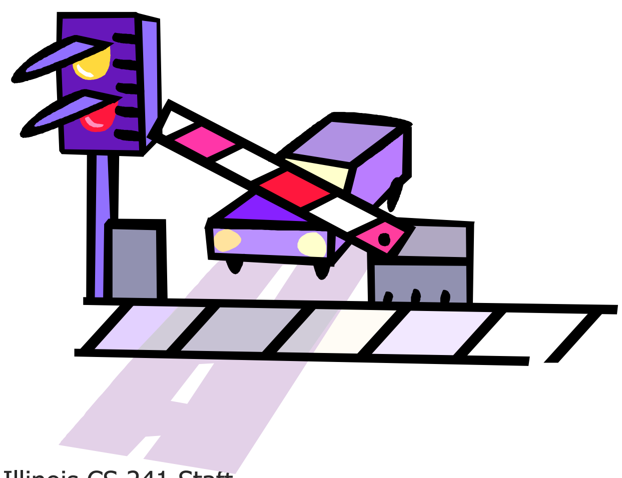
Introduction to Synchronization



Conflict exam

- Tuesday March, 11th morning @ 8am

- If you have Physics 214 Final on March 10th, you can take cs241 conflict exam next day.
- You need to register for the conflict exam by sending an email to cs241help with subject “conflict exam” and explaining the reason of your conflict
- We will double check each request making sure it is a valid conflict. Don’t fake it, we will find out…

Do we really need synchronization with threads?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
#define NUM_THREADS 2


int cnt = 0;

void * worker( void *ptr ) {
   int i;
   for (i = 0;
        i < 50000; i++)
      cnt++;
   pthread_exit(NULL);
}
```

```c
int main(void) {
   pthread_t threads[NUM_THREADS];
   int i, res;

   for (i = 0; i < NUM_THREADS; i++)  {
      res = pthread_create(&threads[i],
                 NULL, worker, NULL);
   }
   for (i = 0; i < NUM_THREADS; i++) {
      res = pthread_join(threads[i], NULL);
   }
    /* Print result */
    printf("Final value: %d\n", cnt);
}
```

What is the output?

Do threads conflict in practice?

- If everything worked...

```
$ ./20-counter
Final value: 100000
```

```
Output
---------------------------------------
mcaccamo$ ./test1
Final value: 62354
mcaccamo$ ./test1
Final value: 57718
mcaccamo$ ./test1
Final value: 55632
mcaccamo$ ./test1
Final value: 50801
```

Do threads conflict in practice?

- Q: What do you think is the minimum final value?

Deconstructing the Counter

C code for counter loop for thread **i**

```
for (i=0; i < 50000; i++)
    cnt++;
```

Corresponding assembly code

```
        movl  (%rdi),%ecx
        movl  $0,%edx
        cmpl  %ecx,%edx
        jge  .L13
.L11:
        movl  cnt(%rip),%eax
        incl  %eax
        movl  %eax,cnt(%rip)
        incl  %edx
        cmpl  %ecx,%edx
        jl  .L11
.L13:
```

Head (H_i)

Load **cnt** (L_i)
Update **cnt** (U_i)
Store **cnt** (S_i)

Critical section: reading or writing shared variable

Tail (T_i)

What is wrong with the shared counter?

- We just saw that processes / threads can be preempted at arbitrary times
  - The previous example might work, or not

Shared state:	Thread 1:	Thread 2:
`int x=0;`	`x++;`	`x++;`

Let's look at possible concurrent interleaving of thread code

Incrementing Variables

- How is **`x++`** compiled?
- A possible sequence of compiled pseudo-code is:

```
register1 = x                 (atomic)
register1 = register1 + 1     (atomic)
x = register1                 (atomic)
```

What could happen?

```
x++:    r1 = x
        r1 = r1 + 1
        x = r1
```

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	`r1`	`r2`	`x`
`r1 = x`		0		0
`r1 = r1+1`		1		0

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x = r1`		1		1

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x  = r1`		1		1
	`r2 = x`		1	1

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
r1 = x		0		0
r1 = r1+1		1		0
x = r1		1		1
	r2 = x		1	1
	r2 = r2+1		2	1

This could happen...

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x = r1`		1		1
	`r2 = x`		1	1
	`r2 = r2+1`		2	1
	`x = r2`		2	2

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	`r1`	`r2`	`x`
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x = r1`		1	1	1

But this could happen too!

Thread 1: `x++;`	Thread 2: `x++;`	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x  = r1`		1	1	1
	`x  = r2`	1	1	1

Introducing: Critical Section

