1 Using Condition Variables

Condition variables are a synchronization primitive often used in conjunction with a **mutex** (or **guard lock**) to allow threads to wait until a specific condition (defined by some **guard variables** or state variables) is true.

For an overview, please first look at this acquire-release diagram. The Acquire-Release paradigm is a standard paradigm, and you'll find almost all of your usage of condition variables will fit into this paradigm.

1.1 The Waiting Room

In this diagram, you see a "Waiting Room". Threads that call `pthread_cond_wait()` and block are effectively placed in a conceptual "**Waiting Room**." They remain there until another thread signals or broadcasts the condition variable.

1.2 Core Functions and Syntax

The three main functions for condition variables are: `pthread_cond_wait()`, `pthread_cond_signal()`, and `pthread_cond_broadcast()`.

Function	Syntax	Purpose
`pthread_cond_wait()`	`int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex);`	**Blocks** the calling thread until the condition variable `cond` is signaled.
`pthread_cond_signal()`	`int pthread_cond_signal( pthread_cond_t *cond);`	**Wakes up at least one** thread waiting on the condition variable `cond`.
`pthread_cond_broadcast()`	`int pthread_cond_broadcast( pthread_cond_t *cond);`	**Wakes up all** threads waiting on the condition variable `cond`.

1.3 The Mutex Argument in `pthread_cond_wait()`

The function `pthread_cond_wait()` requires a **mutex argument** (`&guard_mutex`). This is because `pthread_cond_wait()` must **atomically release the mutex** and put the calling thread into the "Waiting Room". When the waiting thread is woken up (signaled/broadcasted), it **re-acquires the mutex** before returning from `pthread_cond_wait()`.

1.4 Why `pthread_cond_signal()` Does Not Need a Mutex

The functions `pthread_cond_signal()` and `pthread_cond_broadcast()` simply send a notification to a thread or all threads (for `broadcast`) in the waiting room. The thread calling `signal` or `broadcast` is expected to already hold the mutex so it can safely modify the shared state (guard variables). The `pthread_cond_signal()` function itself does not need to handle the release or re-acquisition of the lock.

1.5 Implementation of `pthread_cond_signal()` and Spurious Wakeups

In terms of implementation, the functions `pthread_cond_signal()` and `pthread_cond_broadcast()` send signals to one or all threads that are currently in the waiting room. This works because a thread in the waiting room must have called `pthread_cond_wait()`. In most implementations, that "wait" function causes the thread to sleep indefinitely. The "signal" and "broadcast" functions send Linux signals to the threads in the waiting room using `pthread_kill()`. Each thread has a signal handler (defined by `libpthread.so`) that responds to the signal. This forces the thread out of the SLEEPING state. When the signal handler returns, the waiting thread is again in the function `pthread_cond_wait()`. That function then calls `pthread_mutex_unlock` to leave the waiting room. The function `pthread_mutex_unlock()` may block, but eventually it returns with success, and then `pthread_cond_wait()` returns.

Finally, a thread in the waiting room may receive an unrelated signal, called a *spurious wakeup*. A good example is the SIGWINCH signal. When a user reshapes a terminal window (or other), the operating system sends a SIGWINCH to all processes interacting with the window. Such a process (e.g., the "vi" editor) can then declare a signal handler allowing it to redisplay after a window is resized. But that SIGWINCH can also wake up a thread in the waiting room!

---

2 Acquire and Release Pattern

The simple, robust pattern for using condition variables is divided into three logical steps: Acquire, Use, and Release. Please refer back to the acquire-release diagram for this part.

Further, here is some typical code that implements this paradigm. In this example, we want to limit the number of active threads, perhaps because we have only MAX_ACTIVE_THREADS many CPU cores, and we do not wish to overcommit CPU resources.

