CS 471 Operating Systems

Yue Cheng

George Mason University
Spring 2019

Announcement

- Homework 1 posted on BB
- Due next Wednesday, Feb 20, 9:00am, on BB

Review: Synchronization Worksheet

Review: Semaphores

Semaphores

- Motivation: Avoid busy waiting by blocking a process execution until some **condition** is satisfied

- Two operations are defined on a semaphore variable s:

```
sem_wait(s)   (also called P(s) or down(s))
sem_post(s)   (also called V(s) or up(s))
```

Semaphore as a General Synchronization Tool

- Semaphores provide a general synchronization mechanism beyond the “critical section” problem

- Example problem: Execute *B* in P_j only after *A* executed in P_i
- Use semaphore flag initialized to 0
- Code:

```
Pi :                    Pj :
 ⋮                       ⋮
 A                       sem_wait(flag)
sem_post(flag)           B
```

Join with Semaphore

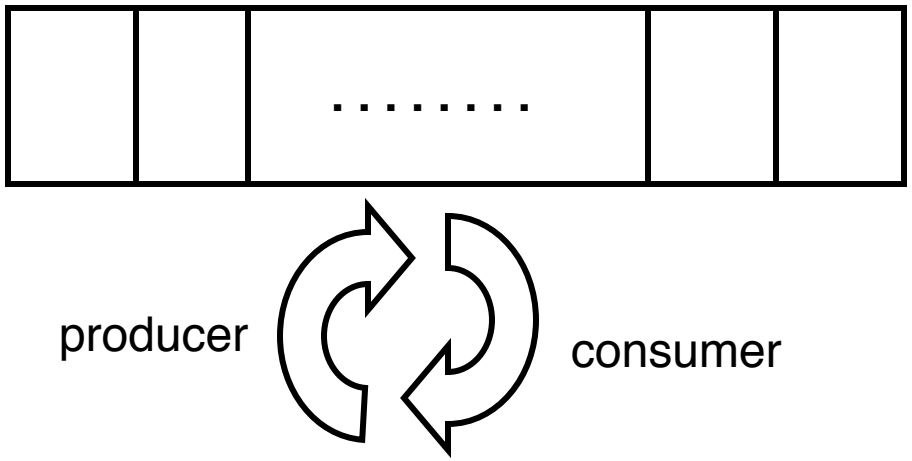
```
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s);
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0);
