TCSS 422: OPERATING SYSTEMS

Completely Fair Scheduler, Introduction to Concurrency, Threads, Thread API

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

January 24, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Homework 0 Questions
- Tutorial 1 Questions
- Homework 1 Questions
- Active Reading Quiz 1
- Feedback from 1/22
- Linux Completely Fair Scheduler
- Ch. 26
  - Introduction to concurrency, threads
- Ch. 27
  - Thread API
- Ch. 28
  - Locks

January 24, 2018	TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma	L6.2

SELECTED FEEDBACK FROM 1/22

- Can you go over more examples of priority boosy and preventive gaming?

- Sample problem next slide

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

FEEDBACK - 2

- Is the lottery scheduler ever useful?
  - Biggest benefit: ease of implementation
- What is the purpose of the user prioritizing jobs (in the ticket mechanisms example) if the OS will handler prioritizing?
  - If the user has multiple jobs, this allows the user to provide priority for their own set of jobs
  - For example: the user may have one job with HIGH priority, and another which is ***VERY LOW…***

FEEDBACK - 3

- How does scheduling relate to virtualization?
  - With virtual machines, there is often a separate scheduler which coordinates sharing the CPU among multiple CPUs
  - For Amazon Cloud, "XEN" is the program (called a hypervisor) used to host the virtual machines (VMs)
  - Akin to Virtual Box but designed for use on servers
  - "XEN" provides its own operating system kernel complete with schedulers to share the CPU and I/O devices among all guest VMs
- How does the OS reassign tickets when more processes join? Does it avoid inflation?
  - The OS distributes tickets from a fixed pool.
  - Presumably the OS will need to redistributed tickets to all jobs as the ratios change
  - Tickets provide an analogy to the CPU time share of a job

FEEDBACK - 4

- **How is the Stride Scheduler not just a convoluted priority queue?**
  - Queues arrange jobs in a first in / first out fashion
  - Time is delineated among jobs in a round-robin fashion with each job receiving an equal share of the CPU (e.g. ***time slice***)
  - The stride scheduler allows assignment of tickets to influence the time share of each job
  - Round robin queues have no such feature

FEEDBACK - 5

- **Will there be a practice midterm?**
  - Tentative plan – second half of class on Monday February 5th
- **Spending a lot of time on feedback seems a bit detrimental to the content you intended to cover**
  - Covering every topic once and never reviewing would helps increase the total volume of content (chapters) covered…
  - ***. . . at the cost of student retention***
  - While it may seem redundant for some to review already familiar topics, some students may be seeing things for the first time
  - Ideally, there would be time to cover everything twice, once in lecture, and again in an activity or open discussion

CHAPTER 9 - PROPORTIONAL SHARE SCHEDULER

January 24, 2018
TCSS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma
L6.9

STRIDE SCHEDULER

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

January 24, 2018
TCSS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma
L6.10

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
  - Job B has 50 tickets → B_{stride} = 10000/50 = 200
  - Job C has 250 tickets → C_{stride} = 10000/250 = 40
- Stride scheduler tracks "pass" values for each job (A, B, C)

STRIDE SCHEDULER - 3

- Basic algorithm:
  1. Stride scheduler picks a 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)
- 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

- Each job tracks its pass value with a counter
- Each time a job runs we increment its counter by its stride to track when it should next run
- Start by randomly choosing A (all pass values=0)

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

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 receives a lot of opportunities to run…

STRIDE SCHEDULER EXAMPLE - 4

- Job counters support determining which job to run next
- Over time jobs are selected 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	…

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 receives exactly 1/n th of the CPU time
- Scheduling classes (runqueues)
  - Each has specific priority: default, real-time
  - Scheduler picks highest priority task in highest scheduling class
  - Time quantum based on proportion of CPU time (%), not fixed time allotments
  - Quantum calculated using NICE value

COMPLETELY FAIR SCHEDULER - 2

- Time slice: Linux ***"Nice value"***
  - Nice value predates the CFS scheduler
  - Top shows nice values
  - Process command: `Ps ax -o pid,ni,cmd,%cpu`
- Nice Values: from -20 to 19
  - Lower is ***higher*** priority, default is 0
  - Scheduling quantum is calculated using nice value
  - Target latency:
    - Interval during which task should run at least once
    - Automatically increases as number of jobs increases

COMPLETELY FAIR SCHEDULER - 3

- Challenge:
  - How do we map a nice value to an actual CPU timeslice (ms)
  - What is the best mapping?
    - O(1) scheduler (< 2.6.23)
      - tried to map nice value to timeslice (fixed allotment)
    - Linux completely fair scheduler
      - maps nice value based on time proportion
      - with fewer jobs in a runqueue, the time quantum is larger

