Condition Variables

Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Condition Variables

- There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution
- 즉, Multi-thread 프로그램에서, 각 thread가 어떤 조건이 true일 때까지 기다려야하는 상황 발생
- Example:
  - A parent thread might wish to check whether a child thread has *completed*.
  - This is often called a `join()` (`join()`함수를 어떻게 설계할 수 있을지도 고민해보자!)

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      }
```

What we would like to see here is:

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

Parent waiting fore child: 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      }
```

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

How to wait for a condition

- **Condition variable**
  - Queue of threads (대기 중인 thread들의 큐)
  - **Waiting** on the condition
    - An explicit queue that threads can put themselves on when some state of execution is not as desired. (원하는 상태가 아닐 때, thread가 스스로를 explicit queue에 넣고 잠드는 동작)
  - **Signaling** on the condition
    - Some other thread, *when it changes it state*, can wake one of those waiting threads and allow them to continue. (다른 thread가 상태를 바꿨을 때, 대기중인 thread 하나를 깨워 실행을 재개)
- **Three in a package**
  - condition variable c;
  - state variable done;
  - lock L;

각각이 왜 필요할까?

Definition and Routines

- Declare condition variable

```
pthread_cond_t c;
```

  - Proper initialization is required.
  - Contains information about threads waiting for a signal

- Operation (the POSIX calls)

Wait → Sleep + Unlock
Signal → Wakeup + lock

```
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.
    - Wait() has to be called while holding a lock → The wait() call release the lock
    - When the thread wakes up, it must re-acquire the lock

Lock이 없으면 무슨 문제가 발생?

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
```

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:**
  - Creates the child thread and continues running itself.
  - Calls into `thr_join()` to wait for the child thread to complete.
    - Acquires the lock.
    - Checks if the child is done.
    - Puts itself to sleep by calling `wait()`.
    - Releases the lock.
- **Child:**
  - Prints the message "child".
  - Calls `thr_exit()` to wake up the parent thread.
    - Grabs the lock.
    - Sets the state variable `done`.
    - Signals 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 up

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       }
```

Without "lock"

	Parent thread	Child thread
1	Check done == 0	
2		done == 1
3		Signal(c)
4	Wait(c, m)	
…	**Never wakes up**	
	…	

- 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 (하나 이상의 생산자 thread)**
  - Produce data items / Wish to place data items in a buffer (데이터를 만들어 buffer에 넣는 thread)
- **Consumer (하나 이상의 소비자 thread)**
  - Grab data items out of the buffer consume them in some way (buffer 에서 데이터를 꺼내 처리하는 thread)
- 즉, Producer/Consumer thread들이 buffer를 동시에 접근 → Race condition
- Bounded Buffer를 통해 해결

Bounded buffer

- 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* (grep은 매칭되는 라인을 출력)
    - The wc process is the *consumer* (wc는 word 수를 셈)
    - Between them is an in-kernel bounded buffer.
  - Bounded buffer is Shared resource → **Synchronized access** is required.
    - The shared Buffer is full → Threads should be blocked
    - The shared Buffer is empty → Threads should blocked
    - Threads access the shared buffer at the same time → Lock + wait queue (sleep)

The Put and Get Routines (Version 1)

```
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*.

Producer/Consumer Threads (Version 1)

```
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.

A single condition variable `cond` and associated lock `mutex`

d If Statement

```
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)                         // 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
20              if (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      }
```

- **p1-p3**: A producer waits for the buffer to be empty.
- **c1-c3**: A consumer waits for the buffer to be full.
- With just *a single producer* and *a single consumer*, the code works.

Thread Trace: Broken Solution (Version 1)

```
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)                        // 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
20                if (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        }
```

T_{c1} state	T_{p1} state	Cnt	Comment
C1 run		0	
C2 run		0	
C3 sleep		0	Buffer is empty
	P1 run	0	
	P2 run	0	
	P4 run	1	Buff is now full
	P5 run	1	T_{c1} awoken
	P6 run	1	
	P1 run	1	
	P2 run	1	
	P3 sleep	1	
C4 run		0	Buffer is now empty

Thread Trace: Broken Solution (Version 1)

```
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)                        // 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
20                if (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        }
```

T_{c1} state	T_{p1} state	Cnt	Comment
C1 run		0	
C2 run		0	
C3 sleep		0	Buffer is empty
	P1 run	0	
	P2 run	0	
	P4 run	1	Buff is now full
	P5 run	1	T_{c1} awoken
	P6 run	1	
	P1 run	1	
	P2 run	1	
	P3 sleep	1	
C4 run		0	Buffer is now empty

If we have more than one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

```
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)                        // 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
20                if (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        }
```

T_{c1}	State	T_{c2}	State	T_p	State	Cour	*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

Thread Trace: Broken Solution (Version 1)

