Concurrency: Condition Variables

Sridhar Alagar

Synchronization Objectives

- Mutual exclusion (e.g., A and B don't run at same time)
    - solved with locks

- Ordering (e.g., B runs after A does something)
    - solved with condition variables and semaphores

Ordering Example: Join

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n");
return 0;
```

how to implement join()?

Condition Variables

- Condition Variable: queue of waiting threads (CV)

- B waits for condition CV before running
  - wait(CV, …)

- A sends signal to CV when the condition is met
  - signal(CV, …)

Join Implementation

Parent:

```
void thread_join() {
            Cond_wait(&c);   // y
}
```

Child:

```
void thread_exit() {
            Cond_signal(&c);        // b
}
```

Example schedule:

```
Parent:      y
Child:                 a
```

Works!

Join Implementation

Parent:

```
void thread_join() {
        Cond_wait(&c);   // y
}
```

Child:

```
void thread_exit() {
        Cond_signal(&c);        // b
}
```

Can you construct ordering that does not work?

Example broken schedule:

```
Parent:                   y

Child:                b
```

parents waits forever

Rule of Thumb 1

- Keep state in addition to CV's!
- CV's are used to signal threads when state changes
- If state is already as needed, thread doesn't wait for a signal!

Join Implementation (Attempt 2)

Parent:

```
void thread_join() {

        if (done == 0)                          // x
                Cond_wait(&c);   // y

}
```

Child:

```
void thread_exit() {

        done = 1                    // b
        Cond_signal(&c);            // c

}
```

Fixes previous broken ordering

Example schedule:

```
Parent:                              x       y

Child:                b       c
```

Join Implementation (Attempt 2)

Parent:

```
void thread_join() {

        if (done == 0)                    // x
                Cond_wait(&c);   // y

}
```

Child:

```
void thread_exit() {

        done = 1                    // b
        Cond_signal(&c);            // c

}
```

```
Parent:     x                  y

Child:            b      c
```

Use mutex to ensure no race between "interacting with state" and wait/signal

Join Implementation (Attempt 3)

Parent:

```
void thread_join() {
        Mutex_lock(&m);                 // w
        if (done == 0)                  // x
                Cond_wait(&c);          // y
        Mutex_unlock(&m);               // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);                 // a
        done = 1                        // b
        Cond_signal(&c);                // c
        Mutex_unlock(&m);               // d
}
```

Parent:	w	x	y	
Child:				a

Both parent and child will waits

Join Implementation (Attempt 3)

Parent:

```
void thread_join() {
        Mutex_lock(&m);                 // w
        if (done == 0)                  // x
                Cond_wait(&c, &m);      // y
        Mutex_unlock(&m);               // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);         // a
        done = 1                // b
        Cond_signal(&c);        // c
        Mutex_unlock(&m);       // d
}
```

Parent:	w	x	y					z
Child:				a	b	c	d	

Rule of Thumb 2

- Acquire lock before checking/updating state and subsequent calling of wait/signal

Condition Variables

- Pthread_cond_wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning

- Pthread_cond_signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

Producer Consumer Problem

- Class of problems where producer generates data/jobs and consumer consumes/services

- Synchronization is required among producers and consumers

Example UNIX Pipes

- A pipe may have many writers and readers

- Internally, there is a finite-sized buffer

- Writers add data to the buffer
  - Writers have to wait if buffer is full

- Readers remove data from the buffer
  - Readers have to wait if buffer is empty

Example UNIX Pipes

Implementation Outline:

- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait

Producer Consumer Solution

Simple case:

- One producer thread
- One consumer thread
- Shared buffer of size 1

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



                do_fill(i);



        }
}
```

```
void *consumer(void *arg) {
        while(1) {



                int tmp = do_get();



                printf(“%d\n”, tmp);
        }
}
```

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);


                do_fill(i);

                Mutex_unlock(&m);
        }
}
```

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);


                int tmp = do_get();

                Mutex_unlock(&m);
                printf(“%d\n”, tmp);
        }
}
```

numfull=0

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);

                Mutex_unlock(&m);
        }
}
```

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);


                int tmp = do_get();
                 Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf(“%d\n”, tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
➡       for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
➡       while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
→       for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                   Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
→               Mutex_lock(&m);
                if(numfull == 0)
                   Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
➡       for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                   Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
➡               if(numfull == 0)
                   Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf(“%d\n”, tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
→       for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                   Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
            →      Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
  ➡   for (int i=0; i<loops; i++) {
            Mutex_lock(&m);
            if(numfull == max)
               Cond_wait(&cond, &m);
            do_fill(i);
            Cond_signal(&cond);
            Mutex_unlock(&m);
      }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
      while(1) {
            Mutex_lock(&m);
            if(numfull == 0)
      ➡      Cond_wait(&cond, &m);
            int tmp = do_get();
            Cond_signal(&cond);
            Mutex_unlock(&m);
            printf("%d\n", tmp);
      }
}
```

numfull=0

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
➡               Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
➡                 Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
      →         if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
      →           Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf(“%d\n”, tmp);
        }
}
```

numfull=1

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
   ➡            Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
       ➡          Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=1

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
        →       Mutex_unlock(&m);
        }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
             →    Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=1

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
    →           Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
    →             Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=1

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
      ➡         if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
      ➡           Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=1

[BLOCKED]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
          →       Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNABLE]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
          →       Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=1

[BLOCKED]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
          →       Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
            →   int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[BLOCKED]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
           ➡      Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[RUNNING]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
                  Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
             ➡  Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNABLE]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
          →       Cond_wait(&cond, &m);
                do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
          →       Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

numfull=0

[RUNNING]

