CS 471 Operating Systems

Yue Cheng

George Mason University
Fall 2019

Review: Threads

Threads

- **Processes** vs. **threads**
  - Parent and child processes do not share address space
  - Inter-process communication w/ message passing or shared memory

  - Threads created by one process share address space, open files, global variables, etc.
  - Much cheaper and more flexible inter-thread communication and cooperation

A Simple Example Using `pthread`

```c
#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;
}
```

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	

Thread Trace 1

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		

Thread Trace 1

main	Thread 1	Thread2
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

Thread Trace 1

main	Thread 1	Thread2
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"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

What would a 3rd thread trace look like?

Synchronization

- Race Conditions
- The Critical Section Problem
- Synchronization Hardware and Locks

Threaded Counting Example

```c
#include <stdio.h>
#include "common.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char*) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
$ git clone https://github.com/tddg/demo-ostep-code
$ cd demo-ostep-code/threads-intro
$ make
$ ./t1 <loop_count>
```

Try it yourself

Back-to-Back Runs

Run 1…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 10706438)

Run 2…
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 11852529)

What exactly Happened??

What exactly Happened??

```
% otool -t -v thread_rc
```
[Mac OS X]

```
% objdump -d thread_rc
```
[Linux]

...

0000000100000d52	movl	0x2f8e %eax
0000000100000d58	addl	$0x1, %eax
0000000100000d5b	movl	%eax, 0x2f8e

...

```
counter = counter + 1;
```

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51

Time

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt			
Save T1's state			
Restore T2's state			
		movl 0x2f8e, %eax	
		addl $0x1, %eax	
		movl %eax, 0x2f8e	

Time

Concurrent Access to the Same Memory Address

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1's state Restore T2's state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51

Time

Concurrent Access to the Same Memory Address

Time

OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1's state Restore T2's state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51
Interrupt Save T2's state Restore T1's state			

Concurrent Access to the Same Memory Address

Time / OS	Thread 1	Thread 2	Value
	Enter into critical section		
	movl 0x2f8e, %eax		50
	addl $0x1, %eax		51
Interrupt Save T1’s state Restore T2’s state			
		movl 0x2f8e, %eax	50
		addl $0x1, %eax	51
		movl %eax, 0x2f8e	51
Interrupt Save T2’s state Restore T1’s state			
	movl %eax, 0x2f8e		

Concurrent Access to the Same Memory Address

Time	OS	Thread 1	Thread 2	Value
		Enter into critical section		
		movl 0x2f8e, %eax		50
		addl $0x1, %eax		51
	Interrupt Save T1's state Restore T2's state			
			movl 0x2f8e, %eax	50
			addl $0x1, %eax	51
			movl %eax, 0x2f8e	51
	Interrupt Save T2's state Restore T1's state			
		movl %eax, 0x2f8e		**51**

Concurrent Access to the Same Memory Address

Time	OS	Thread 1	Thread 2	Value
		Enter into critical section		
		movl 0x2f8e, %eax		50
		addl $0x1, %eax		51
	Interrupt Save T1's state Restore T2's state			
			movl 0x2f8e, %eax	50
			addl $0x1, %eax	51
			movl %eax, 0x2f8e	51
	Interrupt Save T2's state Restore T1's state			
		movl %eax, 0x2f8e		**51**

Race Conditions

- Observe: In a **time-shared** system, **the exact instruction execution order** cannot be predicted
  - Deterministic vs. **Non-deterministic**

