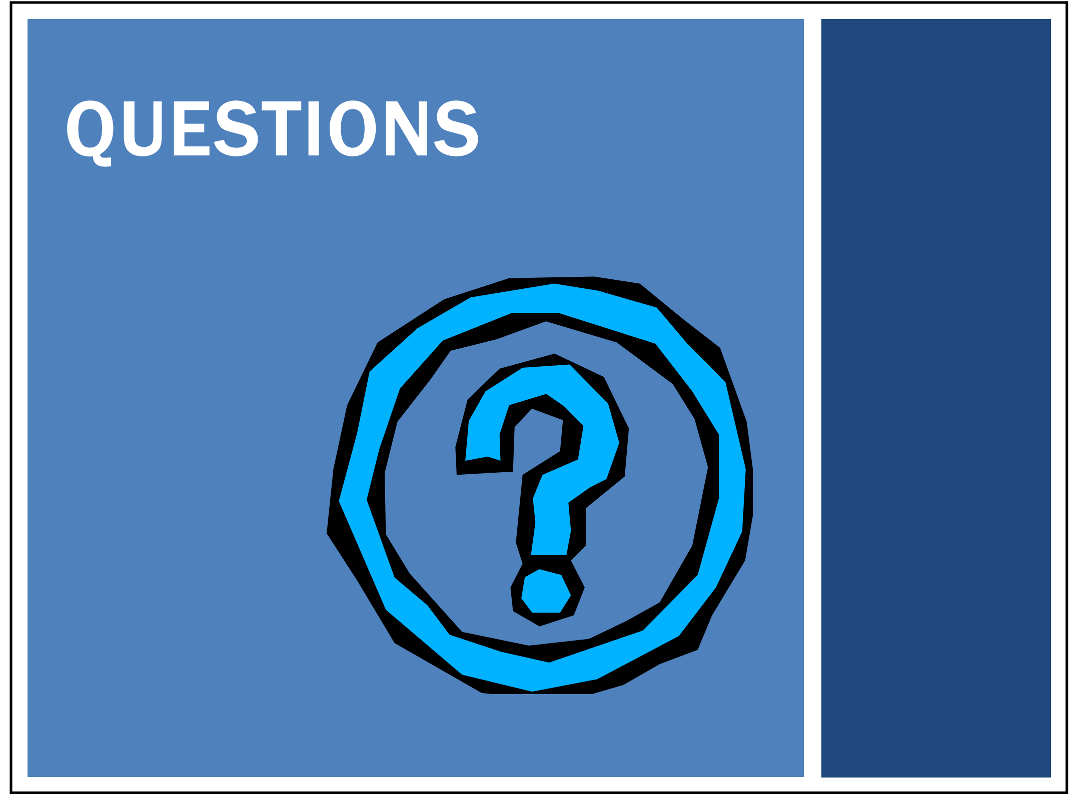
PTHREADS LIBRARY

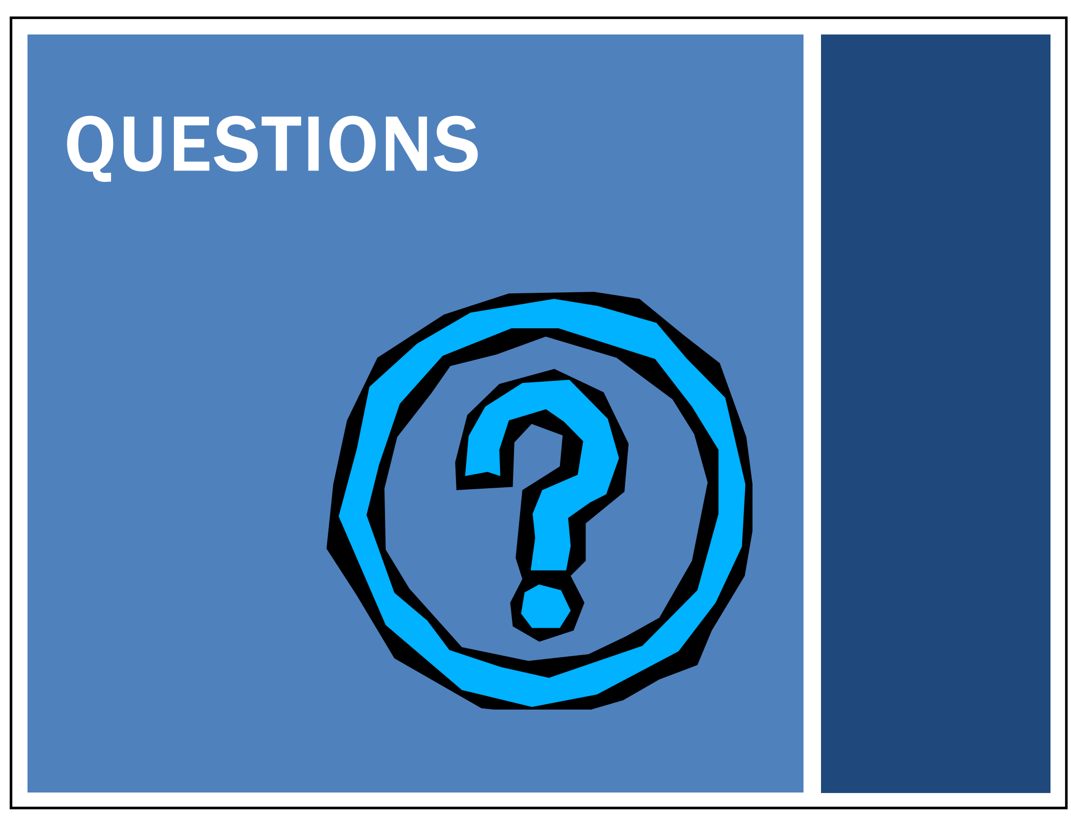
- Compilation:
  gcc requires special option to require programs with pthreads:
  - gcc –pthread pthread.c –o pthread
  - Explicitly links library with compiler flag
  - RECOMMEND: using makefile to provide compiler arguments
- List of pthread manpages
  - man –k pthread

SAMPLE MAKEFILE

```
CC=gcc
CFLAGS=-pthread -I. -Wall

binaries=pthread pthread_int pthread_lock_cond pthread_struct

all: $(binaries)

pthread_mult: pthread.c pthread_int.c
	$(CC) $(CFLAGS) $^ -o $@

clean:
	$(RM) -f $(binaries) *.o
```

- Example builds multiple single file programs
  - All target
- pthread_mult
  - Example if multiple source files should produce a single executable
- clean target




QUESTIONS

TCSS 422
OFFICE HOURS
PLEASE SAY HELLO

OFFICE HOURS
HAVE STEPPED OUT
WILL RETURN
SHORTLY