

Synchronization, Part 5: Condition Variables

Arun Prakash Jana edited this page on Mar 21 · 3 revisions

Warm up

Name these properties!

- "Only one process(/thread) can be in the CS at a time"
- "If waiting, then another process can only enter the CS a finite number of times"
- "If no other process is in the CS then the process can immediately enter the CS"

See [Synchronization, Part 4: The Critical Section Problem](#) for answers.

What is the 'exchange instruction' ?

The exchange instruction ('XCHG') is an atomic CPU instruction that exchanges the contents of a register with a memory location. This can be used as a basis to implement a simple mutex lock.

```
// *Pseudo-C-code* for a simple busy-waiting mutex
// that uses an atomic exchange function
int lock = 0; // initialization

// To enter the critical section you need to read a lock value of zero.
// 'xchg' function doesn't exist, but imagine this function is built on the atomi
// i.e. it writes '1' into the lock variable and returns the previous contents of
while (xchg( 1, &lock)) { /*spin spin spin*/ }
/* Do Critical Section stuff*/
lock = 0;
```

What are condition variables? How do you use them? What is Spurious Wakeup?

- Condition variables allow a set of threads to sleep until tickled! You can tickle one thread or all threads that are sleeping. If you only wake one thread then the operating system will decide which thread to wake up. You don't wake threads directly instead you 'signal' the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable.
- Condition variables are used with a mutex and with a loop (to check a condition).
- Occasionally a waiting thread may appear to wake up for no reason (this is called a *spurious wake*)! This is not an issue because you always use `wait` inside a loop that tests a condition that must be true to continue.

Edit

New Page

▼ Pages 51

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)


File System, Part 3: Permissions

File System, Part 4: Working with directories

File System, Part 5: Virtual file systems

Show 36 more pages...

Clone this wiki locally

 Clone in Desktop

- Threads sleeping inside a condition variable are woken up calling `pthread_cond_broadcast` (wake up all) or `pthread_cond_signal` (wake up one). Note despite the function name, this has nothing to do with POSIX `signal` s!

What does `pthread_cond_wait` do?

The call `pthread_cond_wait` performs three actions:

- unlock the mutex and atomically...
- waits (sleeps until `pthread_cond_signal` is called on the same condition variable)
- Before returning, locks the mutex

(Advanced topic) Why do Condition Variables also need a mutex?

Condition variables need a mutex for three reasons. The simplest to understand is that it prevents an early wakeup message (`signal` or `broadcast` functions) from being 'lost.'
Imagine the following sequence of events (time runs down the page) where the condition is satisfied _just before _ `pthread_cond_wait` is called. In this example the wake-up signal is lost!

| Thread 1 | Thread 2 |
|---------------------------------------|--------------------------------|
| <code>while(answer < 42) {</code> | |
| | <code>answer++</code> |
| | <code>p_cond_signal(cv)</code> |
| <code>p_cond_wait(cv,m)</code> | |

If both threads had locked a mutex, the signal can not be sent until *after* `pthread_cond_wait(cv, m)` is called (which then internally unlocks the mutex)

A second common reason is that updating the program state (`answer` variable) typically requires mutual exclusion - for example multiple threads may be updating the value of `answer` .

A third and subtle reason is to satisfy real-time scheduling concerns which we only outline here: In a time-critical application, the waiting thread with the *highest priority* should be allowed to continue first. To satisfy this requirement the mutex must also be locked before calling `pthread_cond_signal` or `pthread_cond_broadcast` . For the curious, a longer and historical discussion is [here](#).

Why do spurious wakes exist?

For performance. On multi-CPU systems it is possible that a race-condition could cause a wake-up (signal) request to be unnoticed. The kernel may not detect this lost wake-up call but can detect when it might occur. To avoid the potential lost signal the thread is woken up so that the program code can test the condition again.

Example

Condition variables are *always* used with a mutex lock.

Before calling *wait*, the mutex lock must be locked and *wait* must be wrapped with a loop.

```
pthread_cond_t cv;
pthread_mutex_t m;
int count;

// Initialize
pthread_cond_init(&cv, NULL);
pthread_mutex_init(&m, NULL);
count = 0;

pthread_mutex_lock(&m);
while (count < 10) {
    pthread_cond_wait(&cv, &m);
    /* Remember that cond_wait unlocks the mutex before blocking (waiting)! */
    /* After unlocking, other threads can claim the mutex. */
    /* When this thread is later woken it will */
    /* re-lock the mutex before returning */
}
pthread_mutex_unlock(&m);

//later clean up with pthread_cond_destroy(&cv); and mutex_destroy

// In another thread increment count:
while (1) {
    pthread_mutex_lock(&m);
    count++;
    pthread_cond_signal(&cv);
    /* Even though the other thread is woken up it cannot not return */
    /* from pthread_cond_wait until we have unlocked the mutex. This is */
    /* a good thing! In fact, it is usually the best practice to call */
    /* cond_signal or cond_broadcast before unlocking the mutex */
    pthread_mutex_unlock(&m);
}
```

Implementing counting semaphores

- We can implement a counting semaphore using condition variables.
- Each semaphore needs a count, a condition variable and a mutex

```
typedef struct sem_t {
    int count;
    pthread_mutex_t m;
    pthread_condition_t cv;
} sem_t;
```

Implement `sem_init` to initialize the mutex and condition variable

```
int sem_init(sem_t *s, int pshared, int value) {
    if (pshared) { errno = ENOSYS /* 'Not implemented' */; return -1;}
}
```

```
s->count = value;
pthread_mutex_init(&s->m, NULL);
pthread_cond_init(&s->cv, NULL);
return 0;
}
```

Our implementation of `sem_post` needs to increment the count. We will also wake up any threads sleeping inside the condition variable. Notice we lock and unlock the mutex so only one thread can be inside the critical section at a time.

```
sem_post(sem_t *s) {
    pthread_mutex_lock(&s->m);
    s->count++;
    pthread_cond_signal(s->cv); /* See note */
    /* A woken thread must acquire the lock, so it will also have to wait until we

    pthread_mutex_unlock(&s->m);
}
```

Our implementation of `sem_wait` may need to sleep if the semaphore's count is zero. Just like `sem_post` we wrap the critical section using the lock (so only one thread can be executing our code at a time). Notice if the thread does need to wait then the mutex will be unlocked, allowing another thread to enter `sem_post` and waken us from our sleep!

Notice that even if a thread is woken up, before it returns from `pthread_cond_wait` it must re-acquire the lock, so it will have to wait a little bit more (e.g. until `sem_post` finishes).

```
sem_wait(sem_t *s) {
    pthread_mutex_lock(&s->m);
    while (s->count == 0) {
        pthread_cond_wait(&s->cv, &s->m); /*unlock mutex, wait, relock mutex*/
    }
    s->count--;
    pthread_mutex_unlock(&s->m);
}
```

Wait `sem_post` keeps calling `pthread_cond_signal` won't that break `sem_wait`? Answer: No! We can't get past the loop until the count is non-zero. In practice this means `sem_post` would unnecessary call `pthread_cond_signal` even if there are no waiting threads. A more efficient implementation would only call `pthread_cond_signal` when necessary i.e.

```
/* Did we increment from zero to one- time to signal a thread sleeping inside s
if (s->count == 1) /* Wake up one waiting thread!*/
    pthread_cond_signal(&s->cv);
```

Other semaphore considerations

- Real semaphores implementation include a queue and scheduling concerns to ensure fairness and priority e.g. wake up the highest-priority longest sleeping thread.
- Also, an advanced use of `sem_init` allows semaphores to be shared across

processes. Our implementation only works for threads inside the same process.

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

