[Chapter 6]

경영학과/2018100233/김정민

I. Synchronization

A. Multithreaded programs에서는 쓰레드들이 "협업"한다.

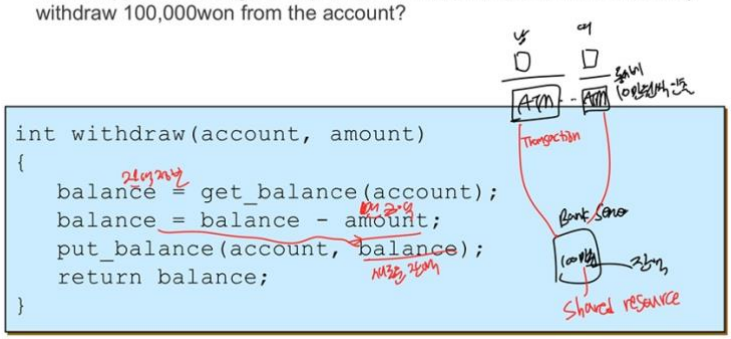
1. 여러 쓰레드가 자원을 공유하기 위해 shared data structures에 접근한다.
   - Shared memory를 사용하게 되면, OS단에서 관리가 어렵다. 따라서 개발자들이 개발 과정에서 미리 정한 룰을 따라야 한다.
2. 하나의 job을 수행하기 위해 job을 쪼개서 여러 쓰레드가 협업하는 경우도 있다. (시간적 동기화)
   - IPC(shared memory, message queue)

B. 쓰레드 간 협업은 공유된 자원의 정확성을 보장하기 위해 관리가 필요하다.

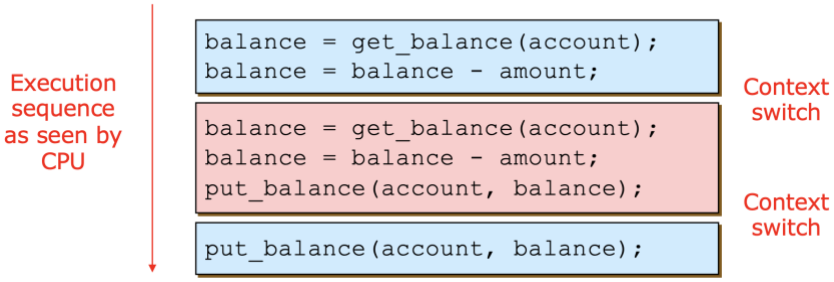
1. 쓰레드들의 스케줄링은 프로그램 작성한 사람의 통제 영역 밖이다.
   - 쓰레드들이 어떤 순서로 일을 처리하는지에 따라서 결과가 달라질 수 있다.
2. Synchronization을 통해서 관리한다.
   - 실행 과정에서 스레드들이 임의로 교차되는 것을 제한한다.
3. 쓰레드뿐만 아니라, 프로세스에도 적용되는 개념이다.
   - 또한 분산 시스템에서 각각의 컴퓨터들끼리의 협업에도 적용될 수 있다.

C. Synchronization이 필요한 예시1

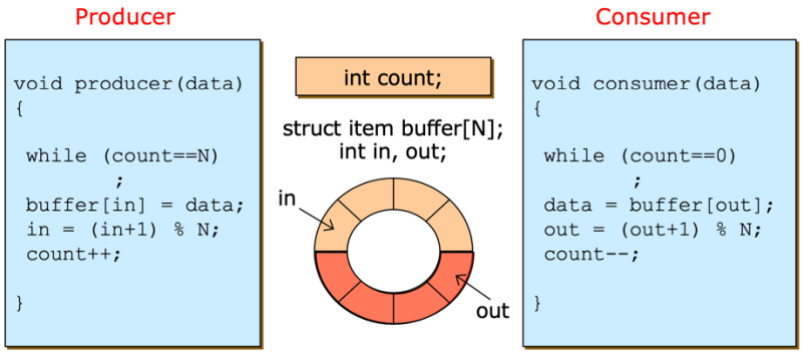
- ✓ Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won
- ✓ What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?