NOTE: This code could also have been implemented using a semaphore, and initializing its count to MAX_ACTIVE_THREADS. The thread would call `sem_wait()` before DO_TASK, and `sem_post()` after DO_TASK. However, we will see that this new implementatoin is much more flexible for more general problems.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t mutex;
pthread_cond_t cond_var;
int num_active_threads = 0;
const int NUM_THREADS = 3;
const int MAX_ACTIVE_THREADS = 2;

void *worker_thread_func(void *not_used) {
    // ACQUIRE the resource (the permission to do a task)
    pthread_mutex_lock(&mutex);
    while (num_active_threads >= MAX_ACTIVE_THREADS) {
        pthread_cond_wait(&cond_var, &mutex);
    }
    num_active_threads++;
    pthread_mutex_unlock(&mutex);
```

```
    // USE the resource (i.e., DO_TASK)
    sleep(2);

    // RELEASE the resource (the permission to do a task)
    pthread_mutex_lock(&mutex);
    num_active_threads--;
    pthread_cond_signal(&cond_var);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
int main() {
    pthread_t threads[3];
    long i;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_var, NULL);
    for (i = 1; i <= NUM_THREADS; i++) {
        pthread_create(&threads[i-1], NULL, worker_thread_func, (void*)i);
    }
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

2.1 A. Acquire Resource

This section manages waiting until the resource is safe to use.

1. **Acquire Guard Mutex:** Lock the mutex to protect access to the shared **guard variables**.

   ```
   pthread_mutex_lock(&guard_mutex);
   ```

2. **Check Safety Condition:** Repeatedly check if the `safety_condition` (based on the guard variables) is `False`.

   ```
   while (safety_condition(guard_vars) == False) {
     pthread_cond_wait(&cond, &guard_mutex); // Go into the "Waiting Room"
   }
   ```

3. **Update Guard Variables:** After the condition is met, update the guard variables (e.g., update the number of threads using the resource and decrement the number of threads that are waiting).

4. **Release Guard Mutex:** Unlock the mutex.

   ```
   pthread_mutex_unlock(&guard_mutex);
   ```

2.2 B. Use Resource

The thread now has safe access to the resource.

- If a read resource was acquired, any writers should be guaranteed to be waiting.
- If a write resource was acquired, any readers and other writers should be guaranteed to be waiting.

2.3 C. Release Resource

This section signals to other threads that the resource is now available or that the state has changed.

1. **Acquire Guard Mutex:** Lock the mutex.

   ```
   pthread_mutex_lock(&guard_mutex);
   ```

2. **Signal Waiting Threads:** Call `pthread_cond_broadcast()` to wake up all threads in the "Waiting Room". While `pthread_cond_signal()` can be used for performance optimization, **`pthread_cond_broadcast()` is always safe**.

   ```
   pthread_cond_broadcast(&cond);
   ```

3. **Update Guard Variables:** Update the guard variables to reflect the resource release (e.g., update the number of threads that are using the resource).

4. **Release Guard Mutex:** Unlock the mutex.

   ```
   pthread_mutex_unlock(&guard_mutex);
   ```

   Deadlock and Safety Risk: The only danger when using guard variables in a critical section for a mutex is if there is bad logic in the `safety_condition()` or in updating the guard/state variables. This can lead to **deadlock** or a **safety violation**.

Would you like to know more about the important trap of confusing condition variables with semaphores, as mentioned in the document?

It's important to understand the significant trap: A condition variable is **not** a semaphore. Trying to use a condition variable where you really want a semaphore is tempting but incorrect.

---

3 The Reader-Writer Problem

We now use the reader-writer problem to show some the Acquire-Release paradigm in action. It assumes that at any moment, there should either be one writer executing a task, or else arbitrarily many readers (possibly even zero) executing a task.

The code has two optimizations. First, The code below could have always used `pthread_cond_broadcast()` instead of `pthread_cond_signal()`. One should always implement first, for correctness, and only apply optimizatoins in a later versoin. The second optimization is that we have two waiting rooms, labelled by `readers_cond` and `writers_cond`. If the condition logic in the `while` statements is correct, then one can use a single waiting room (`general_cond`). In that situation, we always want to use `pthread_cond_broadcast()` and wake both readers and writers. We then let each type of thread (reader or writer) implement specific logic in the `while` condition.

3.1 Reader-Writer Poblem (Reader Preferred)

Using the above principles, we now show the reader-writer problem (reader preferred). Note that we optimize the code by having two "waiting rooms" (condition variables), one for the readers and one for the writers. And we further optimize the code by using `pthread_cond_signal` to wake just one writer, instead of broadcasting to all writers.

