TCSS 422: OPERATING SYSTEMS

Multi-level Feedback Queue II, Proportional Share Schedulers, Introduction to Concurrency

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

1

TEXT BOOK COUPON

- 15% off textbook code: **EARTHWEEK15**
  (*through Friday Apr 21*)
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-softcover-version-100/paperback/product-14mjrrgk.html
- With coupon textbook is only $18.70 + tax & shipping

2

OFFICE HOURS – SPRING 2023

- **Tuesdays:**
  - 2:30 to 3:30 pm - CP 229 / Zoom
- **Fridays**
  - *1:30 to 2:30 pm – Zoom / (CP 229-on some days)
- Also available after class
- Or email for appointment

> Office Hours set based on Student Demographics survey feedback
** time may be occasionally rescheduled due to faculty meeting conflicts*

3

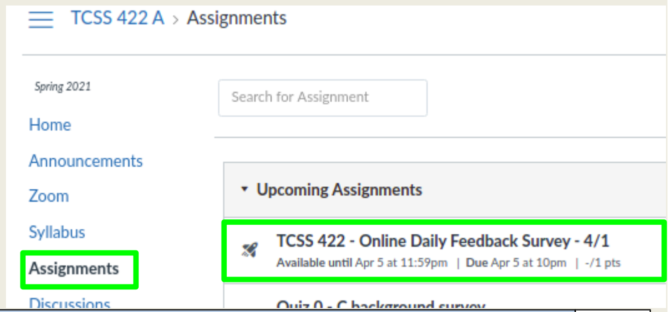
OBJECTIVES – 4/18

- **Questions from 4/13**
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

4

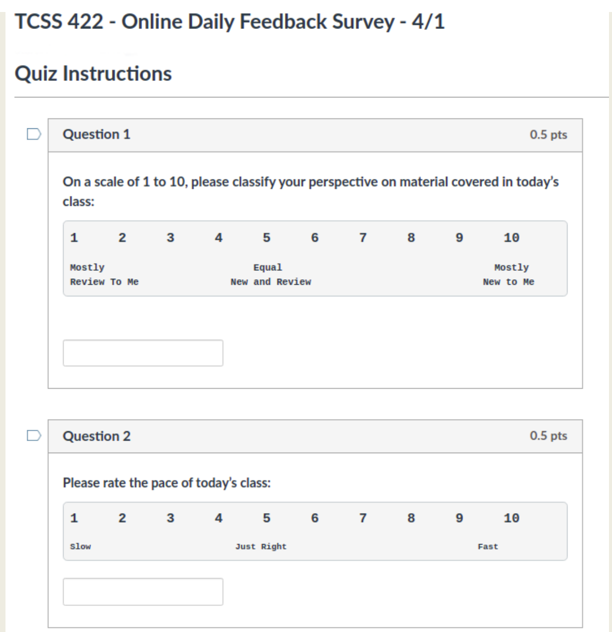
ONLINE DAILY FEEDBACK SURVEY

- 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



April 18, 2023 | TCSS422: Computer Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L7.5

5



April 18, 2023 | TCSS422: Computer Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L7.6

6

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (47 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 7.26 (↓ - previous 7.64)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.79 (↓ - previous 6.31)**

FEEDBACK FROM 4/13

- ***What does round robin do if multiple jobs arrive at the same time? Is this something that can happen or is there always a time difference?***
  - For the scheduling problems we solve, a distinct job arrival sequence will always be provided
  - Jobs may share the same arrival time (t=0), but an arrival sequence will be specified "A B C"
  - For a scheduling problem that leads to equally probable scheduling actions, both will be considered as legitimate in problem grading

FEEDBACK - 2

- ***For the multi-level feedback queue scheduler, is there a way to manually force a Low Priority job back to a High Priority queue?***
- ***For example, if our "Weather Simulation" is being a CPU hog and it gets pushed to the bottom, is there a way to force it back to the Highest Priority queue, say if I need the simulation to ~~finish immediately~~ respond to user input and just don't care about anything else?***
  - We will next introduce solutions to '***gaming the scheduler***' which are situations where jobs may become starved for execution time on the CPU