1. 100만원이 잔액으로 있는 계좌(shared resource)가 있다.
2. 남녀가 각자 다른 ATM 기계에서 10만원씩 동시에 인출한다.
   - 잔액은 80만원이 남아야 한다.
3. 하지만 put_balance() 함수를 수행하기 전에 context switch가 발생하면, 새로운 프로세스(다른 ATM기기)에서의 초기 잔액은 다시 100만원이고, 10만원을 인출하면 put_balance()를 통해 잔액을 90만원으로 변경한다.
   - 다시 context switch가 발생해서 기존 프로세스로 돌아와서 90만원인 balance 변수를 put_balance()의 파라미터로 넣어서 호출하면, 잔액은 90만원이 된다. (동기화 문제)



D. Synchronization이 필요한 예시2 (Bounded buffer)



1. Producer는 버퍼에 넣기만 하고, Consumer는 버퍼에서 데이터를 빼기만 한다.
2. Shared data인 count가 critical section이다.
   - 예시 1과 같이 count++, count—를 수행하기 전에 context switch가 발생하면 count값이 제대로 된 값을 갖지 못한다.
   - Count가 5인 상태에서 하나씩 넣고 빼면, count는 5가 되어야 하는데, 4나 6이 될 수도 있다.

E. Synchronization Problem

1. 복수의 쓰레드 혹은 프로세스가 동기화 장치 없이 공유된 자원에 접근하는 경우
   - Race condition
   - 동기화 없다고 무조건 해당 문제가 생기는 것은 아니다. (non-deterministic)
2. 여러 쓰레드가 동시에 공유 자원에 접근하는 것을 관리할 필요가 있다.
3. 어느 공유 자원에나 동기화는 필요하다.
   - Shared data structure (buffers, queues, lists 등)
   - Critical section problem (동기화 문제가 생길 수 있는 코드 섹션에 대한 동기화 필요)

II. Synchronization Tools

A. Requirements for Synchronization Tools (3가지 조건)

1. Mutual Exclusion
   - 상호배제
   - 한 쓰레드가 critical section에 진입해서 수행하고 있는 경우, 다른 어떠한 쓰레드들도 critical section에 있는 코드를 수행할 수 없다.
2. Progress
   - Critical section의 코드를 수행하고 있는 프로세스가 없고, 수행을 원하는 프로세스가 있다면, 해당 프로세스는 바로 critical section에 진입해서 코드를 수행할 수 있다. (무한정 기다리지 않는다.)
3. Bounded Waiting
   - 프로세스는 공유 자원에 접근할 때 무한정 대기하는 것이 아니다.
   - 한정된 순번으로 공유 자원에 접근할 수 있어야 한다.

B. Synchronization Tools

1. Locks (low level mechanism)
   - OS에서 lock/unlock 시스템 콜 제공된다.
     - lock() : lock이 풀릴 때까지 다른 쓰레드는 접근 못하도록 기다리게 한다.
     - unlock() : lock()에서 대기 중인 쓰레드가 들어올 수 있다.
   - 크리티컬 섹션에 진입하기 전에 lock()을 호출하고, 크리티컬 섹션을 빠져나올 때 unlock()을 호출
   - 최대 한 개의 쓰레드만이 동시에 lock을 유지할 수 있다.
   - 종류
     - spin (spinlock : low level mechanism)

- block (mutex : high level mechanism)

- 초기 lock() 코드 (spinlocks)

```
struct lock { int held = 0; }

void lock(struct lock *l)   {
   while (l->held)
       ;
   l->held = 1;
}
void unlock(struct lock *l)    {
   l->held = 0;
}
```

The caller "busy-waits", or spins for locks to be released, hence spinlocks

  - While문의 held를 통해 대기할지 결정한다.
  - 여러 프로세스가 held에 동시에 접근할 수 있다. (busy-waits, spinlocks)
  - 하지만 이 코드는 정상적으로 동작하지 않는다.
  - lock() 코드 자체에도 critical section이 존재하기 때문이다. (held가 공유자원)

- Spinlocks의 단점 (시스템 낭비가 크다.)
  - Busy-waiting -> 쓰레드는 lock에서 대기 중일 때 다른 코드를 수행할 수 없다.
  - 크리티컬 섹션이 길수록 더 많이 기다린다.
  - Lock holder가 방해받을 확률이 높다.

