Condition Variables

순서를 지정할 필요가 생기게 됨.

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Condition Variable: Concept

동기화 (synchronization)을 lock 말고 구현하기는 어려움

- **There are many cases where a thread wishes to check whether a condition is true before continuing its execution**
  - e.g.) a parent thread might wish to check whether a child thread has completed before continuing

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

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

main thread

What we want to see:

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

→ 최적화하지 말라는 뜻 (컴파일러 보고)

```
volatile int done = 0;

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

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

전역변수

의미가 없네! 없어.

done = 1
~~done = 0~~
done = 1
done = 1

HW 순으로

spin

- **We could try using a shared variable**

done.

  - This solution will generally work, but it is hugely inefficient as a parent spins, wasting CPU time

완전 상태

→ 그럼 스레드를 sleep으로 재워.

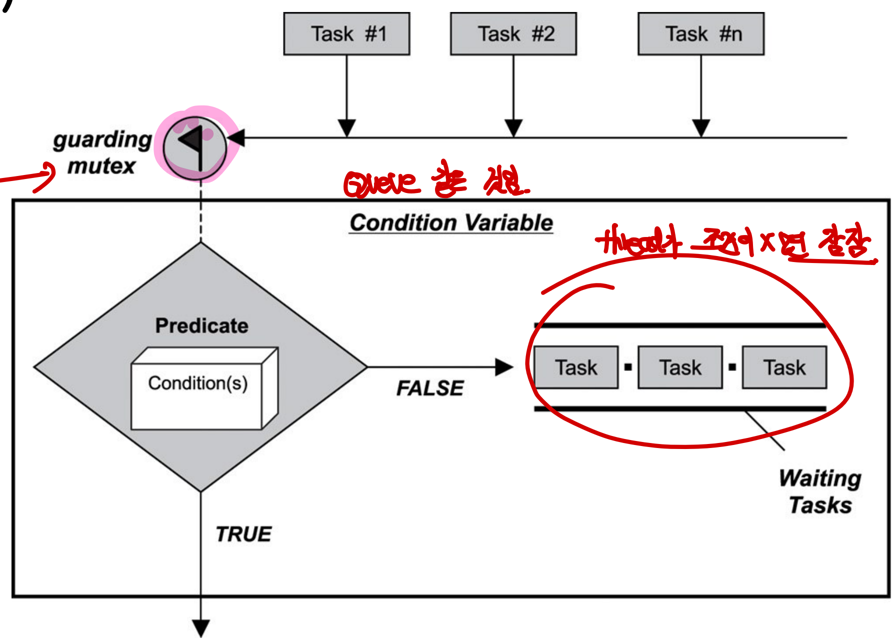
Definition and Routines

쓰레드가 특정 조건이 만족되지 않았을 때 큐 안에서 대기하게 만드는 장치

→ 다른 스레드에서 큐 안에서 sleep 상태로 대기 중인 스레드를 깨움. 다시 Ready 상태가 됨

- **To wait until condition is met, thread can use condition variable**
  - It is an explicit queue that threads can put themselves on when some state of execution (i.e. some condition) is not as desired by waiting on the condition
  - Some other thread, when it changes the state, can then wake one (or more) of the waiting threads and thus allow them to continue by signaling on the condition

- **To use condition variable, you simply declare and initialize first**
  - Two operations: `wait()` and `signal()`
  - `wait()` to put itself to sleep when a condition is not met
  - `signal()` to wait a sleeping thread waiting on this condition
  - "`wait()`" needs a mutex as parameter
    1) When calling, mutex must be locked
    2) Then, it releases the lock and put the thread to sleep atomically
    3) When waking up, it must re-acquire the lock before returning



Parent Waiting for Child: Use a Condition Variable

- Let's look at a simple example: two cases to consider

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
(4)  Pthread_mutex_lock(&m);
     done = 1;
     Pthread_cond_signal(&c);
     Pthread_mutex_unlock(&m);
}

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

→조건 변수

이 전에 잠든 친구를 깨워라.

조건변수를 가진 스레드에게 signal을 보내서 깨운다

