Condition Variables

- 실행 전 조건이 참인지 확인해야하는 스레드가 많음
  - 부모 스레드가 자식 스레드가 끝났는지 확인해야하는 경우
  - join()

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

```
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      }
```

- 비효율적임(계속해서 Spin을 해 CPU 시간을 낭비함)

Using Condition Variable

- Waiting : 실행 대기
- Signaling : 다른 기다리고 있는 스레드를 깨워주는 것
- Condition Variable 선언

```
pthread_cond_t c;
```

- 연산

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

wait() : mutex를 인자로 받음, lock을 release 한 뒤 호출한 스레드를 sleep상태로 만듬

  - ▪ 다시 깨어날려면 lock을 얻어야 함
- 사용예시

```
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      }
```

- Parent
  - Child를 만들고, 계속 실행
  - thr_join()을 호출해서 Child가 끝날때까지 기다림
    - lock을 받아서, child가 끝난지 확인하고, wait()를 호출해서 sleep으로 들어가서 lock을 release
- Child
  - thr_exit()를 호출해서 Parent 스레드를 깨움
  - lock을 받아서, done을 바꾼 뒤, Parent에게 signal

- Done 변수의 중요성

```
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      }
```

- done이 없으면, child가 호출 즉시 작업을 완료해 signal하는 케이스에서 부모가 이를 받지 못한 뒤 나중에 sleep에 들어가면 stuck됨 -> 절대 못깨어남

```
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       }
```

- 미묘한 Race Condition 발생
  - 부모쪽에서 join 호출 중 done의 상태를 체크하고 wait를 호출하기전, 인터럽트 발생으로 자식으로 흐름이 넘어가버리면
  - done을 그때서야 자식이 1로 바꾸고, 시그널보냄
  - 아까랑 똑같이 부모 스레드가 stuck됨

Producer / Consumer (Bound Buffer) Problem

- Producer : 데이터 아이템을 만드는 역할, 버퍼에 데이터 아이템을 넣고싶어 함
- Consumer : 버퍼에서 데이터 아이템을 가져와 자신의 방식대로 소비하길 원함

Bounded Buffer

- 한 프로그램의 아웃풋을 pipeout해서 다른 프로그램으로 보내는 것
- 공유자원이기 때문에, 동기화된 접근이 필요함

```
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      }
```

- Put
  - count값이 0일때만 버퍼에 데이터를 넣음
- Get
- count값이 1일때만 버퍼에서 데이터를 가져옴
- 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      }
```

  - Producer는 루프를 돌면서 정수를 버퍼에 여러번 넣고
  - Consumer는 버퍼에서 데이터를 가져옴
- Single C/V and 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      }
```

  - 하나의 Condition Variable, lock mutex
  - 하나의 Producer / 하나의 Consumer를 대상으로는 코드가 잘 작동함
- Multiple Producer/Consumer

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	T_{c1} 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	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	**Oh oh! No data**

    - T_c1이 데이터를 받아야하는데, 중간에 T_c2가 받아버림
    - Mesa Semantics : 깨워진 스레드가 원하는 값을 받는다는 확신이 없음
    - Hoare Semantics : 깨워진 스레드가 원하는 값을 받을 가능성이 높아짐(깨운 스레드를 바로 실행)
- Single C/V 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
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      }
```

- 조건문에 while 구문을 사용하는 것
- 여전히 버그가 있음

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	T_{c1} 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	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**
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 ...**

    - Consumer가 Consumer를 깨우는 일이 없어야함(Producer만이 깨울 수 있음)
- Single Buffer P/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);
    }
}

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);
    }
}
```

- Producer는 empty 컨디션일때 wait, fill 컨디션일 때 signal
- Consumer는 fill 컨디션일때 wait, empty 컨디션일 때 signal

- Final Producer/Consumer

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

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

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

```
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      }
```

- 버퍼 슬롯을 더해서, Concurrency / Efficiency를 향상
  - 동시에 Producing / Consuming을 가능하게 함
  - Context Switch를 줄임
- Proucer는 모든 버퍼가 fill 일때만 sleep
- Conusmer는 모든 버퍼가 empty 일때만 sleep

