[Chapter 7,8]

경영학과/2018100233/김정민

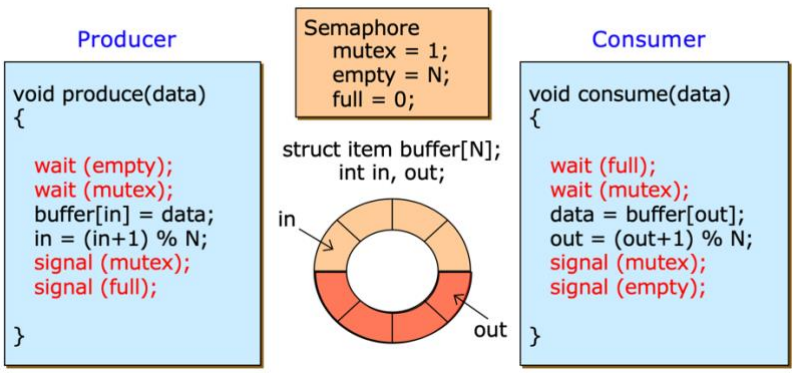
I. Classic Problems of Synchronization

A. Bounded Buffer Problem

1. 별다른 동기화 없는 경우
   - Count(공유 자원)에 접근했다가, 업데이트하기 전에 컨텍스트 스위치가 일어나면 문제가 발생한다. (이전 챕터)
   - 공유 자원의 값이 정상적인 값을 갖지 못할 수도 있다.

2. Binary Semaphores 활용

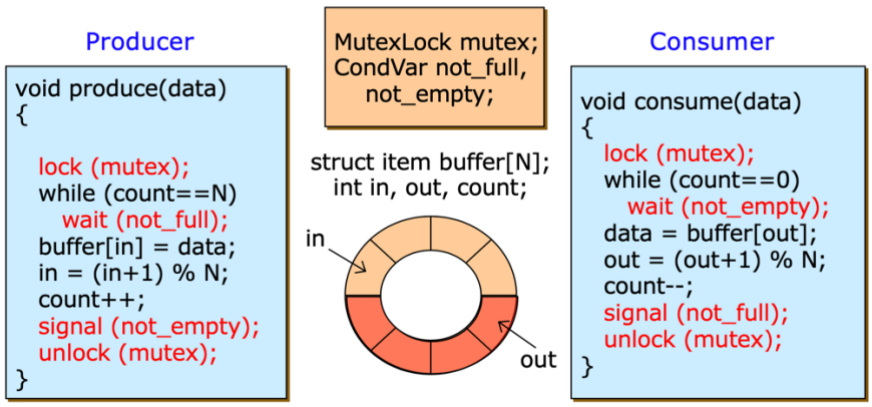
   ■ Implementation with semaphores



   - Count값 대신 mutex를 활용해서, 비정상적인 count값 때문에 발생되는 혼란 방지
   - 앞뒤로 wait()과 signal()을 활용해서 empty와 full 상태를 감지 -> overflow / underflow 방지 (시간적 동기화)

3. Mutex lock과 conditional variables 활용

   ■ Implementation with mutex lock and condition variables

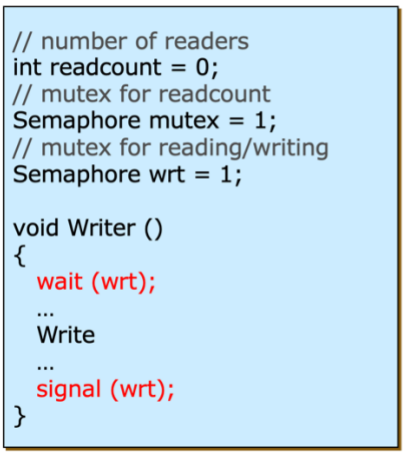


   - Condition variables -> not_full, not_empty
   - lock(mutex) : 티켓이 있는지 확인 후 있으면 입장(없으면 대기) 및 티켓 소진
   - wait(not_full) : count가 N인 경우 (버퍼가 꽉 차있는 경우)에는 하나라도 비어있을 때까지 대기한다.
   - signal(not_empty) : 하나라도 버퍼에 들어가있다고 알림
   - unlock(mutex) : 티켓 하나를 반납한다.
   - 시스템 콜 및 데이터를 복사하는 과정에서 커널 오버헤드 증가

B. Readers-Writers Problem

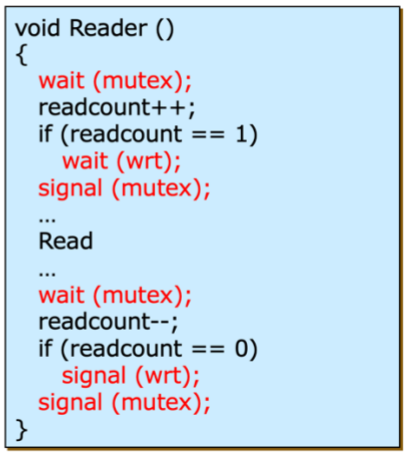
1. Readers
   - 동시에 여러 프로세스 접근해도 된다.

