Semaphores

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Semaphores: A Definition

상호 배제 + 동기화 한번에!

→ OS concurrency의 핵심.

- A semaphore invented by Dijkstra is an object with an integer value that we can manipulate with two routines
  - In the POSIX standard, these routines are `sem_wait()` and `sem_post()`
  - The semaphore must be initialized since its initial value determines the behavior

↳ 초기화 값에 따라 → lock으로도 쓰일 수 있고 condition variable로 쓰일 수 있음

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

semaphore를 공유

This code declares a semaphore s and initialize it to the value 1
0 indicates a shared semaphore between threads in the same process

- After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`

→ 조건 값이 내부적으로 condition을 결정

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}
```

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

  - **`sem_wait()`** either returns right away (s--) or causes the caller to suspend execution waiting (if s<0) for a subsequent post (s++)
  - **`sem_post()`** simply increases the value of the semaphore (s++) and then, wakes one of them up if there is any waiting thread
  - The semaphore value, when negative, is equal to the number of waiting threads

Binary Semaphores (Locks)

- **Consider a code that tries to use a semaphore as a lock**
  - What should the initial value of the semaphore X be? By definition, X=1
- **Consider scenario with two threads**
  - T_0 calls **sem_wait()** (m=0), enters critical section, calls **sem_post()** (m=1)
  - **1)** T_0 calls **sem_wait()** (m=0), begins critical section, **2)** T_1 runs, tries critical section, calls **sem_wait()** (m=-1), sleeps, **3)** T_0 runs, ends critical section, calls **sem_post()** (m=0), wakes, **4)** T_1 runs, critical section, calls **sem_post**()(m=1)

①

```
sem_t m;
sem_init(&m, 0, X); // initialize to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

초기값을 1로 해주면 lock 가능

= binary semaphore

Value of Semaphore	Thread 0	Thread 1
1		
1	call sem_wait()	
0	sem_wait() returns	
0	(crit sect)	
0	call sem_post()	
1	sem_post() returns	

sleep전의 스위칭 가능

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)	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

1개가 기다리고 있다

sleep

바로 switching해줘야 해!.

Semaphores For Ordering → initial value 0 (조건 변수로 사용)

- **Semaphores are useful to order events in a concurrent program**
  - What should the initial value of this semaphore be? It should be set to 0
- **There are two cases to consider for the parent-child program**
  - Parent→Child: **1)** parent calls **`sem_wait()`** (s=-1), sleeps, **2)** child runs, calls **`sem_post()`** (s=0), wakes (P), **3)** parent runs, returns from **`sem_wait()`**
  - Child→Parent: **1)** child runs, calls **`sem_post()`** (s=1), wakes (nothing), **2)** parent runs, calls **`sem_wait()`** (s=0) and **`sem_wait()`** returns immediately due to s≥0

```
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;
}
```

깨움. 스레드는 없음.

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

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

The Producer/Consumer Problem: First Attempt

→ 이거 세마포어로 잘 되나?

- **The first attempt introduces two semaphores, empty and full**
  - Image there is two threads, a producer and a consumer, with MAX=1
  - **1)** consumer runs, calls **`sem_wait(&full)`** (f=-1), sleeps, **2)** producer runs, calls **`sem_wait(&empty)`** (e=0), **`put()`**, **`sem_post(&full)`** (f=0), wakes: Then, **3)** if the producer continues, calls **`sem_wait(&empty)`** (e=-1), sleeps
    or **3)** if the consumer runs, returns from **`sem_wait(&full)`**, consumes
  - It works with two or more threads, but what if multiple threads with MAX > 1?
  - Imagine two producers (P_a, P_b) call **`put()`**: **1)** P_a puts a value to **`buffer[0]`** and interrupted before **`fill++`**, **2)** P_b puts a value to **`buffer[0]`**?...NO!

semaphore 2개

buffer의 크기 1인 상황.

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

void put(int value) {
    buffer[fill] = value;    // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];   // Line G1
    use = (use + 1) % MAX;   // Line G2
    return tmp;
}
```

또 produce 하도. sleep 해제 할 수 있다.

→ mutual exclusion 이 구현되어야 한다.

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);         // Line C1
        tmp = get();             // Line C2
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}
```

소비자를 깨움

생산자를 깨움

MAX가 1보다 크면 문제.

