

CSE 344 System Programming

Week 10

Synchronizing with threads

- Mutexes
- Condition variables
- Classic problems and their solutions in terms of threads

Thread synchronization

Threads are a beautiful concept, they are:

- “**lighter**” than processes in terms of memory use
- **faster** in terms of creation and context switching
- **easier** to establish inter-thread communication as they share the process memory.

However, all this comes at a cost, and that cost is solving the associated synchronization problems. You have already seen (counting) semaphores. You can use Posix/IPC semaphores with threads as well to solve **some** of your synchronization problems.

Posix threads offer us two more synchronization tools that enable us to solve (theoretically) **any** synchronization problem: **mutexes** and **condition variables**.

Thread synchronization

When faced with a critical section (i.e. a block of code executed in parallel, posing a synchronization risk), there is **no way** of avoiding an eventual rescheduling by the kernel.

All we can (and should) do, is make sure no other thread/process accesses the critical section while there is already a thread/process inside it.

We achieved this with semaphores by surrounding critical sections with wait and post calls:

```
wait(s)
//critical section, common variable, common data structure, etc
post(s)
```

Thread synchronization

Threads employ **mutexes** (**mutual exclusion**) for the same purpose.

A mutex m can be in either of two states: **locked (owned)** or **unlocked (not owned)**. It can change states through `lock()` and `unlock()` calls.

Once a mutex m is locked/owned by a thread t_1 , all other threads trying to lock m will block, until t_1 unlocks m .

Thread t_1

`lock(m)`

`push(v)`

`unlock(m)`

Thread t_2

`lock(m)`

`v = pop()` `// critical section`

`unlock(m)`

Thread synchronization

Creating a mutex (**dynamically**, with eventually custom attributes)

```
pthread_mutex_t mutex;  
pthread_mutex_init (&mutex, NULL);    // NULL for default attrs.
```

```
#include <pthread.h>  
  
int pthread_mutex_init(pthread_mutex_t * m ,  
                        const pthread_mutexattr_t * attr);
```

Returns 0 on success, or a positive error number on error

or **statically** (with default attributes; convenient for global variables)

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Thread synchronization

Attention: calling `pthread_mutex_init()` on an already initialized mutex leads to undefined behavior!

Once you no longer need to use a mutex, call `pthread_mutex_destroy()`

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t * mutex );
```

Returns 0 on success, or a positive error number on error

It is not necessary to call `pthread_mutex_destroy()` on a mutex that was statically initialized using `PTHREAD_MUTEX_INITIALIZER`

!!! Make sure **it's not locked** when you call it !!!

Thread synchronization

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t * m );
```

```
int pthread_mutex_unlock(pthread_mutex_t * m );
```

Both return 0 on success, or a positive error number on error

Assuming that m is a mutex, and t_1 , t_2 are threads using it.

What happens when ->	t1 calls lock	t1 calls unlock	t2 calls lock	t2 calls unlock
While m is owned by t_1	depends on the type of m	m is unlocked	t_2 is blocked	undefined/error depending on type
while m is not owned	t_1 locks m	undefined/error depending on type	t_2 locks m	undefined/error depending on type

Thread synchronization

Rules (the exact behavior depends on the type of the mutex):

- A single thread may not lock the same mutex twice.
- A thread may not unlock a mutex that it doesn't currently own (i.e., that it did not lock).
- A thread may not unlock a mutex that is not currently locked.

Normal/Fast: default, no checks, deadlocks in case of double lock (waits for itself)

Recursive: keeps count of locks, and only unlocks when the count is zero

Errorcheck: slower, but returns errors on all 3 scenarios

Thread synchronization

Example on how to set the mutex type:

```
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
int s, type;

s = pthread_mutexattr_init(&mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_init");

s = pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_ERRORCHECK);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_settype");

s = pthread_mutex_init(&mtx, &mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutex_init");

s = pthread_mutexattr_destroy(&mtxAttr);          /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_mutexattr_destroy");
```

Thread synchronization

Common errors with mutexes

- Different orders of locks across threads

Thread A

```
1. pthread_mutex_lock(mutex1);  
2. pthread_mutex_lock(mutex2);  
blocks
```

Thread B

```
1. pthread_mutex_lock(mutex2);  
2. pthread_mutex_lock(mutex1);  
blocks
```

- Relocking an already locked mutex
- Trying to unlock another thread's mutex

Tip: you can check safely whether a mutex is locked using `pthread_mutex_trylock()`; it returns zero if the lock is acquired or an immediate error otherwise.

Thread synchronization

Important: is a mutex **identical** to a binary semaphore?

No. Because conversely to semaphores, mutexes revolve around the core concept of ownership. Semaphores don't have an owner, they can be increased through “`post()`” by any thread/process. A mutex can be unlocked/released **only** by its owner.

Mutexes are excellent for controlling access to critical sections.