Covering Conditions

- 남은공간이 0바이트라고 가정
  - 스레드 T_a가 allocate(100) 호출
  - 스레드 T_b가 allocate(10) 호출
  - 두 스레드 모두 wait, sleep에 들어감
  - 스레드 T_c가 free(50) 호출
    - 어떤 스레드가 깨워져야 할까?

```
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      }
```

- Lampson / Redell이 제안한 해결법
  - pthread_cond_signal을 pthread_cond_broadcast()로 변경
    - wait하고 있는 를 **모든 스레드를** 깨움
    - 비용 : 너무 많은 스레드가 일어날 수 있음
    - 일어나면 안되는 스레드들은 일어나서 조건을 확인한뒤 다시 sleep할 것임

Semaphore

- 정수값을 가지는 객체
- 초기화

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

  - semaphore s를 선언하고 1으로 초기화
  - 두번째 인수 0은 Semaphore가 같은 프로세스 내의 스레드들 간에 공유된다는 것을 뜻함
  - 초기화는 딱 한번만
- 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   }
```

- 호출됐을 때 semaphore의 값이 1보다 크거나 같으면 바로 return
- 호출자의 실행을 멈추고, 다음 post를 기다리게 함
- 음수일때는, semaphore의 값이 wait하고 있는 스레드의 숫자와 동일

- 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  }
```

  - semaphore의 값을 1 증가시킴
  - 깨워야 하는 스레드들이 있으면 해당 스레드 중 하나를 깨움

Binary Semaphores(Locks)

- Mutual Exclusion을 위해 초기값을 1로 설정

```
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);
```

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 Condition Variables

```c
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
    return 0;
}
```

- X의 값은 0이 됨

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

- 자식이 sem_post()를 호출하기 전에 부모가 sem_wait()를 호출하는 경우

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

- 부모가 sem_wait()를 호출하기 전에 자식이 모든 실행을 끝마치는 경우

Producer/Consumer Problem

```
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  }
```

- Producer의 put : 데이터를 넣기 위해 버퍼가 empty될 때까지 기다림
- Consumer의 get : 데이터를 사용하기 위해 버퍼가 fill될 때까지 기다림
- Full과 Empty condition을 사용하는 방법

```
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    }
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    }
```

- MAX값이 1보다 클 경우
  - producer가 다수일 경우, f1 line에서 경쟁조건 발생
  - 데이터가 **덮어쓰기** 됨
- Mutual Exclusion : 버퍼를 채우는 것 / 인덱스의 증가하는 부분이 Critical Section이여야 함

- Adding Mutual Exclusion(Incorrect)

```
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  }
```

    - Consumer가 mutex를 획득
    - Consumer가 sem_wait()호출, Block되어서 CPU 양보(mutex는 계속 가지고 있음)
    - Producer가 mutex를 가지고 오기위해 대기 시작
    - 서로 영원히 대기함 (**Deadlock**)
- Working Solution

```
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  }
```

- sem_wait의 위치를 살짝씩 조정해준 형태

Reader-Writer Locks

- Insert
  - List의 상태를 바꿈
  - 전통적인 Critical Section
- Lookup
  - 자료구조를 읽어옴
  - 진행하고 있는 Insert작업이 없음을 확인해야 함
  - 여러개의 Lookup 작업을 Concurrent하게 실행
- 하나의 Writer만 Lock을 얻을 수 있음
- Reader가 Lock을 얻으면
  - 다른 Reader들도 Lock을 얻을 수 있음
  - Writer는 모든 Reader들이 끝낼때까지 기다려야 함

```
1   typedef struct _rwlock_t {
2       sem_t lock;       // binary semaphore (basic lock)
3       sem_t writelock;  // used to allow ONE writer or MANY readers
4       int readers;      // count of readers reading in critical section
5   } rwlock_t;
6
7   void rwlock_init(rwlock_t *rw) {
8       rw->readers = 0;
9       sem_init(&rw->lock, 0, 1);
10      sem_init(&rw->writelock, 0, 1);
11  }
12
13  void rwlock_acquire_readlock(rwlock_t *rw) {
14      sem_wait(&rw->lock);
15      …
```

```
15      rw->readers++;
16      if (rw->readers == 1)
17              sem_wait(&rw->writelock); // first reader acquires writelock
18      sem_post(&rw->lock);
19  }
20
21  void rwlock_release_readlock(rwlock_t *rw) {
22      sem_wait(&rw->lock);
23      rw->readers--;
24      if (rw->readers == 0)
25              sem_post(&rw->writelock); // last reader releases writelock
26      sem_post(&rw->lock);
27  }
28
29  void rwlock_acquire_writelock(rwlock_t *rw) {
30      sem_wait(&rw->writelock);
31  }
32
33  void rwlock_release_writelock(rwlock_t *rw) {
34      sem_post(&rw->writelock);
35  }
```