- Any possible orders can happen, which result in different output across runs

Race Conditions

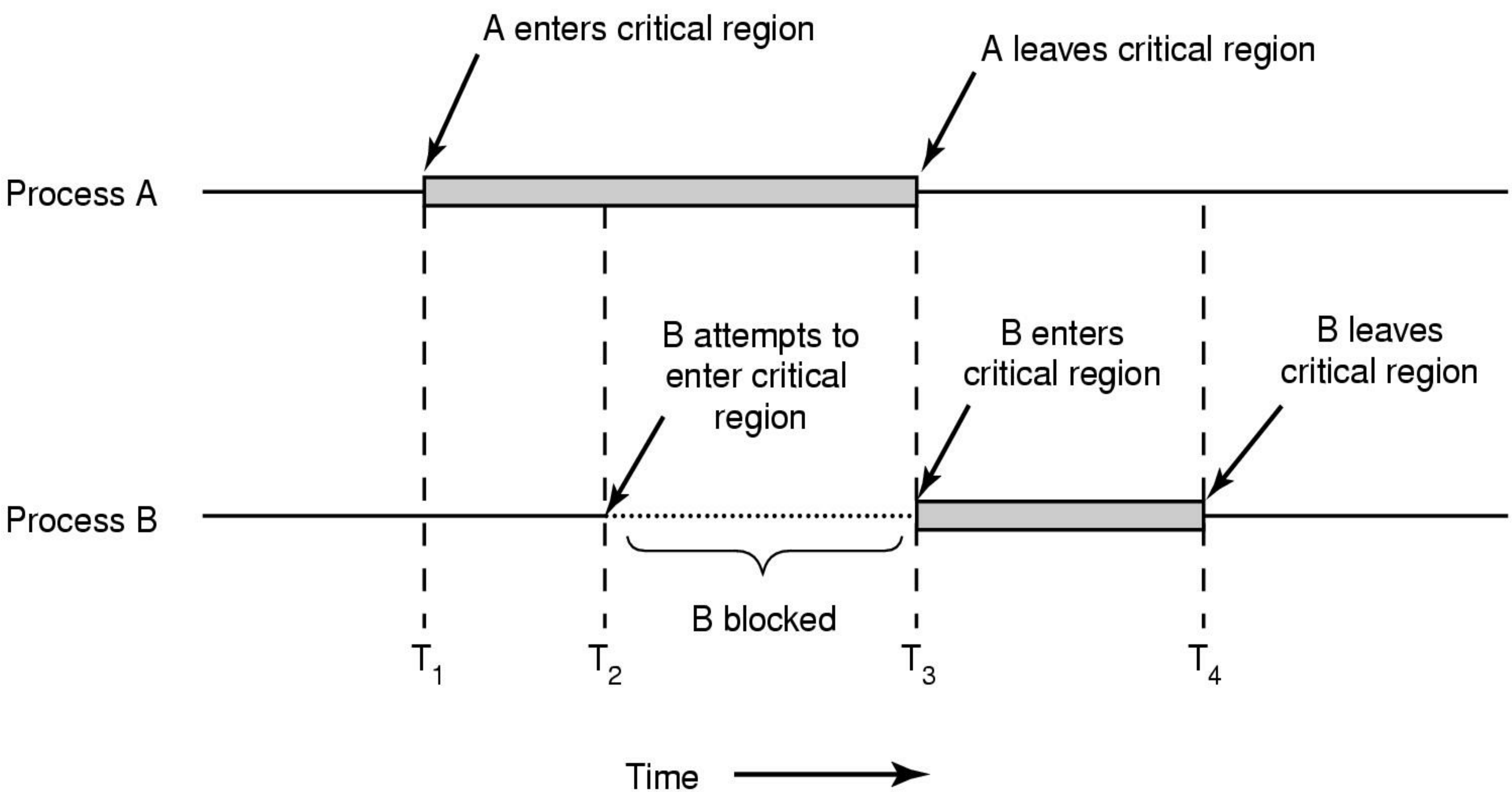
- Situations like this, where multiple processes are writing or reading some shared data and the final result depends on who runs precisely when, are called **race conditions**
  - A serious problem for any concurrent system using shared variables

- Programmers must make sure that some **high-level** code sections are executed **atomically**
  - Atomic operation: It completes in its entirety without worrying about interruption by any other potentially conflict-causing process

The Critical-Section Problem

- *N* processes/threads all competing to access the shared data
- Each process/thread has a code segment, called critical section (critical region), in which the shared data is accessed
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in that critical section
- The execution of the critical sections by the processes must be mutually exclusive in time

Mutual Exclusion



Solving Critical-Section Problem

Any solution to the problem must satisfy **four conditions**!

Mutual Exclusion:

No two processes may be simultaneously inside the same critical section

Bounded Waiting:

No process should have to wait forever to enter a critical section

Progress:

No process executing a code segment unrelated to a given critical section can block another process trying to enter the same critical section

Arbitrary Speed:

No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed)

Using **Lock** to Protect Shared Data

- Suppose that two threads A and B have access to a shared variable "`balance`"

Thread A:

```
balance = balance + 1
```

Thread B:

```
balance = balance + 1
```

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks

- A lock is a **variable**

- Two states
  - Available or free
  - Locked or held

- **`lock()`:** tries to acquire the lock
- **`unlock()`:** releases the lock that has been acquired by caller

Building a Lock

- Needs help from hardware + OS
- A number of hardware primitives to support a lock
- Goals of a lock
  - Basic task: **Mutual exclusion**
  - Fairness
  - Performance

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

→ **A spin lock** (`; // spin-wait (do nothing)`)

First Attempt: A Simple Flag

- How about just using `Loads/Stores` instructions?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

A spin lock

What's the problem?

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	

First Attempt: A Simple Flag

Flag is 0 initially

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)

First Attempt: A Simple Flag

Flag is set to 1 by T2

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

First Attempt: A Simple Flag

Flag is set to 1 again! Two threads both in Critical Section

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Reason:
Lock operation is not atomic!
And therefore, no mutual exclusion!

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
        critical section;
    unlock()
        remainder section;
} while (1);
```

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Getting Help from the Hardware

- One solution supported by hardware may be to use interrupt capability

```
do {
    lock()
        critical section;
    unlock()
        remainder section;
} while (1);
```

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Are we done??

Synchronization Hardware

- Many machines provide special hardware instructions to help achieve mutual exclusion

- The `TestAndSet(TAS)` instruction tests and modifies the content of a memory word **atomically**

- TAS returns old value pointed to by `old_ptr` and updates said value to `new`

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Operations performed atomically!

Mutual Exclusion with **TAS**

- Initially, `lock`'s flag set to 0

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

→ **A correct spin lock**

Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "`lock`" variable that uses busy waiting is called a spin lock
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness?
  - Performance?

Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary "`lock`" variable that uses busy waiting is called a spin lock
  - Processes that find the lock unavailable "spin" at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness? (**A**: No. Heavy contention may cause starvation)
  - Performance? (**A**: Busy waiting wastes CPU cycles)

A Simple Approach: Just Yield ~~(Win)~~!

- When you are going to Spin, just give up the CPU to another process/thread

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Lock Worksheet