

Synchronization

CS 241

```
int X = 0; /**< Global variable used for counting. */

/**
 * Increments global variable X by 1 a total of TOTAL times.
 */
void* count_up( void *ptr )
{
    int i = 0;
    for (i=0; i < TOTAL; i++)
        X++;

    return NULL;
}

void main()
{
    pthread_t tid[2]; int i;
    for (i=0; i<2; i++) {
        pthread_create(&tid[i], NULL, count_up, NULL); }
    for (i=0; i<2; i++) {
        pthread_join(tid[i], NULL); }

    printf("%d\n", X);
}
```

Assembly Code

C code for counter loop for thread i

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

Corresponding assembly code

<pre> movl (%rdi), %ecx movl \$0, %edx cmpl %ecx, %edx jge .L13</pre>	}	Head (H_i)
<pre>--- .L11: movl cnt(%rip), %eax incl %eax movl %eax, cnt(%rip)</pre>		
<pre>--- incl %edx cmpl %ecx, %edx jl .L11</pre>	}	Tail (T_i)
<pre>.L13:</pre>		

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

Critical Section

- A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Assembly Code

- One possible ordering...

Thread 1	Thread 2	%eax ₁	%eax ₂	cnt
H		-	-	0
L		0	-	0
U			-	0
S			-	1
	H		-	1
	L		1	1
	U		2	1
	S		2	2
	T		2	2
T			-	2



Thread 1
critical section



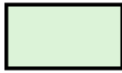
Thread 2
critical section


OK!

Assembly Code

- A second possible ordering...

Thread 1	Thread 2	%eax ₁	%eax ₂	cnt
H		-	-	0
L		0	-	0
U			-	0
	H		-	0
	L		0	0
S				
T		-		
	U	-		
	S	-		
	T	-	-	

 Thread 1
critical section

 Thread 2
critical section

Oops!

```
int X = 0; /**< Global variable used for counting. */

/**
 * Increments global variable X by 1 a total of TOTAL times.
 */
void* count_up( void *ptr )
{
    int i = 0;
    for (i=0; i < TOTAL; i++)
    {   atomic {   X++;   }   }
        /* atomic doesn't exist, how do we simulate it? */

    return NULL;
}

void main()
{
    pthread_t tid[2]; int i;
    for (i=0; i<2; i++) {
        pthread_create(&tid[i], NULL, count_up, NULL); }
    for (i=0; i<2; i++) {
        pthread_join(tid[i], NULL); }

    printf("%d\n", X);
}
```

Mutex

- The pthread library provides us with a mutex, a variable that is “locked” or “unlocked”.
- Key operation: **pthread_mutex_lock()**
 - When **locked**: wait until the variable is unlocked before locking and continuing.
 - When **unlocked**: lock the variable and continue.
- Also: **pthread_mutex_unlock()**


```
int X = 0; /**< Global variable used for counting. */
pthread_mutex_t mutex;

/**
 * Increments global variable X by 1 a total of TOTAL times.
 */
void* count_up( void *ptr )
{
    int i = 0;
    for (i=0; i < TOTAL; i++) {
        pthread_mutex_lock(&mutex);
        X++;
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

void main()
{
    pthread_mutex_init(&mutex, NULL);
    ...
    pthread_mutex_destroy(&mutex);
}
```

In Hardware...

- Every system has a different way of implementing the atomic nature of **pthread_mutex_lock()**.

- Fundamentally, we abstract it into a function that tests and sets a lock variable:

```
/**
 * A C-code representation of an atomic hardware instruction.
 *
 * If the value contained in lock is UNLOCKED (0), we LOCK (1) it
 * and return SUCCESS (0). If the value was LOCKED (1), we
 * return FAILURE (1).
 *
 * In x86, this is done via an XCHG or LOCK opcodes.
 */
int testandset(int *lock)
{
    if (*lock == 0)        /* If our lock is unlocked... */
    {
        *lock = 1;        /* ...lock it, */
        return 0;         /* ...and return that we locked it. */
    }
    else
        return 1;        /* Otherwise, we can't lock it. */
}
```

Two Terms in Synchronization

- **Mutual Exclusion:** At most, only one thread is accessing the critical section at any time.
- **Progress:** If a thread wants to enter the critical section and no other thread is in the critical section, it must have access to the critical section.

Violations in Synchronization

- **Violation of Mutual Exclusion:** At any time, two or more threads have access to the critical section.
- **Violation of Progress:** A thread is indefinitely blocked from entering the critical section when no other thread is executing the critical section.

Synchronization Examples

- In each example:
 - Assume x , $x1$, and $x2$ are initially set to 0.
 - Each thread may run any number of times, in any order.
 - One thread may finish before the other thread (eg: the system may only have one of the two threads executing after a period of time).