- Fairness 문제 존재
  - reader가 writer를 starve시킬 가능성이 높음
  - writer가 대기중일때 다른 새로운 reader들이 못들어오게 해야함

Basic Solution(Using Condition Variable)

- 여기서 시험문제 하나 나옴
- Writer가 없을 때 Reader가 데이터베이스에 접근 가능
- Writer는 Reader/Writer가 없을 때 데이터베이스에 접근 가능
- 한번에 하나의 스레드만 상태 변수를 조종할 수 있음

기본구조

- Reader()
  - Writer가 없을때까지 기다리고
  - 데이터베이스에 접근한뒤

    - Check Out하면서 기다리고 있는 Writer를 깨움
- Writer()
    - Active Reader/Writer가 없을때까지 기다리고
    - 데이터베이스에 접근한뒤
    - Check Out하면서 기다리고 있는 Readers/Writer를 깨움

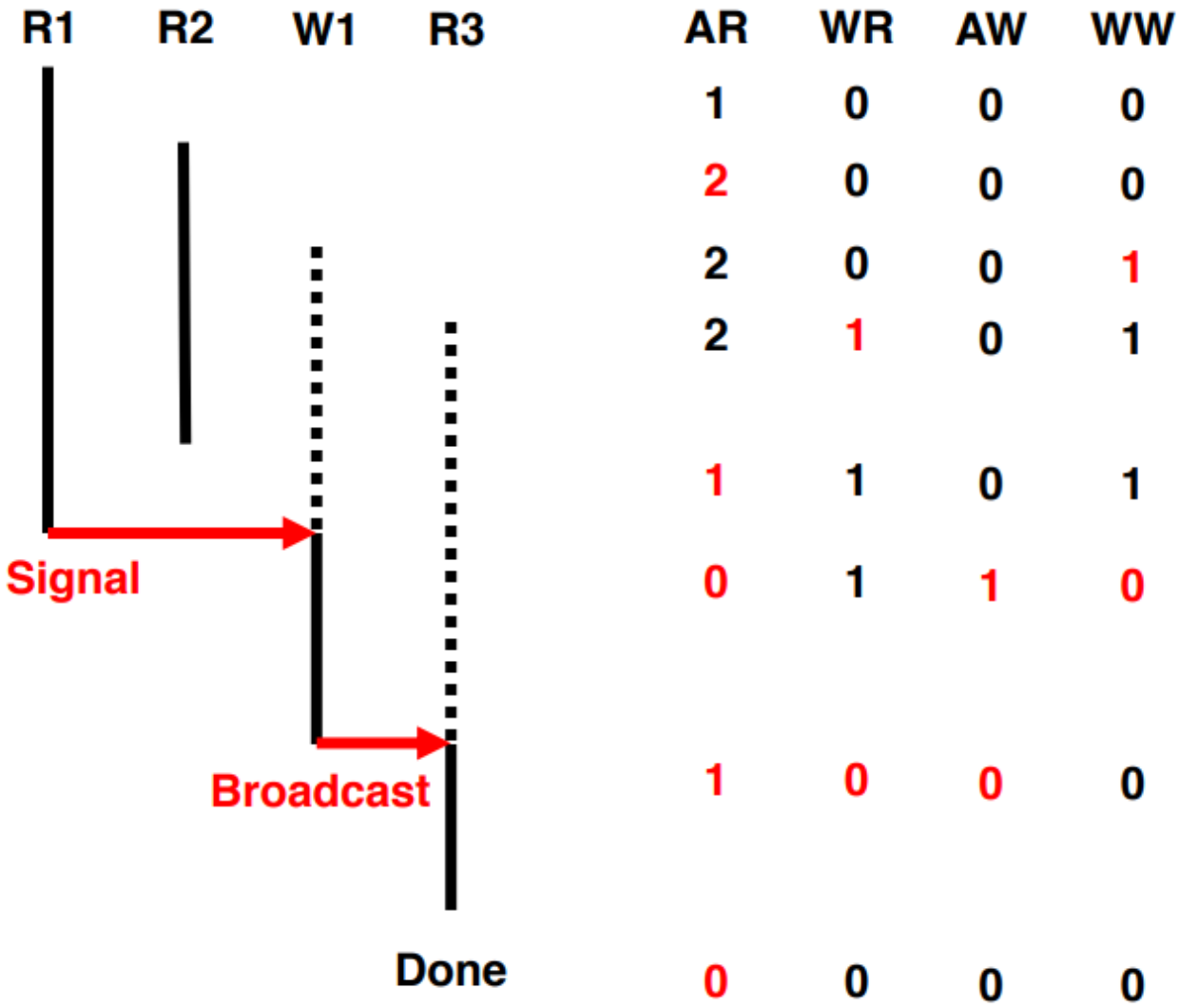
- State Variables
    - AR : Active Reader의 수 / 초기값 0
    - WR : Wating Reader의 수 / 초기값 0
    - AW : Active Writer의 수 / 초기값 0
    - WW : Waiting Writer의 수 / 초기값 0
    - 상태 okToRead = NIL
    - 상태 okToWrite = NIL