Semaphores are great for implementing shared counters.

In practice however we encounter far more complex situations; for those we have **condition variables**.

Thread synchronization

Examples of real-world synchronization problems:

- wait for t to acquire the value of 17
- wait for $x+y > 13$
- wait for some buffer to fill up to at least 85%... etc.

Common denominator of all problems: **wait for a certain condition or logical predicate to be satisfied.**

A condition variable c possesses three main methods:

- 1) $cwait(c,?)$: blocks the thread ($?$: is a hidden parameter, *more on this soon*)
- 2) $signal(c)$: awakens a thread blocked on c because it called $cwait(c,?)$
- 3) $broadcast(c)$: awakens all threads blocked on c because they called $cwait(c,?)$

Thread synchronization

A condition variable **knows nothing** about the condition that it's waiting.

Example

T1, T2 and T3: threads, c: condition variable, $x=y=0$

T3 must not advance unless $x+y > 13$

<u>T1</u>	<u>T2</u>	<u>T3</u>
<code>x += 5;</code>	<code>y += 7;</code>	<code>while (!(x + y > 13))</code>
<code>signal(c);</code>	<code>signal(c);</code>	<code>cwait(c, ?);</code>

With every call of `signal()`, T3 checks whether the condition is satisfied; if yes, it continues, otherwise it goes back to sleep.

Thread synchronization

T3: test condition	FALSE	cwait and sleep again	x=0, y=0
T1: x+=5	signal	T3 awake and ready	x=5, y=0
...			
T3: test condition	FALSE	cwait and sleep again	x=5, y=0
T2: y+=7	signal	T3 awake and ready	x=5, y=7
T3: test condition	FALSE	cwait and sleep again	x=5, y=7
T1: x+=5	signal	T3 awake and ready	x=10, y=7
T3: test condition	TRUE	exit loop	x=10, y=7

The while clause could be about any logical predicate, that's what enables condition variables to be used with any problem.

Thread synchronization

Issues

The variables involved in the condition are shared between T1, T2 and T3. Let's say the condition is satisfied, and T3 exits the while loop. How do we know the condition is still satisfied at that point? We don't!

All we know is that the condition was satisfied certainly for at least a brief moment in time. Some thread could have possibly changed them back to unwanted values.

Conclusion: condition variables are **never used alone**, but **always with a mutex**.

- 1) Signal and broadcast calls must be made always under an exclusive lock.
- 2) The condition must be tested under an exclusive lock.

Thread synchronization

Conclusion:

- 1) Signal and broadcast calls must be made always under an exclusive lock.
- 2) The condition must be tested under an exclusive lock.

T1

lock (m)

x += 5;

signal (c);

unlock (m)

T2

lock (m)

y += 7;

signal (c);

unlock (m)

T3

lock (m)

while (!(x + y > 13)) {

unlock (m)

cwait (c, ?);

lock (m)

}

...

unlock (m)

Are we done?

Thread synchronization

No! What if T1 or T2 call `signal` while T3 is between `unlock` and `cwait`? The signal will be lost, and T3 will wait for the next signal which might never arrive.

T1	T2	T3
<code>lock (m)</code>	<code>lock (m)</code>	<code>lock (m)</code>
<code>x += 5;</code>	<code>y += 7;</code>	<code>while (!(x + y > 13)) {</code>
<code>signal (c);</code>	<code>signal (c);</code>	<code> unlock (m)</code>
<code>unlock (m)</code>	<code>unlock (m)</code>	<hr/> <code> cwait (c, ?);</code>
		<code> lock (m)</code>
		<code>}</code>
		<code>...</code>
		<code>unlock (m)</code>

Thread synchronization

The truth is you cannot solve this at user level. That is why `cwait` receives not one but two parameters: **the condition variable to operate on and a mutex**.

And thus we talk about a monitor = a lock and zero or more condition variables

```
cwait(c,m)
```

This way when a signal arrives (due to `signal()` or `broadcast()`) the mutex is first locked and then `cwait` returns (**atomically**).

AND

When calling `cwait`, the mutex is first unlocked (**atomically**).

Thread synchronization

In other words, all three statements are combined into one.

T3

```
lock (m)
```

```
while (!(x + y > 13)) {
```

```
    unlock (m)
```

```
    cwait (c, ?) ;
```

```
    lock (m)
```

```
}
```

```
...
```

```
unlock (m)
```

T3

```
lock (m)
```

```
while (!(x + y > 13)) {
```

```
    cwait (c, m)
```

```
}
```

```
... // safe to proceed on x, y
```

```
unlock (m)
```

Thread synchronization

Example: the **bounded producer-consumer** problem.

`int count=0`: number of products, `N` upper limit, mutex `m` for storage

Condition variables `empty` and `full`.

