Concurrency: Condition Variables

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2024

Administrivia

- Project 3 graded
- Project 4 out – due Nov 5th
    - Some tests coming out later this week
    - Piazza post with video tracing scheduler in xv6
- Exam 1 grades coming later this week

Review: Locked Data Structures

- Making thread safe data structures
  - One big lock, then worry above performance if still not performing efficiently enough
- Approximate Counter, Linked List, Queue, Hash Table examples

Quiz: Locked Data Structures

https://tinyurl.com/cs537-fa24-q11

Condition Variables Agenda

A mechanism for threads to check if a condition is true before continuing execution

- Example in pthread_join()
- Example in producer/consumer (bounded buffer) problem

CV Definition

A **condition variable** is an explicit queue that threads can put themselves on when some state of execution is not as desired. Some other thread, when it changes said state, can wake one of the waiting threads thus allowing them to continue.

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Parent Waiting for Child

```
void *child(void *arg) {
   printf("child\n");
   thr_exit(); //OUR IMPLEMENTATION OF EXIT
   return NULL;
}
int main(int argc, char *argv[]) {
   printf("parent: begin\n");
   pthread_t p;
   Pthread_create(&p,NULL,child,NULL);
   thr_join(); //OUR IMPLEMENTATION OF JOIN
   printf("parent: end\n");
   return 0;
}
```

Parent Waiting for Child (cont)

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
   Pthread_mutex_lock(&m);
   done = 1;
   Pthread_cond_signal(&c);
   Pthread_mutex_unlock(&m);
}

void thr_join() {
   Pthread_mutex_lock(&m);
   while (done == 0)
      Pthread_cond_wait(&c,&m);
   Pthread_mutex_unlock(&m);
}
```

`wait()` and `signal()`

The `wait()` call assumes the mutex is locked when it is called. The call releases the lock and puts the thread to sleep.

After another thread calls `signal()`, the thread is awoken, the lock is reacquired before returning from the call to `wait()`.

In the implementation of `thr_join()` and `thr_exit()` all three variables are important for this correct implementation: done, mutex, and cv.

- Imagine no `done` variable and child runs immediately and calls `thr_exit()`
- Imagine no mutex (and if statement) and parent calls `thr_join()`, checks done, but before calling signal child runs.

Producer Consumer Example

- One or more **producer threads** generating data items and placing them in a buffer
- One or more **consumer threads** grabbing items from the buffer and consuming them in some way.

Lots of problems can be seen in this way:

- Multi-threaded web server
  - producer takes incoming requests and puts them in a queue
  - consumer threads take requests from the queue to process them
- Linux Pipes: `grep foo file.txt | wc -l`
  - First program is producer (given output to OS)
  - Second program is consumer (reading data from the OS)

Producer/Consumer Setup

To understand this scenario we will use a buffer of a single integer and a counter that will be one if the buffer is full (holding a value) or 0 if it is empty. Idea can be extended to larger buffers.

```
int buffer;
int counter = 0; //initially empty

void put(int value) {
   assert(count == 0);
   count = 1;
   buffer = value;
}

int get() {
   assert(count == 1);
   count = 0;
   return buffer;
}
```

Producer/Consumer Threads

Now we need the code for the producer and consumer threads.

```
void *producer(void *arg) {
   int i;
   int loops = (int) arg;
   for(i=0;i<loops; i++) {
      put(i);
   }
}

void *consumer(void *arg) {
   int i;
   while(1) {
      int tmp = get();
      printf("%d\n",tmp);
   }
}
```

A clear problem: producer can `put()` into a full buffer, consumer can `get()` from an empty buffer.

Attempt 2 (still broken)

```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        if (count == 1)                       // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        if (count == 0)                       // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

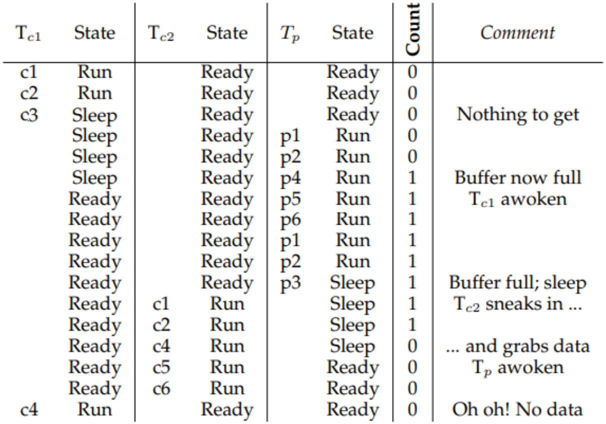
Broken Solution Analysis

- Producer waits for buffer to be empty (lines p1-p3), then puts a value in the buffer, signals, and releases the lock (lines p4-p6)
- Consumer waits for buffer to be full (lines c1-c3), then gets a value, signals, and releases the lock (lines c4-c6)

This works for a single producer thread and single consumer thread.

What happens with 2 consumers and one producer – The 2nd consumer might context-switch and get the value while the first consumer is waiting.

Broken Solution Thread Trace



T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

Meaning of Signal

When a thread signals, just hint that state has changed – no guarantee state is what thread wants (**Mesa semantics**).

Hoare semantics guarantees that the woken thread will run immediately upon being woken.

Virtually every system ever built employs Mesa semantics.