    pthread_t c;
    printf("parent: begin\n");
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

Classical Problems of Synchronization

- Producer-Consumer Problem
  - Semaphore version
  - Condition Variable
    - A CV-based version
- Readers-Writers Problem
- Dining-Philosophers Problem

Producer-Consumer Problem

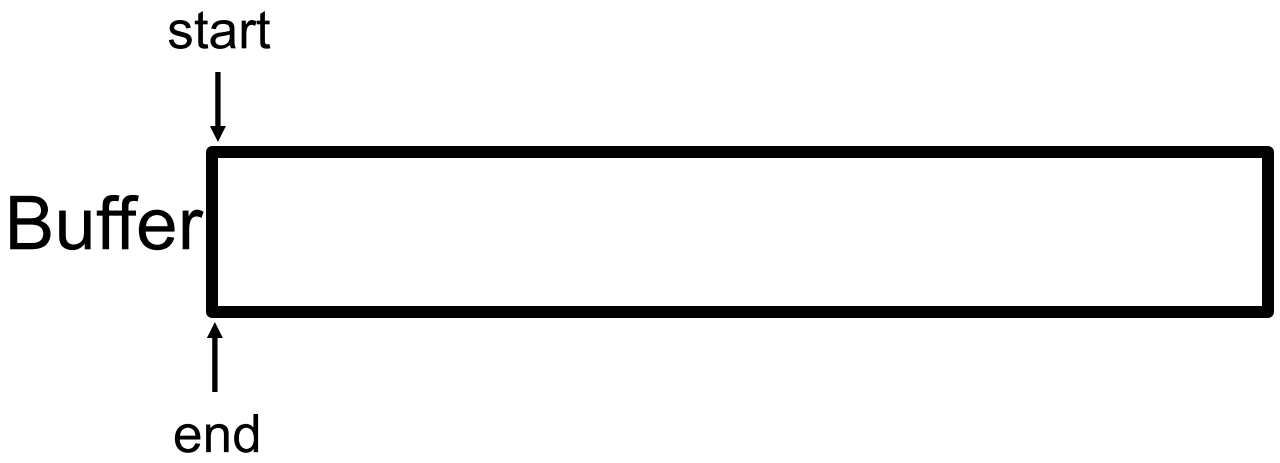
- The *bounded-buffer* producer-consumer problem assumes that there is a buffer of size *N*
- The producer process puts items to the buffer area