OBJECTIVES – 4/18

- Questions from 4/13
- **Assignment 0 - Due Fri Apr 21**
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

ASSIGNMENT 0 - DUE FRI APR 21

- Due Friday April 21 @ 11:59pm
- Grace period: submission ok until Sun Apr 23 @ 11:59 AM
- Late submissions thru Tuesday Apr 25 @ 11:59pm

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Thursday April 27th at 11:59pm
- Link:
- https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_s2023_quiz_1.pdf

QUIZ 2

- Canvas Quiz – Practice CPU Scheduling Problems

- Posted in Canvas
- Unlimited attempts permitted
- Due Tuesday May 2nd at 11:59pm

- **Link:**
- https://canvas.uw.edu/courses/1642522/assignments/8316759

COMING SOON...

- Assignment #1
  - To be posted for next class, Thursday Apr 20

- Midterm Exam
  - Thursday May 4th
  - In Class

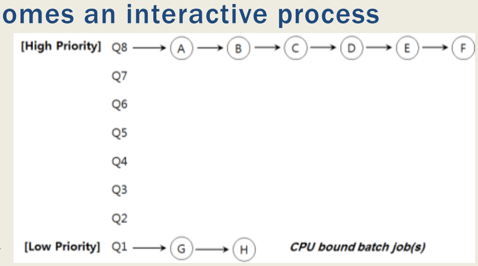
OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - **Gaming the Scheduler**
  - Examples
- 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

17

MLFQ: ISSUES - 2

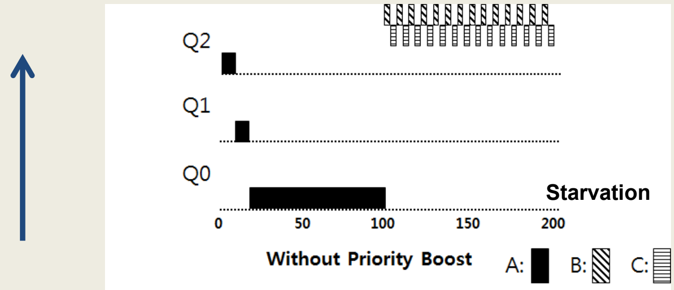
- Gaming the scheduler
  - Issue I/O operation at 99% completion of the time slice
  - Keeps job priority fixed – never lowered
- Job behavioral change
  - CPU/batch process becomes an interactive process



Priority becomes stuck

18

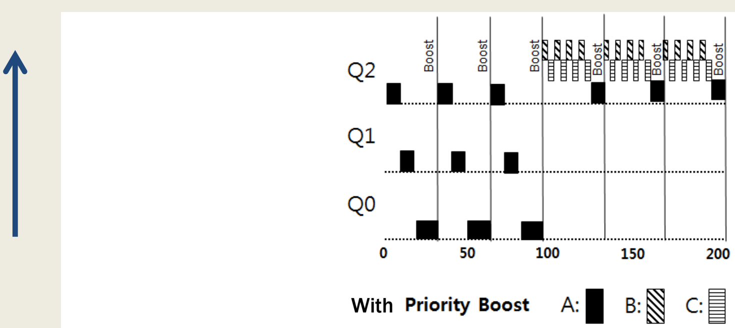
RESPONDING TO BEHAVIOR CHANGE



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

RESPONDING TO BEHAVIOR CHANGE - 2

- With priority boost
  - Prevents starvation



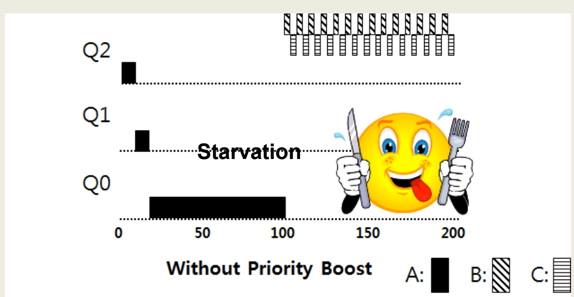
KEY TO UNDERSTANDING MLFQ – PB

- Without priority boost:
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
- **KEY**: If time quantum of a higher queue is filled, then we don't run any jobs in lower priority queues!!!