Initialization

```
sem_init(&empty, 0, MAX);
sem_init(&full, 0, 0);
```

P_a P_b

max가 증가하면

A Solution: Adding Mutual Exclusion

(dead-lock ...)

상호 배제를 고려해 추가함 → binary semaphore

- **Filling a buffer and increasing the index is a critical section**
  - We can use the binary semaphore to add some locks (left code)
  - Unfortunately, it does not work due to deadlock; image a produce, a consumer: **1)** consumer calls **`sem_wait(&mutex)`** (m=0), **`sem_wait(&full)`** (f=-1), sleeps: holds the lock, **2)** producer runs, calls **`sem_wait(&mutex)`** (m=-1), sleeps; stuck ➔ consumer holds lock and waiting for signal full and producer is waiting for lock
  - The solution is to reduce the scope of the lock (see right code)

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);      // Line P0 (NEW LINE)
        sem_wait(&empty);      // Line P1
        put(i);                // Line P2
        sem_post(&full);       // Line P3
        sem_post(&mutex);      // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);      // Line C0 (NEW LINE)
        sem_wait(&full);       // Line C1
        int tmp = get();       // Line C2
        sem_post(&empty);      // Line C3
        sem_post(&mutex);      // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

mutex 1 → 0 → producer sleep
full 0 → -1 → consumer sleep
emty 1

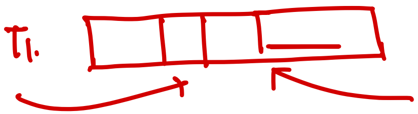
dead-lock x
lock의 범위를 줄여줌

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);      // Line P1
        sem_wait(&mutex);      // Line P1.5 (MUTEX HERE)
        put(i);                // Line P2
        sem_post(&mutex);      // Line P2.5 (AND HERE)
        sem_post(&full);       // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);       // Line C1
        sem_wait(&mutex);      // Line C1.5 (MUTEX HERE)
        int tmp = get();       // Line C2
        sem_post(&mutex);      // Line C2.5 (AND HERE)
        sem_post(&empty);      // Line C3
        printf("%d\n", tmp);
    }
}
```

mutex 1 → 0 → 1
full 0 → -1 → sleep
empty 1 → 0

Reader-Writer Locks

T1.  T3

읽기만 하면 동시에 읽어도 상관X
그리고 write 할때는 lock.

- **Read-writer lock allows many concurrent reads and single write**
  - A single writer can acquire the lock (**`writelock`**) to update the data structure
  - When acquiring a read lock, the reader first acquires **`lock`** and increase the **`reader`** variable to track how many reads are currently inside the data structure
  - →읽는 동안에 쓰면 안되니까.
  - When the first reader acquires the lock, the reader also acquires the write lock (**`writelock`**), and then will release the lock later after the last reader is done
  - Once a reader has **`lock`**, more readers can acquire, but any thread to write has to wait until finishing all readers ➔ relatively easy for readers to starve writers

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in 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_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}
```

→ writer를 위한 세마포어

→ read 할때 lock 걸.