A good way to test yourself on your understanding is to verify for yourself that the `assert` statements in the code will never trigger a failure.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
int shared_data = 0;
int reader_count = 0;
int waiting_reader_count = 0;
int writer_count = 0;
pthread_mutex_t mutex_lock;
pthread_cond_t readers_cond;
pthread_cond_t writers_cond;

void *reader(void *not_used) {
    while (1) {
        // ACQUIRE
        pthread_mutex_lock(&mutex_lock);
        waiting_reader_count++;
        while (writer_count > 0) {
            pthread_cond_wait(&readers_cond, &mutex_lock);
        }
        waiting_reader_count--;
        reader_count++;
        assert(writer_count == 0);
        pthread_mutex_unlock(&mutex_lock);
        // USE (DO_TASK)
        usleep(100000);
        printf("I read\n");
        // RELEASE
        pthread_mutex_lock(&mutex_lock);
        reader_count--;
        assert(writer_count == 0);
        if (reader_count == 0) {
            pthread_cond_signal(&writers_cond);
        }
        pthread_mutex_unlock(&mutex_lock);
        usleep(1000000);
    }
    return NULL;
}
void *writer(void *not_used) {
    while (1) {
        // ACQUIRE
        pthread_mutex_lock(&mutex_lock);
        while (reader_count > 0 || waiting_reader_count > 0 ||
               writer_count > 0) {
            pthread_cond_wait(&writers_cond, &mutex_lock);
        }
        assert(reader_count == 0 && waiting_reader_count == 0 &&
```

```
            writer_count == 0);
        writer_count++;
        pthread_mutex_unlock(&mutex_lock);
        // USE (DO_TASK)
        shared_data++;
        usleep(200000);
        printf("I wrote\n");
        // RELEASE
        pthread_mutex_lock(&mutex_lock);
        writer_count--;
        assert(reader_count == 0 && writer_count == 0);
        if (waiting_reader_count > 0) {
            pthread_cond_broadcast(&readers_cond);
        } else {
            pthread_cond_signal(&writers_cond);
        }
        pthread_mutex_unlock(&mutex_lock);
        usleep(100000);
    }
    return NULL;
}
#define NUM_READERS 5
#define NUM_WRITERS 2
int main() {
    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];
    int reader_ids[NUM_READERS];
    int writer_ids[NUM_WRITERS];
    pthread_mutex_init(&mutex_lock, NULL);
    pthread_cond_init(&readers_cond, NULL);
    pthread_cond_init(&writers_cond, NULL);
    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i + 1;
        if (pthread_create(&readers[i], NULL, reader, &reader_ids[i]) != 0) {
            return 1;
        }
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i + 1;
        if (pthread_create(&writers[i], NULL, writer, &writer_ids[i]) != 0) {
            return 1;
        }
    }
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }
    pthread_mutex_destroy(&mutex_lock);
    pthread_cond_destroy(&readers_cond);
```