STARVATION EXAMPLE

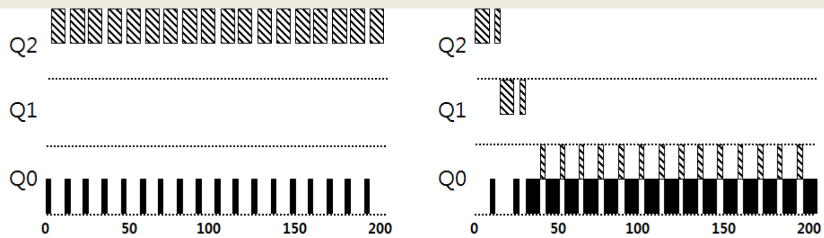
- **Consider 3 queues:**
- Q2 – HIGH PRIORITY – Time Quantum 10ms
- Q1 – MEDIUM PRIORITY – Time Quantum 20 ms
- Q0 – LOW PRIORITY – Time Quantum 40 ms

- Job A: 200ms no I/O
- Job B: 5ms then I/O
- Job C: 5ms then I/O
- Q2 fills up, starves Q1 & Q0
- A makes no progress



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

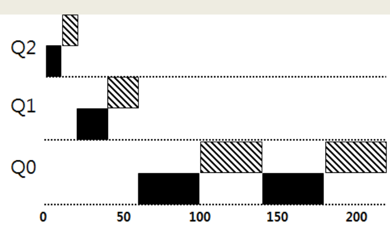


Without(Left) and With(Right) Gaming Tolerance

23

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

24

PRACTICAL EXAMPLE

- Legacy Oracle Solaris (Unix) MLFQ implementation (v2.6)
  - 60 Queues → w/ slowly increasing time slice
    (high to low priority)
    20ms high priority time slice
    100ms low priority time slice
    boost every second
  - Provides sys admins with set of editable table(s)
  - Supports adjusting time slices, boost intervals, priority changes, etc.
- Giving the scheduler 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 in the highest priority queue.
- **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 one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - **Examples**
- 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

27

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.

<u>SANITY CHECK</u>: Consider the timing graph x-axis should not exceed the combined job length of all jobs.

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

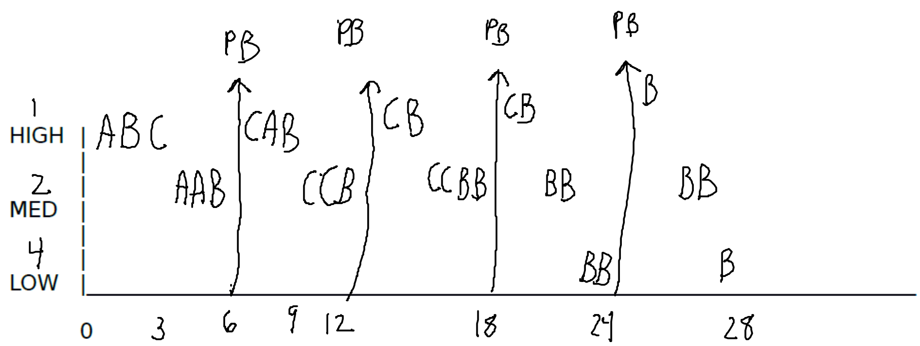
28

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	
		28	

time slice is job time before c/s

(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.