- **Spinlocks의 해결법**
  - Software-only algorithms
    - Algorithm 1,2,3 (2개의 프로세스용)
    - Bakery algorithm (복수의 프로세스용)
    - 소프트웨어를 이용한 해결법은 운영체제 오버헤드가 증가하고 시스템 성능이 낮아진다.
    - 또한 알고리즘은 CPU 자원을 많이 써서 오버헤드가 높기 때문에 사용하지 않는다.
  - Hardware atomic operation (Primitive lock for OS in multi-processors)
    - Spinlock임에도 불구하고 인터럽트를 직접적으로 건드리는 것보다는 안정적이다.
    - OS 안에서만 사용한다.
    - 공유자원(held) 접근하는 코드를 하나의 인스트럭션으로 해결하면 중간에 context switch가 발생하지 않는다.
      - Test-and-Set

```
struct lock { int held = 0; }

void lock(struct lock *l)   {
   while (TestAndSet(l->held))
       ;
}

void unlock(struct lock *l)    {
   l->held = 0;
}
```

    -

- Compare-and-Swap

```
struct lock { int held = 0; }


void lock(struct lock *l)  {
   key = true;
   while (CompareAndSwap(l->held, key))
        ;
}

void unlock(struct lock *l)   {
   l->held = 0;
}
```

  - ◆
- Disable/re-enable interrupts (Primitive lock for OS in single processor)
  - ◆ Context switch를 발생시키는 인터럽트를 무효화시킨다.
  - ◆ 이 방법은 개발자가 인터럽트를 임의로 건드릴 수 있는 것이기에 뒤죽박죽이 될 수도 있다.
  - ◆ 운영체제(커널) 코드에서만 사용한다.
  - ◆ 멀티프로세서 환경에서는 유용하지 않다.
  - ◆ Critical section이 길면 중요한 이벤트의 인터럽트를 놓칠 수 있다.
    - 무시된 인터럽트들을 나중에 다시 복구할 방법이 없다.
  - ◆ 신중하게 사용해야 한다.

2. high level mechanism의 필요성
- Spinlocks와 인터럽트를 무시하는 방법은 critical section이 굉장히 짧을 때만 효용성이 있다.
  - "primitive" -> mutual exclusion 이외에 할 수 있는 게 없다. (low level)
- Higher-level synchronization이 필요하다.
  - Block waiters
  - 인터럽트를 무시하지 않고 사용 가능해야 한다.
- 목표
  - Shared resource 보호
  - 2개의 프로세스가 시간을 딱딱 맞춰서 사용해야 될 때까지도 동기화가 가능해야 한다.
- 2가지 방법 (high level로 구현, lock()은 OS primitive한 방법)
  - Semaphores (binary(mutex), counting)
  - Monitors (mutexes and condition variables)

3. (Counting) Semaphores
- 카운터 기반 동기화
  - 카운터 : 복수의 프로세스가 접근 가능한 정수 변수 (크리티컬 섹션에 진입할 수 있는 티켓 개념)
- Operations 종류
  - Wait or P (lock)
  - Signal or V (unlock)
- 절차
  - 우선 공유자원에 접근할 수 있는 semaphore(티켓)을 확인한다.
  - Semaphore의 값이 양수라면, 프로세스는 공유자원에 접근할 수 있다.
    - ◆ 접근하게 되면 semaphore의 값은 1씩 감소한다.
  - Semaphore의 값이 0이면, 프로세스는 값이 1 이상이 될 때까지 기다린다. (sleep)
    - ◆ Semaphore 값이 양수가 되면 프로세스는 처음 과정부터 수행한다. (wake up)
- wait(mutex) 및 signal(mutex) 사용법
  - wait(mutex) : 티켓이 생길 때까지 기다린다.
  - signal(mutex) : 할 일 끝났으니 티켓 하나 반납한다.

```
Semaphore mutex = 1;

int withdraw(account, amount)
{
   wait(mutex);
   balance = get_balance(account);
   balance = balance - amount;
   put_balance(account, balance);
   signal(mutex);
   return balance;
}
```

