


TCSS 422: OPERATING SYSTEMS

Proportional Share Schedulers, Introduction to Concurrency



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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

OBJECTIVES – 4/22

- **Questions from 4/20**
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.2

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

Discussions

Upcoming Assignments

TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | /1 pts

Quiz 0 - C background survey

April 22, 2021	TCSS422: Computer Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.3
----------------	--	------

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 22, 2021	TCSS422: Computer Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.4
----------------	--	------

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (48 respondents):
 - 1-mostly review, 5-equal new/review, 10-mostly new
 - **Average – 6.67 (↓ - previous 7.07)**
- Please rate the pace of today's class:
 - 1-slow, 5-just right, 10-fast
 - **Average – 5.53 (↑ - previous 5.48)**

April 22, 2021

TCCS422: Computer Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.5

FEEDBACK

- **Do a lot of malicious softwares operate off the creation and execution of batch commands?**
- **2012:** Can a .sh file be a virus or something harmful?
 - <https://security.stackexchange.com/questions/15585/can-a-sh-file-be-malware>
- .sh files are shell scripts. They are analogous to .bat files under Windows. All of these scripts are executable programs; if you run one, it can do anything you can do. So yes, shell scripts can be harmful. Treat a shell script (or Perl, Python, or Ruby script, etc.) with the same suspicion as any other application.
- It's a bit harder to hide malware in a shell script without looking suspicious, because scripts can be read by people with knowledge of the scripting language. But it is not much harder.
- **2020:** We have started to notice an increase in the script changes and quality. Plain text links are replaced by Base64-encoded text, while some of the code chunks were downloaded or encoded payloads. This is done to hide direct payload links, evade security rules used for their identification, and make analysis more difficult.
- https://www.trendmicro.com/en_us/research/20/i/the-evolution-of-malicious-shell-scripts.html

April 22, 2021

TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.6

BONUS SESSION – EXAMPLE SCHEDULER PROBLEMS

- **Bonus session: Tuesday April 27 starting at 6:30pm**
 - Approximately ~1 hour
- **Will solve a series of example scheduling problems**
 - Focus on: FIFO, SJF, STCF, RR, MLFQ
- **Video will be recorded and posted**

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.7

QUIZ 1

- **Active reading on Chapter 9 – Proportional Share Schedulers**
- **Posted in Canvas**
- **Due Friday April 30th at 11:59pm**
- **Grace period til Sunday May 2nd at 11:59 ** AM ****
- **Late submissions til Tuesday May 4th at 11:59pm**
- **Link:**
- **http://faculty.washington.edu/wlloyd/courses/tcss422/TCSS422_s2021_quiz_1.pdf**

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.8

OBJECTIVES – 4/22

- Questions from 4/20
- **Assignment 0**
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.9

ASSIGNMENT 0

- Due at Thursday April 22 @ 11:59pm
- Grace period: submission ok til Sat April 24 @ **11:59 AM**
- Late submissions: ok til Monday April 26 @ 11:59pm

- OFFICE HOURS FRIDAY April 23rd
- 1:30 to 2:30pm
- *Hours adjusted for this Friday April 23*

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.10

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- **C Tutorial - Pointers, Strings, Exec in C**
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.11
----------------	---	-------

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- **Assignment 1**
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.12
----------------	---	-------

CHAPTER 9 - PROPORTIONAL SHARE SCHEDULER



April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.13

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - **Lottery scheduler**
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.14

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.15

LOTTERY SCHEDULER

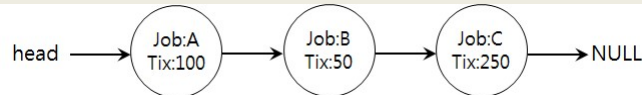
- Simple implementation
 - Just need a random number generator
 - Picks the winning ticket
 - Maintain a data structure of jobs and tickets (list)
 - Traverse list to find the owner of the ticket
 - Consider sorting the list for speed

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.16

LOTTERY SCHEDULER IMPLEMENTATION



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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.17

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - **Ticket mechanisms**
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.18

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)

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.19

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.

