Operating Systems

Lecture 21: Condition Variables

Nipun Batra

Oct 23, 2018

Concurrency Objectives

Concurrency Objectives

- Mutual exclusion

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time
  - Solved using locks

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time
  - Solved using locks
- Ordering

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time
  - Solved using locks
- Ordering
  - A runs after B

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time
  - Solved using locks
- Ordering
  - A runs after B
  - Solved with?

Concurrency Objectives

- Mutual exclusion
  - A & B don't run at the same time
  - Solved using locks
- Ordering
  - A runs after B
  - Solved with?
    - Join

Concurrency Objectives

- Mutual exclusion
  - A & B don’t run at the same time
  - Solved using locks
- Ordering
  - A runs after B
  - Solved with?
    - Join
      - Implemented using condition variables

How to Join (run child before continuing parent)

```
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); // create child
    //how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

How to Join: Spin based approach

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

How to Join: Spin based approach

Initialising

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

How to Join: Spin based approach

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

Set done to 1 in child

How to Join: Spin based approach

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

Spin wait for child to finish

How to Join: Spin based approach

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

How to Join: Spin based approach

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

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

Spin waiting is very costly!

Condition Variables

Condition Variables

- **Condition variable**

Condition Variables

- **Condition variable**
  - Queue of sleeping threads

Condition Variables

- **Condition variable**
  - Queue of sleeping threads
  - Thread add themselves to queue with **wait**

Condition Variables

- **Condition variable**
  - Queue of sleeping threads
  - Thread add themselves to queue with **wait**
  - Thread wake up threads on the queue with **signal**

Condition Variables

- **Condition variable**
  - Queue of sleeping threads
  - Thread add themselves to queue with **wait**
  - Thread wake up threads on the queue with **signal**

Condition Variables

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
  - Wake a single waiting thread

Condition Variables

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
  - Wake a single waiting thread
  - If there is no waiting thread, just return, do nothing

Exercise: order using condition variables Write thread_exit() an thread_join() using CVs

- **wait (cond_t *cv, mutex_t *lock)**
  - Assumes lock is held when wait() is called
  - Puts caller to sleep + atomically releases lock
  - When awoken, reacquires lock before returning
- **signal (cond_t *cv)**
  - Wake a single waiting thread
  - If there is no waiting thread, just return, do nothing

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }


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

Exercise: order using condition variables Attempt #1

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables
Attempt #1

```
void thread_exit {
     mutex_lock(&m)         //a
    cond_signal(&c)        //b
    mutex_unlock(&m) } //c
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) } //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) }    //c
```

```
void thread_join {
    mutex_lock(&m)        //x
    cond_wait(&c, &m)     //y
    mutex_unlock(&m) }    //z
```

```
1   void *child(void *arg) {
2       printf("child\n");
3       thread_exit()
4       return NULL; }

7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL); // create child
11      thread_join()
12      printf("parent: end\n");
13      return 0; }
```

Order

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) } //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }


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

Order

Parent	Child

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) }    //c
```

```
void thread_join {
    mutex_lock(&m)           //x
    cond_wait(&c, &m)        //y
    mutex_unlock(&m) }       //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	
	a

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) }    //c
```

```
void thread_join {
    mutex_lock(&m)        //x
    cond_wait(&c, &m)     //y
    mutex_unlock(&m) }    //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	
	a
	b

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	
	a
	b
	c

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)       //a
    cond_signal(&c)      //b
    mutex_unlock(&m) }   //c
```

```
void thread_join {
    mutex_lock(&m)         //x
    cond_wait(&c, &m)      //y
    mutex_unlock(&m) }     //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	
	a
	b
	c
z	

Exercise: order using condition variables Attempt #1

```
void thread_exit {
     mutex_lock(&m)        //a
   cond_signal(&c)        //b
   mutex_unlock(&m) } //c
```

```
void thread_join {
     mutex_lock(&m)            //x
   cond_wait(&c, &m)        //y
   mutex_unlock(&m) }       //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
x	
y	
	a
	b
	c
z	

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) }    //c
```

```
void thread_join {
    mutex_lock(&m)            //x
    cond_wait(&c, &m)         //y
    mutex_unlock(&m) }        //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)        //a
    cond_signal(&c)       //b
    mutex_unlock(&m) }    //c
```

```
void thread_join {
    mutex_lock(&m)            //x
    cond_wait(&c, &m)         //y
    mutex_unlock(&m) }        //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b
	c