Thread 1:

```
while (x > 0) { }  
x++;  
/* critical section */  
x--;
```

Thread 2:

```
while (x > 0) { }  
x++;  
/* critical section */  
x--;
```

Mutual Exclusion?

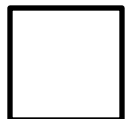
Progress?

Thread 1:

```
while (x1 != 0) { }  
x2 = 1;  
/* critical section */  
x2 = 0;
```

Thread 2:

```
while (x2 != 0) { }  
x1 = 1;  
/* critical section */  
x1 = 0;
```



Mutual Exclusion?



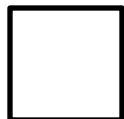
Progress?

Thread 1:

```
x2 = 1;  
while (x1 != 0) { }  
/* critical section */  
x2 = 0;
```

Thread 2:

```
x1 = 1;  
while (x2 != 0) { }  
/* critical section */  
x1 = 0;
```



Mutual Exclusion?



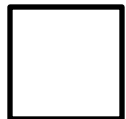
Progress?

Thread 1:

```
if (x % 2 == 1)
{
    /* critical section */
    x = 2;
}
```

Thread 2:

```
if (x % 2 == 0)
{
    /* critical section */
    x = 1;
}
```



Mutual Exclusion?



Progress?

Thread 1:

```
while (x == 1) {}  
x = 1;  
/* critical section */
```

Thread 2:

```
while (x == 1) {}  
x = 1;  
/* critical section */
```

Mutual Exclusion?

Progress?

Thread 1:

```
while (testandset(&x)) { }  
/* critical section */  
x = 0;
```

Thread 2:

```
while (testandset(&x)) { }  
/* critical section */  
x = 0;
```

Mutual Exclusion?

Progress?

Thread 1:

```
while (pthread_mutex_lock(&m))  
    { }  
/* critical section */  
pthread_mutex_unlock(&m);
```

Thread 2:

```
while (pthread_mutex_lock(&m))  
    { }  
/* critical section */  
pthread_mutex_unlock(&m);
```

Mutual Exclusion?

Progress?

Mutexes are limited...

- How do we allow two threads to enter a code region (as opposed to 1)?
- How do we allow the same thread to enter a code region multiple times (but not any other threads)?
 - Equivalent to the **synchronized** keyword in Java.
- How do we allow any general condition?

Conditional Variables

- Idea:
 - Any number of threads can **_wait()** for a condition.
 - When the condition has changed, the thread changing the condition **_signal()**s one thread or **_broadcast()**s to all the threads.
 - The condition itself is contained a critical section, allowing only one thread to access it.

pthread_cond_wait()

- In **pthread_cond_wait()**:
 - Takes two arguments:
 - **pthread_cond_t**: The conditional variable.
 - **pthread_mutex_t**: The mutex for the critical section.
 - When **pthread_cond_wait()** is called:
 - Unlocks the mutex, (so the mutex must be locked)
 - Waits for a signal, (via **_signal()** or **_broadcast()**)
 - Locks the mutex before running again

Conditional Variables

- Scenario #1:
 - Block all threads until at least four threads arrive. Upon the fourth thread, allow all threads (blocked and future) to continue.

```
pthread_mutex_t mutex; /* _mutex_init() called elsewhere */
pthread_cond_t cond; /* _cond_init() called elsewhere */
int threads_seen = 0;
```

```
void roadblock_four()
{
    pthread_mutex_lock(&mutex);
    threads_seen++;
    pthread_cond_broadcast(&cond);

    while ( threads_seen < 4 )
        pthread_cond_wait(&cond, &mutex);

    pthread_mutex_unlock(&mutex);
}
```

Conditional Variables

- Scenario #2:
 - Create a blocking queue data structure.
 - Any operation to **_dequeue()** should block until data is available.

```
typedef struct _blockingqueue_t
{
    queue_t *q;    /* Standard queue. */
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} blockingqueue_t;

void *blockingqueue_dequeue(blockingqueue_t *q)
{
    pthread_mutex_lock( &(q->mutex) );

    while ( queue_size(q->q) == 0 )
        pthread_cond_wait( &(q->cond), &(q->mutex) );

    void *ret = queue_dequeue( q->q );
    pthread_mutex_unlock( &(q->mutex) );

    return ret;
}

void blockingqueue_enqueue(blockingqueue_t *q, void *item)
{
    pthread_mutex_lock( &(q->mutex) );
    queue_enqueue(q->q, item);
    pthread_cond_signal( &(q->cond) );
    pthread_mutex_unlock( &(q->mutex) );
}
```