COS40003 Concurrent Programming

Lecture 7: Condition Variable

Outline

- Why condition variable ?
- What is condition variable ?
- How to use condition variable ?
  - (apply to) Parent/Child
  - Producer and Consumer Problem (important)
- Java methods

Recall: Lock

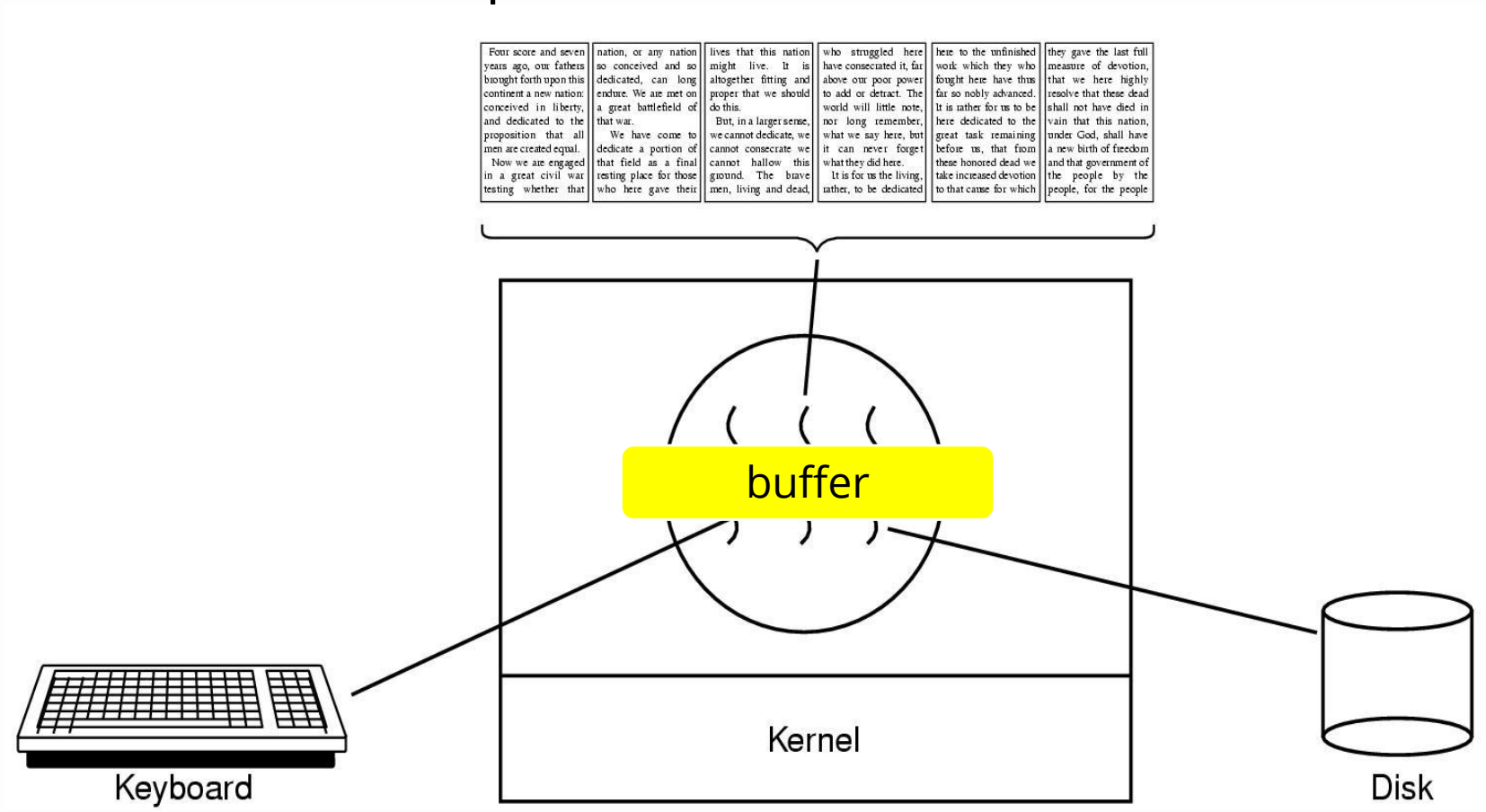
- What lock can do
  - guarantees that only a single thread ever enters a critical section, avoiding races, that is to guarantee mutual exclusion.
- What lock cannot do (but needed by concurrent programs)
  - Thread coordination
  - One common interaction: one thread must wait for another to complete some action before it can continue.
  - For example, when a parent thread creates a child thread to perform a disk I/O and the parent is put to sleep; when the child completes the I/O, the parent needs to be waken up and continue.

