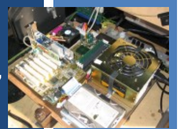
TCSS 422: OPERATING SYSTEMS

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



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

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] 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

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.2

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

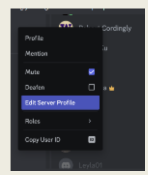
> Office Hours set based on Student Demographics survey feedback
** time may be occasionally rescheduled due to faculty meeting conflicts*

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.3

3

TCSS 422 DISCORD SERVER

- Please join the TCSS 422 A – Spring 2023 Discord Server
- https://discord.gg/hqNanxEQ
- Under Edit Server Profile:
  Please update your 'Server Nickname' to your real name or UW NET ID
  THANK YOU



April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.4

4

OBJECTIVES – 4/20

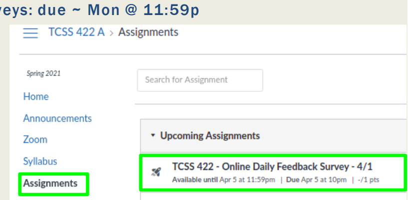
- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.5

5

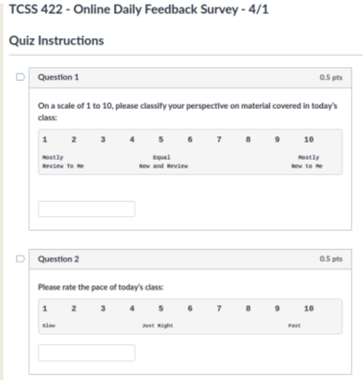
ONLINE DAILY FEEDBACK SURVEY

- 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 20, 2023 | TCSS422: Computer Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.6

6



April 20, 2023 | TCSS422: Computer Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.7

7

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (43 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 7.58 (↑ - previous 7.26)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 6.14 (↑ - previous 5.79)**

April 20, 2023 | TCSS422: Computer Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.8

8

FEEDBACK FROM 4/18

- I wonder if you can explain again how to draw the graph which is the example in class about the priority boost? I'm not sure how to get that result, such as B being the last job.

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.9

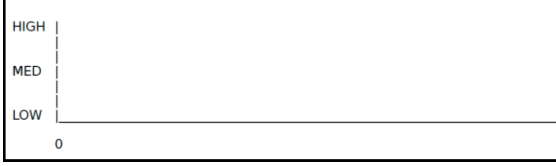
9

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.

***SANITY CHECK*: 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.



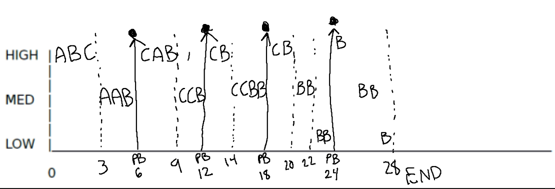
10

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.

***SANITY CHECK*: 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.



11

FEEDBACK - 2

- ***If we will not be working with legacy systems, my question is: What is the usefulness of, and how important is it for our long-term careers to remember these older methods of scheduling that are no longer considered to be relevant or have been replaced with schedulers such as CFS?***
- Understanding fundamental schedulers such as round-robin is core-CS. It is possible to implement a system that will need use this or a similar load distribution algorithm. Understanding traditional schedulers is important when innovating new ones.
- For example, a former student in 2021 implemented a load distribution algorithm based on reinforcement learning a type of deep-learning for a serverless cloud computing platform.
  - One of the baseline schedulers we compared against in the paper was round-robin load distribution: **https://tinyurl.com/4ure98ff**

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.12

12

FEEDBACK - 3

- It is difficult to understand concepts of the Linux Completely Fair Scheduler (CFS) without understanding core scheduling concepts, like RR, fairness, context-switching, job time share, etc.
- Understanding how schedulers have evolved helps us understand the problems encountered and the corresponding solutions

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.13

13

FEEDBACK - 4

- ***I need some clarification on Assignment 0 question 2. What is the purpose of the command?:***

```
./a0.sh > a0.out
```

- ***It keeps saying no file in directory***
- This is how you run the "a0.sh" script and direct the output of the script to go to the text file called 'a0.out'.
- If you have not yet created the 'a0.sh' script in the working directory, there will be no script to run… *i.e. "no file in directory"….*
- Use an editor such as gedit, nano, or vim/vi to create a0.sh and a0_answers.txt

```
gedit a0.sh
```

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.14

14

OBJECTIVES – 4/20

- Questions from 4/18
- **C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28**
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.15

15

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- **Assignment 0 - Due Fri Apr 21 | Assignment 1**
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.16

16

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

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.17

17

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- **Assignment 0 - Due Fri Apr 21 | Assignment 1**
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.18

18

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- **Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)**
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

19

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

20

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

21

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - **Linux Completely Fair Scheduler**
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

22

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 !

