CS 3214: Computer Systems
Lecture 16: Condition Variable

Instructor: Huaicheng Li

Oct 13 2022

VIRGINIA TECH™

Recap

- ❑ Lock
- ❑ Semaphore
- ❑ A need for proper "wait()"

```
void *child(void *arg) {
    printf("child\n");
    /* how to indicate we are done? */
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    printf("parent: end\n");
    return 0;
}
```

Method #1: A Spin-based Approach

- Does it work?
- Is it a good implementation?
  - Busy-waiting (spinning) wastes CPU cycles

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    /* how to indicate we are done? */
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    while (done == 0) ; /* spin */
    printf("parent: end\n");
    return 0;
}
```

Instead of spinning, can we figure out a way to wait for the done flag to be set true? e.g., sleeping and then waking up …

Condition Variable

- Definition:
  - A condition variable is a queue of sleeping threads
  - When condition not true, threads can put them to sleep on the queue
  - When condition becomes true, wake up thread(s) from the queue
- APIs
  - Declaration: *pthread_cond_t c = PTHREAD_COND_INITIALIZER;*
  - Wait() operation: *pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)*
    - *Assume lock m locked when wait() is called*
    - *Wait() releases the lock, put the calling thread to sleep*
  - Wake-up / Signal() operation: *pthread_cond_signal(pthread_cond_t *c)*
    - *Signal() must re-acquire the lock before returning to the caller*

Method #2: Condition Variable based Waiting

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
void *child(void *arg) {
    printf("child\n");
    /* how to indicate we are done? */
    pthread_mutx_lock(&m);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

What could go wrong with this code?

Method #2: Condition Variable based Waiting

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
void *child(void *arg) {
    printf("child\n");
    /* how to indicate we are done? */
    pthread_mutx_lock(&m);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

Problem: What if child()'s *critical section* runs before main()'s?

Method #4: Condition Variable based Waiting

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
int done = 0;

void *child(void *arg) {
    printf(“child\n”);
    /* how to indicate we are done? */
    done = 1;
    pthread_cond_signal(&c);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    printf(“parent: begin\n”);
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    if (done == 0)
        pthread_cond_wait(&c, NULL);
    printf(“parent: end\n”);
    return 0;
}
```

Problem: A subtle race condition!   Always hold the lock for wait()/signal()!

Method #5: Condition Variable based Waiting

```
int done = 0;
Pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
Pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
    printf("child\n");
    /* how to indicate we are done? */
    pthread_mutx_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);
    /* how to wait for child? */
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
    printf("parent: end\n");
    return 0;
}
```

The Producer/Consumer Problem

- Also known as *bounded buffer* problem, uses cases like
  - e.g., web servers
  - e.g., pipe ("grep foo file.txt | wc –l")
- Problem Statement
  - A buffer, limited size
  - One or more producer threads: add items to the tail of the buffer
  - One or more consumer threads: consume items from the head of the buffer
  - Correct behaviors
    - Producers can only add new items when buffer not full
    - Consumer can only consume items when buffer not empty
    - Producers won't add new items to the same location of the buffer
    - Consumers won't consume the same item from the same buffer location

Producer/Consumer

```
int buffer;
int count = 0; /* initially empty */

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

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

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

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

Producer/Consumer Threads (#1)

```
pthread_cond_t cond;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 1)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Does it work for 1 producer + 1 consumer?

How about more than 1 producers/consumers? e.g., 1 producer and 2 consumers

- Reasons why the previous code fails
  - the state of the bounded buffer changed betwee
    - signal() from the producer, and
    - the actually running of the first consumer
  - signal() is just a hint to wake up a sleeping thread, but doesn't guarantee when a thread runs, the state remains the same … (***Mesa semantics, de-facto***)

Producer/Consumer Threads (#2)

```
pthread_cond_t cond;
pthread_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(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        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(&cond, &mutex);
        int tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Always safe to use "while" to check status change with cond. var.!

IEEE Std. 1003.1

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the `pthread_cond_timedwait()` or `pthread_cond_wait()` functions may occur. Since the return from `pthread_cond_timedwait()` or `pthread_cond_wait()` does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

POSIX.1 2008

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, POSIX.1-2008 explicitly documents that spurious wakeups may occur.

Producer/Consumer Threads (#2)

```
pthread_cond_t cond;
pthread_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(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        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(&cond, &mutex);
        int tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

But the code above is still problematic! Related to the single cond. var.

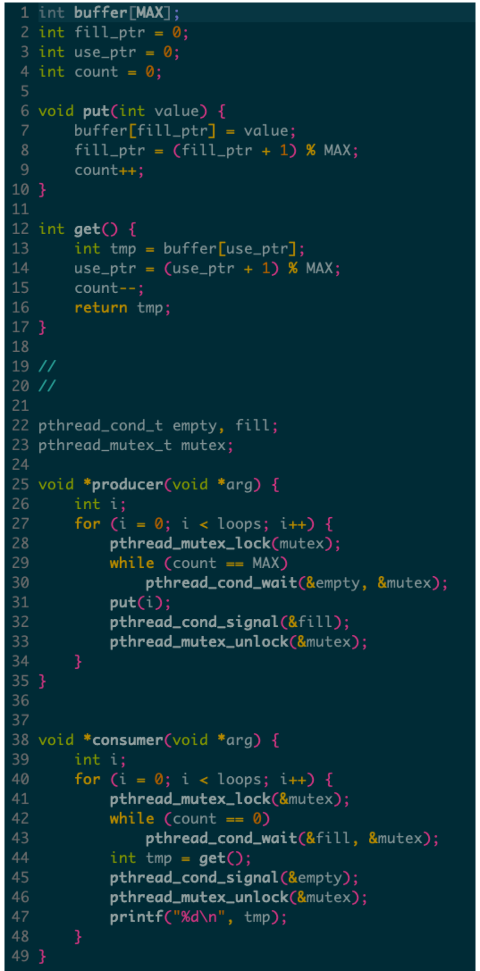
Producer/Consumer Threads (#3)

```
pthread_cond_t empty, fill;
pthread_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);
    }
}
```

Producer/Consumer Threads (#4)



```c
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;
}

//
//

pthread_cond_t empty, fill;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(mutex);
        while (count == MAX)
            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);
    }
}
```

Generalized ...

Signal() and Broadcast()

- Signal() doesn't specify which sleeping thread to wake up
- And it might end up waking up the wrong thread …

```
int bytesLeft = MAX_HEAP_SIZE;
pthread_cont_t c;
pthread_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); // who to signal()
    pthread_mutex_unlock(&m);
}
```