CONDITION VARIABLES AND THE PRODUCER/CONSUMER PROBLEM

Kai Mast
CS 537
Fall 2022

RECAP: LOCKS AND CONDITION VARIABLES

- Locks enable **mutual exclusion** of threads
  - Only at most one thread can execute a critical section

- Condition variables allow **ordering** of thread execution
  - Threads can wait() on the condition variable and will be woken up if

RECAP: CONDITION VARIABLES

Interface

- `wait()`: block current thread until woken up by another thread
- `signal/broadcast()`: wake up (at least) one thread or wake up all threads

Common Errors

- No (re-)checking of state before calling wait
- Not holding the lock while changing shared state
- `if` instead of `while`; does not protect against spurious wake-ups

Spurious Wake-ups

- Problem: Signal might wake up more than one waiter
- Or waiters get woken up without even when there was no signal (less common)
- Why? Might have been more efficient to implement

RECAP: JOIN() IMPLEMENTATION

```
void thread_exit(thread_t *t) {
    mutex_lock(&t->mutex);
    t->done = 1;
    cond_signal(&t->cond);
    mutex_unlock(&t->mutex);
}
```

```
void thread_join(thread_t *t) {
    mutex_lock(&t->mutex);
    while (t->done == 0) {
        cond_wait(&t->cond, &t->mutex);
    }
    mutex_unlock(&t->mutex);
}
```

Why do we need the done-variable?

- Thread might already have terminated before we call thread_join()

Why do we need the `while`-loop?

- To prevent against spurious wake-ups
- Generally, always (re-)check state after acquiring a lock

A MORE COMPLEX EXAMPLE: UNIX PIPES

What happens when we do this?

```
prompt> cat myfile.txt | grep “cs537”
```

- OS maps `cat`’s standard output to a pipe
- OS maps `grep`’s standard input to the same pipe

- Why is this useful?
  - Allows performing **complex tasks** using a combination of **simple tools**

- Pipe is a limited size buffer provided by the OS
  - Why limited? Dangerous to let process fill an unlimited kernel buffer

A MORE COMPLEX EXAMPLE: UNIX PIPES

```
prompt> cat myfile.txt | grep "cs567"
```

- A pipe can have multiple readers and writers
- Internally, there is a finite-size, circular buffer
  - We'll see later what "circular" means
- Writers add data to the buffer: May have to wait if buffer is full
- Readers remove data from the buffer: May have to wait if buffer is empty

MORE GENERAL: PRODUCERS & CONSUMERS

Producers generate data (e.g., pipe writers)
Consumers grab data and process it (e.g., pipe readers)

Producer/consumer problems are frequent in systems (e.g., web servers)

General strategy use condition variables to:

- Make producers wait when buffers are full
- Make consumers wait when buffers are empty

Handle case where producer and consumer work at much different speeds

SINGLE PRODUCER/CONSUMER

We start with an easy case:

- One producer thread
- One consumer thread
- At most one item that can be produced or consumed

Keep track of array state using shared state

- `num_full` indicates the available items to “consume”
- Can only be 0 or 1

SINGLE PRODUCER/CONSUMER

```
void* producer(void *arg) {
    while (!done) {
        mutex_lock(&m);
        while (num_full > 0)
            cond_wait(&cond, &m);
        fill_buffer();
        num_full += 1;
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

```
void* consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (num_full == 0)
            cond_wait(&cond, &m);
        use_buffer();
        num_full -= 1;
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

Will the producer be stuck waiting for `mutex_lock()`?

What happens to consumer when producer calls signal?

PRODUCER/CONSUMER WITH MULITPLE BUFFERS

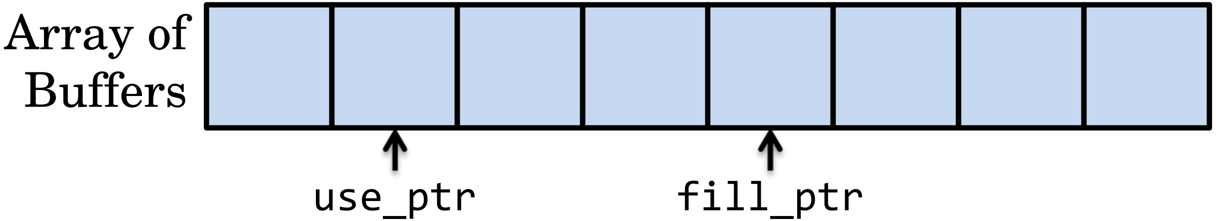
Now a slightly more complicated case:

- One producer thread
- One consumer thread
- An array of multiple buffers that can be filled or consumed

Keep track of array state using shared state

- `num_full` indicates the available items to “consume”
- `max` indicates the size of the array
- `num_full` can be 0 or any integer `<=max`

PRODUCER/CONSUMER WITH MULITPLE BUFFERS




```
void fill_next_buffer(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % max;
}
```

```
int use_next_buffer() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % max;
    return tmp;
}
```

- `fill_ptr` tracks the current position of the producer
- `use_ptr` tracks the current position of the consumer
- When reaching the end of the buffer, pointers wrap around

PRODUCER/CONSUMER WITH MULITPLE BUFFERS

```
void* producer(void *arg) {
     while (!done) {
          mutex_lock(&m);
          while (num_full == max)
               cond_wait(&cond, &m);
          fill_next_buffer();
          num_full += 1;
          cond_signal(&cond);
          mutex_unlock(&m);
     }
}
```

```
void* consumer(void *arg) {
     for (1) {
          mutex_lock(&m);
          while (num_full == 0)
               cond_wait(&cond, &m);
          use_next_buffer();
          num_full -= 1;
          cond_signal(&cond);
          mutex_unlock(&m);
     }
}
```

Does this behave correctly?