```
void thr_join() {
(2)  Pthread_mutex_lock(&m);
     while (done == 0)
         Pthread_cond_wait(&c, &m);
     Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
     printf("parent: begin\n");
     pthread_t p;
     Pthread_create(&p, NULL, child, NULL);
(1)  thr_join();
(5)  printf("parent: end\n");
     return 0;
}
```

기다리는 동안, lock은 해제하고 해당 스레드는 sleep.

이 모든 과정은 원자적으로 진행된다.

parent | child

1) The parent creates the child thread but continues running itself (parent→child):
   (1) The parent immediately calls **`thr_join()`** to wait for the child to complete
   (2) **`thr_join()`** acquires the lock and puts itself to sleep by calling **`wait()`** →child로 context-switch
   (3) The child runs, print "child", and call **`thr_exit()`** to wake the parent
   (4) **`thr_exit()`** acquires the lock, done=1, and signals to parent
   (5) the parent run, print "parent: end" and return
2) The child runs immediately upon creation (child→parent): →자고 있는 친구 X
   (1) The child performs (3) and (4) above but nothing sleeping, and it just returns
   (2) The parent runs, calls **`thr_joins()`**, sees done=1, and doesn't wait and returns

Parent Waiting: No State Variable & No Lock

done이 없으면?

→ lock 없다면..?
done은 있었다!

- **Why do we need the state variable done?**
  - Consider the case where the child run immediately and calls `thr_exit()`
  - The child signals but there is no thread asleep on the condition → When the parent runs, it will just call `wait()` and be stuck; no thread will ever wake it

No State Variable

child parent

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

깨울 쓰레드가 없는 경우

No Lock

child 수행

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

parent에서 재개

race condition 발생.

atomic 하지 않아서 문제가 발생

사이에서 interrupt.

- **What if threads does not hold a lock to signal and wait?**
  - The issue here is a subtle race condition
  - Assume that the parent runs upon the child thread creation:
    (1) It calls `thr_join()`, sees done=0, is interrupted before sleep and the child runs
    (2) The child makes done=1 and signals but no thread is waiting and nothing wakes
    (3) When the parent runs again, it sleeps forever

parent child

interrupt

done = 1

- **From these, we saw basic requirements for condition variable**

The Producer/Consumer Problem

→ bounded problem.

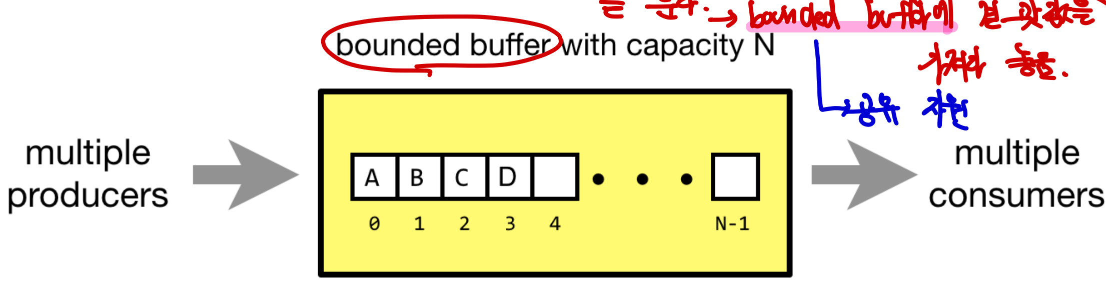
→ buffer 크기의 한정

- **The producer/consumer problem, is sometimes called bounded buffer problem, is a famous synchronization problem**
  - Imagine one or more producer thread and one or more consumer threads
  - Producers generate data items and place them in a buffer
  - Consumers grab said items from the buffer and consume them in some way
  - e.g.) multi-threaded web server (producer: http requests, consumer: processing requests), pipe (`grep foo file.txt | wc -l` → connected through in-kernel bounded buffer)

둘 사이의 sink를 맞춰야 한다.

kernel에 내부적으로 buffer를 둔다. → bounded buffer에 결과값을 가져와 출력.

여러 consumer.



ㄴ 공유 자원 = 여러 스레드에서 접근 가능

- **The bounded buffer is a shared resource** = race-condition이 발생할 수 있음.
  - We must require synchronized access to it, otherwise, a race condition arise

Producer/Consumer Threads: Initial

- **The first thing we need is a shared buffer, into which a producer puts data, and out of which a consumer takes data**
  - Let's use a single integer for simplicity and we will generalize it later

→ int 1개 buffer.

→ 상호 배제가 구현 되어 있지 않음.

buffer

count=0/1

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

→ 0이 아니면 끝남 → 함수 종료

producer가 data를 넣는 것.

마찬가지

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

← 계속을 대비해 get() 호출.

  - **put()** checks if the buffer is empty and puts a value into buffer and count=1
  - **get()** does the opposite, setting the buffer to empty and returning the value
- **We need two types of threads: producer and consumer threads**

2개의 thread.

  - **producer()** puts an integer into the shared buffer loops times
  - **consumer()** gets the data out of the buffer (forever) and each time printing
  - However, this code does not work properly and an "assertion will fire"

→ 0이 되지 않음.

A Broken Solution (동작 x)

$T_p \rightarrow$ buffer $\rightarrow T_{C1}$, T_{C2} → 여러개의 consumer를 대기를 생각해 봐야 함.

- **Consider mutual exclusion (lock) and order (condition variable)**
  - Consider two consumers (T_{c1}, T_{c2}) and a producer (T_p): **1)** T_{c1} sleeps due to no data, **2)** T_p runs, produces a value, and sleeps due to buffer full, **3)** T_{c2} runs and consumes (then no data in buffer), **4)** T_{c1} runs and tries to consume but nothing
  - Mesa semantics: no guarantee that when the woken thread runs, the state will still be as desired (virtually every system ever built employs this semantics)
  - Hoare semantics: strong guarantee that the woken thread will run immediately

정리: Tc1이 실행되기 전에 버퍼의 상태가 변경되었음. → 이것이 if문 사용했을 때의 문제점임

This code with a single producer and a single consumer works fine