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {  // Is it safe to read?
    WR++;                  // No. Writers exist
    okToRead.wait(&lock);  // Sleep on cond var
    WR--;                  // No longer waiting
  }
  AR++;                    // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                    // No longer active
  if (AR == 0 && WW > 0)   // No other active readers
    okToWrite.signal();    // Wake up one writer
  lock.Release();
}
```

Why release lock here?

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```

- R1, R2, W1, R3 순으로 온다고 가정



The Dining Philosophers

- 5명의 철학자가 테이블을 둘러싸고 있다고 가정
  - 2명의 철학자 사이에는 하나의 포크가 존재(총 5개)
  - 각 철학자는 생각할때(포크 필요없음), 먹을때(포크 2개 필요)가 있음
  - 이 포크를 위한 투쟁
- Key Challenge
  - Deadlock은 없어야 함
  - 어떤 철학자도 Starve하는 일은 없어야 함
  - Concurrency가 높다

```
while (1) {
        think();
        getforks();
        eat();
        putforks();
}
```

Basic loop of each philosopher

```
// helper functions
int left(int p) { return p; }

int right(int p) {
        return (p + 1) % 5;
}
```

Helper functions (Downey's solutions)

- 왼쪽의 포크를 요청할 때 left, 오른쪽은 right 호출

```
1    void getforks() {
2         sem_wait(forks[left(p)]);
3         sem_wait(forks[right(p)]);
4    }
5
6    void putforks() {
7         sem_post(forks[left(p)]);
8         sem_post(forks[right(p)]);
9    }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

- 포크 하나별로 Semaphore 하나가 필요함
  - 데드락 발생
    - 모든 철학자가 자신 왼쪽의 포크를 집었을경우, 모두 영원히 데드락
  - 포크를 얻는방법을 바꿔야 함
    - 4번째 철학자는 포크를 얻는방법을 다르게 바꿈(오른쪽 먼저)
    - 데드락 발생 가능성이 없어짐

```
1    void getforks() {
2         if (p == 4) {
3                   sem_wait(forks[right(p)]);
4                   sem_wait(forks[left(p)]);
5         } else {
6                   sem_wait(forks[left(p)]);
7                   sem_wait(forks[right(p)]);
8         }
9    }
```

How to Implement Semaphores

- Zemaphore 라는걸 만듬

```
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   …
```

```
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   }
```

- 값이 0보다 작아질 수 없음
- 구현하기 쉽고, 현재의 리눅스 구현과 일치

Semaphore에서 Monitor를 만들수 있을까?

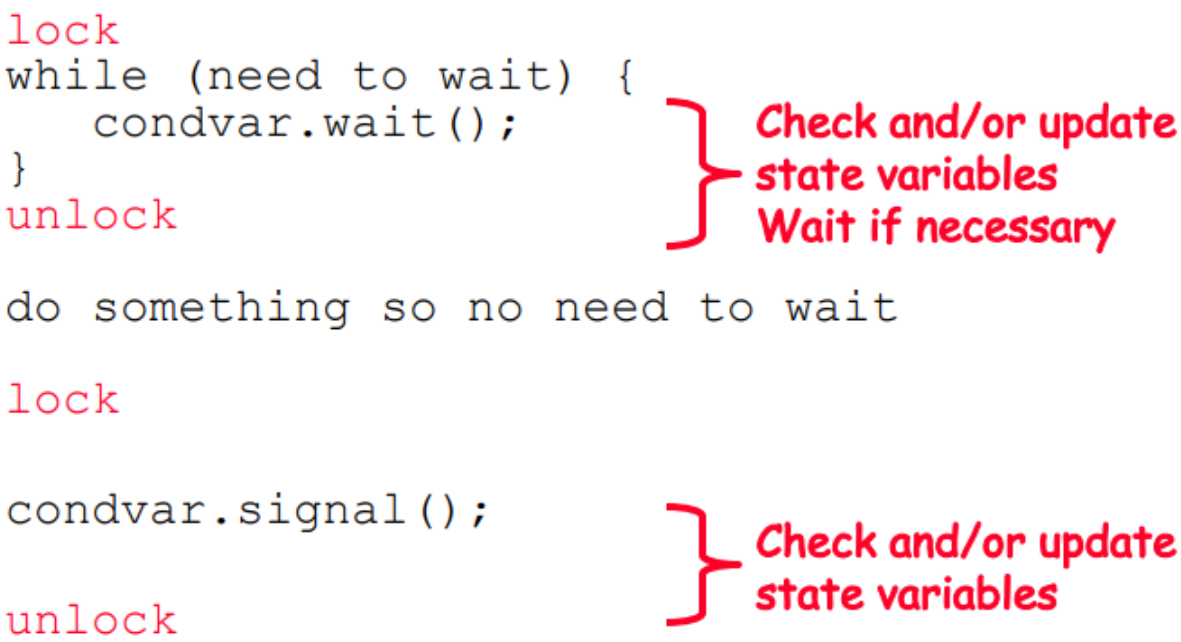
- 보기에는 쉬움 : Mutex 사용하기

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Wait가 Lock을 가진채로 Sleep에 들어갈 가능성이 있음

```
Wait(Lock *lock) {
    lock->Release();
    semaphore.P();
    lock->Acquire();
}
Signal() { semaphore.V(); }
```

- 상태변수는 History가 없고, Semaphore는 있다
  - 스레드 하나가 Signal하고, 아무도 안기다리고 있으면?
- 슬라이드 참조 너무 어려움 ㅅㅂ;
- 결론
  - 모니터는 프로그램의 로직을 뜻함
    - 필요하면 Wait
    - 뭔가 바꾸면 Signal, 다른 스레드가 일하도록
  - Monitor-Based 프로그램의 기본 구조



  - Lock + 하나 이상의 Condition Variables
    - 공유 데이터에 접근할때 lock을 꼭 획득하고
    - Critical Section 안에서 대기해야할때 Condition Variable 사용
      - Wait(), Signal(), Broadcast()

Common Concurrency Problems

- 실제 상황에서 발생하는 Concurrency Problem에 주목

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

Non-Deadlock Bugs

- 대부분의 Concurrency Bug를 차지함

Atomicity Violation

- 다양한 메모리 접근 요청으로부터 요구되는 Serializability가 위반됨

```
1      Thread1::
2      if(thd->proc_info){
3         …
4         fputs(thd->proc_info , …);
5         …
6      }
7
8      Thread2::
9      thd->proc_info = NULL;
```

