


TCCS 422: OPERATING SYSTEMS

Concurrency Intro, Linux Thread API

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



April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
Tacoma

OBJECTIVES – 4/21

- **Questions from 4/16**
- **C Tutorial**
- **Active Reading Quiz – Ch. 7**
- **Assignment 0**
- **Assignment 1**
- **Chapter 26: Concurrency: An Introduction**
 - Introduction
 - Race condition
 - Critical section
- **Chapter 27: Linux Thread API**
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/_signal/_broadcast

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.2

MATERIAL / PACE

- 4/16 feedback survey for Lecture 6 – not properly posted on CANVAS
- Survey has been reposted and is available
- Please classify your perspective on material covered in today's class (? respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – ? (?↓ from 7.6)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – ? (?↑ from 5.45)**


April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.3

FEEDBACK FROM 4/16

- **From 4/14: When does software benefit from turnaround time more than response time and vice versa?**
- **Compute-bound batch jobs** benefit from CPU schedulers that improve **turnaround time**
- Examples:
 - Genomic sequencing compute jobs (i.e. DNA analysis/alignment)
 - Earth science models (e.g. weather, climate, hydrology, erosion)
 - Forecast models (e.g. stock market, economics)
- These jobs require LONG uninterrupted access to the CPU
- **Event processing** for user interfaces benefits from CPU scheduler that improve **response time**
- These jobs involve frequent short bursts of computation

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.4

FEEDBACK - 2



- **What are the main advantages to using a multilevel queue for scheduling?**
- Ch. 7 schedulers require knowing job runtime in advance
- In practice, we don't know **how long** jobs require to execute
 - Many unknowns: job resource requirements, state of the system, etc.
- **UW Tacoma MSCSS Student Sonia Xu:** working on MS thesis to predict runtime of bioinformatics jobs on different cloud VMs
- Multi-level feedback queue (MLFQ) adjusts scheduling of jobs dynamically based on job behavior
- A job can actually change its behavior from batch to event-based and back and MLFQ adapts the scheduling!
- CPU schedulers must be adaptive unless having a **crystal ball**

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.5

FEEDBACK - 3

- From email: about the 'Multilevel Feedback Queue Scheduler', I feel I don't fully understand how this algorithm works

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.6

MLFQ - PRIORITY BOOST

- With priority boost
 - Prevents starvation

↑

Q2

Q1

Q0

0 50 100 150 200

With Priority Boost A: B: C:

April 14, 2020

TCS442: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L5.8

CHAPTER 9 QUESTIONS ?		
<ul style="list-style-type: none">■ Chapter 9: Proportional Share Schedulers<ul style="list-style-type: none">▪ Lottery scheduler▪ Ticket mechanisms▪ Stride scheduler▪ Linux Completely Fair Scheduler		
April 21, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L7.10

OBJECTIVES – 4/21		
■ Questions from 4/16		
■ C Tutorial		
■ Active Reading Quiz – Ch. 7		
■ Assignment 0		
■ Assignment 1		
■ Chapter 26: Concurrency: An Introduction		
■ Introduction		
■ Race condition		
■ Critical section		
■ Chapter 27: Linux Thread API		
■ pthread_create/_join		
■ pthread_mutex_lock/_unlock/_trylock/_timelock		
■ pthread_cond_wait/_signal/_broadcast		
April 21, 2020	TCSS422: Operating Systems (Spring 2020) School of Engineering and Technology, University of Washington – Tacoma	L7.12

OBJECTIVES – 4/21

- Questions from 4/16
- C Tutorial
- Active Reading Quiz – Ch. 7
- Assignment 0
- Assignment 1
- Chapter 26: Concurrency: An Introduction
 - Introduction
 - Race condition
 - Critical section
- Chapter 27: Linux Thread API
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/_signal/_broadcast

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.13

OBJECTIVES – 4/21

- Questions from 4/16
- C Tutorial
- Active Reading Quiz – Ch. 7
- Assignment 0
- Assignment 1
- Chapter 26: Concurrency: An Introduction
 - Introduction
 - Race condition
 - Critical section
- Chapter 27: Linux Thread API
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/_signal/_broadcast

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.14

OBJECTIVES – 4/21


- Questions from 4/16
- C Tutorial
- Active Reading Quiz – Ch. 7
- Assignment 0
- Assignment 1
- Chapter 26: Concurrency: An Introduction
 - Introduction
 - Race condition
 - Critical section
- Chapter 27: Linux Thread API
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/_signal/_broadcast

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.15

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION



April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.16

OBJECTIVES – 4/21

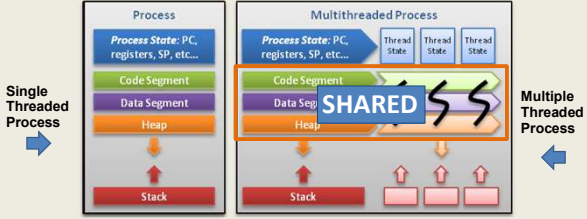
- Questions from 4/16
- C Tutorial
- Active Reading Quiz – Ch. 7
- Assignment 0
- Assignment 1
- Chapter 26: Concurrency: An Introduction
 - Introduction
 - Race condition
 - Critical section
- Chapter 27: Linux Thread API
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/_signal/_broadcast

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.17

THREADS



April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.18

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.19

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 status word

Register contents

Main memory

Resources

Process priority

Accounting

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.20

SHARED ADDRESS SPACE

- Every thread has it's own stack / PC

0KB

1KB

2KB

15KB

16KB

Program Code

Heap

(free)

Stack (1)

The code segment:
where instructions live

The heap segment:
contains malloc'd data
dynamic data structures
(it grows downward)

(it grows upward)
The stack segment:
contains local variables
arguments to routines,
return values, etc.