Why condition variable?

- There are many cases where a thread wishes to check whether a **condition** is true before continuing.
  - Buffer is full: stop putting;
  - Buffer is empty: stop getting;

Example one: word processor

A word processor with three threads



Example two: video download and play

- Youtube download

- One thread is downloading;
- One thread is playing;

- Pause the video, downloading is stopped (when buffer is full)

Example three: parent waiting for child (implementing join)

```
1   void *child(void *arg) {
2       printf("child\n");
3       // XXX how to indicate we are done?
4   return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL);
11      // create child
11      // XXX how to wait for child?
12      printf("parent: end\n");
13      return 0;
14 }
```

Example three: parent waiting for child (implementing join)

```
1    volatile int done = 0;
2
3    void *child(void *arg) {
4        printf("child\n");
5        done = 1;
6        return NULL;
7    }
8
9    int main(int argc, char *argv[]) {
10       printf("parent: begin\n");
11       pthread_t c;
12       Pthread_create(&c, NULL, child, NULL); // create child
13       while (done == 0);
14               // spin
15       printf("parent: end\n");
16       return 0;
17 }
```

Correct?

Correct!
But, inefficient

How can we do better, with condition variable?

Outline

- Why condition variable ?
- What is condition variable ?
- How to use condition variable ?
  - (apply to) Parent/Child
  - Producer and Consumer Problem (important)
- Java methods

What is condition variable?

- A **condition variable** is an explicit queue that
- Threads can put themselves on when some state (i.e., some **condition**) is not as desired
  - (by **waiting** on the condition);
- Some other thread, when it changes that state, can then wake one (or more) of those waiting threads and thus allow them to continue
  - by **signalling** on the condition.

POSIX condition variable

- Pthread_cond_t c;
  - declares c as a condition variable
- pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
  - The wait() call is executed when a thread wishes to put itself to sleep
- pthread_cond_signal(pthread_cond_t *c);
  - The signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

Outline

- Why condition variable ?
- What is condition variable ?
- How to use condition variable ?
  - (apply to) Parent/Child
  - Producer and Consumer Problem (important)
- Java methods

Parent waiting for child using condition variable (main)

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

int main(int argc, char *argv[]) {
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        thr_join();
        printf("parent: end\n");
        return 0;
}
```

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Case one: child scheduled before parent

Child starts:

1. done=1

2. signal(&c), no waiting thread yet, so nothing happened

Child finishes.

Parent starts:

3. done == 0 is false, wait() is skipped

Parent finishes.

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Case two: child scheduled after parent

Parent starts:

1. Since done==0,

2. wait(), put itself in the waiting queue;

Child starts:

3. done = 1

4. signal(), signal the parent thus waking it

Child finishes

Parent:

5. Waken up from waiting

Parent finishes

Several questions regarding parent/child example

- Question 1
  - Why the child can acquire the lock?
- Question 2
  - Why “unlock m, put thread to sleep and waiting for c” should be atomic?
- Question 3
  - Why signal() inside lock and unlock?
- Question 4
  - Why “while (done==0)” instead of “if (done==0)”?

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 1:

While the parent is waiting, since the lock has been acquired by the parent, how can the child acquire the lock?

Answer:

wait(&c , &m){

1. Unlock m, put itself sleeping, waiting for c (atomically)

   ...

2. After return, lock m

}

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

wait(&c , &m)

1. Unlock m, put itself sleeping, waiting for c (atomically)

...

2. After return, lock m

Question 2:

Why "unlock m, put thread to sleep and waiting for c" should be atomic?

Answer: otherwise

Unlock m;

-- context switch --

Child runs and finishes;

-- context switch back --

Parent waiting forever

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 3:

Why signal() is inside lock and unlock?

Answer:

Explained later in producer-consumer problem

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 4:

Why “while (done==0)”?

Why not “if (done==0)”?

Answer:

Explained later in producer-consumer problem

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 5:

Without the state variable done, is it correct?

Comparison

Original