- 두개의 스레드가 동시에 thd->proc_info에 접근
- 해결법 : 해당 부분에 Lock을 더한다

Order Violation

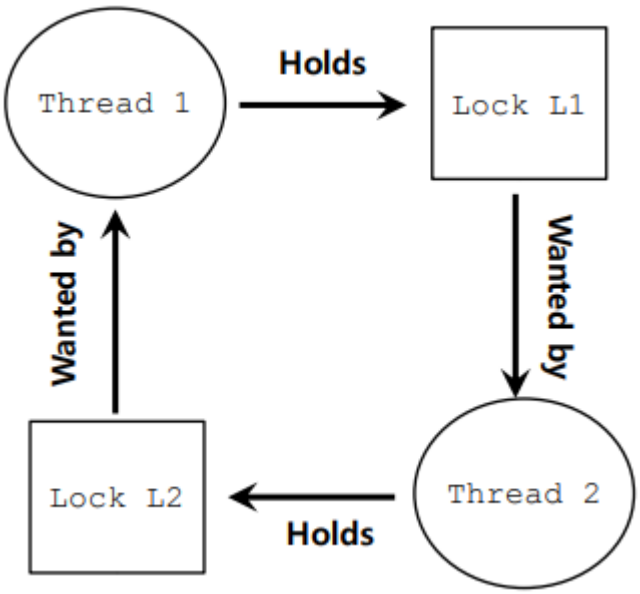
- 메모리 접근의 순서가 요구한대로 안되고, 뒤바뀜

```
1    Thread1::
2    void init(){
3         mThread = PR_CreateThread(mMain, …);
4    }
5
6    Thread2::
7    void mMain(…){
8         mState = mThread->State
9    }
```

- Thread2에서 실행되는 문장은 초기화 된 후 실행 돼야함
- 해결법 : Condition Variable을 사용해서 Ordering을 강제함

```
1   pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2   pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3   int mtInit = 0;
4
5   Thread 1::
6   void init(){
7       …
8       mThread = PR_CreateThread(mMain,…);
9
10      // signal that the thread has been created.
11      pthread_mutex_lock(&mtLock);
12      mtInit = 1;
13      pthread_cond_signal(&mtCond);
14      pthread_mutex_unlock(&mtLock);
15      …
16  }
17
18  Thread2::
19  void mMain(…){
21      // wait for the thread to be initialized …
22      pthread_mutex_lock(&mtLock);
23      while(mtInit == 0)
24              pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      …
29  }
```

Deadlock Bugs



- 사이클의 존재
  - Thread1이 L1을 가지고있고, Thread2는 L2를 가지고 있고, 서로 기다리는 상황
- 발생하는 이유

    - 큰 코드 베이스의 경우, Component들 사이에 복잡한 의존성이 존재
    - Encapsulation의 특성 때문
        - 모듈화를 위해 구체적인 구현을 숨기려함
        - locking과는 잘 맞지 않음
- 예시) 자바 벡터 클래스

```
1    Vector v1,v2;
2    v1.AddAll(v2);
```

    - v1, v2를 위한 lock이 각각 필요함
    - 다른 스레드가 v2.AddAll(v1)을 거의 동시에 실행하게 되면, Deadlock 발생
- Deadlock의 발생조건

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
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 thread holds one more resources that are being requested by the next thread in the chain

    - 위 4가지 조건중 하나라도 만족이 안되면, 데드락은 발생할 수 없음

Circular Wait

- Lock의 획득에 Total Ordering을 제공
    - Global Locking Strategy에 조심스러운 디자인이 필요함
- 예시
    - 시스템에 락이 2개 (L1, L2) 있음
    - L2를 얻기 전 항상 L1을 얻게 함으로써 데드락 예방 가능

Hold-and-Wait

```
1    lock(prevention);
2    lock(L1);
3    lock(L2);
4    ...
5    unlock(prevention);
```

- 모든 락을 한번에 획득하게 함(Atomically)
    - Lock을 획득할 때 스레드 스위치가 없게 해야 함
- 문제점
    - 어떤 락이 필요한지 확인하고, 미리 획득해놔야 함