- The consumer process consumes items from the buffer
- The producer and the consumer execute *concurrently*



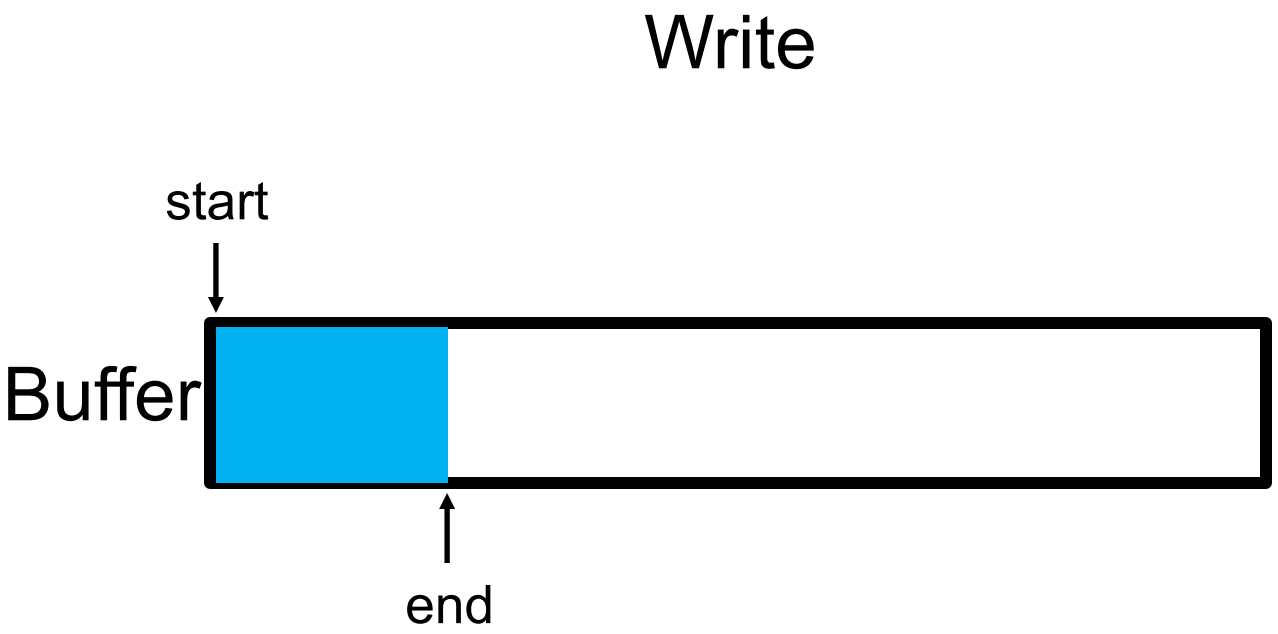
Example: UNIX Pipes

- A pipe may have many writers and readers
- Internally, there is a finite-sized buffer
- Writers add data to the buffer
- Readers remove data from the buffer

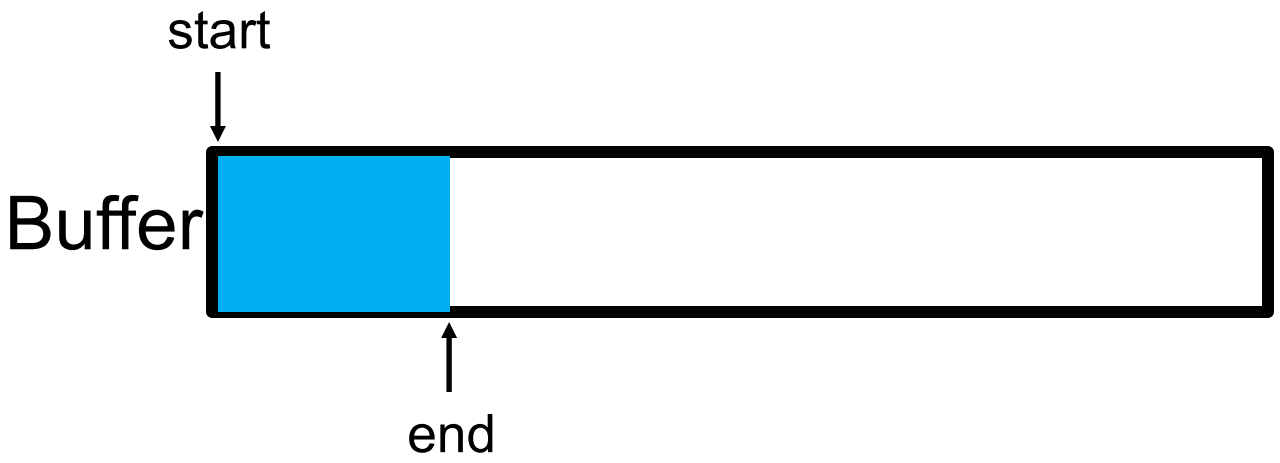
Example: UNIX Pipes



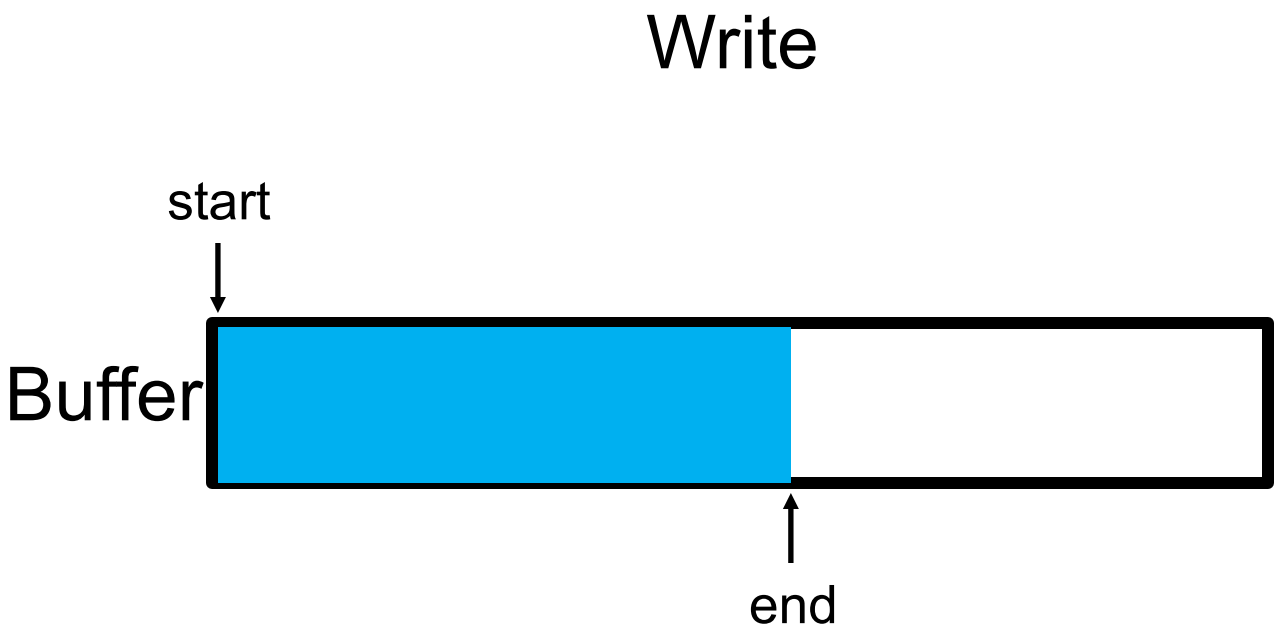
Example: UNIX Pipes



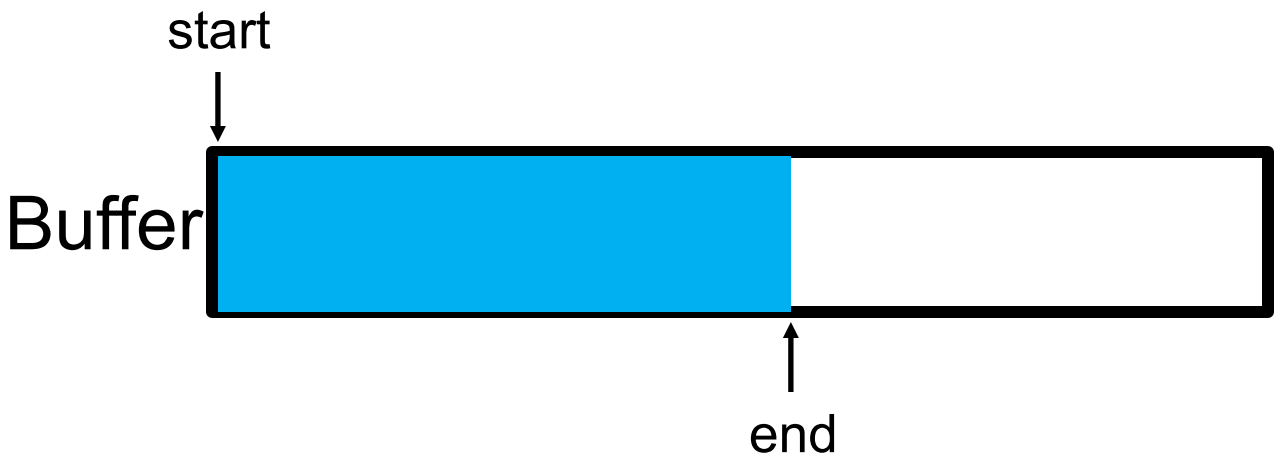