```
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)                        // 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
```

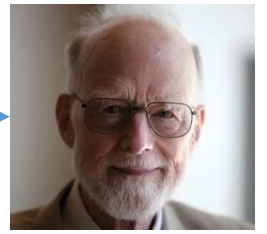
```
16        void *consumer(void *arg) {
17            int i;
18            for (i = 0; i < loops; i++) {
19                Pthread_mutex_lock(&mutex);            // c1
20                if (count == 0)                        // c2
21                    Pthread_cond_wait(&cond, &mutex);// c3
22    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

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

Problem! (Race condition
→ 결론적으로는 Deadlock)

Thread Trace: Broken Solution (Version 1)

- The problem arises for a simple reason:
  - After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer *changed by* T_{c2}.
  - **Mesa semantics vs. Hoare semantics**: OS의 철학 차이



Tony Hoare
(1934~26.3.5)

  - There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics. (signal은 즉시 실행이 보장 X)
    - Virtually every system ever built employs *Mesa semantics* (Windows, Linux, …)
    - 즉, signal을 통해 어떤 thread를 깨울지 알 수 없음 (즉, 이러한 상황을 고려해서 구현해야 한다)
  - Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken (대부분 사용 X, 구현이 너무 복잡함)

Producer/Consumer: Single CV and While

- Consumer T_{c1} wakes up and re-checks the state of the shared variable.
  - If the buffer is empty, the consumer simply goes back to sleep.

CV and 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
```

(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      }
```

- A simple rule to remember with condition variables is to **always use while loops**
- **However**, this code still has a bug (*next page*)

Producer/Consumer: S... CV and 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
```

```
(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        }
```

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...

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

Producer/Consumer: S... CV and 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
```

```
(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        }
```

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...

Race condition

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

The single Buffer Producer/Consumer Solution

- 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
```

(*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      }
```

```
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           (P1) → Pthread_mutex_lock(&mutex);
8           (P2) → while (count == 1)
9           (P3) →     Pthread_cond_wait(&empty, &mutex);
10          (P4) → put(i);
11          (P5) → Pthread_cond_signal(&fill);
12          (P6) → Pthread_mutex_unlock(&mutex);
13            }
14        }
15
```

```
(Cont.)
16        void *consumer(void *arg) {
17            int i;
18            for (i = 0; i < loops; i++) {
19          (C1) → Pthread_mutex_lock(&mutex);
20          (C2) → while (count == 0)
21          (C3) →     Pthread_cond_wait(&fill, &mutex);
22          (C4) → int tmp = get();
23          (C5) → Pthread_cond_signal(&empty);
24          (C6) → Pthread_mutex_unlock(&mutex);
25          (C7) → printf("%d\n", tmp);
26            }
27        }
```

T_{c1} state	T_{c2} state	T_p state	Cnt	Comment
C1 run			0	
C2 run			0	
C3 sleep			0	Buffer is empty
	C1 run		0	
	C2 run		0	
	C3 sleep		0	Buffer is empty
		P1 run	0	
		P2 run	0	
		P4 run	1	Buffer is full
		P5 run	1	T_{c1} is woken up
		P6 run	1	
		P1 run	1	
		P2 run	1	
		P3 sleep	1	
C2 run			1	
C4 run			0	Buffer is now empty
C5 run			0	T_{c2} is not woken up
C6 run			0	
C7 run			0	
	...			

The Final Producer/Consumer Solution

- 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      }
```

Solution (Cont.)

```
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      }
```

- p2: **A producer** only sleeps if all buffers are currently filled.
- c2: **A consumer** only sleeps if all buffers are currently empty.
- Bottle lock?
  - Concurrent Queue가 유리 (Multi-CPUs, Thread ↑)
  - → **Parallel Computing**

Covering Conditions

- Signal이 어떠한 thread를 깨워야 하는지 알 수 없는 경우

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

```
1       // how many bytes of the heap are free?
2       int bytesLeft = MAX_HEAP_SIZE;
3
4       // need lock and condition too
5       cond_t c;
6       mutex_t m;
7
8       void *
9       allocate(int size) {
10          Pthread_mutex_lock(&m);
11          while (bytesLeft < size)
12              Pthread_cond_wait(&c, &m);
13          void *ptr = ...;                // get mem from heap
14          bytesLeft -= size;
15          Pthread_mutex_unlock(&m);
16          return ptr;
17      }
18
19      void free(void *ptr, int size) {
20          Pthread_mutex_lock(&m);
21          bytesLeft += size;
22          Pthread_cond_signal(&c);        // whom to signal??
23          Pthread_mutex_unlock(&m);
24      }
```

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads.**
    - Cost: too many threads might be woken.
    - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep.