# 18. Semaphores

lock 혹은 condition variable로 사용 ⇒ init 값에 따라 사용 용도가 달라짐

- POSIX semaphores

```
#include <semaphore.h>

int sem_init(sem_t *s, int pshared, unsigned int value);
//pshared : 하나의 process 내부에 있는지
//or 다른 address space를 가지는 여러 process 내부에 있는지 check
//pshared == 0 -> 하나의 process 내부(address space 공유) ⇒ 이때만 신경써서 봅시다!
//pshared == o.w -> semaphore가 반드시 공유 메모리에 할당되어야 함

int sem_wait(sem_t *s); → s--
//s가 음수면 잠들어버림

int sem_post(sem_t *s); → s++
//자고 있는 thread가 있다면 하나를 깨워 줌
```

## ▼ Semaphore 사용 방식

### ▼ 1. Lock으로 사용하기 → Binary semaphores

- Semaphore → 초기값 system 내에서 결정 x, 사용 용도에 따라 다르게 사용됨 ⇒ lock 과 가장 다른 점

```
sem_t m;
sem_init(&m, 0, 1);
sem_wait(&m);
// critical section here
sem_post(&m);
```

semaphore 초기 값 ⇒ lock or condition variable

thread 간의 address space 공유 (한 process 내부)

- 첫번째 진입
  - 감소해도 음수가 x → thread no sleep → critical section 진입 good
- 두번째 진입 이후
  - 감소하면 음수가 됨 → thread sleep ⇒ sem_post()로 다시 양수로 만들어줘야 함!!
  - 잠든 thread가 다시 깨워져서 critical section 내부로 진입

⇒ lock한 놈이 lock 해줘야 함.

| 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 | *Interrupt; Switch→T1* | Ready | | Run |
| 0 | | Ready | call sem_wait() | Run |
| -1 | | Ready | decr sem | Run |
| -1 | | Ready | (sem<0)→sleep | Sleep |
| -1 | | Run | *Switch→T0* | Sleep |
| -1 | (crit sect end) | Run | | Sleep |
| -1 | call sem_post() | Run | | Sleep |
| 0 | incr sem | Run | | Sleep |
| 0 | wake(T1) → 잠든 thread 대신 wake | Run | | Ready |
| 0 | sem_post() returns | Run | | Ready |
| 0 | *Interrupt; Switch→T1* | 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 |

## ▼ 2. Condition Variable로 사용하기 → semaphores for ordering

thread 간의 실행 순서 지키도록!!

```c
sem_t s;

void * child(void *arg) {
  printf("child\n");
  sem_post(&s); → Semaphore ++ → 자고 있는 thread가 있다면 깨워줌.
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t c;
  sem_init(&s, 0, X); // what should X be? -> 0이 되어야 함! → condition variable로 사용!!
  printf("parent: begin\n");
  pthread_create(&c, NULL, child, NULL);
  sem_wait(&s);
  printf("parent: end\n");
  return 0;
}
```

Condition var → sem_post, sem_wait가 다른 thread에서 호출되는게 일반적

| Val | Parent | State | Child | State |
|---|---|---|---|---|
| 0 | create(Child) | Run | *(Child exists, can run)* | Ready |
| 0 | call sem_wait() | Run | | Ready |
| -1 | decr sem | Run | | Ready |
| -1 | (sem<0)→sleep | Sleep | | Ready |
| -1 | *Switch→Child* | Sleep | child runs | Run |
| -1 | | Sleep | call sem_post() | Run |
| 0 | | Sleep | inc sem | Run |
| 0 | | Ready | wake(Parent) | Run |
| 0 | | Ready | sem_post() returns | Run |
| 0 | | Ready | *Interrupt→Parent* | Ready |
| 0 | sem_wait() returns | Run | | Ready |