Example: UNIX Pipes



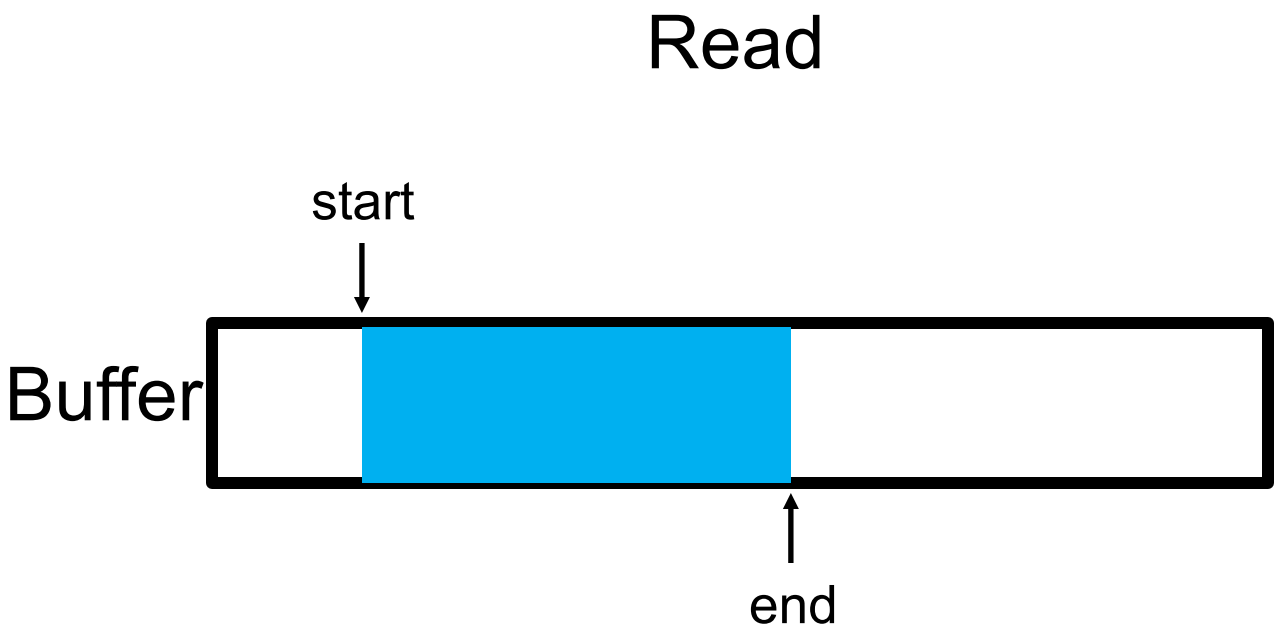
Example: UNIX Pipes



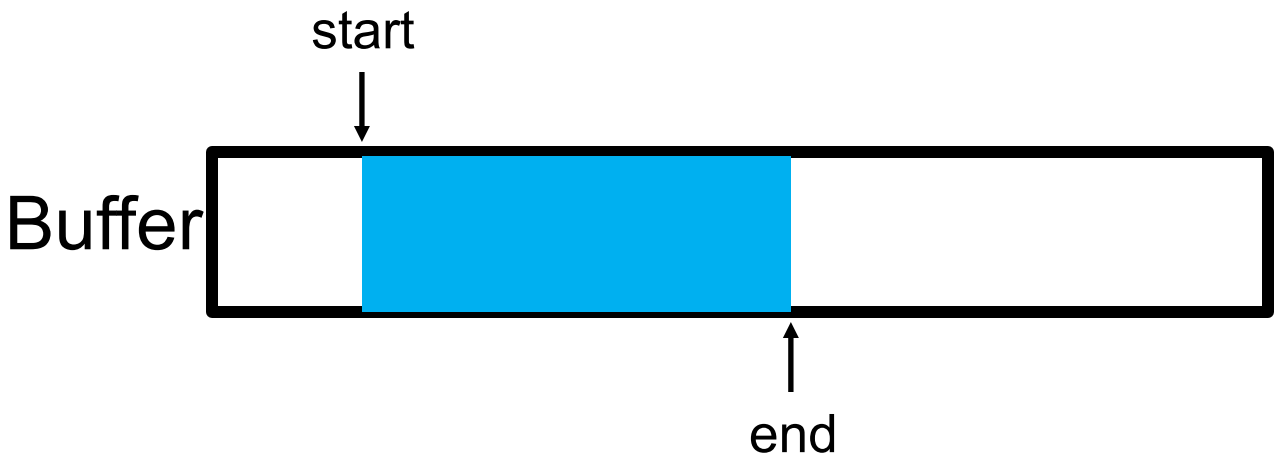
Example: UNIX Pipes



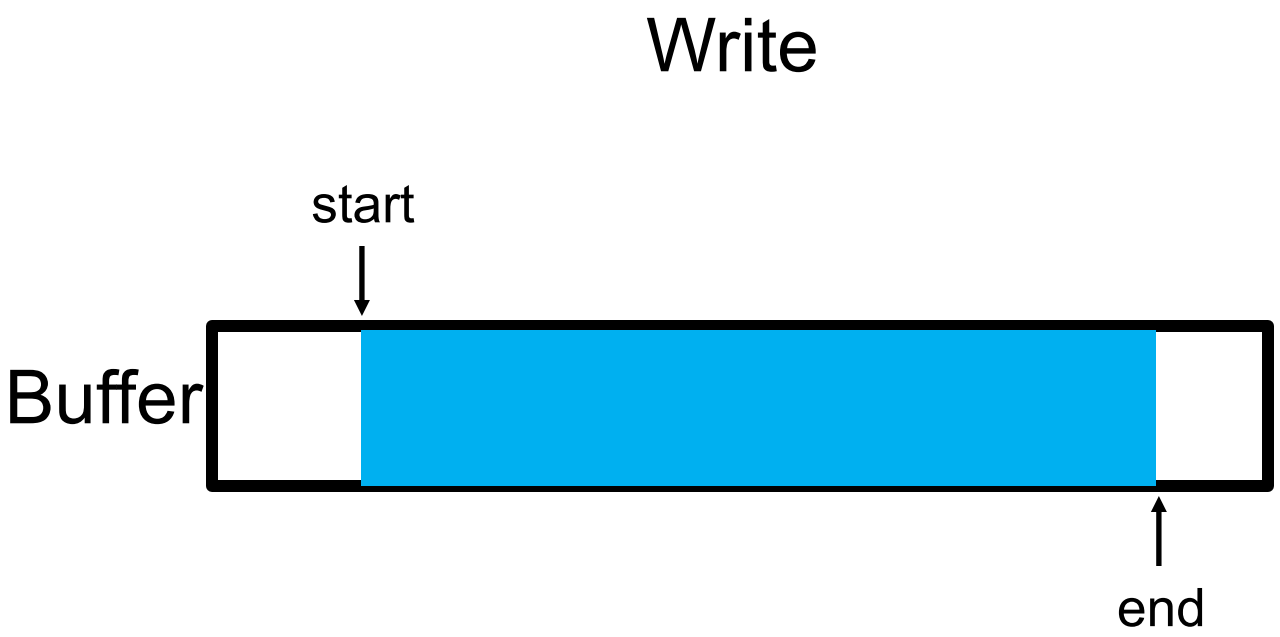
Example: UNIX Pipes



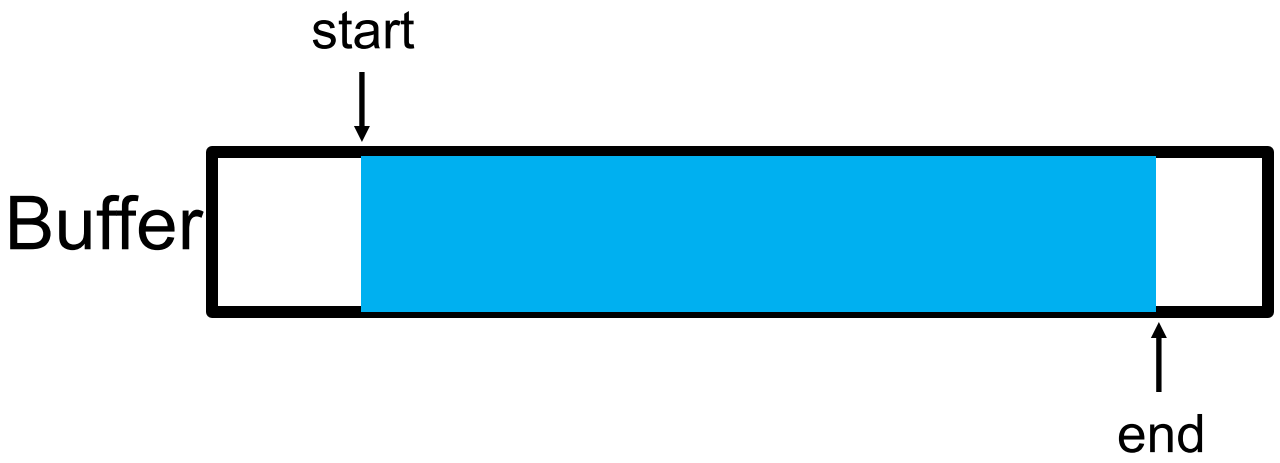
Example: UNIX Pipes



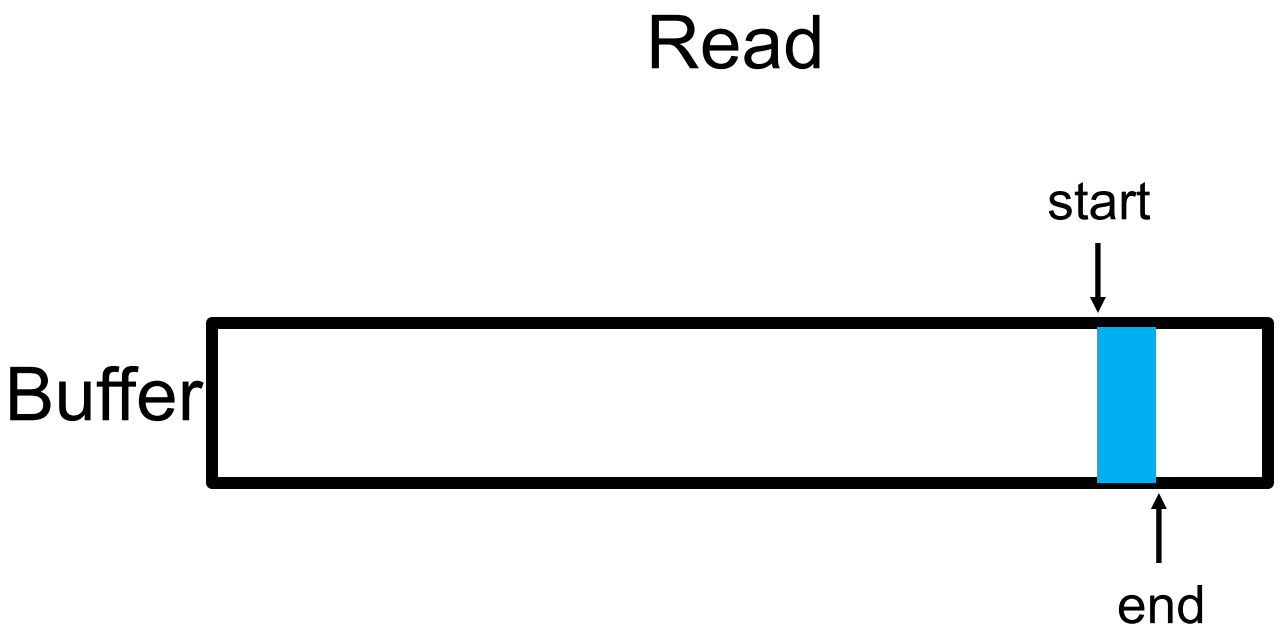
Example: UNIX Pipes



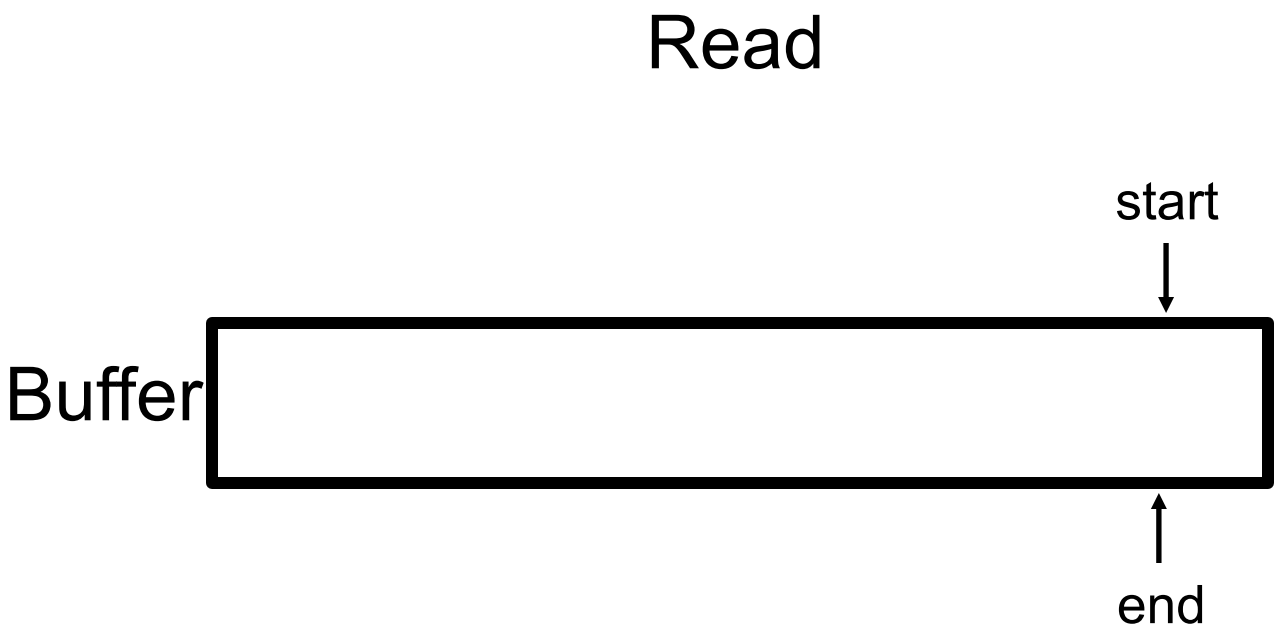
Example: UNIX Pipes



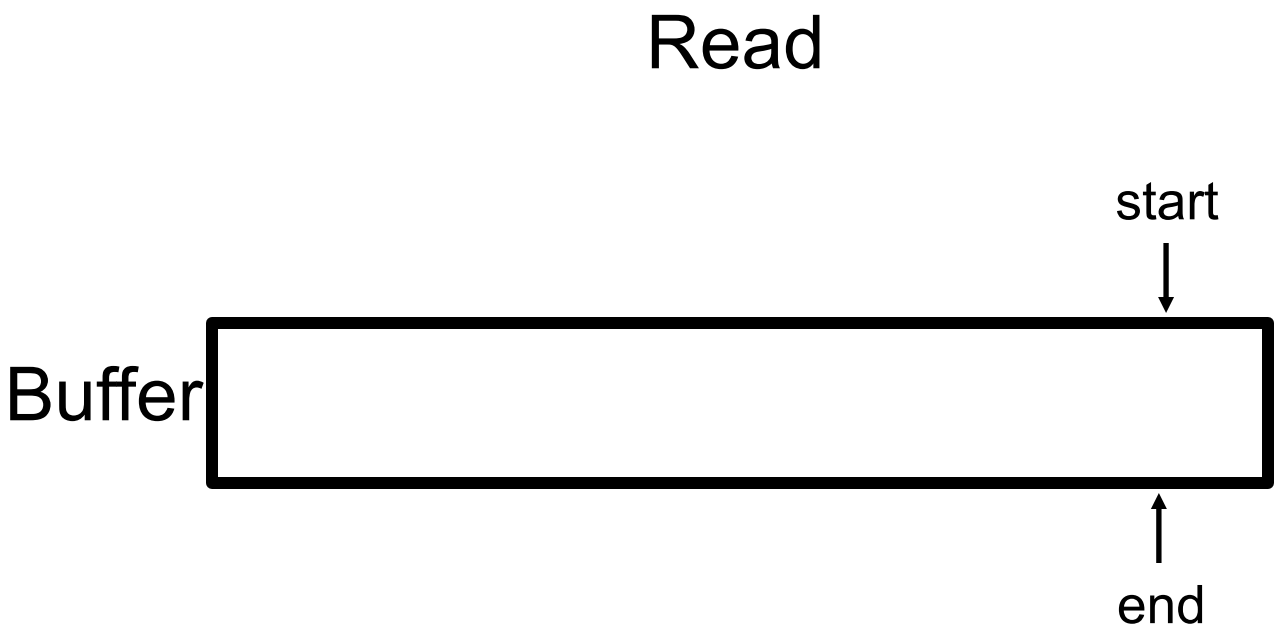
Example: UNIX Pipes



Example: UNIX Pipes