```
void *child(void *arg) {
    Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
    return NULL;
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
         Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Alternative

```
void *child(void *arg) {
    Pthread_mutex_lock(&m);

    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
    return NULL;
}
void thr_join() {
    Pthread_mutex_lock(&m);

    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 5:

Without the state variable done, is it correct?

Answer: No

If child runs before parent, parent will be waiting for ever.

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    while (done == 0)
        Pthread_cond_wait(&c);
}
```

Question 6:

Have to hold a lock in order to signal and wait?

Comparison

Original

```
void *child(void *arg) {
    Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
    return NULL;
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Alternative

```
void *child(void *arg) {

    done = 1;
    Pthread_cond_signal(&c);

    return NULL;
}
void thr_join() {

    while (done == 0)
        Pthread_cond_wait(&c);

}
```

Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    while (done == 0)
        Pthread_cond_wait(&c);
}
```

Question 6:

Have to hold a lock in order to signal and wait?

Answer: Yes

Otherwise,

Parent checks done==0,

-- contex switch --

Child runs and finishes

-- contex switch back --

Parent calls wait() and will wait forever

Outline

- Why condition variable ?
- What is condition variable ?
- How to use condition variable ?
  - (apply to) Parent/Child
  - Producer and Consumer Problem (important)
- Java methods

Producer and Consumer problem (bounded buffer problem)

- One or more producer threads and one or more consumer threads.
  - Producers generate data items and place them in a buffer;
  - Consumers grab said items from the buffer and consume them in some way.
- Example:
  - in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them

Producer-Consumer

- An example without concurrency
  - What is the problem?
- An example with concurrency control, i.e. lock and condition variable
  - What is the problem?
- The correct example

A single buffer

```
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;
}
```

The first thing we need is a shared buffer, into which a producer puts data, and out of which a consumer takes data.

assert(boolean expression);

The argument boolean expression of *assert()* must be true when the assert() macro is executed, otherwise the program aborts and prints an error message.

Routines to write and read the buffer

```
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 producer that puts an integer into the shared buffer loops number of times,

A consumer gets the data out of that shared buffer (forever)

Producer-Consumer

- An example without concurrency
  - What is the problem?
- An example with concurrency control, i.e. lock and condition variable
  - What is the problem?
- The correct example

Routines to write and read the buffer with lock and condition variable

```
cond_t cond;
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);
    }
}
```

Routines to write and read the buffer with lock and condition variable

```
cond_t cond;
mutex_t 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);
    }
}
```

Any problems?

Problem 1: spurious wakeup

- Assume two consumers C1 and C2, one producer P1
- (1) C1 runs, and finds buffer empty, and goes to sleep;
- (2) P1 comes, checks buffer not full, fills the buffer and signals that a buffer has been filled;
- (3) C1 is moved from sleeping on a condition variable to the ready queue; C1 is now able to run;
- (4) P1 continues until realizing the buffer is full, at which point it sleeps;

Problem 1: spurious wakeup

- The problem occurs:
- (5) another consumer C2 sneaks in and consumes the value in the buffer;
- (6) At this time, C1 returning from the wait, runs, it re-acquires the lock and find the current buffer is empty.

- The problem arises for a simple reason: after the producer woke C1, but before C1 ever ran, the state of the buffer changed

- Clearly, we should somehow prevent C1 from trying to consume because C2 sneaked in and consumed the value in the buffer.

While, Not If

```
cond_t cond;
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);
    }
}
```

While, Not If

```
cond_t cond;
mutex_t mutex;

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
              Pthread_cond_wait(&cond, &mu

        int tmp = get();

        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Mesa semantics

(there is no guarantee that when the woken thread runs, the state will still be as desired)