```
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);
    }
}
```

while 대신 if문 사용하면 문제 발생

스케줄러는 다시 produce 해야하는지 모름.

while로

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

wait

signal

CS

put() 호출

c4부터 시작하는데 데이터가 없음!

Better, But Still Broken: While, Not If

- **The earlier issue is fixed by changing `if` to `while` for re-check**
  - However, a situation where **1)** T_{c1}, T_{c2} run, sleep (no data), **2)** T_p runs, produces a value, wakes T_{c1}, **3)** T_p sleeps (full), **4)** T_{c1} runs, re-checks, consumes, signals to T_{c2} (not T_p), sleeps (no data), **5)** T_{c2} runs, sleeps ➔ everyone asleep…
  - Signaling is clearly needed, but must be more directed; a consumer should not wake other consumers, only producers, and vice-versa

Thanks to Mesa semantics, a rule to remember for condition variable is to always use while loops

```
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);
    }
}
```

producer를 깨웠으면 했지만 다른. consumer를 깨움

consumer를 깨우면 안됨

get()

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...

wait

CS

wait

CS

signal

CS

signal

wait

CS

wait

put()

다시 cont 체크

둘다 sleep..?

The Single Buffer Producer/Consumer Solution

- **The solution is a small one: use two condition variables, not one**
  - Then, threads can properly signal (which type of thread should be wake up) when the state of the system changes

생산자는 소비자만 깨우고
소비자는 생산자만 깨우게 해야함

```c
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}
```

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

consumer가 consumer를 깨우지 않음

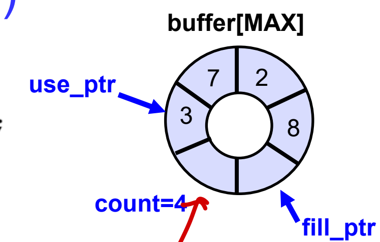
  - The producer threads wait on the condition empty, and signals fill
  - Conversely, the consumer threads wait on fill and signal empty
  - By doing so, a consumer can never accidentally wake a consumer, and a producer can never accidentally wake a producer

- **Now, we know that the use of two condition variables and while is necessary to properly handle the produce/consumer problem**

The Correct Producer/Consumer Solution

- **The last change is to enable more concurrency and efficiency**
  - We add more buffer slots, so that multiple values can be produced before sleeping, and similarly multiple values can be consumed before sleeping
  - The first change for this correct solution is within the buffer structure itself and the corresponding **put()** and **get()** → circular buffer (queue)

"circular buffer"




```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;
```

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

```
int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

  - We also slightly change the conditions that producers check to determine whether to sleep or not

(buffer가 꽉차면 못 넣음)

꽉차 있으면 producer 멈춤

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // p1
        while (count == MAX)                     // p2
            Pthread_cond_wait(&empty, &mutex);   // p3
        put(i);                                  // p4
        Pthread_cond_signal(&fill);              // 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(&fill, &mutex);    // c3
        int tmp = get();                         // c4
        Pthread_cond_signal(&empty);             // c5
        Pthread_mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
}
```

Covering Conditions

memory allocation
(malloc) consume
(free) produce

- Consider a simple multi-threaded memory allocation library
  - When a thread calls into the memory allocation code, it might have to wait for more memory to become free
  - Then, which of waiting threads should be woken up when a thread frees memory?
  - A scenario that when there is zero byte free, 1) T_a calls **allocate(100)**, T_b calls **allocate(10)**, and both wait on the condition and sleep, 2) T_c calls **free(50)** and what if it signals to T_a instead of T_b?
    → T_a sleeps again

    Ta는 signal 가고 다시 wait

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t  c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...; // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c); // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

T_A : 100B → T_B : 10B → T_C : 50B free.

스케줄러가 뭘 깨울지 모름.

요청한 것보다 사용가능한 메모리가 적으면 할당 안되고 대기

어떤 스레드를 깨워야 할까.

모든 thread를 깨움. but 깨지 않아야 하는 친구도 깸.

  - The solution is simple: replace **signal()** to **broadcast()**, waking up all
    → sleep의 상태인 모든 스레드를 깨움
  - This scheme guarantees that any threads that should be woken are
  - However, the threads that should not be awake will also wake up, re-check the condition, and then go immediately back to sleep → performance↓ context-switch. condition check.
  - Covering condition: covers all the cases where a thread needs to wake up