Operating Systems COMS(3010A) Condition variables and Semaphores

Branden Ingram

branden.ingram@wits.ac.za

Office Number : ???

Recap

- Locks
- Synchronization Challenges
- Test and Set
- LLSC
- Ticket Locks
- Using Queues

Condition variable

• There are many cases where a thread wishes to check whether a condition is true before continuing its execution.

Condition variable

• **There are many cases where a thread wishes to check whether a condition is true before continuing its execution.**

Example:

A parent thread might wish to check whether a child thread has *completed*.
This is often called a `join()`.

Condition variable

• **There are many cases where a thread wishes to check whether a condition is true before continuing its execution.**

Example:

A parent thread might wish to check whether a child thread has *completed*. This is often called a `join()`.

• **A Condition variable is a synchronization object that lets a thread efficiently wait for a change to shared state that is protected by a lock**

Condition variable

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      }
```

What we would like to see here is:

```
parent: begin
child
parent: end
```

Spin-based Approach

```
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      }
```

^ volatile indicates to compiler its going to be used in multi-threaded systems

- This is hugely inefficient as the parent spins and wastes CPU time

How do condition variables work

- **A condition variable has two states**

  Waiting on the condition

  An explicit queue that threads can put themselves on when some state of execution is not as desired.

  Signaling on the condition

  Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue.

How do condition variables work

- **A condition variable has three methods**

  Function which puts itself to sleep

  - **CV::wait(Lock *lock) :** This automatically releases the lock and suspends execution of the calling thread, placing it on the condition variables waiting list until the thread is re-enabled

  - **CV::signal() :** This takes one thread off the condition variables waiting list and marks it eligible to run (puts it on the schedulers ready list)

  - **CV::broadcast() :** Takes all threads off the condition variables waiting list and marks them as eligible to run

How do condition variables work

- **Declare condition variable**

```
Pthread_cond_t c;
```

  - Proper initialization is required.

- **Operation (the POSIX calls)**

Pass lock into wait so we release lock via wait

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

- **The wait() call takes a mutex as a parameter.**
  - The wait() call release the lock and put the calling thread to sleep.
  - When the thread wakes up, it must re-acquire the lock.

Parent waiting for Child: Use a condition variable

```
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
```

we give up the lock here

Parent waiting for Child: Use a condition variable

```
(cont.)
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      }
```

Parent waiting for Child: Use a condition variable

- **Parent:**
  - **Create the child thread and continues running itself.**
  - **Call into thr_join() to wait for the child thread to complete.**
    - Acquire the lock
    - Check if the child is done
    - Put itself to sleep by calling wait()
    - Release the lock
- **Child:**
  - **Print the message "child"**
  - **Call thr_exit() to wake the parent thread**
    - Grab the lock
    - Set the state variable done
    - Signal the parent thus waking it.

The importance of the state variable “done”

```
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      }
```

`thr_exit()` and `thr_join()` without variable `done`

- **Imagine the case where the child runs immediately.**
  - The child will signal, but there is no thread asleep on the condition.
  - When the parent runs, it will call wait and be stuck.
  - No thread will ever wake it.

Another poor implementation

```
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       }
```

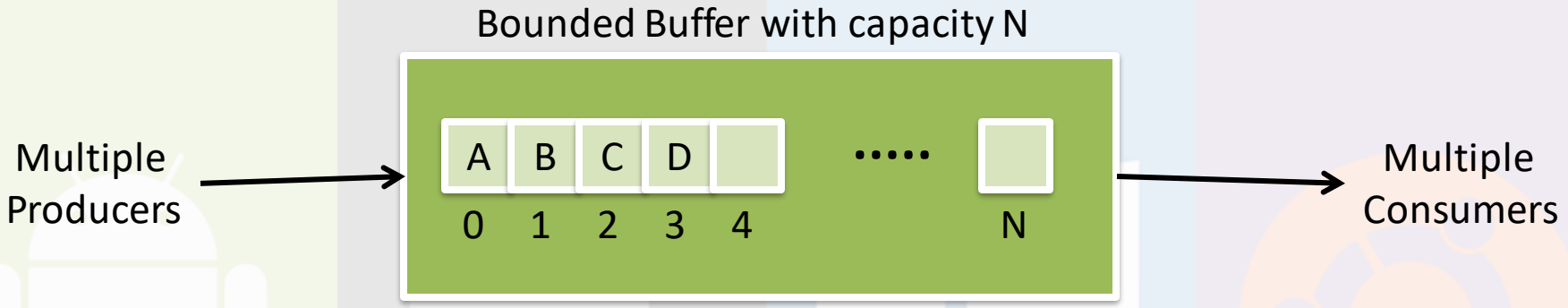
- **The issue here is a subtle race condition.**
  - The parent calls thr_join().
    - The parent checks the value of done.
    - It will see that it is 0 and try to go to sleep.
    - Just before it calls wait to go to sleep, the parent is interrupted and the child runs.
  - The child changes the state variable done to 1 and signals.
    - But no thread is waiting and thus no thread is woken.
    - When the parent runs again, it sleeps forever.

Another poor implementation