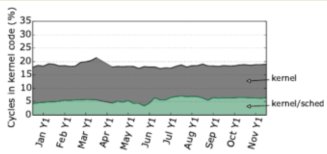


Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

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

23

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/nth 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)

24

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`

25

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
```


26

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/

27

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

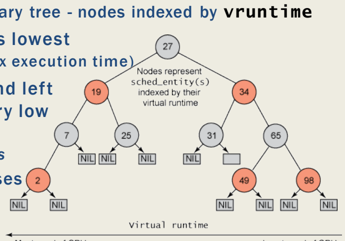
- **LOW** **sched_min_granularity_ns** (timeslice)
  sched_latency_ns
  sched_wakreup_granularity_ns

  CFS features increased context switching → more overhead
  better near-term fairness

28

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



29

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,
};
```


30

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!***

31

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

32

OBJECTIVES – 4/20

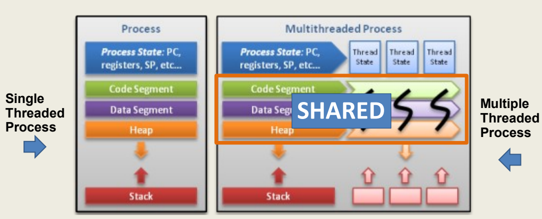
- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

33

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

34

THREADS



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

35

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?***

36

PROCESS AND THREAD METADATA

- Thread Control Block vs. Process Control Block

Thread identification
Thread state
CPU information:
    Program counter
    Register contents
Thread priority
Pointer to process that created this thread
Pointers to all other threads created by this thread

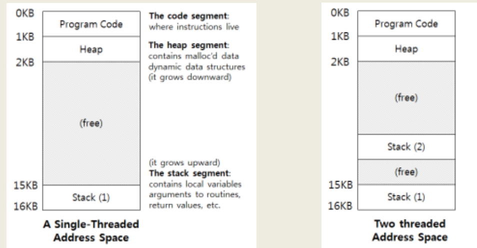
Process identification
Process status
Process state:
    Process status word
    Register contents
    Main memory
    Resources
    Process priority
Accounting

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.37

37

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC



April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.38

38

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.39

39

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.40

40

POSSIBLE ORDERINGS OF EVENTS - 2

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

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.41

41

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?

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.42

42

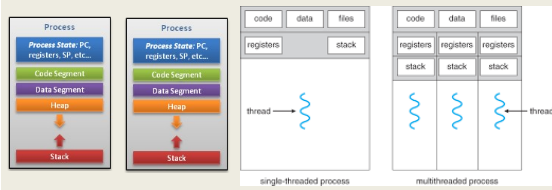
COUNTER EXAMPLE

- 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?***

43

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



44

WE WILL RETURN AT 4:55PM

45

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - **Race condition**
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

46

RACE CONDITION

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

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	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

47

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - **Critical section**
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

48

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***

49

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

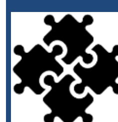
50

COUNTER EXAMPLE

- With locks
  - 2 threads count to 16 million
  - ~1.4 seconds
  - COUNT IS CORRECT – no data loss
- Without locks
  - 2 threads count to 16 million
  - ~0.03 seconds
  - COUNT IS INCORRECT - DATA IS LOST
- Correct version is 46.6 x slower
  - Cost is 16 million Lock & Unlock API calls

51

CHAPTER 27 - LINUX THREAD API



52

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

53

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*)

54

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);
        …
}
```


55

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?

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


56

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 ??***

57

What will this code do?

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

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

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

Data on thread stack

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

How can this code be fixed?

58

How about this code?

```
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;
}
```

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


59

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);
```


60

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.61

61

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - pthread_cond_wait/_signal/_broadcast

April 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.62

62

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.63

63

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.64

64

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.65

65

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 20, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L8.66

66

OBJECTIVES – 4/20

- Questions from 4/18
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 0 - Due Fri Apr 21 | Assignment 1
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: An Introduction
  - Race condition
  - Critical section
- Chapter 27: Linux Thread API
  - pthread_create/_join
  - pthread_mutex_lock/_unlock/_trylock/_timelock
  - **pthread_cond_wait/_signal/_broadcast**

67

CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

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

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

68

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()

69

CONDITIONS AND SIGNALS - 3

- Wait example:

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

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

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

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

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

70

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)

71

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

72

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

73

QUESTIONS

74