2. Writers
   - 한 번에 하나만 접근해야 한다. (Mutual exclusion)
3. Reader는 다른 reader의 작업이 끝날 때까지 기다릴 필요가 없다.
4. Writer가 작업을 수행할 땐, 공유 혹은 방해 없이 그 작업을 빠르게 수행한다.
5. Semaphore를 활용한 구현
   - Readcount : 데이터를 읽는 쓰레드의 개수
   - Mutex : readcount에 접근할 수 있는 티켓 개념
   - Wrt : reading 혹은 writing의 대한 권한(티켓) 개념
     - Implementation with semaphores



```
// number of readers
int readcount = 0;
// mutex for readcount
Semaphore mutex = 1;
// mutex for reading/writing
Semaphore wrt = 1;

void Writer ()
{
   wait (wrt);
   ...
   Write
   ...
   signal (wrt);
}
```



```
void Reader ()
{
   wait (mutex);
   readcount++;
   if (readcount == 1)
      wait (wrt);
   signal (mutex);
   ...
   Read
   ...
   wait (mutex);
   readcount--;
   if (readcount == 0)
      signal (wrt);
   signal (mutex);
}
```

   -
   - Writer()에서는 mutex에 대한 wait, signal이 필요없다.
   - Reader()에서는 mutex에 대한 동기화도 필요하다.
   - wait(mutex) : mutex(티켓)가 있으면 크리티컬 섹션에 진입한다.
     - readcount++ : 읽고있는 프로세스 수 증가
   - if(readcount == 1) : 방금 크리티컬 섹션에 진입한 프로세스가 처음으로 읽기 시작한 프로세스라면
   - wait(wrt) : 쓰기 작업 못하도록 wrt 티켓을 가져온다.
   - signal(mutex) : readcount라는 크리티컬 섹션에 대해서 처리가 완료됐기 때문에 mutex 반납한다.
   - readcount가 2,3,4...이라면?
     - 이미 1일 때 쓰기작업 못들어오게 보장해줬으니 아무 작업 없어도 된다.
   - 다 읽고 나서 wait(mutex) : readcount 감소를 위한 mutex 획득
   - if (readcount == 0) : 더 이상 읽고있는 프로세스가 없다면
   - signal(wrt) : 읽거나 쓸 수 있는 티켓 반납
   - signal(mutex) : readcount 동기화 끝났으니 mutex 반납
6. 정리
   - Writer가 있다면?
     - 첫 번째 reader는 wrt를 획득하기 위해 대기한다. wait(wrt)
     - 다른 reader들은 mutex조차 획득하지 못하고 대기한다. wait(mutex)
   - Writer가 퇴장하면, 모든 reader들은 크리티컬 섹션에 진입 가능하다.
     - Writer가 wrt, mutex 모두 반환
     - 첫번째 reader는 wrt 획득하고 mutex 반환 -> 이후 모든 reader들은 mutex 획득 후 반환 반복 -> 동시에 여러 reader 동작 가능
   - 마지막 reader가 wrt, mutex를 반환하고 나오면? (writer가 wrt 획득을 위해 기다리는 상황)
     - 바로 writer는 wrt 획득 후 writing 진행
   - Writer가 할 일을 마치고 wrt를 반환했을 때, Writer와 reader가 동시에 기다리고 있는 상황에서는 스케

줄러에 따라서 wrt를 획득한다.

C. Dining Philosopher Problem (생각하는 철학자)

1. 개요

- Dijkstra, 1965
- 반복과정
  - Thinking
  - Getting hungry
  - Getting two chopsticks
  - Eating

2. 간단한 솔루션 (deadlock 유발 가능)

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
   while (1)  {
      think ();
      wait (chopstick[i]);
      wait (chopstick[(i+1) % N];
      eat ();
      signal (chopstick[i]);
      signal (chopstick[(i+1) % N];
   }
}
```

⇒ Problem: causes deadlock

-
- 생각하다가 배가 고파지면 자신의 왼쪽과 오른쪽의 젓가락을 짚는다. (semaphore 획득)
- 먹고나서 반환한다.
- 만약에 동시에 모든 철학자들이 wait(chopstick[i])를 수행하면, wait(chopstick[(i+1)%N]을 수행할 수 없이 모두가 교착상태가 된다.

3. Deadlock free version

- Deadlock-free version: starvation?