```
void * worker(void *ptr)  {
    while (true) {
        ENTER CRITICAL SECTION
        Access shared variable;
        LEAVE CRITICAL SECTION
        Do other work
    }
}
```

- Instructions inside the critical section should not be interleaved with other threads' critical section.
- How do you enforce mutually exclusive execution of critical sections?

➔ Classic solution: Dijkstra's P and V operations on semaphores

Introducing: P(s) and V(s)

- A semaphore **s** is a non-negative integer that can only be manipulated by P and V operations
- **P(s)**: (simple version)

  while(1) {**[if (s>0) {s--; break;}]** usleep(usec); }
- **V(s)**: (simple version)

  [s++;]
- OS guarantees that instructions within brackets [ ] are executed atomically

Introducing: P(s) and V(s)

```
Semaphore s=1; // semaphore initialized as unlocked

Thread {
    while (true) {
        P(s)
        Access shared variable; ← critical section
        V(s)
        Do other work
    }
}
```

- Semaphore **s** guarantees the mutually exclusive execution of each critical section it is protecting.

→ it prevents race conditions among concurrent threads

Critical Section Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

Critical Section Requirements

- Mutual Exclusion
  - At most one thread in critical section
  - No other thread may execute within the critical section while a thread is in it
- Progress
- Bounded Wait

Critical Section Requirements

- Mutual Exclusion
- Progress
  - If no thread is executing inside its critical section and some threads are trying to get into their critical section, then one of them should be able to enter its critical section
- Bounded Wait

Critical Section Requirements

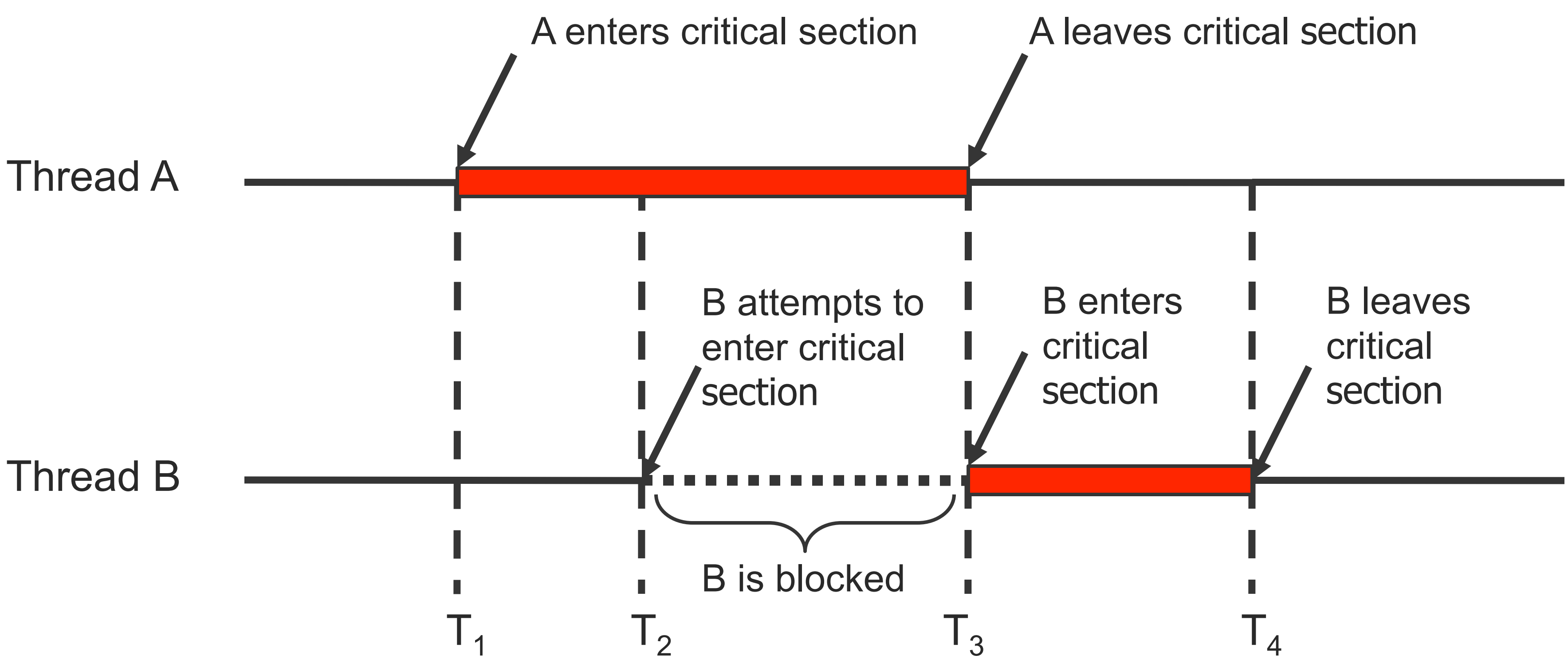
- Mutual Exclusion
- Progress
- Bounded Wait
  - A thread requesting entry to a critical section should only have to wait for a bounded number of other threads to enter and leave the critical section

Critical Section Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

Must ensure these requirements without assumptions about number and speed of CPUs, or scheduling policy!

Summarizing Critical Sections



Threads A and B have both a critical section guarded by the same semaphore **S**