Operating System: Semaphores

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

Semaphore: A definition

- Semaphore
  - An object with an integer value, S
- We can manipulate with two routines;
  - sem_wait(S)
    - originally called P(S)
  - sem_post(S)
    - originally called V(S)

lock 과 같은 기능

unlock 과 같

```
1    sem_t m;
2    sem_init(&m, 0, X);
3
4    sem_wait(&m);
5    //critical section here
6    sem_post(&m);
```

Using semaphore

Semaphore: Initialization

- Initialization

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

초기화

0: 프로세스내 스레드 들끼리 세마포어 공유

  - Declare a semaphore `s` and initialize it to the value 1
  - The second argument, 0, indicates that the semaphore is shared between *threads in the same process*

Semaphore: Interact with semaphore

- `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  }
```

값이 감소
세마포어 음수면 기다림.

  - If the value of the semaphore was *one* or *higher* when called `sem_wait()`, **return right away** 바로리턴
  - It will cause the caller to suspend execution waiting for a subsequent post post 호출까지 기다리기
  - When negative, the value of the semaphore is equal to the number of waiting threads

음수면 그 음수값이 기다리고 있는 thread의 수

Semaphore: Interact with semaphore (Cont.)

- `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  }
```

  - Simply **increments** the value of the semaphore 값증가
  - If there is a thread waiting to be woken, **wakes** one of them up 기다리는 것을 하나깨우기

Binary Semaphores (Locks)

- What should X be?
  - The initial value should be 1

```
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);
```

binary semaphore는 1개만 관리해야하니까

X가 1이어야하겠지?

Thread Trace: Single Thread Using a 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	

실행예시

Thread Trace: Two Threads Using a Semaphore

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

Semaphores As Condition Variables

```
1   sem_t s;
2
3   void *
4   child(void *arg) {
5       printf("child\n");
6       sem_post(&s); // signal here: child is done
7       return NULL;
8   }
9
10   int
11   main(int argc, char *argv[]) {
12      sem_init(&s, 0, X); // what should X be?
13      printf("parent: begin\n");
14      pthread_t c;
15      pthread_create(c, NULL, child, NULL);
16      sem_wait(&s); // wait here for child
17      printf("parent: end\n");
18      return 0;
19   }
```

A Parent Waiting For Its Child

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

The execution result

- What should x be?
  - The value of semaphore should be set to is 0

Thread Trace: Parent Waiting For Child (Case 1)

- The parent call `sem_wait()` before the child has called `sem_post()`

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

Thread Trace: Parent Waiting For Child (Case 2)

- The child runs to completion before the parent call `sem_wait()`

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

Case 1, 2 모두 child가 먼저 실행되게끔 설정함.

interrupt가 언제일어나느냐에 따라 case에서 다른결과값이 나올것임.

Classical Problems of Synchronization

- Solutions with semaphore for the following problems
  - Producer/Consumer (Bounded buffer) Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

이것들이 semaphore 가 해결할 수 있는 문제!

Bounded buffer problem

- Producer
  - Produce data items
  - Wish to place data items in a buffer

buffer에 데이터를 생성해서 저장하는 생산자

- Consumer
  - Grab data items out of the buffer consume them in some way

buffer의 데이터를 가져감.

- Example: Multi-threaded web server
  - *A producer* puts HTTP requests in to a work queue
  - *Consumer threads* take requests out of this queue and process them

서버 한쪽에서 들어오는 예시.