```
#define N        5
#define L(i)     ((i+N-1)%N)
#define R(i)     ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
  }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

-
- 철학자는 생각, 픽업, 먹기, 내려놓기의 과정을 반복한다.
- pickup(i)
  - wait(mutex) : 크리티컬 섹션에 진입하기 위해 mutex 획득 대기
    - binary mutex이므로 티켓을 획득하면 다른 철학자들은 상태를 변경할 수 없다.
  - state[i] = HUNGRY : state라는 공유 자원을 HUNGRY로 바꾼다.
  - test(i)
    - 본인이 배고프고, 양쪽에 있는 사람이 먹고있는 상태(양쪽 젓가락을 점유)가 아니라면
    - 본인의 상태를 먹고 있는 것으로(양쪽 젓가락을 점유) 바꾼다.
    - signal(s[i]) : 본인의 state를 바꿀 수 있는 semaphore를 반납한다.(하나 생성한다)

- signal(mutex) : 크리티컬 섹션에서 빠져나올 때 mutex를 반납한다.
- wait(s[i]) : test(i)에서 조건을 만족하지 못했다면, signal(s[i])가 실행되지 않아서 semaphore를 획득할 수 없다. 획득할 때까지 대기한다.

- Putdown(i)
  - wait(mutex) : 철학자의의 상태(크리티컬 섹션)을 업데이트하기 위해 mutex 획득 대기
  - state[i] = THINKING
    - 본인의 상태를 EATING에서 THINKING으로 바꿔준다.
  - test(L(i)), test(R(i)) : 좌우 사람의 기준에서는 본인이 각각 우, 좌에 위치한 사람이다.
    - 좌우 사람들에 대해서 각각 우, 좌에 있는 사람(본인)의 상태가 EATING이 아니기 때문에 젓가락을 집을 수 있는 상태가 될 수 있다.
  - signal(mutex) : 크리티컬 섹션을 빠져나오면서 mutex 반환한다.

4. Monitor implementation

```
monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i);
    void putdown(int i);
    void test(int i);
    void init() {
       for (int i = 0; i < 5; i++)
       state[i] = EATING;
    }
}
```

-
```
void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        self[i].wait();
}
```

```
void putdown(int i) {
    state[i] = THINKING;
    // test left and right
    test((i+4) % 5);
    test((i+1) % 5);
}
```

```
void test(int i) {
    if ( (state[(i + 4) % 5] != EATING) &&
      (state[i] == HUNGRY) &&
      (state[(i + 1) % 5] != EATING)) {
        state[i] = EATING;
        self[i].signal();
    }
}
```

-
- 프로그래밍 언어를 이용해서 함수를 통해 공유 자원(state 배열)에 접근한다.
- mutex를 사용하지 않는다.
- 이외에는 같은 원리로 작동한다.

II. Synchronization Tools in Real World

A. POSIX synchronization (라이브러리로 구현)

1. POSIX Semaphores

- #include <semaphore.h>
- int sem_init(sem_t * sem, int pshared, unsigned int value);
- int sem_wait(sem_t *sem);
- int sem_trywait(sem_t *sem);

- int sem_post(sem_t *sem);
- int sem_getvalue(sem_t *sem, int *sval);
- int sem_destroy(sem_t *sem);
- 성공적으로 수행되면 0 반환, 0이 아닌 값은 에러를 나타낸다.

2. POSIX Mutex Locks
   - #include <pthread.h>
   - int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mattr);
   - int pthread_mutex_destroy(pthread_mutex_t *mutex);
   - int pthread_mutex_lock(pthread_mutex_t *mutex);
   - int pthread_mutex_trylock(pthread_mutex_t *mutex);
   - int pthread_mutex_unlock(pthread_mutex_t *mutex);
   - 성공적으로 수행되면 0 반환, 0이 아닌 값은 에러를 나타낸다.
3. POSIX Condition Variables
   - #include <pthread.h>
   - int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cattr);
   - int pthread_cond_destroy(pthread_cond_t *cond);
   - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
   - int pthread_cond_signal(pthread_cond_t *cond);
   - int pthread_cond_broadcast(pthread_cond_t *cond);
   - 성공적으로 수행되면 0 반환, 0이 아닌 값은 에러를 나타낸다.

B. Semaphores, Mutexes & Condition variables (C, C++/ JAVA)

1. Bounded Buffer Problem with POSIX semaphores

```
sem_t mutex, full, empty;
/*   sem_init (&mutex, 0, 1);
     sem_init (&full, 0, 0); sem_init (&empty, 0, N);  */
buffer resources[N];

void producer (resource x)  {
   sem_wait (&empty);
   sem_wait (&mutex);
   add "x" to array "resources";
   sem_post (&mutex);
   sem_post (&full);
}
void consumer (resource *x)  {
   sem_wait (&full);
   sem_wait (&mutex);
   *x = get resource from array "resources"
   sem_post (&mutex);
   sem_post (&empty);
}
```