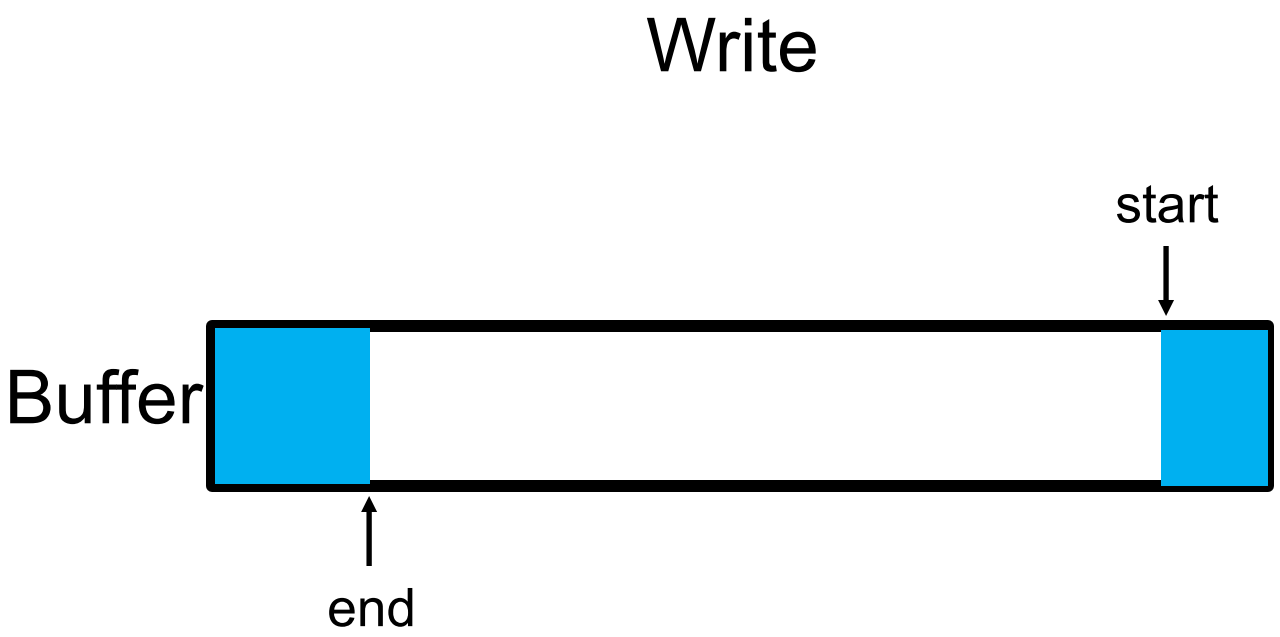


Example: UNIX Pipes

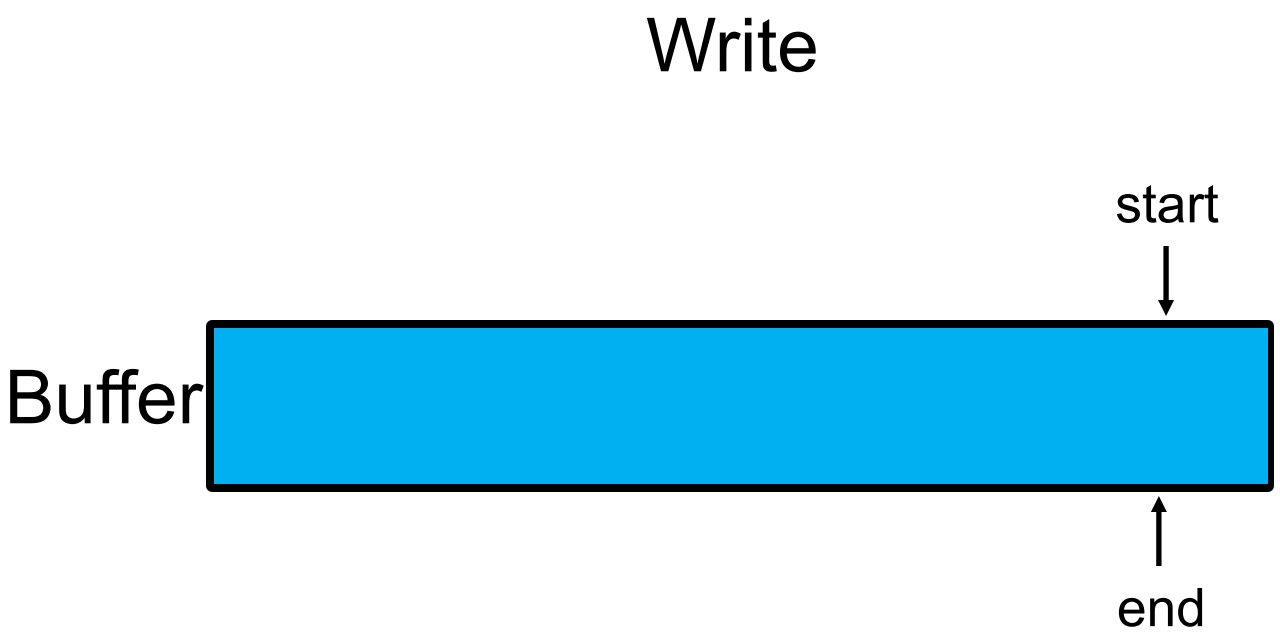


Note: reader must **wait**

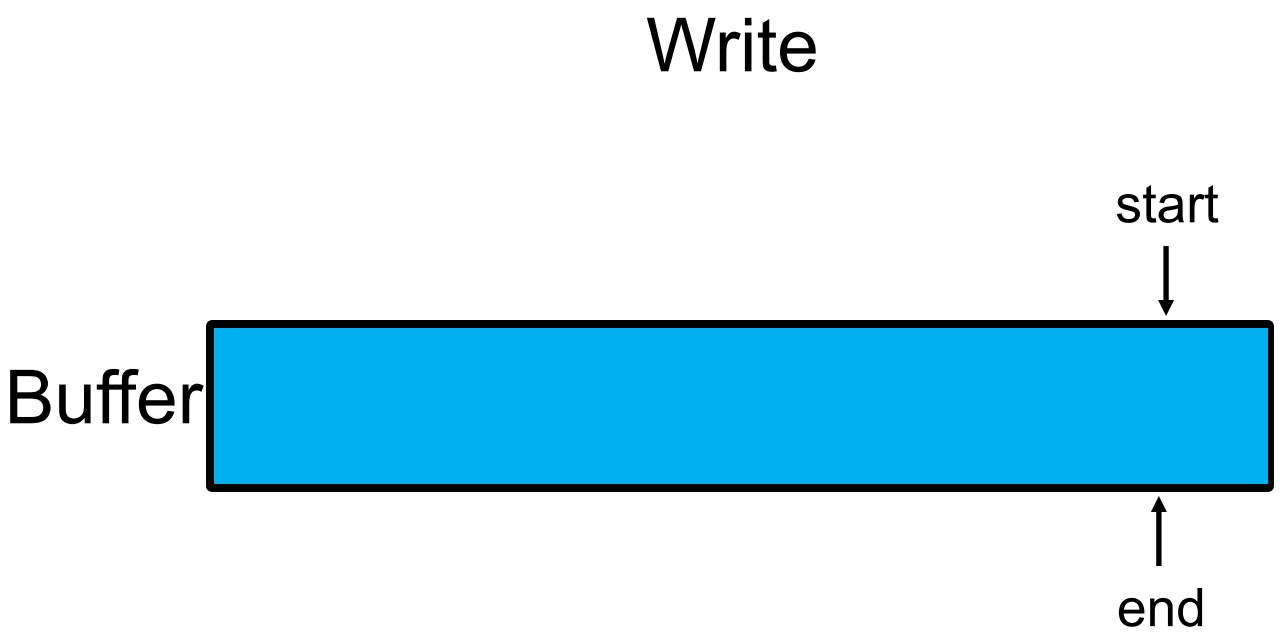
Example: UNIX Pipes



Example: UNIX Pipes



Example: UNIX Pipes



Note: writer must **wait**

Example: UNIX Pipes

- Implementation
  - Reads/writes to buffer require **locking**
  - When buffers are **full**, writers (producers) **must wait**
  - When buffers are **empty**, readers (consumers) **must wait**

Example: UNIX Pipes

Demo