■

- 두 프로세스 간 wait(mutex), signal(mutex) 활용법 (시간적 동기화)

■ General synchronization using semaphores

✓ Execute *B* in P_j only after *A* executed in P_i

✓ Use semaphore *flag* initialized to 0

✓ Code:

```
Semaphore flag = 0;

Pi                  Pj
:                   :
A                   wait(flag);
signal(flag);       B
```

■

■ Pi에서 A가 실행되고 있을 때 Pj에서 B를 실행하고 싶으면 wait(flag)를 통해 flag의 값이 양수가 될 때까지 B의 실행을 지연시킨다.

■ 그러다가 A의 수행이 끝나고 signal(flag)를 통해 flag의 값이 1이 되면, B가 수행을 시작한다.

- 구현

■ Block : 크리티컬 섹션에 접근하고 싶은 프로세스들을 지연시킨다. (웨이팅 큐 대기)

■ Wakeup(P) : block된 프로세스를 일깨워서 웨이팅 큐에서 레디 큐로 보낸다.

- No busy waiting

```
wait(semaphore *S) {
        lock(lock); S->value--; unlock(lock);
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
signal(semaphore *S) {
        lock(lock); S->value++; unlock(lock);
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

■

■ Wait의 경우

◆ 티켓(S->value) 자체가 공유자원이다. 따라서 업데이트하는 부분 앞뒤로 lock, unlock 과정이 필요하다.

◆ 이후 티켓의 값이 0보다 작다면, 들어오려는 프로세스를 웨이팅 큐에 넣고, block()을 때린다.

■ Signal의 경우

- ◆ 마찬가지로 티켓 값 업데이트하는 부분 공유자원이다.
- ◆ 티켓 값을 하나 올렸는데도 티켓 값이 0보다 작거나 같으면, 들어오고 싶은 프로세스(P)가 기다리고 있다는 것이다.
- ◆ P를 웨이팅 큐에서 제거하고 wakeup()을 통해 레디 큐로 보내준다.

4. Mutex locks (Binary semaphores)

- Semaphore의 mutex값이 binary(1 혹은 0)인 방법이다.

```
MutexLock mutex;

int withdraw(account, amount)
{
   lock(mutex);
   balance = get_balance(account);
   balance = balance - amount;
   put_balance(account, balance);
   unlock (mutex);
   return balance;
}
```

- ■
- 티켓이 1장뿐이므로, 크리티컬 섹션에 진입할 수 있는 프로세스는 하나뿐이다.
- 구현하기가 counting semaphore에 비해 쉽다.
- I/O-burst 관점에서 최소화 good

5. Monitors

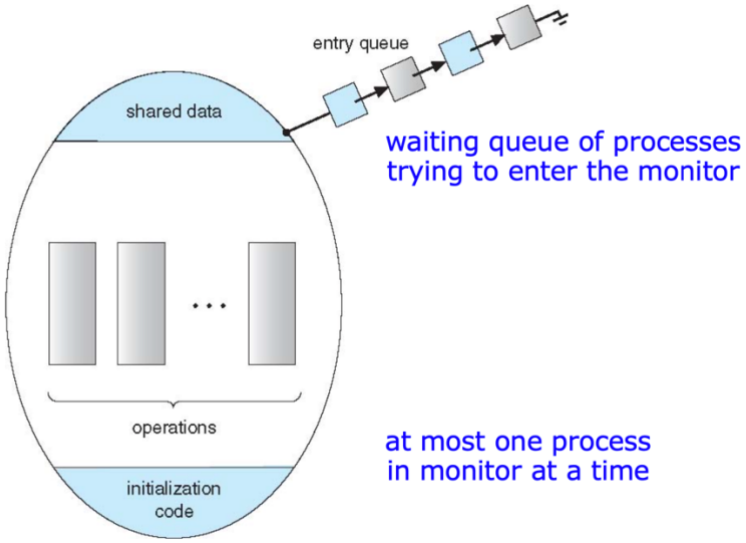
- JAVA에서 활용하는 개념
- C, C++은 지원하지 않기 때문에 semaphore 활용
- 프로그래밍 언어 구조를 통해 공유 자원에 대한 접근을 통제한다.
- 컴파일러에 동기화 코드가 추가되고 런타임에 실행된다.
- 프로세스 간에 추상적인 데이터 타입의 안전한 공유를 가능하게 해준다.
- 소프트웨어 모듈 (아래 요소들을 추상화)
  - ■ 공유자원 구조
  - ■ 공유자원에 대해서 작동하는 함수
  - ■ 함수들을 호출하는 프로세스들 간의 동기화
- 비정형적인 접근으로부터 데이터를 보호한다.
  - ■ 오직 함수 콜을 통해서만 데이터에 접근할 수 있다.
- 모니터의 구조

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (…) {
      . . .
    }
    procedure body P2 (…) {
      . . .
    }
    procedure body Pn (…) {
       . . .
    }
    {
      initialization code
    }
}
```