Exercise: order using condition variables Attempt #1

```
void thread_exit {
     mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) } //c
```

```
void thread_join {
     mutex_lock(&m)              //x
    cond_wait(&c, &m)           //y
    mutex_unlock(&m) }          //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b
x	c

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b
x	c
y	

Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

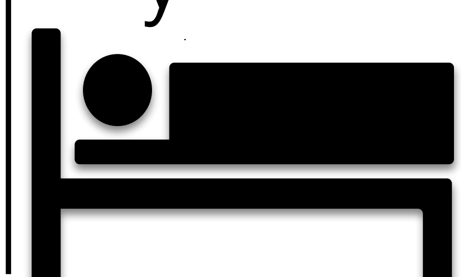
```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b
x	c
y	



Exercise: order using condition variables Attempt #1

```
void thread_exit {
    mutex_lock(&m)          //a
    cond_signal(&c)         //b
    mutex_unlock(&m) }      //c
```

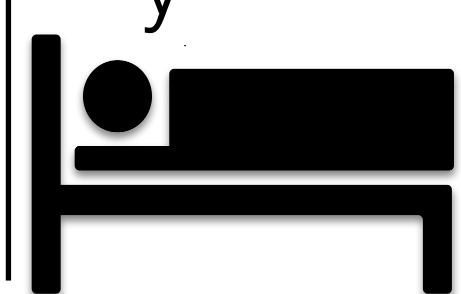
```
void thread_join {
    mutex_lock(&m)          //x
    cond_wait(&c, &m)       //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
	a
	b
x	c
y	



Rule of Thumb #1

Rule of Thumb #1

- In addition to condition variables use another variable to capture state

Rule of Thumb #1

- In addition to condition variables use another variable to capture state
- CVs can be used to nudge threads when state changes

Exercise: order using condition variables Attempt #2

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1                //a
   cond_signal(&c)        //b
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #2

```
void thread_exit {
    Done  = 1              //a
    cond_signal(&c)        //b
```

```
void thread_join {
    mutex_lock(&m)             //w
    If (done==0)               //x
        cond_wait(&c, &m)      //y
    mutex_unlock(&m) }         //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #2

```
void thread_exit {
    Done  = 1                 //a
    cond_signal(&c)           //b
```

```
void thread_join {
    mutex_lock(&m)             //w
    If (done==0)               //x
        cond_wait(&c, &m)      //y
    mutex_unlock(&m) }         //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1                 //a
   cond_signal(&c)              //b
```

```
void thread_join {
    mutex_lock(&m)              //w
   If (done==0)                 //x
      cond_wait(&c, &m)         //y
   mutex_unlock(&m) }           //z
```

```
void *child(void *arg) {
   printf("child\n");
   thread_exit()
   return NULL; }

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

Order

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1               //a
   cond_signal(&c)           //b
```

```
void thread_join {
      mutex_lock(&m)           //w
     If (done==0)              //x
         cond_wait(&c, &m)     //y
     mutex_unlock(&m) }        //z
```

```
void *child(void *arg) {
   printf("child\n");
   thread_exit()
   return NULL; }

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

Order

Parent

Exercise: order using condition variables Attempt #2

```
void thread_exit {
    Done = 1                //a
    cond_signal(&c)         //b
```

```
void thread_join {
    mutex_lock(&m)          //w
    If (done==0)            //x
        cond_wait(&c, &m)   //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order	
Parent	Child

Exercise: order using condition variables Attempt #2

```
void thread_exit {
    Done  = 1                  //a
    cond_signal(&c)            //b
```

```
void thread_join {
    mutex_lock(&m)             //w
    If (done==0)               //x
        cond_wait(&c, &m)      //y
    mutex_unlock(&m) }         //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	

Exercise: order using condition variables Attempt #2

```
void thread_exit {
    Done  = 1              //a
    cond_signal(&c)        //b
```

```
void thread_join {
    mutex_lock(&m)             //w
    If (done==0)               //x
        cond_wait(&c, &m)      //y
    mutex_unlock(&m) }         //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	

Exercise: order using condition variables Attempt #2

```
void thread_exit {
        Done  = 1                //a
     cond_signal(&c)             //b
```

```
void thread_join {
    mutex_lock(&m)               //w
    If (done==0)                 //x
        cond_wait(&c, &m)        //y
    mutex_unlock(&m) }           //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	
	a

Exercise: order using condition variables Attempt #2