```
% ps aux | less
```

Pipe

```
% cat file | grep <str>
```

Pipe

Producer-Consumer Model: Parameters

- Shared data:

```
sem_t full, empty;
```

- Initially:

```
full = 0        /* The number of full buffers */
empty = MAX     /* The number of empty buffers */
```

First Attempt: MAX = 1

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           sem_wait(&empty);           // line P1
8           put(i);                     // line P2
9           sem_post(&full);            // line P3
10      }
11  }
12
13  void *consumer(void *arg) {
14      int i, tmp = 0;
15      while (tmp != -1) {
16          sem_wait(&full);            // line C1
17          tmp = get();                // line C2
18          sem_post(&empty);           // line C3
19          printf("%d\n", tmp);
20      }
21  }
22
23  int main(int argc, char *argv[]) {
24      // ...
25      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26      sem_init(&full, 0, 0);    // ... and 0 are full
27      // ...
28  }
```

```
1   int buffer[MAX];
2   int fill = 0;
3   int use  = 0;
4
5   void put(int value) {
6       buffer[fill] = value;
7       fill = (fill + 1) % MAX;
8   }
9
10  int get() {
11      int tmp = buffer[use];
12      use = (use + 1) % MAX;
13      return tmp;
14  }
```

Put and Get routines

First Attempt: MAX = 10?

```
1   sem_t empty;
2   sem_t full;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           sem_wait(&empty);           // line P1
8           put(i);                     // line P2
9           sem_post(&full);            // line P3
10      }
11  }
12
13  void *consumer(void *arg) {
14      int i, tmp = 0;
15      while (tmp != -1) {
16          sem_wait(&full);            // line C1
17          tmp = get();                // line C2
18          sem_post(&empty);           // line C3
19          printf("%d\n", tmp);
20      }
21  }
22
23  int main(int argc, char *argv[]) {
24      // ...
25      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
26      sem_init(&full, 0, 0);    // ... and 0 are full
27      // ...
28  }
```

```
1   int buffer[MAX];
2   int fill = 0;
3   int use  = 0;
4
5   void put(int value) {
6       buffer[fill] = value;
7       fill = (fill + 1) % MAX;
8   }
9
10  int get() {
11      int tmp = buffer[use];
12      use = (use + 1) % MAX;
13      return tmp;
14  }
```

Put and Get routines

First Attempt: MAX = 10?

fill = 0

empty = 10

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

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

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
    buffer[fill] = value;
    Interrupted …
    fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: **Sleeping**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
    buffer[fill] = value;
    Interrupted …
    fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0

empty = 9

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
      buffer[fill] = value;
      Interrupted …
      fill = (fill + 1) % MAX;
}
```

First Attempt: MAX = 10?

fill = 0
Overwrite!
empty = 8

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

```
void put(int value) {
    buffer[fill] = value;
    Interrupted …
    fill = (fill + 1) % MAX;
}
```

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

One More Parameter: A mutex lock

- Shared data:

```
sem_t full, empty;
```

- Initially:

```
full = 0;     /* The number of full buffers */
empty = MAX; /* The number of empty buffers */
mutex = 1;    /* Semaphore controlling the access
              to the buffer pool */
```

Add “Mutual Exclusion”

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);           // line p0 (NEW LINE)
9           sem_wait(&empty);           // line p1
10          put(i);                     // line p2
11          sem_post(&full);            // line p3
12          sem_post(&mutex);           // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);           // line c0 (NEW LINE)
20          sem_wait(&full);            // line c1
21          int tmp = get();            // line c2
22          sem_post(&empty);           // line c3
23          sem_post(&mutex);           // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33      // ...
34  }
```

Add “Mutual Exclusion”