29

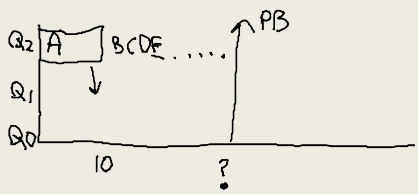
EXAMPLE

- Question:
- Given a system with a quantum length of 10 ms ***for all jobs*** in its highest queue, how often would you have to boost job A (the first job to arrive and run) back to the highest priority level to guarantee that job A, a long-running (and potentially starving) job gets at least 5% of the CPU assuming that on priority boost job execution resets to the front of the queue?

30

EXAMPLE

- Question:
- Given a system with a quantum length of 10 ms ***for all jobs*** in its highest queue, how often would you have to boost job A (the first job to arrive and run) back to the highest priority level to guarantee that job A, a long-running (and potentially starving) job gets at least 5% of the CPU assuming that on priority boost job execution resets to the front of the queue?



$$.05\ PB = 10$$

$$PB = \frac{10}{.05} = 200\ ms$$

April 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.31

31

EXAMPLE

- Question:
- Given a system with a quantum length of 10 ms ***for all jobs*** 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 in highest queue (10 ms)
  - Batch jobs starts, runs for full quantum of 10ms, pushed to lower queue
  - 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***

April 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.32

32

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

33

CHAPTER 9 - PROPORTIONAL SHARE SCHEDULER

34

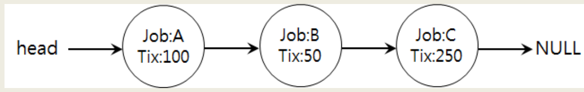
PROPORTIONAL SHARE SCHEDULER

- 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

LOTTERY SCHEDULER IMPLEMENTATION




```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// 'current' is the winner: schedule it...
```

April 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.37

37

WE WILL RETURN AT 4:50PM

April 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.38

38

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

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

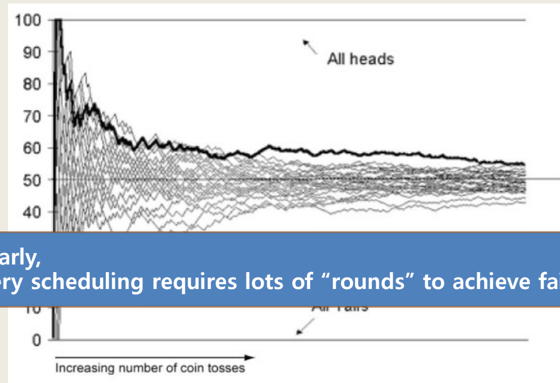
- Scheduler picks a <u>**winning**</u> ticket
  - Load the job with the winning ticket and run it
- Example:
  - Given 100 tickets in the pool
    - Job A has 75 tickets: 0 - 74
    - Job B has 25 tickets: 75 – 99

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!



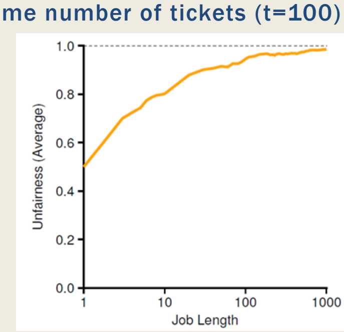
**Similarly,
Lottery scheduling requires lots of "rounds" to achieve fairness.**

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.43

43

LOTTERY FAIRNESS

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



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

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.44

44

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 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.45

45

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.46

46

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…

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 …

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	…

53

Which of the following is NOT a problem with proportional share schedulers?

A. How tickets should be distributed to incoming jobs

B. Lottery scheduler is only eventually fair

C. Given 2 users A and B who both receive a 50% timeshare of the system, the runtime for User A's jobs is dependent on the runtime of User B's.

D. All of the above

E. None of the above

54

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

55

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 !