```
void thread_exit {
        Done  = 1                //a
     cond_signal(&c)        //b
```

```
void thread_join {
     mutex_lock(&m)          //w
    If (done==0)                //x
          cond_wait(&c, &m)  //y
    mutex_unlock(&m) }       //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	
	a
	b

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1                //a
    cond_signal(&c)        //b
```

```
void thread_join {
    mutex_lock(&m)              //w
    If (done==0)                 //x
        cond_wait(&c, &m)  //y
    mutex_unlock(&m) }        //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	
	a
	b
y	

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1                 //a
    cond_signal(&c)             //b
```

```
void thread_join {
      mutex_lock(&m)            //w
     If (done==0)               //x
          cond_wait(&c, &m)     //y
     mutex_unlock(&m) }         //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	
	a
	b
y	

Exercise: order using condition variables Attempt #2

```
void thread_exit {
      Done  = 1                    //a
    cond_signal(&c)          //b
```

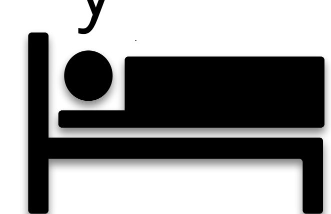
```
void thread_join {
     mutex_lock(&m)             //w
    If (done==0)                 //x
         cond_wait(&c, &m)  //y
    mutex_unlock(&m) }       //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Order

Parent	Child
w	
x	
	a
	b
y	



Exercise: order using condition variables Correct Solution

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Correct Solution

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Exercise: order using condition variables Correct Solution

```
void thread_exit {
    mutex_lock(&m)
    Done  = 1
    cond_signal(&c)
    mutex_unlock(&m)
```

```
void thread_join {
    mutex_lock(&m)          //w
    while (done==0)         //x
        cond_wait(&c, &m)   //y
    mutex_unlock(&m) }      //z
```

```
void *child(void *arg) {
    printf("child\n");
    thread_exit()
    return NULL; }

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

Rule of Thumb #2

Rule of Thumb #2

- Wait and signal while holding the lock

The Producer Consumer Problem

The Producer Consumer Problem

- Producers produce data and place it on a shared resource

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer
    - grep foo file.txt | wc –l

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer
    - grep foo file.txt | wc –l
    - The grep process is the producer.

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer
    - grep foo file.txt | wc –l
    - The grep process is the producer.
    - The wc process is the consumer.

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer
    - grep foo file.txt | wc –l
    - The grep process is the producer.
    - The wc process is the consumer.
    - Between them is an in-kernel bounded buffer.

The Producer Consumer Problem

- Producers produce data and place it on a shared resource
- Example:
  - Multi-threaded web server:
    - Multiple request coming in concurrently - Producers
    - Multiple responses concurrently - Consumers
  - Bounded buffer
    - grep foo file.txt | wc –l
    - The grep process is the producer.
    - The wc process is the consumer.
    - Between them is an in-kernel bounded buffer.
-

The Producer Consumer Problem

Bounded Buffer

The Producer Consumer Problem

Bounded Buffer

The Producer Consumer Problem

Producer adds to the buffer

Bounded Buffer

The Producer Consumer Problem

Producer adds to the buffer

Bounded Buffer

The Producer Consumer Problem

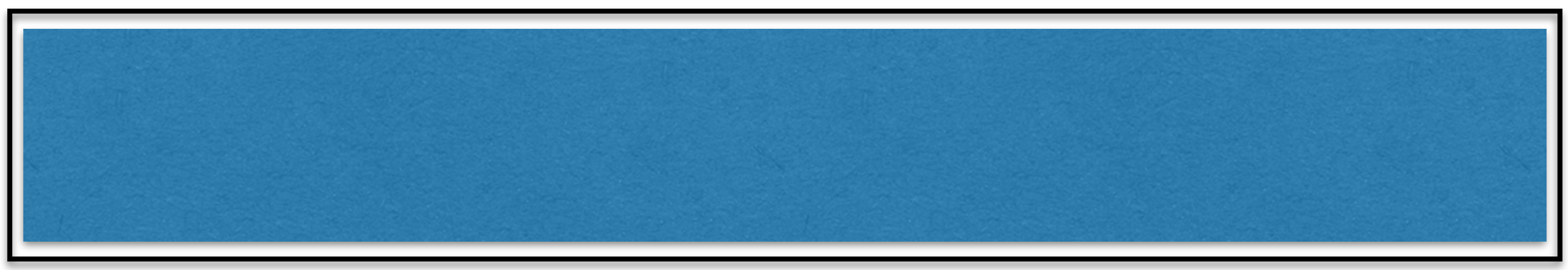
Producer adds to the buffer

Consumer removes from the buffer

Bounded Buffer

The Producer Consumer Problem

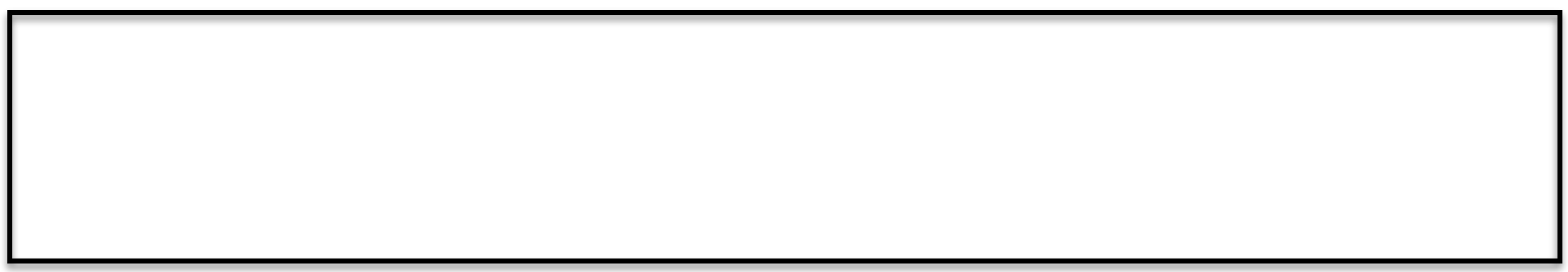
Buffer Full - Producer(s) have to wait



Bounded Buffer

The Producer Consumer Problem

Buffer Empty - Consumer(s) have to wait



Bounded Buffer

The Producer Consumer Problem (Buffer size =1)

```
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 Producer Consumer Problem (Buffer size =1)

```
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;
}
```

Insert into buffer (produce) only if buffer is empty

The Producer Consumer Problem (Buffer size =1)

```
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;
}
```

Delete from buffer (consume) only if buffer is full

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
 }
 }
```

Producer puts an integer into the shared buffer loops number of times.

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```c
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);
}
}
```

Consumer gets data out of the buffer.

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

What's the problem

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```c
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);
 }
 }
```

What’s the problem with this approach?

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

What’s the problem with this approach?

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

What’s the problem with this approach?

Multiple threads accessing

The Producer Consumer Problem (Buffer size =1) [Attempt #1]

```
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);
}
}
```

What's the problem with this approach?

Multiple threads accessing shared resource without locking

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
int i;
while (1) { Pthread_mutex_lock(&mutex);
int tmp = get();
Pthread_mutex_unlock(&mutex);
printf("%d\n", tmp);
} }
```

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
 int i;
 while (1) { Pthread_mutex_lock(&mutex);
 int tmp = get();
 Pthread_mutex_unlock(&mutex);
 printf("%d\n", tmp);
 } }
```

What’s the problem

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```c
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
 int i;
 while (1) { Pthread_mutex_lock(&mutex);
 int tmp = get();
 Pthread_mutex_unlock(&mutex);
 printf("%d\n", tmp);
 } }
```

What's the problem with this approach?

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
int i;
while (1) { Pthread_mutex_lock(&mutex);
int tmp = get();
Pthread_mutex_unlock(&mutex);
printf("%d\n", tmp);
} }
```

What's the problem with this approach?

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
int i;
while (1) { Pthread_mutex_lock(&mutex);
int tmp = get();
Pthread_mutex_unlock(&mutex);
printf("%d\n", tmp);
} }
```

What's the problem with this approach?

No explicit waiting

The Producer Consumer Problem (Buffer size =1) [Attempt #2]

```
void *producer(void *arg) {
int i;
int loops = (int) arg;
for (i = 0; i < loops; i++) {
Pthread_mutex_lock(&mutex);
put(i);
Pthread_mutex_unlock(&mutex);}
}
  void *consumer(void *arg) {
int i;
while (1) { Pthread_mutex_lock(&mutex);
int tmp = get();
Pthread_mutex_unlock(&mutex);
printf("%d\n", tmp);
} }
```

What's the problem with this approach?

No explicit waiting on empty and full buffer