COMPLETELY FAIR SCHEDULER - 4

- Nice values become relative for determining time slices
  - Proportion of CPU time to allocate is relative to other queued tasks
- Scheduler tracks virtual run time in vruntime variable
- The task on a given runqueue (nice value) with the lowest `vruntime` is scheduled text
- `struct sched_entity` contains `vruntime` parameter
  - Describes process execution time in nanoseconds
  - Perfect scheduler → achieve equal `vruntime` for all processes of same priority

COMPLETELY FAIR SCHEDULER - 5

- CFS uses weighted fair queueing
- Runqueues are stored using a linux rbtree
  - Self balancing binary search tree
  - The leftmost node will have the lowest `vruntime`
  - Walking the tree to find the left most node is only O(log N) for N nodes
  - If tree is balanced, left most node can be cached
- Key takeaway
  identifying the next job to schedule is ***really*** fast!

January 24, 2018 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L6.21

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

January 24, 2018 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L6.22

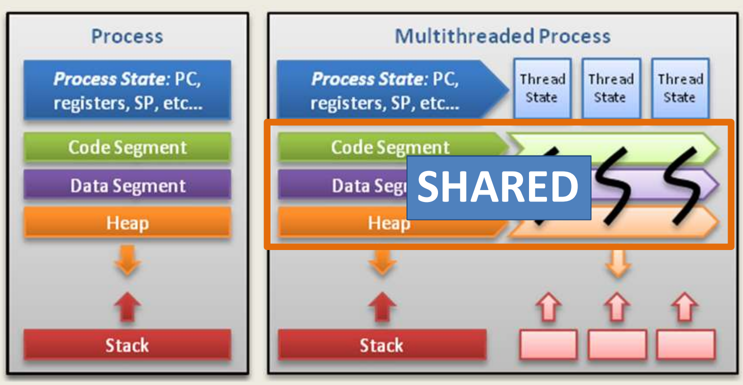
OBJECTIVES

- Introduction to threads
- Race condition
- Critical section
- Thread API

January 24, 2018	TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma	L6.23

THREADS

Single Threaded Process



Multiple Threaded Process

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

January 24, 2018	TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma	L6.24

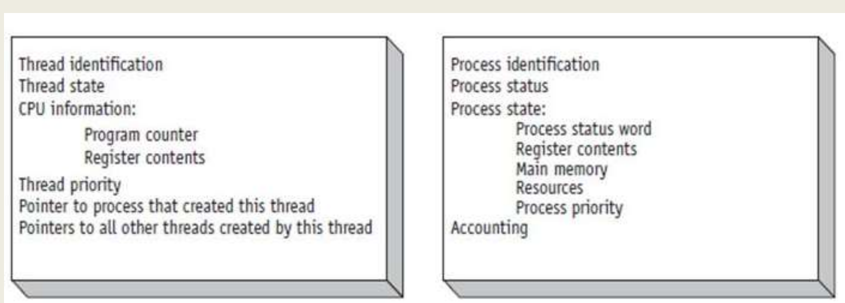
THREADS - 2

- Enables a single process (program) to have multiple "workers"
- Supports independent path(s) of execution within a program
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Code segment, memory, and heap are shared

January 24, 2018 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L6.25

PROCESS AND THREAD METADATA

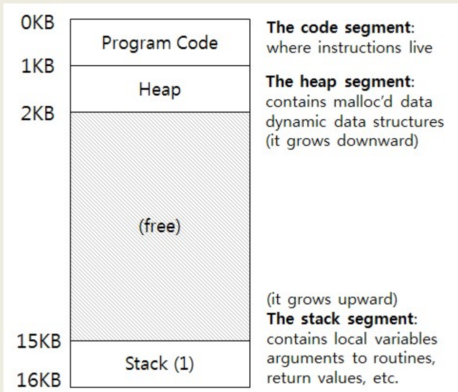
- Thread Control Block vs. Process Control Block



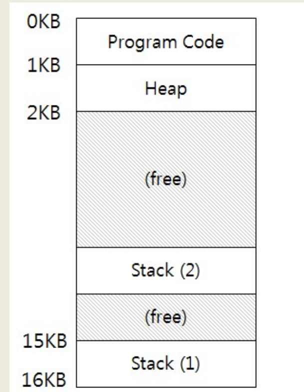
January 24, 2018 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L6.26

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC



A Single-Threaded Address Space



Two threaded Address Space

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'		

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'		

POSSIBLE ORDERINGS OF EVENTS - 3

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

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?