```
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       }
```

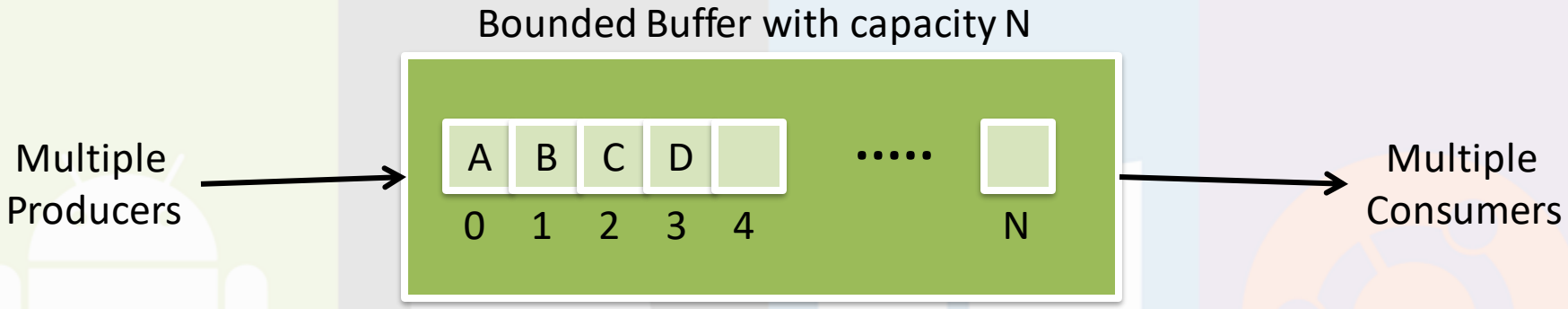
- **The issue here is a subtle race condition.**
  - The parent calls thr_join().
    - The parent checks the value of done.
    - It will see that it is 0 and try to go to sleep.
    - Just before it calls wait to go to sleep, the parent is interrupted and the child runs.
  - The child changes the state variable done to 1 and signals.
    - But no thread is waiting and thus no thread is woken.
    - When the parent runs again, it sleeps forever.

The Producer / Consumer (Bound Buffer) Problem



- **Producer**
  - Produce data items
  - Wish to place data items in a buffer
- **Consumer**
  - Grab data items out of the buffer consume them in some way
- **Example: Multi-threaded web server**
  - A producer puts HTTP requests in to a work queue
  - Consumer threads take requests out of this queue and process them

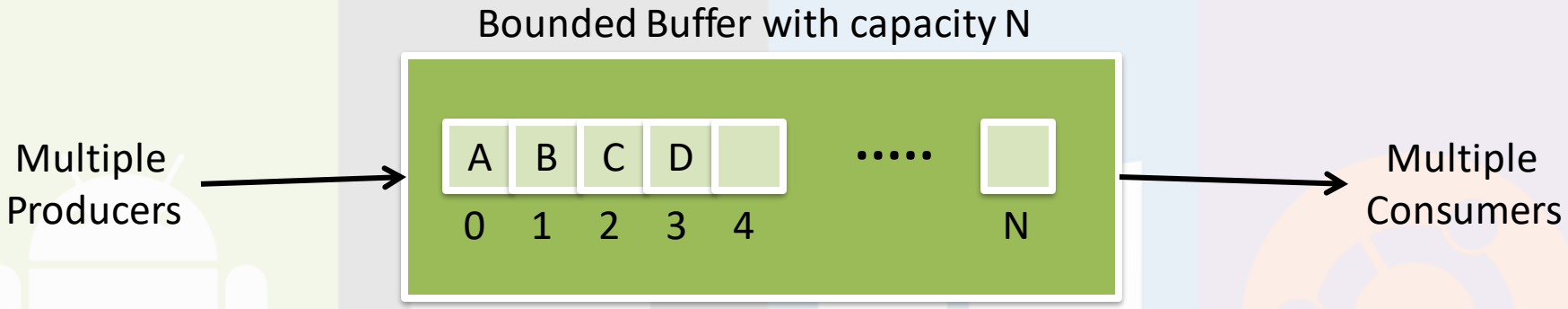
The Producer / Consumer (Bound Buffer) Problem



- **A bounded buffer is used when you pipe the output of one program into another.**
  - **Example: 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.

  - **Bounded buffer is Shared resource -> Synchronized access is required.**

The Producer / Consumer (Bound Buffer) Problem



- **Without proper synchronization the following errors may occur**
  - The producers doesn't block when the buffer is full.
  - A Consumer consumes an empty slot in the buffer.
  - A consumer attempts to consume a slot that is only half-filled by a producer.
  - Two producers writes into the same slot.
  - Two consumers reads the same slot.

Simple Example