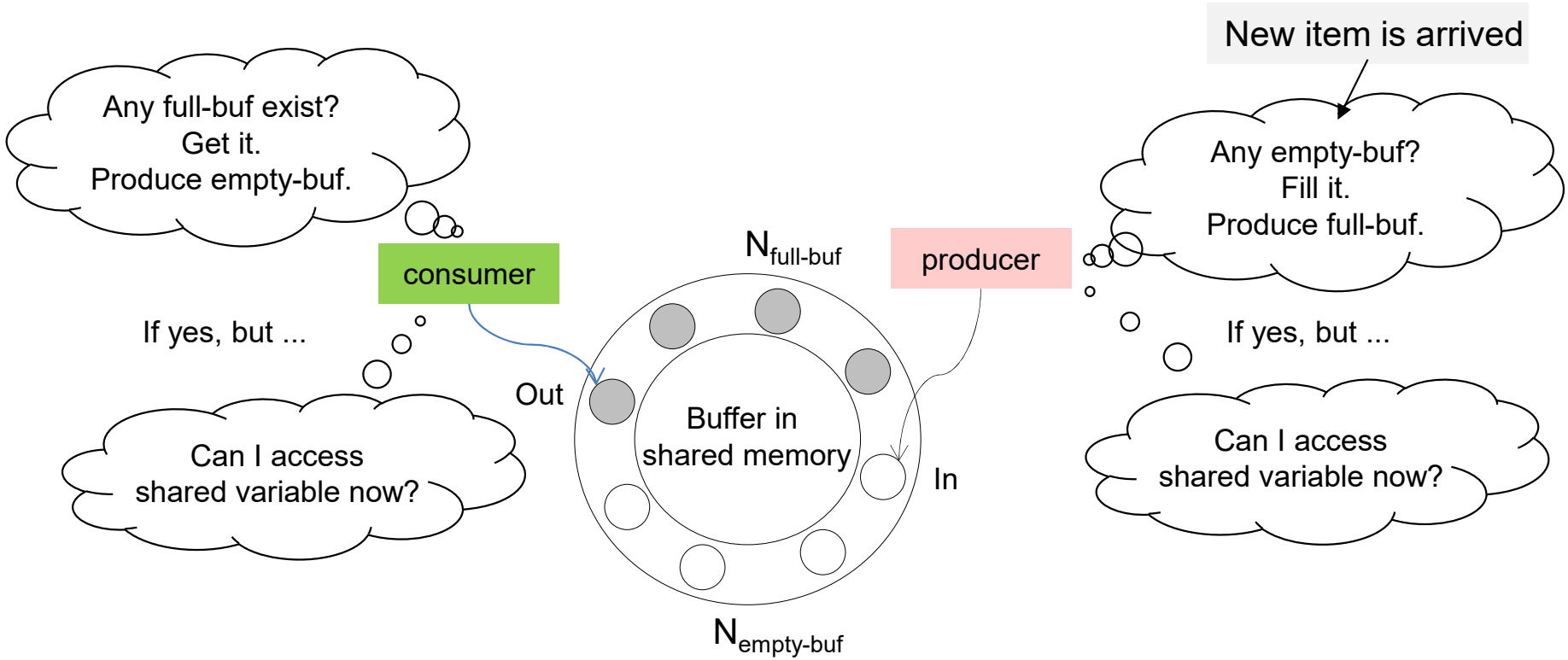
Bounded buffer problem (Cont.)

- 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
    - The wc process is the consumer
    - Between them is an in-kernel bounded buffer
  - Bounded buffer is Shared resource → **Synchronized access** is required

file.txt에서 foo라는 애 찾아서 그 출력을 wc 로 넘기는데 그 사이 파이프로 넘길때 bounded buffer라는 걸 사용해 그래서 이를 bounded buffer는 공유자원 grep은 producer wc는 consumer로 비유함

Bounded buffer problem (Cont.)

- Supposed that there are N buffers



A Solution

- Producer: `put()` interface
  - Wait for a buffer to become *empty* in order to put data into it
- Consumer: `get()` interface
  - Wait for a buffer to become *filled* before using it

```
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  }
```

A Solution (Cont.)

```
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    }
22    …
```

공유자원의 접근을 서로 제어해주니까 producer가 full을 늘려주기 전엔 consumer가 접근을 못하고 consumer가 empty를 늘려주기 전까진 producer가 접근을 못함 따라서 mutual exclusive가 잘된다.

First Attempt: Adding the Full and Empty Conditions

A Solution (Cont.)

```
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   }
```

First Attempt: Adding the Full and Empty Conditions (Cont.)

하지만 producer 혹은 consumer가 여러개일때는 MAX값이 2이상이라 상호배제가 잘 되지 않을 것이다.

- Imagine that `MAX` is greater than 1
  - If there are multiple producers, race condition can happen at line *f1*
  - It means that the old data there is overwritten
- We've forgotten here is **mutual exclusion**
  - The filling of a buffer and incrementing of the index into the buffer is a critical section

A Solution: Adding Mutual Exclusion

```
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
(Cont.)
```

Adding Mutual Exclusion (Incorrectly)

A Solution: Adding Mutual Exclusion

```
(Cont.)
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  }
```

Adding Mutual Exclusion (Incorrectly)

뮤텍스를 추가했으나 이렇게 되면 한쪽은 뮤텍스가 해제되길 기다리는데 다른 한쪽은 세마포어 값을 증가시키길 기다리게됨 deadlock이 생기는거지

A Solution: Adding Mutual Exclusion (Cont.)

- Imagine two thread: one producer and one consumer
  - The consumer **acquire** the `mutex` (line c0)
  - The consumer **calls** `sem_wait()` on the full semaphore (line c1)
  - The consumer is **blocked** and **yield** the CPU
    - ➢ The consumer still holds the mutex!
  - The producer **calls** `sem_wait()` on the binary `mutex` semaphore (line p0)
  - The producer is now **stuck** waiting too. a classic deadlock

```
8   sem_wait(&mutex);       // line p0
9   sem_wait(&empty);       // line p1
10  put(i);                 // line p2
11  sem_post(&full);        // line p3
12  sem_post(&mutex);       // line p4
```

producer

```
19  sem_wait(&mutex);       // line c0
20  sem_wait(&full);        // line c1
21  int tmp = get();        // line c2
22  sem_post(&empty);       // line c3
23  sem_post(&mutex);       // line c4
```

consumer

Finally, A 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
(Cont.)
```

Adding Mutual Exclusion (Correctly)

Finally, A Working Solution

그래서 뮤텍스를 버퍼에 값을 넣기 바로 직전에 삽입하였음