→ 첫번째 reader가 lock 취득

readlock 해제

```
void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

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

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

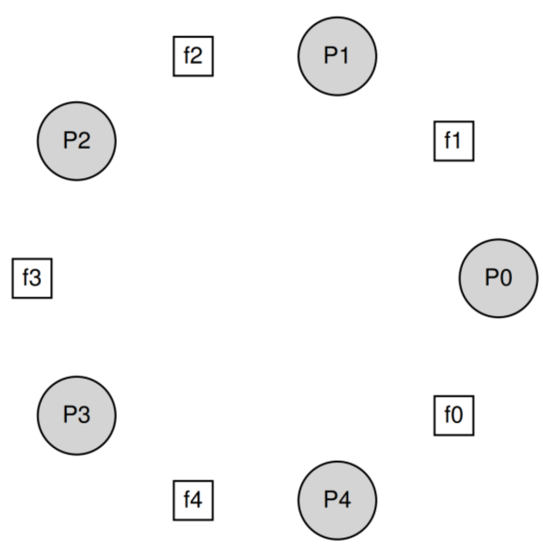
reader가 0명이면 write lock 풀어도 됨.

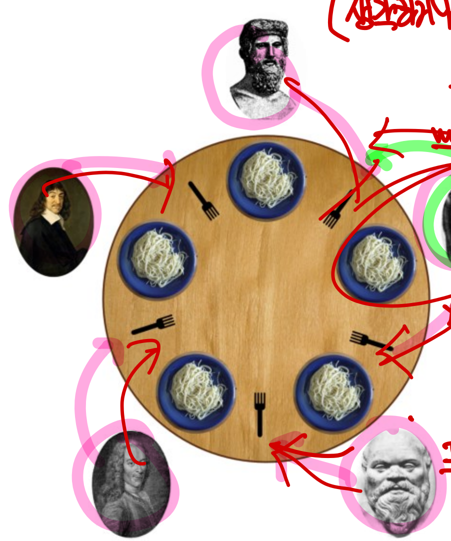
starvation.
writer는 계속 대기할 수 밖에 없음 (reader 우선)

The Dining Philosophers: Introduction

쏘 있는 왕개념!

- **Dining philosopher’s problem is a famous concurrency problem**
  - It is fun and intellectually interesting; however, its practical utility is low
  - Assume there are five “philosophers” on a table and between each pair of philosophers is a single fork (and thus, five forks total)
  - The philosophers each have times where they think, and don’t need any forks, and where they eat, and a philosopher needs two forks to eat (left and right)
    2개의 포크 필요
  - Consider the basic loop of each philosopher; the key is to write **`get_forks()`** and **`put_forks()`** to attain no deadlock, no starvation, and high concurrency
    →어떻게 해야 할까?






```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

Each philosopher has a unique thread identifier **p** from 0 to 4 (inclusive)

The Dining Philosophers: Solutions

- **We need some semaphores to solve the problem**
  - Let's assume we have five semaphores, one for each fork
    5개 세마포어
  - Two helper functions to get left and right forks when philosopher **p** wants to eat

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

왼쪽, 오른쪽 포크 번호는 판단하는?

철학자 번호.

- **Broken solution: deadlock**
  - To pick the forks, we simply grab a lock on each; one on the left and then one on the right and release them when done eating → deadlock

5명 모두 포크를 (left) get 한다고 했을때 ↓ deadlock 무한루프가 발생함

Deadlock

Each will be stuck holding one left fork and waiting for another, forever

```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

왼쪽, 오른쪽 lock을 걸음

lock을 풀

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

한명은 오른쪽 먼저 하면.

마지막은 오른쪽.

No deadlock

- **A solution: breaking the dependency**
  - The simplest way to attack this problem is to change how forks are acquired
  - Assume that philosopher 4 gets the forks in a different order than the others
  - There is no situation where each philosopher grabs one fork and is stuck waiting for another → the cycle of waiting is broken

How to Implement Semaphores

- **Building our own semaphores using lock and condition variable**
  - One lock and one condition variable, plus a state variable to track the value of the semaphore are used to implement a semaphore
  - What the subtle difference between ours and Dijkstra's is that the negative value of the semaphore doesn't indicates the number of waiting threads
  - The value will never be lower than zero → current Linux implementation

lock + condition variable + state variable

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}
```

```
void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Zem_wait

s->value ≤ 0

zem_post

S = -1

→ state variable

semaphore의 값을 tracking

zem_wait

negative로 가지 않음

thread 1 running

thread 2 sleeping

thread wake

thread 1

thread 2

- **Too many threads are running and bogging the system down?**
  - Then, decide upon a threshold for "too many", and then use a semaphore to limit the number of threads concurrently executing the piece of code
  - This approach is called throttling, is a form of admission control

Summary

- **A semaphore an object with an integer value that is manipulated by two routines, `sem_wait()` and `sem_post()`**
  – The semaphore must be initialized sine its initial value determines the behavior
- **A semaphore is like a bouncer at the front of the lineup**
  – The bouncer only allows threads to proceed when instructed to do so