```
1       int buffer;
2       int count = 0;    // initially, empty
3
4       void put(int value) {
5               assert(count == 0);
6               count = 1;
7               buffer = value;
8       }
9
10      int get() {
11              assert(count == 1);
12              count = 0;
13              return buffer;
14      }
```

- **Only put data into the buffer when count is zero.**
  - i.e., when the buffer is empty.
- **Only get data from the buffer when count is one.**
  - i.e., when the buffer is full.

Simple Example

```
1       void *producer(void *arg) {
2               int i;
3               int loops = (int) arg;
4               for (i = 0; i < loops; i++) {
5                       put(i);
6               }
7       }
8
9       void *consumer(void *arg) {
10              int i;
11              while (1) {
12                      int tmp = get();
13                      printf("%d\n", tmp);
14              }
15      }
```

Producer puts an integer into the shared buffer loops number of times.
Consumer gets the data out of that shared buffer.

Simple Example (broken solution)

A single condition variable cond and associated lock mutex **with IF**

```
1       cond_t cond;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);            // p1
8               if (count == 1)  if buffer is full, wait   // p2
9                   Pthread_cond_wait(&cond, &mutex);  // p3
10              put(i);                                // p4
11              Pthread_cond_signal(&cond);            // p5
12              Pthread_mutex_unlock(&mutex);          // p6
13          }
14      }
15
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);            // c1
```

Simple Example (broken solution)

```
20              if (count == 0)        if buffer empty wait till its full   // c2
21                  Pthread_cond_wait(&cond, &mutex);                       // c3
22              int tmp = get();                                            // c4
23              Pthread_cond_signal(&cond);                                 // c5
24              Pthread_mutex_unlock(&mutex);                               // c6
25              printf("%d\n", tmp);
26          }
27      }
```

With just a single producer and a single consumer, the code works.

If we have more than one of producer and consumer?

Simple Example (broken solution-Trace)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	Tc1 awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	Tc2 sneaks in
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	Tp awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Simple Example (broken solution 2)

A single condition variable cond and associated lock mutex with While

```
1         cond_t cond;
2         mutex_t mutex;
3
4         void *producer(void *arg) {
5             int i;
6             for (i = 0; i < loops; i++) {
7                 Pthread_mutex_lock(&mutex);            // p1
8                 while (count == 1)                     // p2
9                     Pthread_cond_wait(&cond, &mutex);  // p3
10                put(i);                                // p4
11                Pthread_cond_signal(&cond);            // p5
12                Pthread_mutex_unlock(&mutex);          // p6
13            }
14        }
15
```

Simple Example (broken solution 2)

A single condition variable cond and associated lock mutex with While

```
(Cont.)
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);             // c1
20              while (count == 0)                      // c2
21                  Pthread_cond_wait(&cond, &mutex);   // c3
22              int tmp = get();                        // c4
23              Pthread_cond_signal(&cond);             // c5
24              Pthread_mutex_unlock(&mutex);           // c6
25              printf("%d\n", tmp);
26          }
27      }
```

Simple Example (broken solution 2-Trace)

Consumer 1 Consumer 2 Producer

c1 relates to lines in code above

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	Tc1 awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	Tc1 grabs data
c5	Running		Ready		Sleep	0	Oops woke Tc2

Simple Example (broken solution 2-Trace)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	(cont.)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep …

A consumer should not wake other consumers, only producers, and vice-versa

Simple Example (Solution 3)

- **Use two condition variables and while**
  - Producer threads wait on the condition empty, and signals fill.
  - Consumer threads wait on fill and signal empty.