```
    pthread_cond_destroy(&writers_cond);
    return 0;
}
```

3.2 Reader-Writer Poblem (Writer Preferred)

Using the above principles, we now show the reader-writer problem (writer preferred). Note that we optimize the code by having two "waiting rooms" (condition variables), one for the readers and one for the writers. And we further optimize the code by using `pthread_cond_signal` to wake just one writer, instead of broadcasting to all writers.

A good way to test yourself on your understanding is to verify for yourself that the `assert` statements in the code will never trigger a failure.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
int shared_data = 0;
int reader_count = 0;
int writer_count = 0;
int waiting_writer_count = 0;
pthread_mutex_t mutex_lock;
pthread_cond_t readers_cond;
pthread_cond_t writers_cond;

void *reader(void *not_used) {
    while (1) {
        // ACQUIRE
        pthread_mutex_lock(&mutex_lock);
        while (waiting_writer_count > 0 || writer_count > 0) {
            pthread_cond_wait(&readers_cond, &mutex_lock);
        }
        reader_count++;
        assert(writer_count == 0 && waiting_writer_count == 0);
        pthread_mutex_unlock(&mutex_lock);
        // USE (DO_TASK)
        usleep(100000);
        printf("I read\n");
        // RELEASE
        pthread_mutex_lock(&mutex_lock);
        reader_count--;
        assert(writer_count == 0);
        if (reader_count == 0) {
            pthread_cond_signal(&writers_cond);
        }
        pthread_mutex_unlock(&mutex_lock);
        usleep(100000);
    }
    return NULL;
}
```

```c
void *writer(void *not_used) {
    while (1) {
        // ACQUIRE
        pthread_mutex_lock(&mutex_lock);
        waiting_writer_count++;
        while (reader_count > 0 || writer_count > 0) {
            pthread_cond_wait(&writers_cond, &mutex_lock);
        }
        waiting_writer_count--;
        assert(reader_count == 0 && writer_count == 0);
        writer_count++;
        pthread_mutex_unlock(&mutex_lock);
        // USE (DO_TASK)
        shared_data++;
        usleep(200000);
        printf("I wrote\n");
        // RELEASE
        pthread_mutex_lock(&mutex_lock);
        writer_count--;
        assert(reader_count == 0 && writer_count == 0);
        if (waiting_writer_count > 0) {
            pthread_cond_signal(&writers_cond);
        } else {
            pthread_cond_broadcast(&readers_cond);
        }
        pthread_mutex_unlock(&mutex_lock);
        usleep(500000);
    }
    return NULL;
}
#define NUM_READERS 5
#define NUM_WRITERS 2
int main() {
    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];
    int reader_ids[NUM_READERS];
    int writer_ids[NUM_WRITERS];
    pthread_mutex_init(&mutex_lock, NULL);
    pthread_cond_init(&readers_cond, NULL);
    pthread_cond_init(&writers_cond, NULL);
    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i + 1;
        if (pthread_create(&readers[i], NULL, reader, &reader_ids[i]) != 0) {
            return 1;
        }
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i + 1;
        if (pthread_create(&writers[i], NULL, writer, &writer_ids[i]) != 0) {
            return 1;
        }
```

```
    }
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }
    pthread_mutex_destroy(&mutex_lock);
    pthread_cond_destroy(&readers_cond);
    pthread_cond_destroy(&writers_cond);
    return 0;
}
```

4 The Trap: Condition Variables vs. Semaphores

Using a simple condition variable wait/signal pattern (like the flawed example below) will fail because of two critical synchronization issues: **spurious wakeups** and **lost wakeups**.

4.1 Flawed Semaphore Pattern (Do Not Use)

```
SEM_WAIT:
  pthread_mutex_lock(guard_vars);
  pthread_cond_wait(&cond, &guard_mutex); // Flaw: No check of state
  pthread_mutex_unlock(guard_vars);

SEM_POST:
  pthread_mutex_lock(guard_vars);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(guard_vars);
```

4.2 1. Spurious Wakeup (Problem A)

A **spurious wakeup** occurs when the system sends a random signal (e.g., `SIGWINCH`) to the thread waiting on the condition variable.

- The waiter wakes up and returns from `pthread_cond_wait()` **without any post** having occurred.
- If the thread doesn't check the shared state (guard variables) after waking up, it will proceed incorrectly.
- **Correction:** This is why the `while` loop (checking the `safety_condition`) is **mandatory** for `pthread_cond_wait()`—it forces the thread to go back to sleep if the condition isn't actually met.

4.3 2. Lost Wakeup (Problem B)

A **lost wakeup** occurs when the thread that posts the signal posts it **first**, and *then* the waiting thread begins to wait.

- The waiter never saw the post.
- Since the condition variable doesn't inherently store the signal, the waiter will be blocked indefinitely, despite the event having already happened.

5 Implementing a Semaphore with Condition Variables (The Correct Way)

To correctly implement a semaphore using condition variables, you must introduce a shared **`count` variable** to maintain the state (the number of available resources or posts).

The preferred approach is to directly use `sem_wait()` and `sem_post()`. However, if you must implement your own:

5.1 SEM_WAIT

The `count` must be decremented *before* checking the condition, and the `while` loop ensures the thread waits only if the count is negative:

```
SEM_WAIT:
  pthread_mutex_lock(guard_vars);
  count--; // Decrement the count (resource claimed)
  while (count < 0) {
    pthread_cond_wait(&cond, &guard_mutex);
  }
  pthread_mutex_unlock(guard_vars);
```

5.2 SEM_POST

The post increments the `count` and signals only one waiting thread:

```
SEM_POST:
  pthread_mutex_lock(guard_vars);
    count++;
    pthread_cond_signal(&cond); // Signal one waiter to wake up
  pthread_mutex_unlock(guard_vars);
```

This ensures that the `count` variable correctly stores any "missed" signals, preventing a lost wakeup.

Would you like to know more about the performance optimization technique of using separate condition variables for readers and writers (writer preference)?