A Single-Threaded
Address Space

0KB

1KB

2KB

15KB

16KB

Program Code

Heap

(free)

Stack (2)

(free)

Stack (1)

Two threaded
Address Space

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.21

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.22

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
Waits for T1		
Prints 'main: end'		

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.23

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.24

POSSIBLE ORDERINGS OF EVENTS - 3

Int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
What if execution order of events in the program matters?		
Waits for T1	Runs	
	Prints 'A'	
	Returns	
Waits for T2		Immediately returns
Prints 'main: end'		

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.25

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.26

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

Process

Process State: PC, registers, SP, etc...

Code Segment

Data Segment

Heap

Stack

Process

Process State: PC, registers, SP, etc...

Code Segment

Data Segment

Heap

Stack

code

data

files

registers

stack

code

data

files

registers

stack

code

data

files

registers

stack

thread

thread

thread

single-threaded process

multithreaded process

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.27

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 0x049a1c, %eax		105 50 50
	add \$0x1, %eax		108 51 50
Interrupt	save r1's state		100 0 50
	restore T2's state		105 50 50
		mov 0x049a1c, %eax	105 50 50
		add \$0x1, %eax	108 51 50
		mov %eax, 0x049a1c	113 51 51
Interrupt	save r2's state		108 51 50
	restore T1's state		113 51 51
	mov %eax, 0x049a1c		113 51 51

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.28

CRITICAL SECTION

- Code that accesses a shared variable must not be **concurrently** executed by more than one thread
- Multiple active threads inside a **critical section** produce a **race condition**.
- Atomic execution** (all code executed as a unit) must be ensured in **critical sections**
 - These sections must be **mutually exclusive**

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.29

LOCKS

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

1 lock_t mutex;

2

3 lock(&mutex);

4 balance = balance + 1;

5 unlock(&mutex);

Critical section


- Counter example revisited

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.30

CHAPTER 27 - LINUX THREAD API



April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.31

OBJECTIVES – 4/21

- Questions from 4/16
- C Tutorial
- Active Reading Quiz – Ch. 7
- Assignment 0
- Assignment 1
- Chapter 26: Concurrency: An Introduction
 - Introduction
 - Race condition
 - Critical section
- Chapter 27: Linux Thread API
 - pthread_create/_join
 - pthread_mutex_lock/_unlock/_trylock/_timelock
 - pthread_cond_wait/ signal/ broadcast

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.32

THREAD CREATION

- pthread_create

```
#include <pthread.h>

int
pthread_create(    pthread_t*    thread,
                  const pthread_attr_t* attr,
                  void*          (*start_routine)(void*),
                  void*          arg);
```

- thread: thread struct
- attr: stack size, scheduling priority... (optional)
- start_routine: function pointer to thread routine
- arg: argument to pass to thread routine (optional)

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.33

PTHREAD_CREATE – PASS ANY DATA

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.34

PASSING A SINGLE VALUE

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

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

```
9 int rc, m;
10 pthread_create(&p, NULL, mythread, (void *) 100);
11 pthread_join(p, (void **) &m);
12 printf("returned %d\n", m);
13 return 0;
14 }
```

April 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.35

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 21, 2020
TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L7.36

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_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

Segmentation fault (core dumped)

How can this code be fixed?

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.37

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

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.38

ADDING CASTS

- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
pthread_int.c: In function 'main':
pthread_int.c:34:20: warning: passing argument 2 of 'pthread_join'
from incompatible pointer type [-Wincompatible-pointer-types]
pthread_join(p1, &p1val);
```
- Example: uncasted return

```
In file included from pthread_int.c:3:0:
/usr/include/pthread.h:250:12: note: expected 'void **' but argument
is of type 'int **'
extern int pthread_join (pthread_t __th, void **__thread_return);
```

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.39

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 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.40

TCSS 422 WILL RETURN
AT ~2:40PM



April 14, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L5.41

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<100000000; i++) {
        int rc = pthread_mutex_lock(&lock);
        assert(rc==0);
        counter = counter + 1;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.42

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.43

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.44

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 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.45

CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

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



pthread_cond_t datatype

pthread_cond_wait()

- Puts thread to "sleep" (waits) (THREAD is BLOCKED)
- Threads added to >FIFO queues<, 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

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.46

CONDITIONS AND SIGNALS - 2

```
int pthread_cond_signal(pthread_cond_t * cond);
int pthread_cond_broadcast(pthread_cond_t * cond);
```

pthread_cond_signal()

- Called to send a "signal" to wake-up first thread in FIFO "wait" queue
- The goal is to unblock a thread to respond to the signal

pthread_cond_broadcast()

- Unblocks **all** threads in FIFO "wait" queue, currently blocked on the specified condition variable
- Broadcast is used when all threads should wake-up for the signal

Which thread is unblocked first?

- Determined by OS scheduler (based on priority)
- Thread(s) awoken based on placement order in FIFO wait queue
- When awoken threads acquire lock as in pthread_mutex_lock()

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.47

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.

April 21, 2020

TCCS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.48

CONDITION AND SIGNALS - 4

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

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

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

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.49

PTHREADS LIBRARY

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

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.50

SAMPLE MAKEFILE

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

binaries=pthread pthread_int pthread_lock_cond pthread_struct
all: $(binaries)

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

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


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

April 21, 2020

TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

L7.51

QUESTIONS



WILL RETURN IN A FEW MINUTES