```
void *producer(void *arg) {
        for (int i=0; i<loops; i++) {
                Mutex_lock(&m);
                if(numfull == max)
                  Cond_wait(&cond, &m);
          ➡     do_fill(i);
                Cond_signal(&cond);
                Mutex_unlock(&m);
        }
}
```

[BLOCKED]

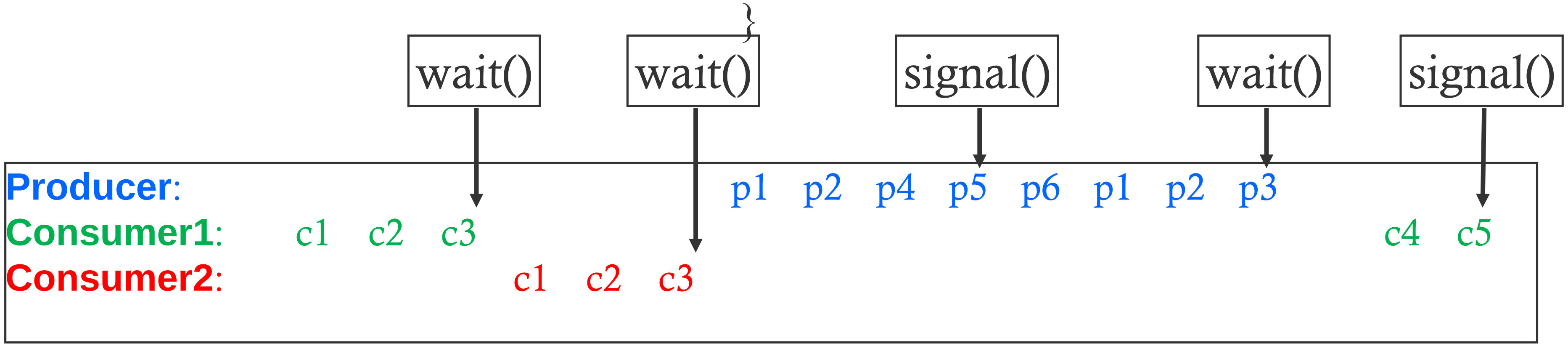
```
void *consumer(void *arg) {
        while(1) {
                Mutex_lock(&m);
                if(numfull == 0)
              ➡   Cond_wait(&cond, &m);
                int tmp = do_get();
                Cond_signal(&cond);
                Mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

How about 2 consumers?

- Will the previous code work?

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```



Producer: p1 p2 p4 p5 p6 p1 p2 p3

Consumer1: c1 c2 c3 c4 c5

Consumer2: c1 c2 c3

does the last signal wake producer or consumer2?

Producer Consumer: Two CVs

```
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                Mutex_lock(&m); // p1
                if (numfull == max) // p2
                   Cond_wait(&empty, &m)
                do_fill(i);  // p4
                Cond_signal(&fill); // p5
                Mutex_unlock(&m); //p6
        }
}
```

```
void *consumer(void *arg) {
        while (1) {
                Mutex_lock(&m);
                if (numfull == 0)
                   Cond_wait(&fill, &m)
                int tmp = do_get();
                Cond_signal(&empty);
                Mutex_unlock(&m);
        }
}
```

Producer Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m)
        do_fill(i);  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m)
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Can you find another bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer1 then reads bad data.

Producer Consumer: Two CVs and while

```
void *producer(void *arg) {
        for (int i = 0; i < loops; i++) {
                Mutex_lock(&m);
                while(numfull == max)
                    Cond_wait(&empty, &m)
                do_fill(i);
                Cond_signal(&fill);
                Mutex_unlock(&m);
        }
}
```

```
void *consumer(void *arg) {
        while (1) {
                Mutex_lock(&m);
                while (numfull == 0)
                    Cond_wait(&fill, &m)
                int tmp = do_get();
                Cond_signal(&empty);
                Mutex_unlock(&m);
        }
}
```

Is this correct?

Correct

- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every do_fill()
- a producer will get to run after every do_get()

Good Rule of Thumb 3

- Whenever a lock is acquired, recheck assumptions about state!
- Possible for another thread to grab lock in between signal and wakeup from wait
- Note that some libraries also have “spurious wakeups” (may wake multiple waiting threads at signal or at any time)

RULES OF THUMB FOR CVs

- Keep state in addition to CV's
- Always do wait/signal with lock held
- Whenever thread wakes from waiting, recheck state

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Arpaci, Prof. Youjip, and other educators. Thanks to all of them.