Producer

```
for (;;)
    lock (m)
    while (count == N)
        cwait (empty, m)
    produce_item()
    count++
    broadcast (full)
    unlock (m)
```

Consumer

```
for (;;)
    lock (m)
    while (count == 0)
        cwait (full, m)
    consume_item
    count--
    broadcast (empty)
    unlock (m)
```

Thread synchronization

Example: **synchronization barrier** with N threads.

Condition variable `c`, mutex `m`, `arrived = 0`

```
lock(m)
```

```
++arrived
```

```
if(arrived < N)    // if this thread is not last
```

```
    cwait(c,m)     // then wait for others
```

```
else
```

```
    broadcast(c)   // i'm last, awaken the other N-1
```

```
unlock(m)
```

Thread synchronization

Example: **readers-writers**

Reader

```
wait until no writers  
access database  
check out -- wake up waiting writer
```

Writer

```
wait until no readers or writers  
access database  
check out -- wake up waiting readers or writer
```

Thread synchronization

State variables

number of active readers AR = 0

number of active writers AW = 0

number of waiting readers WR = 0

number of waiting writers WW = 0

Condition variable `okToRead`

Condition variable `okToWrite`

Lock `m`

Thread synchronization

```
Reader ()
```

```
    lock (m) ;  
    while ((AW + WW) > 0) {    // if any writers, wait  
        WR++;    // waiting reader  
        cwait (okToRead, m) ;  
        WR--;  
    }  
    AR++;    // active reader  
    unlock (m) ;  
    Access DB  
    lock (m) ;  
    AR--;  
    if (AR == 0 && WW > 0)  
        signal (okToWrite, m) ;  
    unlock (m) ;
```


Thread synchronization

```
Writer()  
    lock(m);  
    while ((AW + AR) > 0) { // if any readers or writers, wait  
        WW++;                // waiting writer  
        cwait(okToWrite, m);  
        WW--;  
    }  
    AW++;                    // active writer  
    unlock(m);  
    Access DB  
    lock(m);  
    AW--;  
    if (WW > 0)              // give priority to other writers  
        signal(okToWrite, m);  
    else if (WR > 0)  
        broadcast(okToRead, m);  
    unlock(m);
```

Thread synchronization

The **dining philosophers**: 5 condition variables, one for every fork (resource). The status array stores every fork's status: free or inUse

```
lock(m)
while(status[i]==inUse || status[(i+1) % 5]==inUse)
    cwait(c[i],m);
status[i]=status[(i+1) % 5]=inUse
unlock(m)

//eat

lock(m)
status[i]=status[(i+1) % 5]=free
signal(c[i],m)
signal(c[(i+1) % 5], m)
unlock(m)
```

Thread synchronization

Cigarette smokers: c condition variable, m mutex, s=0 semaphore

Agent

```
for (;;)
    get_two_random_ingredients()
    lock(m)
    deliver_ingredients()
    broadcast(c)      // tell all that the ingredients are here
    unlock(m)         // they can take them
    wait(s)           // wait for the cigarette to be ready
```

Thread synchronization

Smoker

```
for (;;)
    lock(m)
    while(!ingredient_enough())    // while missing ingredients
        cwait(c,m)                // sleep
    obtain_ingredients()
    unlock(m)                      // ingredients taken, no need for lock
    prepare_cigarette()
    post(s)                       // cigarette ready
```

Thread synchronization

Similarly to mutexes, condition variables can be allocated statically or dynamically.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

or

```
#include <pthread.h>
```

```
int pthread_cond_init(pthread_cond_t * cond ,  
                      const pthread_condattr_t * attr );
```

Returns 0 on success, or a positive error number on error

Use `NULL` in place of `attr` for default parameters.

Do not re-initialize (with `init`) an already initialized cond. var. → undefined!

Copying a cond. variable → undefined behavior. Work on the originals!

Thread synchronization

```
#include <pthread.h>
```

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

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

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

All return 0 on success, or a positive error number on error

The thread to be awoken by `pthread_cond_signal()` is unpredictable.

Thread synchronization

```
#include <pthread.h>
```

```
int pthread_cond_timedwait(pthread_cond_t * cond ,  
                           pthread_mutex_t * mutex ,  
                           const struct timespec * abstime );
```

Returns 0 on success, or a positive error number on error

Same as `pthread_cond_wait`, but it waits at most for `abstime`, after which if it's still waiting it returns `ETIMEDOUT`

Thread synchronization

When an automatically or dynamically allocated condition variable is no longer required, then it should be destroyed using `pthread_cond_destroy()`.

```
#include <pthread.h>
```

```
int pthread_cond_destroy(pthread_cond_t * cond );
```

Returns 0 on success, or a positive error number on error

It is not necessary to call `pthread_cond_destroy()` on a condition variable that was statically initialized using `PTHREAD_COND_INITIALIZER`.

Make sure there are no threads waiting for the condition variable before destroying it. You can use it again if you want to; just make sure you call `pthread_cond_init()` first.