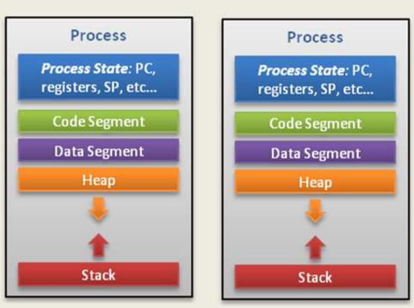
COUNTER EXAMPLE

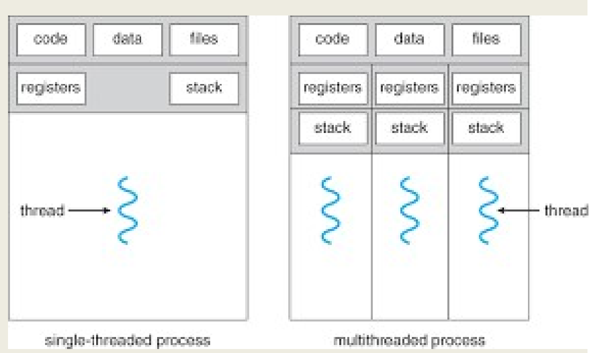
- Show example

- A + B : ordering
- Counter: incrementing global variable by two threads

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 duplicate of code/heap, lightweight execution threads





THREADS - 2

- Enables a single process (program) to have multiple "workers"
- Supports independent path(s) of execution within a program
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Code segment, memory, and heap are shared

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					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

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 produces a ***race condition.***
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***

January 24, 2018	TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma	L6.37

LOCKS

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

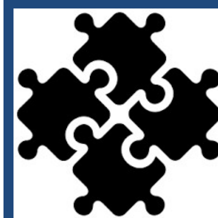
```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- Counter example revisited

January 24, 2018	TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma	L6.38

CHAPTER 27 - LINUX THREAD API



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);
         …
}
```


PASSING A SINGLE VALUE

- Here we "cast" the pointer to pass/return a primitive data type

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

PASSING A SINGLE VALUE

Using this approach on your CentOS 7 VM
How large (in bytes) can the primitive data type be?

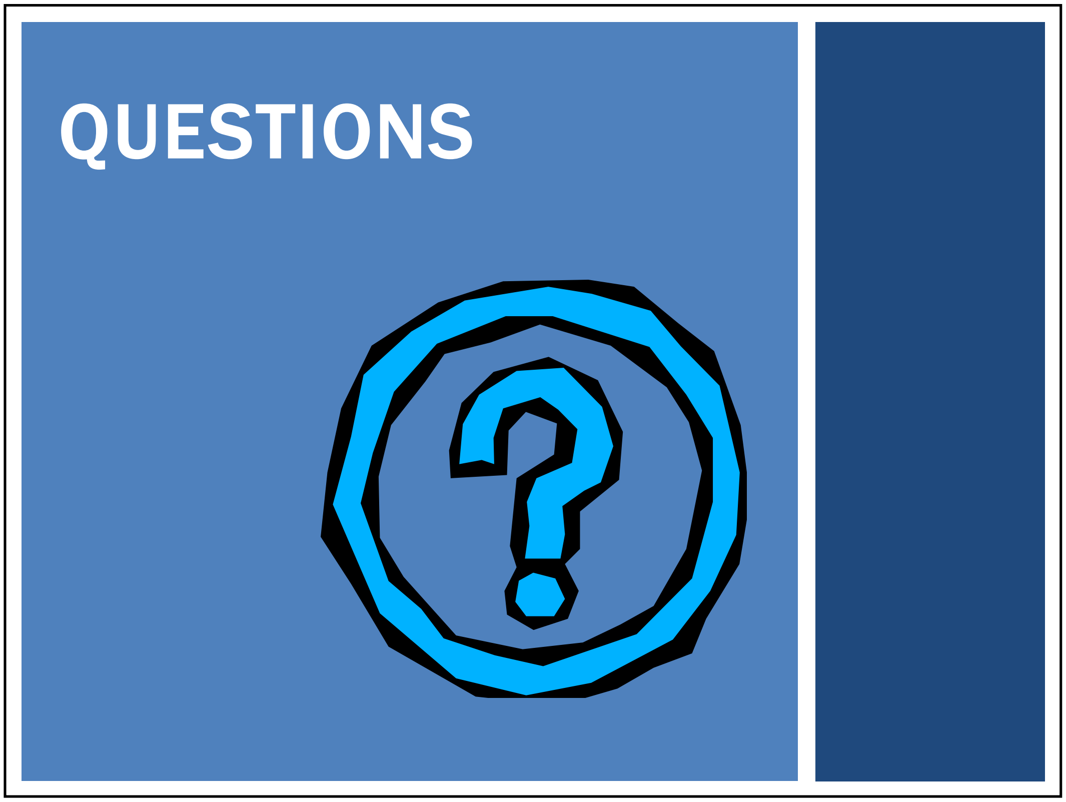
How large (in bytes) can the primitive data type be on a 32-bit operating system?

```
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```


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 ??***


QUESTIONS