- ■
- ■ 동일 시간에 함수를 하나만 호출할 수 있는 구조다.

- 여러 함수들 정의된다.

- 프로세스들이 모니터를 통해 공유 자원에 접근하는 추상적인 모습



-
- 모니터에 접근하기 위해 대기하는 웨이팅 큐 존재
- 한번에 하나씩만 접근 가능

- Conditional variables (모니터에 적용할 수 있는 개념)
  - Rendezvous point
  - 모니터 웨이팅 큐에서 대기하기 위해 조건 변수들이 제공된다.
  - condition x, y;
    - shared data 내에서 x와 y에 대한 레디 큐가 따로 존재한다는 차이점
  - x.wait()
    - 다른 프로세스가 signal()을 보내기 전까지 대기한다.
  - x.signal()
    - 대기하고 있는 프로세스 하나를 모니터에 들어올 수 있게끔 한다.
  - 기다리고 있는 프로세스가 없다면 signal()은 아무 효과가 없다.

6. Monitors와 Semaphores의 비교

- Condition variables는 history에 민감하지 않다.
  - 아무 프로세스가 기다리고 있지 않으면 signal()을 호출해도, 아무런 변화가 일어나지 않는다.
- Semaphores는 history에 민감하다.
  - 아무 프로세스가 기다리고 있지 않을 때 signal()을 호출하면, semaphore(티켓)의 값이 증가한다.

C. semaphore의 문제점

1. Deadlock

- 복수의 프로세스가 무한루프를 돌면서 티켓을 획득하진 못하고, 기다리기만 하는 상태를 말한다. (교착상태)
- 서로가 상대방이 가지고 있는 티켓에 대한 wait()을 동시에 걸면, 이런 상태가 발생할 수 있다.

2. Starvation (indefinite blocking)

- 대기하고 있는 큐 안에 프로세스가 계속해서 티켓을 획득하지 못하고 큐에 남아있는 것을 말한다.

3. 결점 (허점)

- 전역변수의 공유를 활용한다. (소프트웨어 엔지니어링 관점에서 좋지않다.)
  - 어디서나 접근 가능
- Semaphore와 데이터 사이의 관련이 하나도 없다. (그저 안에서 무슨 일이 일어나는지 모르고 티켓 접수만 받는 느낌)

- Critical sections (mutual exclusion)과 협업 (scheduling) 둘 다 사용된다.
- 세마포어 사용에 대한 통제가 없기 때문에, 잘 쓰고있는지 확신할 수 없다.

4. 사용하기 어렵고 버그에 약하다. (이건 그냥 사람(개발자) 탓이다.)
   - Signal, wait 등의 오퍼레이션이 혼란을 준다고하지만, 정확히 사용하면 문제가 없다.
   - 사람 탓이 크다.
   - 새로운 접근 (프로그래밍 언어의 지원을 받는 방법 -> 시스템 콜 사용하기 어려우니 !)
     - Critical region
     - Monitor