Figure 31.7: **Thread Trace: Parent Waiting For Child (Case 1)** child → parent

| Val | Parent | State | Child | State |
|---|---|---|---|---|
| 0 | create(Child) | Run | *(Child exists; can run)* | Ready |
| 0 | *Interrupt→Child* | Ready | child runs | Run |
| 0 | | Ready | call sem_post() | Run |
| 1 | | Ready | inc sem | Run |
| 1 | | Ready | wake(nobody) | Run |
| 1 | | Ready | sem_post() returns | Run |
| 1 | parent runs | Run | *Interrupt→Parent* | Ready |
| 1 | call sem_wait() | Run | | Ready |
| 0 | decrement sem | Run | | Ready |
| 0 | (sem≥0)→awake | Run | | Ready |
| 0 | sem_wait() returns | Run | | Ready |

깨울 thread가 아무도 X

Figure 31.8: **Thread Trace: Parent Waiting For Child (Case 2)** child → parent

# ▼ Producer/Consumer problem

## ▼ single producer/consumer

⇒ semaphore를 이용 : state 변수 관리하지 않아도 됨.

```
int buffer[MAX]; // bounded buffer
int fill = 0;
int use = 0;

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

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

sem_t empty, sem_t full;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);
    tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

```
int main(int argc, char *argv[]) {
  // ...
  //semaphore 초기값 결정이 정말 중요함
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0); // 0 are full
  // ...
}
```

⇒ mutiple일 경우 race condition 발생

여러 thread가 동시에 put, get을 호출하면 값을 동시에 변경할 수 있음.

## ▼ multiple producer/consumer

race condition 제거 ⇒ semaphore 하나더 선언!!

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(i);
    sem_post(&full);
    sem_post(&mutex);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
  }
}
```

⇒ race condition 해결 but 2개 이상의 thread가 waiting인데 어떤 한 thread가 waiting 되어 버린 애 중 하나로만 깨어날 수 있을 때 deadlock

⇒ ① → ② 순서로 실행된다면 서로 계속해서 semaphore를 풀어주기를 기다리기만 함 (영원히 깨어날 수 X) (항상 발생하는 것은 X, 여러 thread일 때 발생함)

## ▼ final

↳ critical을 줄이자!

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
  }
}
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
  }
}
```

## ▼ Reader-Writer Locks

Insert : 기존 상태 바꿈
Lookups : data structure 읽기

사용자별 lock을 구분해서 운영 → 용도에 맞게 구현 ⇒ 훨씬 속도 빠름

- Reader → 여러 개 있어도 race condition ⇒ 발생 X
  - *rwlock_acquire_readlock()*
  - *rwlock_release_readlock()*
- Writer → write가 관여할 때에는 critical section으로 선언해야 함 (하나만!!)
  - *rwlock_acquire_writelock()*
  - *rwlock_release_writelock()*

```
typedef struct _rwlock_t {
  // binary semaphore (basic lock)
  sem_t lock; (reader)
  // used to allow ONE writer or MANY readers
  sem_t writelock; (writer)
  // count of readers reading in critical section
  int readers; → critical section !!
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
  // first reader acquires writelock
  sem_wait(&rw->writelock); (뒤로 writer 더 못 들어오게 막고)
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
  // last reader releases writelock → 더 이상 접근하는 reader가 없도록!!
  sem_post(&rw->writelock); (writer 들어올 수 있도록!)
  sem_post(&rw->lock);
}
```

이 구현은 reader에게 상대적으로 좋고, wirter는 굶주릴 수 있다

## ▼ How to implement semaphores

lock, condition variable 필요한가? → lower-level의 sunchoronization primitives

⇒ semaphore 구현 시에 필요함

```c
typedef struct __Sem_t {
  int value;
  pthread_cond_t cond;
  pthread_mutex_t lock;
} Sem_t;
// only one thread can call this

void Sem_init(Sem_t *s, int value) {
  s->value = value;
  Cond_init(&s->cond);
  Mutex_init(&s->lock);
}

void Sem_wait(Sem_t *s) {
  Mutex_lock(&s->lock);
  while (s->value <= 0)
    Cond_wait(&s->cond, &s->lock);
  s->value--;  (sem 감소)
  Mutex_unlock(&s->lock);
}

void Sem_post(Sem_t *s) {
  Mutex_lock(&s->lock);
  s->value++; (sem 증가)
  Cond_signal(&s->cond);
  Mutex_unlock(&s->lock);
}
```

Sec 31.6 The Dining_philosophers 읽어보기!!