```
1       cond_t empty, fill;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);
8               while (count == 1)
9                   Pthread_cond_wait(&empty, &mutex);
10              put(i);
11              Pthread_cond_signal(&fill);
12              Pthread_mutex_unlock(&mutex);
13          }
14      }
15
```

Simple Example (Solution 3)

- **Use two condition variables and while**
  - Producer threads wait on the condition empty, and signals fill.
  - Consumer threads wait on fill and signal empty.

```
(Cont.)
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);
20              while (count == 0)
21                  Pthread_cond_wait(&fill, &mutex);
22              int tmp = get();
23              Pthread_cond_signal(&empty);
24              Pthread_mutex_unlock(&mutex);
25              printf("%d\n", tmp);
26          }
27      }
```

Simple Example (Final Solution 4)

- **More concurrency and efficiency -> Add more buffer slots.**
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

```
1       int buffer[MAX];
2       int fill = 0;
3       int use = 0;
4       int count = 0;
5
6       void put(int value) {
7           buffer[fill] = value;
8           fill = (fill + 1) % MAX;
9           count++;
10      }
11
12      int get() {
13          int tmp = buffer[use];
14          use = (use + 1) % MAX;
15          count--;
16          return tmp;
17      }
```

```
1       cond_t empty, fill;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);             // p1
8               while (count == MAX)                    // p2
9                   Pthread_cond_wait(&empty, &mutex);  // p3
10              put(i);                                 // p4
11              Pthread_cond_signal(&fill);             // p5
12              Pthread_mutex_unlock(&mutex);           // p6
13          }
14      }
15
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);             // c1
20              while (count == 0)                      // c2
21                  Pthread_cond_wait(&fill, &mutex);   // c3
22              int tmp = get();                        // c4
23              Pthread_cond_signal(&empty);            // c5
24              Pthread_mutex_unlock(&mutex);           // c6
25              printf("%d\n", tmp);
26          }
27      }
```

Semaphore

- An object with an integer value

Semaphore

- **An object with an integer value**
  - We can manipulate with two routines; `sem_wait()` and `sem_post()`.
  - Initialization

```
1   #include <semaphore.h>
2   sem_t s;
3   sem_init(&s, 0, 1); // initialize s to the value 1
```

  - Declare a semaphore `s` and initialize it to the value 1
  - The second argument, 0, indicates that the semaphore is shared between *threads in the same process*.

Semaphore - Interacting

- **sem_wait()**

```
1  int sem_wait(sem_t *s) {
2        decrement the value of semaphore s by one
3        wait if value of semaphore s is negative
4  }
```

  - If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away**.
  - It will cause the caller to suspend execution waiting for a subsequent post.
  - When negative, the value of the semaphore is equal to the number of waiting threads.

Semaphore - Interacting

- sem_post()

```
1  int sem_post(sem_t *s) {
2        increment the value of semaphore s by one
3        if there are one or more threads waiting, wake one
4  }
```

  - Simply increments the value of the semaphore.
  - If there is a thread waiting to be woken, wakes one of them up

Binary Semaphores (Locks)

- **What should X be?**
  - The initial value should be 1.

```
1    sem_t m;
2    sem_init(&m, 0, X); // initialize semaphore to X; what should X be?
3
4    sem_wait(&m);
5    //critical section here
6    sem_post(&m);
```

Thread Trace: Single Thread Using A Semaphore

Value of Semaphore	Thread 0
1	
1	call sema_wait()
0	sem_wait() returns
0	(crit sect)
0	call sem_post()
1	sem_post() returns

Thread Trace: Two Threads Using A Semaphore

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call sem_wait()	Running		Ready
0	sem_wait() retruns	Running		Ready
0	(crit set: begin)	Running		Ready
0	Interrupt; Switch → T1	Ready		Running
0		Ready	call sem_wait()	Running
-1		Ready	decrement sem	Running
-1		Ready	(sem < 0)→sleep	sleeping
-1		Running	Switch → T0	sleeping
-1	(crit sect: end)	Running		sleeping
-1	call sem_post()	Running		sleeping
0	increment sem	Running		sleeping
0	wake(T1)	Running		Ready
0	sem_post() returns	Running		Ready
0	Interrupt; Switch → T1	Ready		Running
0		Ready	sem_wait() retruns	Running
0		Ready	(crit sect)	Running
0		Ready	call sem_post()	Running
1		Ready	sem_post() returns	Running

Semaphore as a Conditional Variable

can use sempahore to simulate conditional variables

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10   int
11   main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      pthread_create(c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19   }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- What should X be?

Semaphore as a Conditional Variable

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10   int
11   main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      pthread_create(c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19   }
```

A Parent Waiting For Its Child

```
parent: begin
child
parent: end
```

The execution result

- **What should X be?**
  - The value of semaphore should be set to is 0.

