TCSS 422: OPERATING SYSTEMS

Introduction to Concurrency, Linux Thread API

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

1

TEXT BOOK COUPON

- 15% off textbook code: **HAPPYPLANET15** (*through Friday Apr 25*)
- https://www.lulu.com/shop/andrea-arpaci-dusseau-and-remzi-arpaci-dusseau/operating-systems-three-easy-pieces-hardcover-version-110/hardcover/product-15gjeeky.html?q=three+easy+pieces+operating+systems&page=1&pageSize=4
- With coupon textbook is only $33.79 + tax & shipping

2

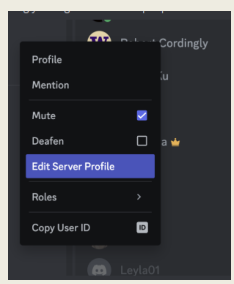
TCSS 422 – OFFICE HRS – SPRING 2025

- **Office Hours plan for Spring (by Zoom):**
- **Monday 11:30am - 12:30p GTA Xinghan**
  - **NEXT MONDAY: Wes**
- **Tuesday 11:30am - 12:30p GTA Xinghan**
- **Wednesday 11:00am - 12:00p Instructor Wes**
- **Friday 12:00pm - 1:00p Instructor Wes or GTA Xinghan**
  - **THIS FRIDAY: Xinghan**
- Instructor is available after class at 6pm in CP 229 each day

3

TCSS 422 DISCORD SERVER

- Please join the TCSS 422 A – Spring 2025 Discord Server
- https://discord.gg/Jh5Cp8TMYn
- Under Edit Server Profile:
  Please update your 'Server Nickname' to your real name or UW NET ID
  THANK YOU



4

OBJECTIVES – 4/24

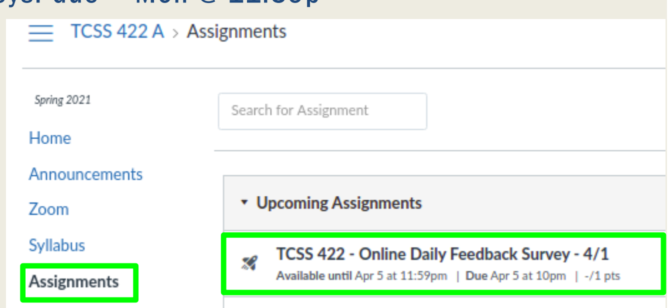
- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- Chapter 26: Concurrency: 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 24, 2025 | TCSS422: Operating Systems [Spring 2025] 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 24, 2025 | TCSS422: Computer Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma | L8.6

6

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

7

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (47 of 63 respondents – 74.6%) :
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.55 (↑ - previous 6.32)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.19 (↑ - previous 4.98)**

8

FEEDBACK FROM 4/22

- ***What would happen in an MLFQ scheduler if there are so many jobs overall that the high priority queue never finishes giving each job a time slice to execute before doing a priority boost?***

- cycle time – total time shared among all jobs in a run queue
- time slice – time an individual job runs for

- From slide 6.50:
  - no rule explicitly describes how the cycle time is divided among jobs
  - No rule explicitly describes how time slice is determine
- Any MLFQ problem having this issue would require rules to describe how this scenario is handled to allow a scheduling graph to be drawn

REVIEW: MLFQ RULE SUMMARY

- The refined set of MLFQ rules:
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

ADDRESSING AN OVERLOADED RUNQUEUE

- One possible way:
  - Cycle time split evenly among jobs in runqueue with no min timeslice
  - For MLFQ, all jobs in runqueue use full timeslice and have priority reduced
  - Not realistic in practice - timeslice becomes too small to be useful
- Another way:
  - Specify min_time_slice (1 ms) per job, and total_cycle_time (10 ms)
  - Job's time_slice = total_cycle_time / jobs_in_runqueue
  - Beyond 10 jobs, other jobs receive no runtime this cycle
  - Jobs receiving no runtime are scheduled first in next cycle
  - Jobs could pile up and experience multi-cycle delays
  - More realistic

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- **Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon**
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

ASSIGNMENT 0 - DUE FRI APR 25 AOE

- Due Friday April 25 AOE (Apr 26 @ 4:59am)
- Grace period: submission ok until Mon Apr 28 @ 4:59 AM
- Late submissions thru Wed Apr 30 @ 4:59am