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.20

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 B A B A

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

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.21
----------------	---	-------

COIN FLIPPING

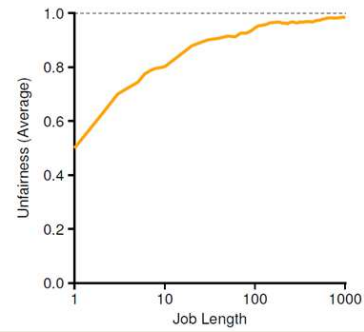
- Equality of distribution (fairness) requires a lot of flips!

Similarly,
Lottery scheduling requires lots of "rounds" to achieve fairness.

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.22
----------------	---	-------

LOTTERY FAIRNESS

- With two jobs
 - Each with the same number of tickets ($t=100$)



Job Length	Average Unfairness
1	0.5
10	0.8
100	0.95
1000	1.0

When the job length is not very long, average unfairness can be quite severe.

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.23
----------------	---	-------

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...

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.24
----------------	---	-------

**WE WILL RETURN AT
4:53PM**



April 22, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L8.25

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - **Stride scheduler**
 - Linux Completely Fair Scheduler
- 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 22, 2021 TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma L8.26

STRIDE SCHEDULER

- Addresses statistical probability issues with lottery scheduling
- Instead of guessing a random number to select a job, simply count...

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.27

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 → $A_{\text{stride}} = 10000/100 = 100$ stride
 - Job B has 50 tickets → $B_{\text{stride}} = 10000/50 = 200$ stride
 - Job C has 250 tickets → $C_{\text{stride}} = 10000/250 = 40$ stride
- Stride scheduler tracks “pass” values for each job (A, B, C)

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.28

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...

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.29

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.30

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 22, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L8.31

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 22, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L8.32

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	...

April 22, 2021
TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.33

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - **Linux Completely Fair Scheduler**
- 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 22, 2021
TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.34

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 !

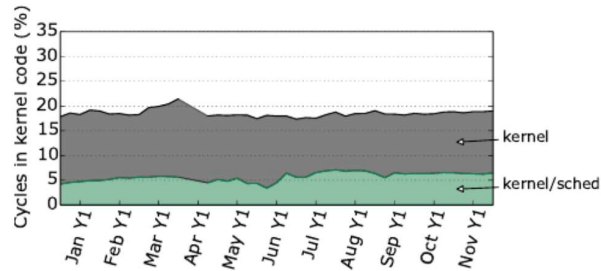


Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

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

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.35
----------------	---	-------

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 $1/n^{\text{th}}$ 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)

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.36
----------------	---	-------

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`

April 22, 2021

TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.37

COMPLETELY FAIR SCHEDULER - 3

- Linux \geq 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

April 22, 2021

TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.38

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/

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.39

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_wakeup_granularity_ns`

increased context switching → more overhead
better near-term fairness

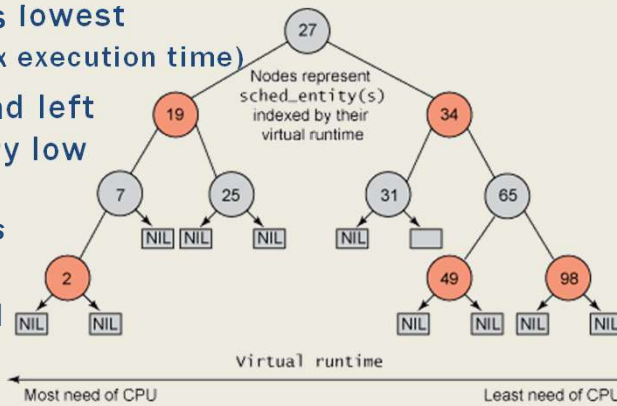
April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.40

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: $\sim O(\log N)$ for N nodes
- Completed processes removed



April 22, 2021

TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma

L8.41

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`
- 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

```
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,
};
```

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma

L8.42

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!

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.43

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>

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.44

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION



April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.45

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.46

THREADS

The diagram illustrates the memory layout of a single-threaded process versus a multithreaded process. On the left, a 'Single Threaded Process' is shown with a vertical stack of memory segments: Process State (PC, registers, SP, etc...), Code Segment, Data Segment, Heap, and Stack. On the right, a 'Multithreaded Process' is shown with a shared memory space containing Code Segment, Data Segment, and Heap, which are labeled as 'SHARED' with lightning bolts. Each thread in the multithreaded process has its own Thread State and Stack. The Process State is also shared.