- Concurrency 하락

No Preemption

- 여러개의 Lock을 한번에 얻으려할때 문제점이 하나의 Lock을 기다릴 때 다른걸 들고있기 때문
- trylock()
  - deadlock-free, ordering-robust lock acqusition protocol을 만들기 위해 사용
  - 가용하면 락을 가져옴
  - 아니면 -1 리턴 (나중에 다시 해야함)

```
1    top:
2         lock(L1);
3         if( tryLock(L2) == -1 ){
4                   unlock(L1);
5                   goto top;
6         }
```

- livelock
  - 계속해서 특정 코드를 일정하게 시도하지만 **성과가없는 경우**(반복)
  - 해결법 : loop back하기전에 랜덤시간 딜레이를 추가함

Mutual Exclusion

- Wait-Free
  - 강력한 하드웨어 명령을 사용
  - 명백한 Locking이 필요없는 자료구조를 만듬

```
1    int CompareAndSwap(int *address, int expected, int new){
2         if(*address == expected){
3                   *address = new;
4                   return 1; // success
5         }
6         return 0;
7    }
```

- 특정값을 일정량만큼 Atomic하게 증가시키고 싶을 때

```
1    void AtomicIncrement(int *value, int amount){
2         do{
3                   int old = *value;
4         }while( CompareAndSwap(value, old, old+amount)==0);
5    }
```

  - 획득하는 Lock이 없고
  - 데드락 가능성도 없음
  - livelock은 발생가능
- 더 복잡한 예시(리스트 삽입)

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n->value = value ;
5         n->next  = head;
6         head     = n;
7    }
```

- 여러 스레드가 동시에 호출할 시, 경쟁 조건 발생
- 해결법
  - Lock Acquire/Release로 감싸기

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n->value = value ;
5         lock(listlock); // begin critical section
6         n->next  = head;
7         head     = n;
8         unlock(listlock) ;  //end critical section
9    }
```

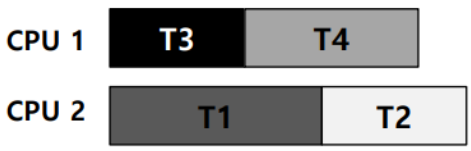
  - Wait-Free 방법(Compare-and-Swap)

```
1    void insert(int value) {
2         node_t *n = malloc(sizeof(node_t));
3         assert(n != NULL);
4         n->value = value;
5         do {
6                   n->next = head;
7         } while (CompareAndSwap(&head, n->next, n));
8    }
```

Deadlock Avoidance via Scheduling

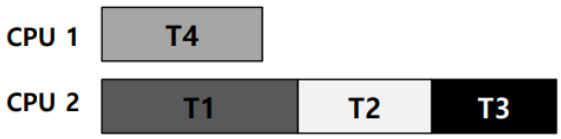
- Global Knowledge가 필요함
  - 각 스레드가 실행될 때 필요로 할 lock들
  - 스레드를 데드락 발생 가능성이 없는 순서로 실행시킴
- 2개의 프로세서 / 4개의 스레드가 있다고 가정

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



  - 똑똑한 스케쥴러는 T1/T2가 동시에 실행되지 않는다면 데드락 발생하지 않는다는걸 암

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no



  - 비슷한 상황, T3도 동시에 실행되면 안되기 때문에 전체 실행 시간이 늘어남

Banker's Algorithm

- ABCD를 다 도와주자
- Safe : 망하기 전
- Unsafe : 다 망함
- 안전상태를 유지할 수 있는 요구만을 수락하고, 불안전 상태를 초래할 요구는 나중에 만족될 수 있을때까지 계속 거절

Detect and Recover

- 데드락이 가끔은 발생하도록 허용하고, 발생할 시 특정 행동 수행
    - 예시) 운영체제가 멈추면, 재부팅
- 다수의 데이터베이스 시스템이 이 방식 채용
    - Deadlock Detector가 주기적으로 실행
    - Resource Graph를 만들어 사이클이 생기는지 확인
    - 데드락 발생시 시스템 재시작