Thread Trace: Parent Waiting For Child (Case 1)

The parent call `sem_wait()` before the child has called `sem_post()`.

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	call sem_wait()	Running		Ready
-1	decrement sem	Running		Ready
-1	(sem < 0)→sleep	sleeping		Ready
-1	Switch→Child	sleeping	child runs	Running
-1		sleeping	call sem_post()	Running
0		sleeping	increment sem	Running
0		Ready	wake(Parent)	Running
0		Ready	sem_post() returns	Running
0		Ready	Interrupt; Switch→Parent	Ready
0	sem_wait() retruns	Running		Ready

Thread Trace: Parent Waiting For Child (Case 2)

The child runs to completion before the parent call sem_wait().

Value	Parent	State	Child	State
0	Create(Child)	Running	(Child exists; is runnable)	Ready
0	Interrupt; switch→Child	Ready	child runs	Running
0		Ready	call sem_post()	Running
1		Ready	increment sem	Running
1		Ready	wake(nobody)	Running
1		Ready	sem_post() returns	Running
1	parent runs	Running	Interrupt; Switch→Parent	Ready
1	call sem_wait()	Running		Ready
0	decrement sem	Running		Ready
0	(sem<0)→awake	Running		Ready
0	sem_wait() retruns	Running		Ready

The Producer/Consumer (Bounded-Buffer) Problem

- **Producer: put() interface**
  - Wait for a buffer to become empty in order to put data into it.
- **Consumer: get() interface**
  - Wait for a buffer to become filled before using it.

```
1   int buffer[MAX];
2   int fill = 0;
3   int use = 0;
4
5   void put(int value) {
6       buffer[fill] = value;      // line f1
7       fill = (fill + 1) % MAX;   // line f2
8   }
9
10  int get() {
11      int tmp = buffer[use];     // line g1
12      use = (use + 1) % MAX;     // line g2
13      return tmp;
14  }
```

The Producer/Consumer (Bounded-Buffer) Problem

```
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    …
```

First Attempt: Adding the Full and Empty Conditions

The Producer/Consumer (Bounded-Buffer) Problem

```
21  int main(int argc, char *argv[]) {
22      // …
23      sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with…
24      sem_init(&full, 0, 0);    // … and 0 are full
25      // …
26  }
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

- **Imagine that `MAX` is greater than 1 .**
  - If there are multiple producers, race condition can happen at line *f1*.
  - It means that the old data there is overwritten.

- **We've forgotten here is mutual exclusion.**
  - The filling of a buffer and incrementing of the index into the buffer is a critical section.

Adding 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
(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

Adding Mutual Exclusion

```
(Cont.)
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  }
```

Adding Mutual Exclusion (Incorrectly)

Adding 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
(Cont.)
```

Adding Mutual Exclusion (Correctly)

Adding Mutual Exclusion

```
(Cont.)
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   }
```

Adding Mutual Exclusion (Correctly)

How To Implement Semaphores

```
1    typedef struct __Zem_t {
2        int value;
3        pthread_cond_t cond;
4        pthread_mutex_t lock;
5    } Zem_t;
6
7    // only one thread can call this
8    void Zem_init(Zem_t *s, int value) {
9        s->value = value;
10       Cond_init(&s->cond);
11       Mutex_init(&s->lock);
12   }
13
14   void Zem_wait(Zem_t *s) {
15       Mutex_lock(&s->lock);
16       while (s->value <= 0)
17       Cond_wait(&s->cond, &s->lock);
18       s->value--;
19       Mutex_unlock(&s->lock);
20   }
21   …
```

Build our own version of semaphores called Zemaphores

How To Implement Semaphores

```
22  void Zem_post(Zem_t *s) {
23      Mutex_lock(&s->lock);
24      s->value++;
25      Cond_signal(&s->cond);
26      Mutex_unlock(&s->lock);
27  }
```

Build our own version of semaphores called Zemaphores

- **Zemaphore don't maintain the invariant that the value of the semaphore.**
  - The value never be lower than zero.
  - This behavior is easier to implement and matches the current Linux implementation.