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- **Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon**
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- **Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)**
- 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

QUIZ 1

- Active reading on Chapter 9 – Proportional Share Schedulers
- Posted in Canvas
- Due Thursday May 1st AOE
- **Link:**
- https://faculty.washington.edu/wlloyd/courses/tcss422/quiz/TCSS422_s2025_quiz_1.pdf

QUIZ 2

- Canvas Quiz – Practice CPU Scheduling Problems
- Posted in Canvas
- Unlimited attempts permitted
- Due Tuesday May 6th AOE
- **Link:**
- https://canvas.uw.edu/courses/1809484/assignments/10329061

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

CATCH UP FROM LECTURE 7

- Switch to Lecture 7 Slides
- Slides L7.56 to L7.61 (Linux CFS)

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

BEYOND CFS → EEVDF SCHEDULER

- Earliest Eligible Virtual Deadline First (EEVDF) Scheduler
  - **Linux kernel version 6.6**, October 29, 2023
  - First described in a research article in 1995
- Like CFS, EEVDF aims to distribute CPU time equally among all runnable tasks with the same priority.
- EEVDF assigns a virtual runtime to each task, creating a "lag" value that is used to determine whether a task has received its fair share of CPU time
  - A task with a positive lag is owed CPU time
  - A task with negative lag has exceeded its timeshare
- EEVDF calculates a virtual deadline (VD) for each task with lag greater or equal to zero
- Task with the earliest virtual deadline is selected to run next
- Virtual deadlines enable latency-sensitive tasks with shorter-time slices to be prioritized more than CFS which helps improve responsiveness
- More info: https://docs.kernel.org/scheduler/sched-eevdf.html

OBJECTIVES – 4/24

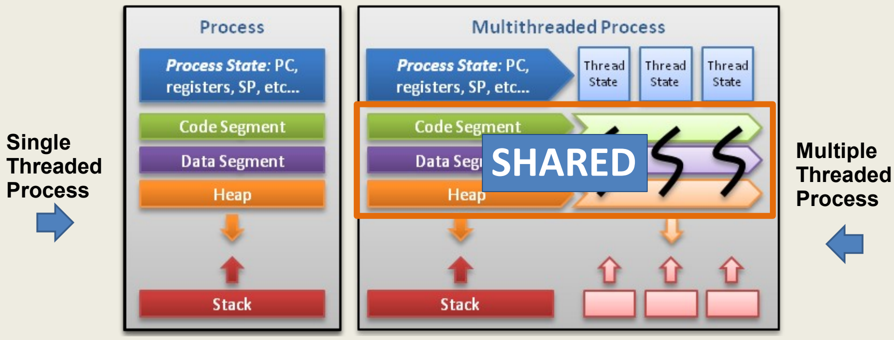
- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- Chapter 9: Proportional Share Schedulers
  - Linux Completely Fair Scheduler
- **Chapter 26: Concurrency: 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

23

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

24

THREADS



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

April 24, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L8.25

25

THREADS - 2

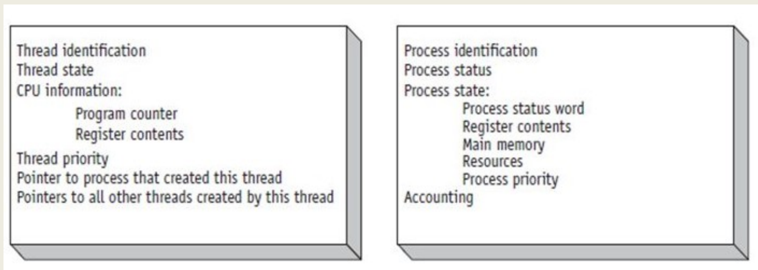
- Enables a single process (program) to have multiple "workers"
  - This is parallel programming...
- Supports independent path(s) of execution within a program *with shared memory ...*
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Threads share code segment,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