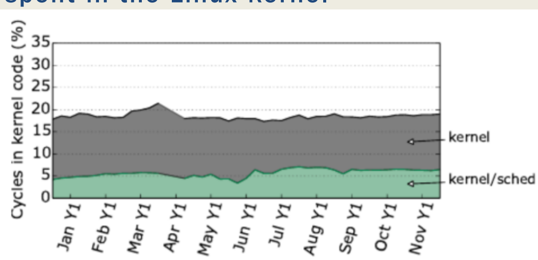


Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

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

56

LINUX: COMPLETELY FAIR SCHEDULER (CFS)

- Loosely based on the stride scheduler
- CFS models system as a Perfect Multi-Tasking System
  - In a perfect system every process of the same priority (class) receives exactly $1/n^{th}$ of the CPU time
- Each scheduling class has a runqueue
  - Groups processes of the same class
  - In the 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 exceeds 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`

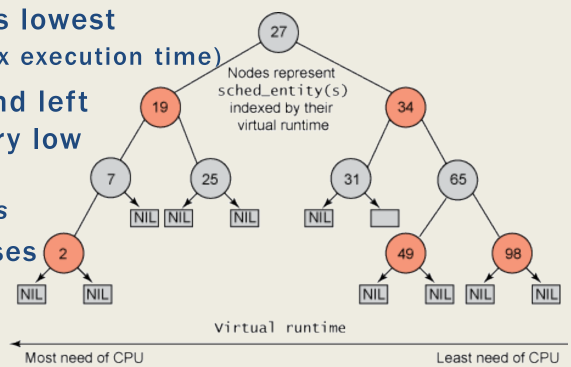
  CFS features reduced context switching → less overhead
  poor near-term fairness

- **LOW** `sched_min_granularity_ns` (timeslice)
  `sched_latency_ns`
  `sched_wakreup_granularity_ns`

  CFS features 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 are removed



CFS: JOB PRIORITY

```
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,
};
```

- 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

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.63

63

COMPLETELY FAIR SCHEDULER - 6

- CFS tracks cumulative job run time with the `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
  - **GOAL:** Perfect scheduler →
    achieve equal `vruntime` for all processes of same priority
- Sleeping jobs: upon return a temporary `vruntime` can be used to increase temporarily the priority of the task
- When tasks wait for I/O they should receive a comparable share of the CPU as if they were performing compute ops when run again
- Key takeaway:
  identifying the next job to schedule is really fast!

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.64

64

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

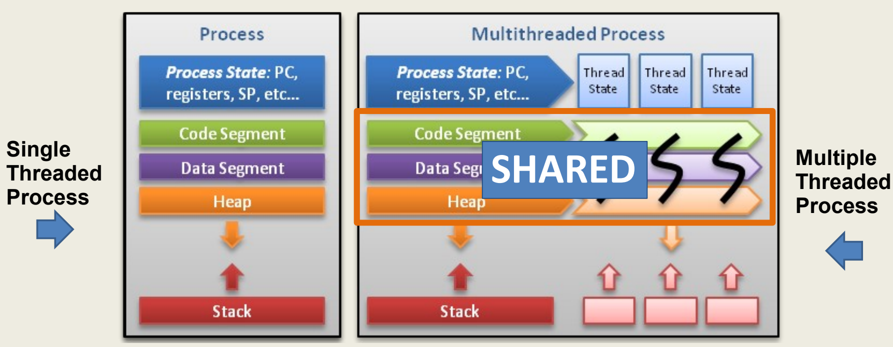
OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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 26 - CONCURRENCY: AN INTRODUCTION

67

THREADS



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

68

THREADS - 2

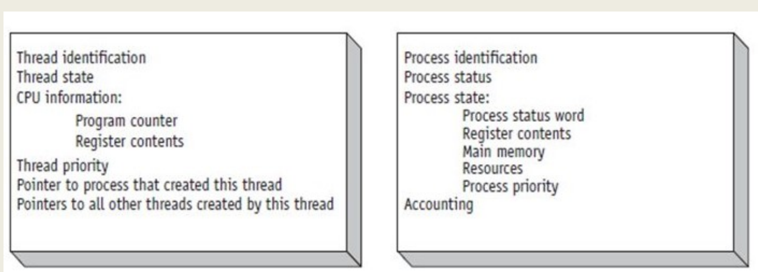
- Enables a single process (program) to have multiple "workers"
  - This is parallel programming…
- Supports independent path(s) of execution within a program *with shared memory …*
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Threads share code segment, memory, and heap are shared
- ***What is an embarrassingly parallel program?***