©Alfred Park, <http://randu.org/tutorials/threads>

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.47
----------------	---	-------

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 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.48
----------------	---	-------

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 22, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.49

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC

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**

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.

**Two threaded
Address Space**

April 22, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.50

THREAD CREATION EXAMPLE

```

#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;
}
    
```

April 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.51
----------------	---	-------

POSSIBLE ORDERINGS OF EVENTS

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'		

April 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.52
----------------	---	-------

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 22, 2021
TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.53

POSSIBLE ORDERINGS OF EVENTS - 3

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		Immediately returns
Prints 'main: end'		

What if execution order of events in the program matters?

April 22, 2021
TCSS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.54

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?


April 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.55
----------------	---	-------

PROCESSES VS. THREADS

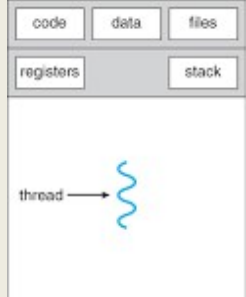
- What's the difference between forks and threads?
 - Forks: duplicate a process
 - Think of **CLONING** - There will be two identical processes at the end
 - Threads: no duplication of code/heap, lightweight execution threads



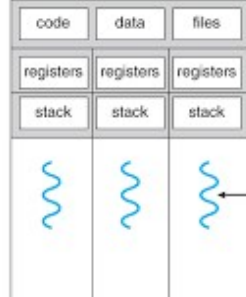
Process



Process



single-threaded process



multithreaded process

April 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.56
----------------	---	-------

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.57
----------------	---	-------

RACE CONDITION

- What is happening with our counter?
 - When counter=50, consider code: counter = counter + 1
 - If synchronized, counter will = 52

	OS	Thread1	Thread2	(after instruction)		
				PC	%eax	counter
{		before critical section		100	0	50
		mov 0x8049a1c, %eax		105	50	50
		add \$0x1, %eax		108	51	50
	interrupt					
{		save T1's state		100	0	50
		restore T2's state		105	50	50
			mov 0x8049a1c, %eax	105	50	50
{			add \$0x1, %eax	108	51	50
			mov %eax, 0x8049a1c	113	51	51
	interrupt					
{		save T2's state		108	51	50
		restore T1's state		113	51	51
		mov %eax, 0x8049a1c		113	51	51

April 22, 2021	TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.58
----------------	---	-------


OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.59
----------------	---	-------

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**



April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.60
----------------	---	-------

LOCKS

- To demonstrate how critical section(s) can be executed “atomically-as a unit” Chapter 27 & beyond introduce locks

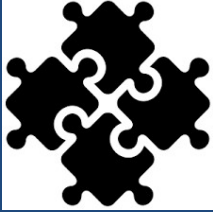
```
1  lock_t mutex;  
2  . . .  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```

Critical section

- Counter example revisited

April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.61
----------------	---	-------

CHAPTER 27 - LINUX THREAD API



April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.62
----------------	---	-------

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.63

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*)

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.64

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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.65

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?

```
9     int rc, m;
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 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.66


```

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**

April 22, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.69

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);
                
```

April 22, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L8.70

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;
```

April 22, 2021

TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.71

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021

TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.72

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;
}
```

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.73

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.74

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.75

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.76

OBJECTIVES – 4/22

- Questions from 4/20
- Assignment 0
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1
- Chapter 9: Proportional Share Schedulers
 - Lottery scheduler
 - Ticket mechanisms
 - Stride scheduler
 - Linux Completely Fair Scheduler
- 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 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.77
----------------	---	-------

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_cond_t datatype
- pthread_cond_wait()
 - Puts thread to “sleep” (waits) (THREAD is BLOCKED)
 - Threads added to >FIFO queue<, lock is released
 - Waits (*llstens*) 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



April 22, 2021	TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma	L8.78
----------------	---	-------

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()`

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.79

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(&cond);  
pthread_mutex_unlock(&lock);
```

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.80

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)

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.81

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.82

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

April 22, 2021

TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma

L8.83

QUESTIONS