```
(Cont.)
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   }
```

Adding Mutual Exclusion (Correctly)

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set
  - Writers – read and write the data set

reader는 여러명이 공유자원에 접근해도 상관없지만 writer가 문제 그래서 이를 해결해보자

- Problem
  - Multiple readers can read at the same time
  - Only one single writer can access the shared data at the same time
- Shared Data
  - data set
  - integer readcount: the number of readers
  - semaphore mutex: mutual exclusion for access to read count
  - semaphore wrt: mutual exclusion for writer

Readers-Writers Problem (Cont.)

- The structures of writer and reader
  - semaphore mutex = 1, wrt = 1;
  - int readcount = 0;

누군가 읽고 있을때 write를 못하게끔 wrt를 감소시키고 또 누군가가 쓰고 있을때에도 write를 못하게끔 wrt를 감소시킨다. 또 다수가 reading을 할때 readcount 값을 증가시키는데에 있어서 경쟁조건을 해소시키기 위해 mutex를 추가한 것

writer

```
while (true) {
        sem_wait(wrt);

        // writing is performed

        sem_post(wrt);
}
```

reader

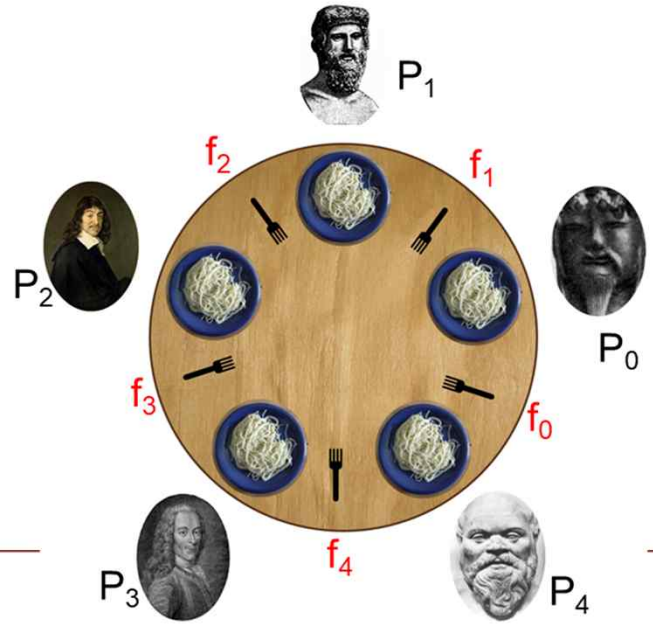
```
while (true) {
        sem_wait(mutex);
        readcount++;
        if (readcount == 1)        sem_wait(wrt);
        sem_post(mutex);

        // reading is performed

        sem_wait (mutex);
        readcount--;
        if (readcount  == 0)       sem_post(wrt);
        sem_post(mutex);
}
```

Dining Philosophers Problem

- Assume there are five "philosophers" sitting around a table
  - Between each pair of philosophers is a single fork (five total)
  - The philosophers each have times where they **think**, and don't need any forks, and times where they **eat**
  - In order to *eat*, a philosopher needs two forks, both the one on their *left* and the one on their *right*
  - **The contention for these forks**



Dining Philosophers (Cont.)

- Key challenge
  - There is **no deadlock**
  - **No** philosopher **starves** and never gets to eat
  - **Concurrency** is high

```
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)

- Philosopher `p` wishes to refer to the fork on their left → call `left(p)`
- Philosopher `p` wishes to refer to the fork on their right → call `right(p)`

Dining Philosophers (Cont.)

- We need some semaphore, one for each fork:
  `sem_t forks[5]`

```
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)

이런 경우에는 deadlock이 발생가능 5명이 모두 왼쪽 포크를 얻은 상황에서 모두가 오른쪽 포크를 얻으려고 한다면 누구도 얻을 수 없을 것

  - Deadlock occur!
    - If each philosopher happens to **grab the fork on their left** before any philosopher can grab the fork on their right
    - Each will be stuck *holding one fork* and waiting for another, *forever*

A Solution: Breaking The Dependency

- Change how forks are acquired
  - Let's assume that philosopher 4 acquire the forks in a *different order*

```
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    }
```

간단하게 4명이 왼쪽 포크를 얻었다면 마지막 한명은 오른쪽 포크부터 얻도록 순서를 약간 조정하면 deadlock문제를 해결 가능

➢ There is no situation where each philosopher grabs one fork and is stuck waiting for another. **The cycle of waiting is broken**