Yes

How much concurrency does this allow?

Only one thread can fill or use a buffer at a time

PRODUCER/CONSUMER WITH MULITPLE BUFFERS

```
void* producer(void *arg) {
    while (!done) {
        mutex_lock(&m);
        while (num_full == max)
            cond_wait(&cond, &m);
        mutex_unlock();
        fill_next_buffer();
        mutex_lock();
        num_full += 1;
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

```
void* consumer(void *arg) {
    for (1) {
        mutex_lock(&m);
        while (num_full == 0)
            cond_wait(&cond, &m);
        mutex_unlock();
        use_next_buffer();
        mutex_lock();
        num_full -= 1;
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

Why can we release the lock here?

- `num_full` guards which buffers will be filled or used
- We only access `num_full` while holding the lock

MULTIPLE PRODUCERS/CONSUMERS

Another slightly more complicated case:

- Multiple producer threads
- Multiple consumer threads
- At most one item that can be produced or consumed

Keep track of array state using shared state

- `num_full` indicates the available items to "consume"
- Can only be 0 or 1

MULTIPLE PRODUCERS/CONSUMERS

```
void* producer(void *arg) {
    while (1) {
        mutex_lock(&m);                     A
        while (num_full > 0)                B
            cond_wait(&cond, &m);           C
        fill_buffer();                      D
        num_full += 1;                      E
        cond_signal(&cond);                 F
        mutex_unlock(&m);                   G
    }
}
```

```
void* consumer(void *arg) {
    while (1) {
        mutex_lock(&m);                     H
        while (num_full == 0)               I
            cond_wait(&cond, &m);           J
        use_buffer();                       K
        num_full -= 1                       L
        cond_signal(&cond);                 M
        mutex_unlock(&m);                   N
    }
}
```

What is the potential problem here?

Producer			A B D E **F** G			A B C	
Consumer 1	H I J			J I K L **M** N	H I J		All stuck on cond_wait()!
Consumer 2		H I J			J I J		

Time

SIMPLE SOLUTION: USE BROADCAST

```
void* producer(void *arg) {
    while (!done) {
        mutex_lock(&m);
        while (num_full > 0)
            cond_wait(&cond, &m);
        fill_buffer();
        num_full += 1;
        cond_broadcast(&cond);
        mutex_unlock(&m);
    }
}
```

```
void* consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (num_full == 0)
            cond_wait(&cond, &m);
        use_buffer();
        num_full -= 1;
        cond_broadcast(&cond);
        mutex_unlock(&m);
    }
}
```

Does this behave correctly?

Yes

Is this an ideal solution?

No, scales poorly with number of threads

BETTER: USE TWO CONDITION VARIABLES

```
void* producer(void *arg) {
    while (!done) {
        mutex_lock(&m);
        while (num_full > 0)
            cond_wait(&consume, &m);
        fill_buffer();
        num_full += 1;
        cond_signal(&produce);
        mutex_unlock(&m);
    }
}
```

```
void* consumer(void *arg) {
    while (1) {
        mutex_lock(&m);
        while (num_full == 0)
            cond_wait(&produce, &m);
        use_buffer();
        num_full -= 1;
        cond_signal(&consume);
        mutex_unlock(&m);
    }
}
```

produce-condition is only signaled by producers and only wakes up consumers

consume-condition is only signaled by consumers and only wakes up producers

CV BEST PRACTICES REVISITED

- Do not use one condition variable across multiple mutexes
- But, you can use multiple condition variables for one mutex
  - Make sure each condition variable has a designated "role"

- As with locks, more condition variables make the code more complex but, potentially, increase performance
  - Find a good middle ground between the two

ANOTHER EXAMPLE: MEMORY MANAGEMENT

- In this example, heap memory is limited
  - Threads might block on `alloc()`

- Very simplified: we only keep track of an integer value; not the actual memory mapping

- Can be generalized to other such non-uniform producer/consumer patterns

MEMORY MANAGEMENT EXAMPLE

```
void* allocate(int size) {
    mutex_lock(&m);
    while (bytes_left < size)
        cond_wait(&c, &m);
    // get mem from heap
    void *ptr = ...;
    bytes_left -= size;
    mutex_unlock(&m);
    return ptr;
}
```

```
void free(void *ptr, int size) {
    mutex_lock(&m);
    bytes_left += size;
    cond_signal(&c);
    mutex_unlock(&m);
}
```

Problem?

- Thread that gets woken up might need more memory than became available

MEMORY MANAGEMENT FIXED

```c
void* allocate(int size) {
    mutex_lock(&m);
    while (bytes_left < size)
        cond_wait(&c, &m);
    // get mem from heap
    void *ptr = ...;
    bytes_left -= size;
    mutex_unlock(&m);
    return ptr;
}
```

```c
void free(void *ptr, int size) {
    mutex_lock(&m);
    bytes_left += size;
    cond_broadcast(&c);
    mutex_unlock(&m);
}
```

- All threads will be woken up
- If there is sufficient memory for a thread, that thread will be able to make progress
- A more efficient, but more complicated, solution probably exists