Can fix this problem using `while` instead of `if` to check the condition again

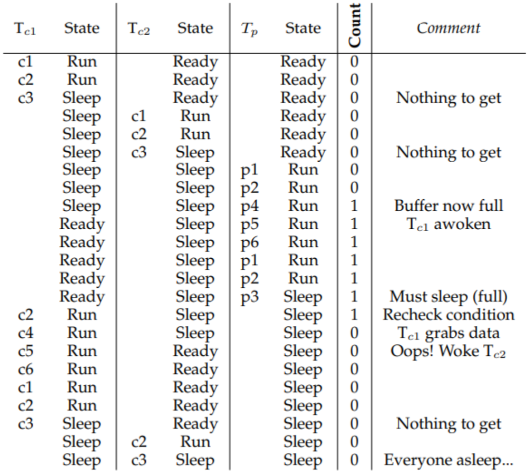
Failed Attempt 3

```
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        while (count == 1)                    // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        while (count == 0)                    // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

Another Broken Solution Analysis

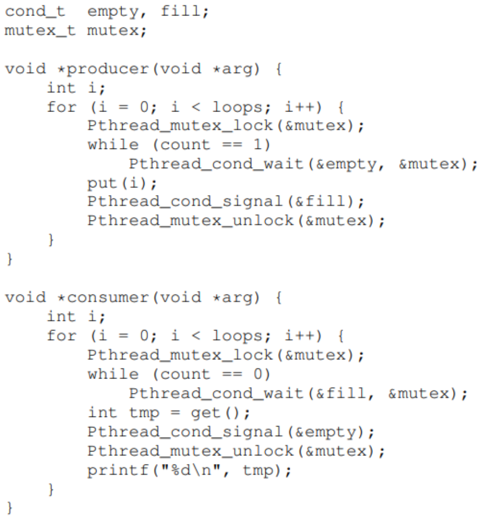
- imagine two consumers, one producer – but both consumers run first, find buffer empty, go to sleep
- Producer runs, buffer is empty, fills it, sends a signal to wake one other thread, loops finds buffer full and sleeps.
- First consumer wakes, reads out data, sends signal (WHO IS WAITING TO BE SIGNALLED?), loop, finds buffer empty and sleeps.
- One of the waiting threads is awoken, if it is consumer 2 then problem. Since buffer is empty it goes back to sleep.
- **EVERYONE IS WAITING**. Signalling needs to be directed (consumers signal producers and vice versa). Need 2 signal variables.

Broken Solution Thread Trace



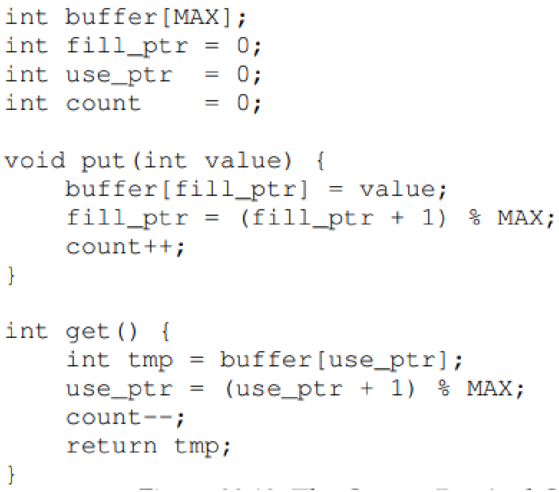
T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Correct Producer / Consumer Solution (One Buffer)



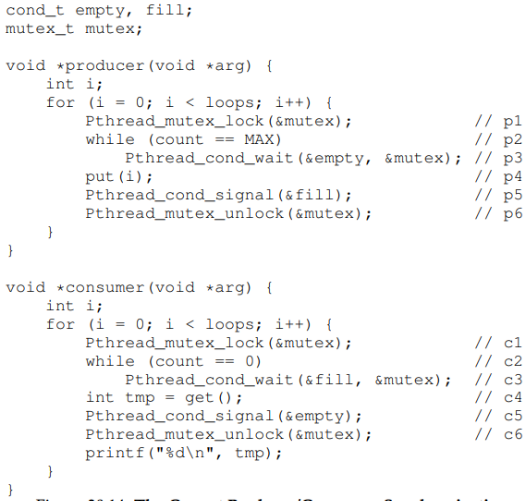
```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Correct Put / Get (Larger Buffer)



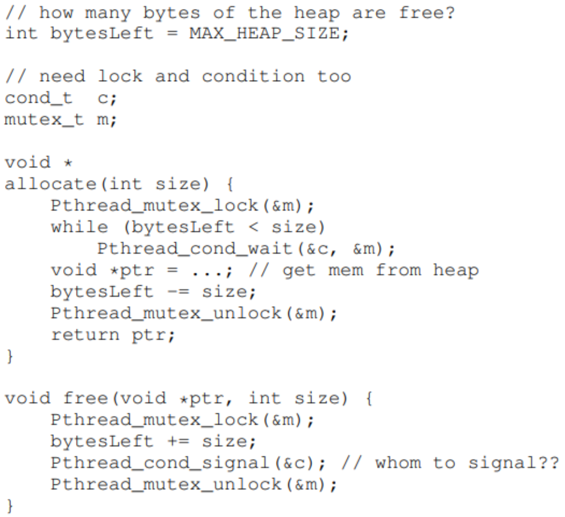
```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Correct Producer / Consumer (Larger Buffer)



```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == MAX)                    // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                 // p4
        Pthread_cond_signal(&fill);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                        // c4
        Pthread_cond_signal(&empty);            // c5
        Pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
}
```

Free Memory / Allocate Memory Example



```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Covering Condition Scenario

Imagine all of memory is full, and a thread calls `allocate(100)` and another thread calls `allocate(10)`.

Now another thread calls `free(50)`, but it wakes the thread that needs 100 (problem) – it just goes back to sleep because there is not enough free memory.

solution – wake all the threads (`pthread_cond_broadcast()`)

- Guaranteed that any thread that should be woken up are.
- Down side is negative performance impact since most threads will just go back to sleep (not enough memory).