April 24, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L8.26

26

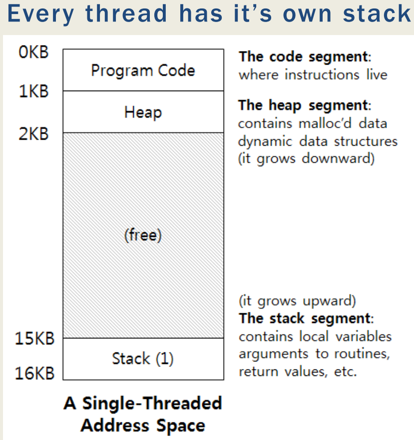
PROCESS AND THREAD METADATA

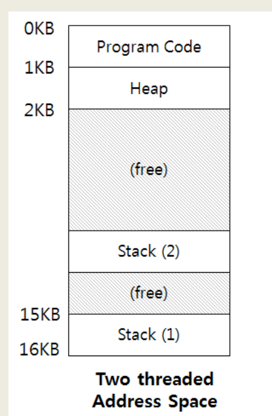
- Thread Control Block vs. Process Control Block



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;
}
```


29

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'		

30

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'		

31

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?

32

COUNTER EXAMPLE

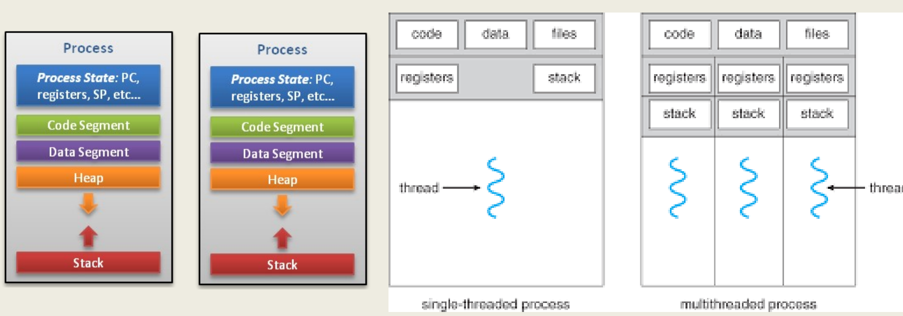
- Counter example
- A + B : ordering
- Counter: incrementing global variable by two threads
- ***Is the counter example embarrassingly parallel?***
- ***What does the parallel counter program require?***

April 24, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L8.33

33

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



April 24, 2025	TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma	L8.34

34

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

RACE CONDITION

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

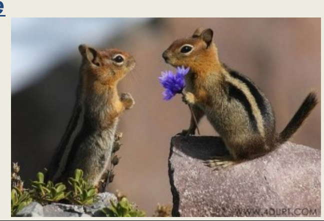
OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax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

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

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

COUNTER EXAMPLE

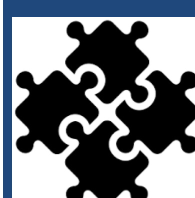
- With locks
  - 2 threads count to 16 million
  - ~1.4 seconds
  - COUNT IS CORRECT – no data loss
- Without locks
  - 2 threads count to 16 million
  - ~0.03 seconds
  - COUNT IS INCORRECT - DATA IS LOST
- Correct version is 46.6 x slower
  - Cost is 16 million Lock & Unlock API calls

WE WILL RETURN AT 5:03PM

April 24, 2025 TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - coma L8.41

41

CHAPTER 27 - LINUX THREAD API



April 24, 2025 TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - T oma L8.42

42

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

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

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

```
        printf("%d\n", m);
```

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 ??***

April 24, 2025 | TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma | L8.47

47

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?

April 24, 2025 | TCSS422: Operating Systems [Spring 2025] School of Engineering and Technology, University of Washington - Tacoma | L8.48

48

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
```


49

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);
```


50

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;
```


OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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

LOCKS

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

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

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


LOCKS - 2

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

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

- Example w/o initialization & error checking

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

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

LOCK INITIALIZATION

- Assigning the constant

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- API call:

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

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

LOCKS - 3

- Error checking wrapper

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

- What if lock can't be obtained?

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

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

OBJECTIVES – 4/24

- Questions from 4/22
- C Tutorial - Pointers, Strings, Exec in C - Due Wed Apr 30 AOE
- Assignment 0 - Due Fri Apr 25 AOE | Assignment 1 soon
- Quiz 1 (Due Thur May 1 AOE) – Quiz 2 (Due Tue May 6 AOE)
- 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**

57

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

58

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()`

59

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.

60

CONDITION AND SIGNALS - 4

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

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

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

PTHREADS LIBRARY

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

SAMPLE MAKEFILE

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

binaries=pthread pthread_int pthread_lock_cond pthread_struct

all: $(binaries)

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

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

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

63

QUESTIONS

64