

# Lecture 15: Condition Variables and Semaphores

Operating Systems

**Content taken from:** <https://pages.cs.wisc.edu/~remzi/OSTEP/>

<https://www.cse.iitb.ac.in/~mythili/os/>

# Last Class

- How should multiple threads concurrently access shared data?
- Avoid Race Conditions in Critical Section
- Mutual Exclusion using Locks
- Waiting and Signaling using Condition Variables

# Condition Variables

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition
- Another thread that makes the condition true can signal the CV to wake up a waiting thread

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

# Example

```
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 }
```

Figure 30.3: Parent Waiting For Child: Use A Condition Variable

# Producer/Consumer or Bounded Buffer Problem

- A common pattern in multi-threaded programs
- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a queue. Worker threads get requests from this queue and process them.
- Setup: one or more producer threads, one or more consumer threads, a shared buffer of bounded size

# Shared Buffer

```
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 }
```

Figure 30.6: The Put And Get Routines (v1)

# Producer and Consumer Threads

```
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    while (1) {
11        int tmp = get();
12        printf("%d\n", tmp);
13    }
14 }
```

Figure 30.7: Producer/Consumer Threads (v1)

# Solution-1

```
1 int loops; // must initialize somewhere...
2 cond_t cond;
3 mutex_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         Pthread_mutex_lock(&mutex); // p1
9         if (count == 1) // p2
10            Pthread_cond_wait(&cond, &mutex); // p3
11         put(i); // p4
12         Pthread_cond_signal(&cond); // p5
13         Pthread_mutex_unlock(&mutex); // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex); // c1
21         if (count == 0) // c2
22            Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get(); // c4
24         Pthread_cond_signal(&cond); // c5
25         Pthread_mutex_unlock(&mutex); // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Figure 30.8: Producer/Consumer: Single CV And If Statement

# Solution-1: Issues

```

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        if (count == 1)                      // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                            // p4
        Pthread_cond_signal(&cond);          // p5
        Pthread_mutex_unlock(&mutex);        // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        if (count == 0)                      // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                   // c4
        Pthread_cond_signal(&cond);          // c5
        Pthread_mutex_unlock(&mutex);        // c6
        printf("%d\n", tmp);
    }
}

```

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

Figure 30.9: Thread Trace: Broken Solution (v1)

# Solution-2

- Always use **while** loops with condition variables

```
1 int loops;
2 cond_t cond;
3 mutex_t mutex;
4
5 void *producer(void *arg) {
6     int i;
7     for (i = 0; i < loops; i++) {
8         Pthread_mutex_lock(&mutex); // p1
9         while (count == 1) // p2
10             Pthread_cond_wait(&cond, &mutex); // p3
11         put(i); // p4
12         Pthread_cond_signal(&cond); // p5
13         Pthread_mutex_unlock(&mutex); // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex); // c1
21         while (count == 0) // c2
22             Pthread_cond_wait(&cond, &mutex); // c3
23         int tmp = get(); // c4
24         Pthread_cond_signal(&cond); // c5
25         Pthread_mutex_unlock(&mutex); // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Figure 30.10: Producer/Consumer: Single CV And While

# Solution-2: Issues

```

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        while (count == 1)                   // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                            // p4
        Pthread_cond_signal(&cond);         // p5
        Pthread_mutex_unlock(&mutex);       // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        while (count == 0)                   // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                  // c4
        Pthread_cond_signal(&cond);          // c5
        Pthread_mutex_unlock(&mutex);        // c6
        printf("%d\n", tmp);
    }
}

```

$T_{c1}$	State	$T_{c2}$	State	$T_p$	State	Count	Comment
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
		Sleep	c1	Run	Ready	0	
		Sleep	c2	Run	Ready	0	
		Sleep	c3	Sleep	Ready	0	Nothing to get
		Sleep		Sleep	p1 Run	0	
		Sleep		Sleep	p2 Run	0	
		Sleep		Sleep	p4 Run	1	Buffer now full
		Ready		Sleep	p5 Run	1	$T_{c1}$ awoken
		Ready		Sleep	p6 Run	1	
		Ready		Sleep	p1 Run	1	
		Ready		Sleep	p2 Run	1	
		Ready		Sleep	p3 Sleep	1	Must sleep (full)
		c2	Run	Sleep	Sleep	1	Recheck condition
		c4	Run	Sleep	Sleep	0	$T_{c1}$ grabs data
		c5	Run	Ready	Sleep	0	Oops! Woke $T_{c2}$
		c6	Run	Ready	Sleep	0	
		c1	Run	Ready	Sleep	0	
		c2	Run	Ready	Sleep	0	
		c3	Sleep	Ready	Sleep	0	Nothing to get
		Sleep	c2	Run	Sleep	0	
		Sleep	c3	Sleep	Sleep	0	Everyone asleep...

Figure 30.11: Thread Trace: Broken Solution (v2)

# Solution-3

- A consumer should wake up only the producers.
- A producer should wake up only the consumers.

```
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
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 }
```

Figure 30.12: Producer/Consumer: Two CVs And While

# A more general Producer/Consumer Solution

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

Figure 30.13: The Correct Put And Get Routines

```
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 }
```

Figure 30.14: The Correct Producer/Consumer Synchronization

# Semaphores

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
  - Up/post increments the counter
  - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative

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

```
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 }
5
6 int sem_post(sem_t *s) {
7     increment the value of semaphore s by one
8     if there are one or more threads waiting, wake one
9 }
```

Figure 31.2: Semaphore: Definitions Of Wait And Post

# Binary Semaphores (Locks)

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

Figure 31.3: A Binary Semaphore (That Is, A Lock)

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call sem_wait ()	Run		Ready
0	sem_wait () returns	Run		Ready
0	(crit sect begin)	Run		Ready
0	<i>Interrupt; Switch→T1</i>	Ready		Run
0		Ready	call sem_wait ()	Run
-1		Ready	decr sem	Run
-1		Ready	(sem<0)→sleep	Sleep
-1		Run	<i>Switch→T0</i>	Sleep
-1	(crit sect end)	Run		Sleep
-1	call sem_post ()	Run		Sleep
0	incr sem	Run		Sleep
0	wake(T1)	Run		Ready
0	sem_post () returns	Run		Ready
0	<i>Interrupt; Switch→T1</i>	Ready		Run
0		Ready	sem_wait () returns	Run
0		Ready	(crit sect)	Run
0		Ready	call sem_post ()	Run
1		Ready	sem_post () returns	Run

Figure 31.5: Thread Trace: Two Threads Using A Semaphore