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&mutex);           // line p0 (NEW LINE)
9           sem_wait(&empty);           // line p1
10          put(i);                     // line p2
11          sem_post(&full);            // line p3
12          sem_post(&mutex);           // line p4 (NEW LINE)
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&mutex);           // line c0 (NEW LINE)
20          sem_wait(&full);            // line c1
21          int tmp = get();            // line c2
22          sem_post(&empty);           // line c3
23          sem_post(&mutex);           // line c4 (NEW LINE)
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock (NEW LINE)
33      // ...
34  }
```

What if consumer gets to run first??

Adding “Mutual Exclusion”

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: **Running**

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Adding “Mutual Exclusion”

Producer 0: Runnable

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: Runnable

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Consumer 0 is waiting for full to be greater than 0

Adding “Mutual Exclusion”

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Consumer 0: Runnable

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Consumer 0 is waiting for full to be greater than 0

Adding “Mutual Exclusion”

Deadlock!!

Producer 0: **Running**

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Producer 0 gets stuck at acquiring mutex which has been locked by Consumer 0!

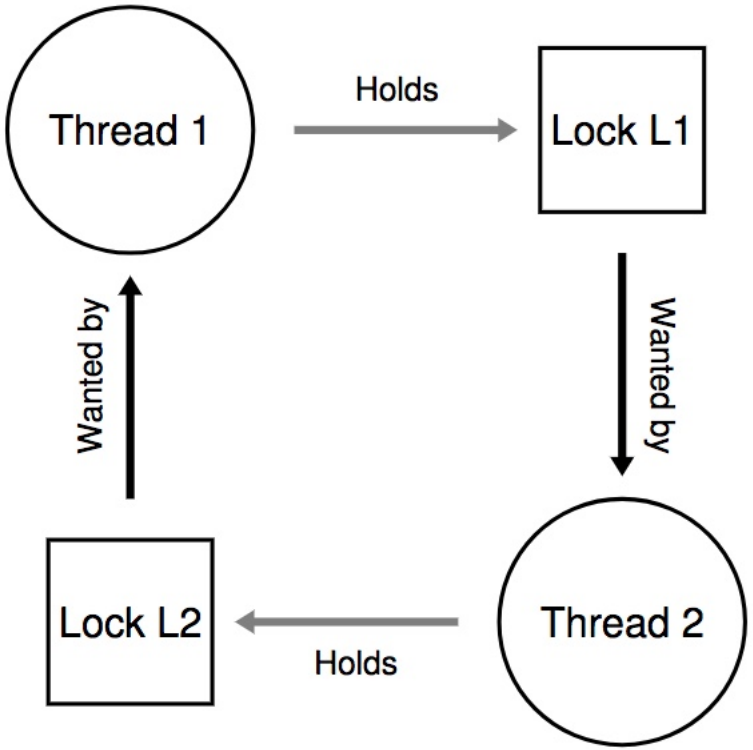
Consumer 0: Runnable

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Consumer 0 is waiting for full to be greater than 0

Deadlocks

- A set of threads are said to be in a ***deadlock*** state when every thread in the set is waiting for an event that can be caused only by another thread in the set



A typical deadlock dependency graph

Conditions for Deadlock

- **Mutual exclusion**
  - Threads claim exclusive control of resources that require (e.g., a thread grabs a lock)
- **Hold-and-wait**
  - Threads hold resources allocated to them while waiting for additional resources
- **No preemption**
  - Resources cannot be forcibly removed from threads that are holding them
- **Circular wait**
  - There exists a circular chain of threads such that each holds one or more resources that are being requests by next thread in chain

Correct Mutual Exclusion

```
1   sem_t empty;
2   sem_t full;
3   sem_t mutex;
4
5   void *producer(void *arg) {
6       int i;
7       for (i = 0; i < loops; i++) {
8           sem_wait(&empty);           // line p1
9           sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
10          put(i);                     // line p2
11          sem_post(&mutex);           // line p2.5 (... AND HERE)
12          sem_post(&full);            // line p3
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          sem_wait(&full);            // line c1
20          sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
21          int tmp = get();            // line c2
22          sem_post(&mutex);           // line c2.5 (... AND HERE)
23          sem_post(&empty);           // line c3
24          printf("%d\n", tmp);
25      }
26  }
27
28  int main(int argc, char *argv[]) {
29      // ...
30      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
31      sem_init(&full, 0, 0);    // ... and 0 are full
32      sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
33      // ...
34  }
```

Mutex wraps just around critical section!

Mutex wraps just around critical section!

Producer-Consumer Solution

- Make sure that
  1. The producer and the consumer do not access the buffer area and related variables at the same time
  2. No item is made available to the consumer if all the buffer slots are empty
  3. No slot in the buffer is made available to the producer if all the buffer slots are full

Condition Variables

Condition Variables

A parent waiting for its child

```
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); // create child
11      // XXX how to wait for child?
12      printf("parent: end\n");
13      return 0;
14  }
```

Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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  }
```

Spin-based Approach

Using a shared variable, parent spins until child set it to 1

```
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  }
```

What's the problem of this approach?

Condition Variables (CV)

- Definition:
  - An explicit queue that threads can put themselves when some **condition** is not as desired (by **waiting** on the condition)
  - Other thread can wake one of those waiting threads to allow them to continue (by **signaling** on the condition)
- Pthread CV

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

CV-based Approach

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

??

```
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;
}
```

??

Broken Implementation 1

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

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
10      Pthread_mutex_unlock(&m);
11  }
```

```
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;
}
```

Broken Implementation 1

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

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
10      Pthread_mutex_unlock(&m);
11  }
```

```
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;
}
```

If parent comes after child, parent sleeps forever

Broken Implementation 2

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

```
1   void thr_exit() {
2       done = 1;
3       Pthread_cond_signal(&c);
4   }
5
6   void thr_join() {
7       if (done == 0)
8           Pthread_cond_wait(&c);
9   }
```

```
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;
}
```

Broken Implementation 2

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

```
1   void thr_exit() {
2       done = 1;
3       Pthread_cond_signal(&c);
4   }
5
6   void thr_join() {
7       if (done == 0)
8           Pthread_cond_wait(&c);
9   }
```

No mutual exclusion, hence child may signal before parent calls cond_wait(). In this case, parent sleeps forever!

```
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;
}
```

Trap 1 When Using CV

Trap 1 When Using CV

Trap 1 When Using CV

Trap 1 When Using CV

Only one thread gets a signal

Trap 2 When Using CV

Condition Variable

Trap 2 When Using CV

Trap 2 When Using CV

Trap 2 When Using CV

Trap 2 When Using CV

Trap 2 When Using CV

Signal lost if nobody waiting at that time

Guarantee

Upon signal, there has to be **at least one** thread waiting;
If there are threads waiting, **at least one** thread will wake

CV-based Parent-wait-for-child Approach

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

CV-based Parent-wait-for-child Approach

Good Rule of Thumb

Always do **1. wait** and **2. signal** while holding the lock

To prevent lost signal

Worksheet