- wait() -> sem_wait()
- signal() -> sem_post()
- sem_init() -> 공유자원 변수 생성

2. Bounded Buffer Problem with POSIX Mutexes & Conditional Vars

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void producer (resource x) {
   pthread_mutex_lock (&mutex);
   while (array "resources" is full)
      pthread_cond_wait (&not_full, &mutex);
   add "x" to array "resources";
   pthread_cond_signal (&not_empty);
   pthread_mutex_unlock (&mutex);
}
void consumer (resource *x) {
   pthread_mutex_lock (&mutex);
   while (array "resources" is empty)
      pthread_cond_wait (&not_empty, &mutex);
   *x = get resource from array "resources"
   pthread_cond_signal (&not_full);
   pthread_mutex_unlock (&mutex);
}
```

-
- lock(mutex) -> pthread_mutex_lock(&mutex)
- wait(not_full) -> pthread_cond_wait(&not_full, &mutex)
- signal(not_empty) -> pthread_cond_signal(&not_empty)
- unlock(mutex) -> pthread_mutex_unlock(&mutex)

C. Monitor (JAVA)

1. Bounded Buffer Problem with JAVA Monitor

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

-
- 무슨 방법을 쓰든 synchronization을 고려한다. 그리고 나아가서 deadlock까지도 고려해야 한다.

III. Deadlock Problems

A. Deadlock이란?

1. 복수의 프로세스가 서로 티켓을 획득하지 못하고 waiting 상태에 빠져있는 상황 (교착 상태)
2. 예시 코드

Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

-
- 두번째 코드인 wait(Q), wait(S)에서 서로 티켓을 보유한 채로 놓아주지 않기 때문에, 하염없이 기다리는

문제가 발생한다.

- 동기화 툴 때문에 데드락이 발생하는 것

B. Deadlock Characterization (4가지 특성)

1. 아래 4가지 특성을 동시에 발생할 때 데드락이 발생한다. (하나만 해결해도 데드락 회피 가능)
2. Mutual exclusion
   - 동시에 오직 하나의 프로세스만 공유자원에 접근할 수 있다.
3. Hold and wait
   - 적어도 하나 이상의 공유 자원을 점유 중인 프로세스가 다른 프로세스에 의해 점유 중인 공유 자원에 접근하기 위해 대기하고 있을 때
4. No preemption
   - 자발적으로 공유 자원에 대한 점유를 해제하지 않을 때
   - 프로세스가 자발적으로 점유를 해제할 때에만 공유 자원에 대한 접근이 가능해진다.
5. Circular wait
   - 꼬리에 꼬리를 무는 waiting 관계에 있을 때
   - P0 -> P1이 점유 중인 자원을 기다림
   - P1 -> P2가 점유 중인 자원을 기다림 ...

C. Handling Deadlocks

1. Deadlock prevention
   - 데드락은 사람의 실수에 의해 유발될 수 있다.
   - 개발자가 예방하는 것이 중요하다.
   - 적어도 하나 이상의 데드락 발생 요건을 방지하게끔 해야 한다.
2. Deadlock avoidance
   - OS가 런타임에 계속해서 모니터링한다. (소극적인 개입)
   - 데드락 발생할 것 같으면 못하게 막는다. (사전처리 개념)
   - 이를 위해서는 공유 자원이 어떻게 요구되는지에 대한 추가적인 정보가 필요하다.
   - 결국 커널 오버헤드가 증가한다.
3. Deadlock detection and recovery
   - 데드락 상태에 진입하면, 그 후에 OS가 고친다. (사후처리 개념, 적극적인 개입)
4. Deadlock ignorance
   - 문제를 싹 다 무시한 채로 진행한다.
   - The Ostrich algorithm

D. Dining Philosopher 코드로 보는 데드락

1. 데드락 유발 simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
   while (1)  {
      think ();
      wait (chopstick[i]);
      wait (chopstick[(i+1) % N];
      eat ();
      signal (chopstick[i]);
      signal (chopstick[(i+1) % N];
   }
}
```

-

- wait(chopstick[i]), wait(chopstick[(i+1) % N])
  - Mutual exclusion : 하나의 공유자원에는 동시에 하나만 접근 가능
  - hold-and-wait : 좌우 사람 중 한 명이라도 EATING 상태라면, 다른 철학자는 젓가락을 위해 대기해야 한다.
  - No preemption : signal()을 통해서만 젓가락이라는 공유자원을 내려놓는다.
- 모든 철학자가 동시에 자신의 왼쪽 젓가락을 집으면?
  - Circular wait

2. Deadlock free version

```
#define N        5
#define L(i)     ((i+N-1)%N)
#define R(i)     ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
  }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

-

- 젓가락을 공유자원으로 두는 것이 아니라, 철학자의 상태를 공유자원으로 활용한다.
  - hold-and-wait가 사라진다.
  - 따라서 데드락이 발생하지 않는다.