While --

The opposite

Hoare semantics

If --

Gold rule:

always use while

Recall: Parent waiting for child using condition variable (join and child)

```
void *child(void *arg) {
        printf("child\n");
        Pthread_mutex_lock(&m);
                done = 1;
                Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
        return NULL;
}
void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

Question 3:

Why signal() is inside lock and unlock?

Answer:

Spurious wakeup

Signal after the unlock

- Thread A starts waiting for items to be added to a thread-safe queue.
- Thread B inserts an item on the queue. After unlocking the queue, but before it issues the signal, a context switch occurs.
- Thread C inserts an item on the queue, and issues the cvar signal.
- Thread A wakes up, and processes both items. It then goes back to waiting on the queue.
- Thread B resumes, and signals the cvar.
- Thread A wakes up, then immediately goes back to sleep, because the queue is empty.

Problem 2: wrong wakeup

- (1) Consumers C1 and C2 run first, and go to sleep;
- (2) Producer P1 runs, put a value in the buffer, and wakes one of C1 and C2. Since the buffer is full now, P1 goes to sleep waiting for the condition.
- (3) C1 wakes up, consumes the buffer value, and signals on the condition.
- Now we have a problem

Problem 2: wrong wakeup

- Recall, C1 finished, C2 and P1 are sleeping queue on the condition.
- Clearly, we should wake P1, rather than C2;
- If, unfortunately C2 is waken up (which is definitely possible, depending on how the wait queue is managed), C2 finds now buffer is empty (because of while), C2 is put to sleep again;
- Then, all C1, C2, P1 are sleeping. – A clear bug

Problem 2: wrong wakeup

- Solution guidline:
  - A consumer should not wake other consumers, only producers, and a producer should only wake consumers.
- Solution idea:
  - Two condition variables (two queues): empty, fill
  - Producer threads wait on the condition **empty**, and signals **fill**. Conversely, consumer threads wait on **fill** and signal **empty**.

Solution: use two condition variables

```
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);
    }
}
```

Solution: use two condition variables

```
cond_t empty, fill;
mutex_t 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);
   }
}
```

Wake up all: broadcast

- Example: memory allocation

- (1) At the beginning, assume there are zero bytes free;
- (2) Thread T_a calls allocate(100),
- (3) Followed by thread T_b calls allocate(10);

  Both T_a and T_b wait on a memory-free condition and go to sleep; there are no enough free bytes to satisfy either of these requests.
- (4) Later, assume a third thread, T_c, calls free(50).
- (5) when T_c calls signal to wake a waiting thread, it might not wake the correct waiting thread, T_b, which is waiting for only 10 bytes; T_a should remain waiting, as not enough memory is yet free. The thread waking other threads does not know which thread (or threads) to wake up.

Wake up all: broadcast

- Solution:
  - Instead of sigal(), using broadcast().
- Pthread_cond_broadcast()
  - which wakes up all waiting threads.
- Drawbacks
  - Negative performance, needlessly wake up many threads;
  - All the threads will simply wake up, re-check the condition, and then go back to sleep immediately. (Recall: use “while()” to handle spurious wakeups)

Outline

- Why condition variable ?
- What is condition variable ?
- How to use condition variable ?
  - (apply to) Parent/Child
  - Producer and Consumer Problem (important)
- Java methods

Java Conditions

- Condition instance are intrinsically bound to a lock.
- To obtain a condition instance for a particular Lock instance use its *newCondition()* method.

```
Class myClass{

   Lock lock = new ReentrantLock();
   Condition myCond  = lock.newCondition();
   void myFunc(){
       lock.lock();
       try {
          myCond.await();
       } finally{ lock.unlock(); }
    }
}
```

Java Condition methods

- *await()* forces the current thread to wait until it's signalled or interrupted.
- *await(long time, TimeUnit unit)* forces the current thread to wait until it's signalled or interrupted, or the specified waiting time elapses.
- *signal()* wakes up one waiting thread.
- *signalAll()* wakes up all waiting threads. (Similar to broadcast() in C)

Java condition variable

- Java producer-consumer example will be given in Lab 7

Acknowledgement

- Chapter 30
  - Operating Systems: Three Easy Pieces
- Java documentation
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

SWINBURNE UNIVERSITY OF TECHNOLOGY

Questions?