April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.69

69

PROCESS AND THREAD METADATA

- Thread Control Block vs. Process Control Block

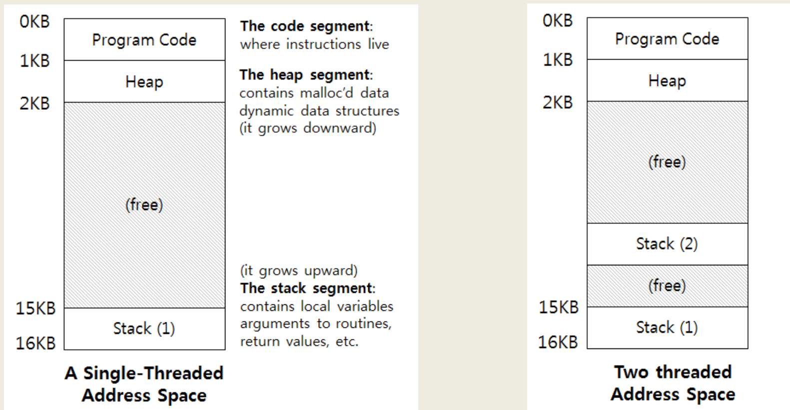


April 18, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L7.70

70

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC



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;
}
```

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 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.73

73

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 18, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L7.74

74

POSSIBLE ORDERINGS OF EVENTS - 3

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
Waits for T		
	Runs	
	Prints 'A'	
	Returns	
Waits for T2		*Immediately returns*
Prints 'main: end'		

What if execution order of events in the program matters?

75

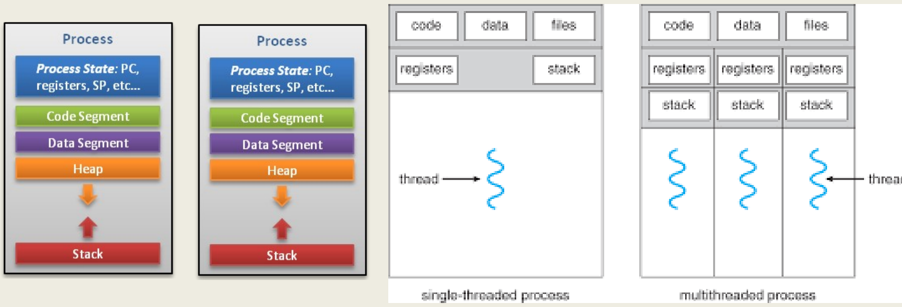
COUNTER EXAMPLE

- 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?***

76

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

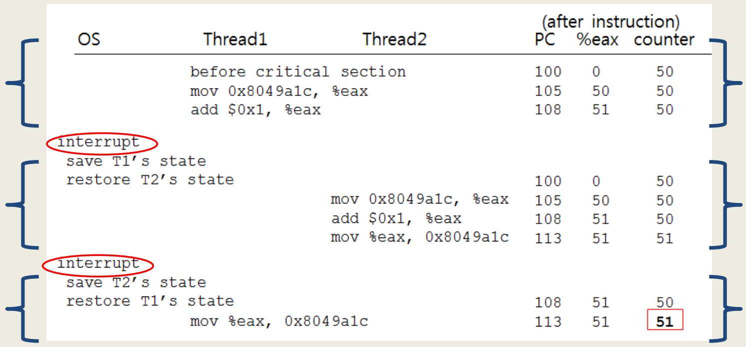


OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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

RACE CONDITION

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



79

OBJECTIVES – 4/18

- Questions from 4/13
- Assignment 0 - Due Fri Apr 21
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Quiz 1 and Quiz 2
- Chapter 8: Multi-level Feedback Queue
  - Gaming the Scheduler
  - Examples
- 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**

80

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

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- Counter example revisited

QUESTIONS

83