동기화 1

동기화란

공유 데이터에 대한 동시 접근은 데이터의 일관성(consistency)를 해치는 결과를 낳을 수 있다.

Race Condition

공유 데이터에 대해 여러 프로세스가 동시에 접근, 변경을 시도하는 상황으로 일관성이 보 장되지 않는 상황

→ 두 개의 상황이 경주를 하는 것이다.

경주는 결과를 알 수 없다.

→ 컴퓨터가 이렇게 동작하는 것은 바람직하지 않다.

→ 일관성 보장이 안 된다.

데이터 일관성을 유지하기 위해서는 수행하는 프로세스들이 순차적으로 데이터를 접근하게 만드는 기법이 필요하다

이것을 동기화(synchronization)라고 함

→ 공유된 데이터는 순차적으로 데이터에 접근하게 한다. (예외가 없다.)

예제

은행의 입출금 문제

1000원의 잔고가 남아있을 때, 500원의 입금과 500원의 출금이 동시에 일어날 경우, 그 결과는?

Balance 가 공유되고 concurrent 하게 수행하는 경우

인터럽트로 인하여 execution interleaving:

순서가 왓다갔다 한다.

최종적으로 500의 잔고가 남게 된다.

Execution 순서에 따라 1500이 될 수도 있다. 즉, 결과를 예측할 수 없다는 문제가 생긴다

→ race condition이 존재하는 상황이다.

→ 바람직하지 않은 상황이다.

→ balance에 대한 consistency가 보장이 안 됨

순서를 달리하면 결과가 달라진다.

지금까지 배운 내용 중에서

Thread programming

→ Data section is shared

IPC : Shared memory

→ Multiple processes can access

커널

→ Data structures shared

공유가 많이 된다. → 동기화가 많이 필요하다.

Critical section

동기화 문제를 Critical Section으로 모델링

Critical section 이란

여러 프로세스들이 **공유하는 데이터에 접근하는 코드 영역**을 Critical section이라고 한다.

한 번에 **오직 하나의 프로세스만이** critical section에 진입해야 함

예: 은행 입출금 문제의 Balance 증가•감소 문장

→ Balance + 500;

Critical Section을 사용하여 모델링하면:

entry section //여러 프로세스가 들어가려고 할 때, 하나만 들어가게 한다.

critical section //공유 데이터 접근해서 처리

exit section //끝났다고 표시를 해준다.

remainder section //다른 녀석이 들어올 수 있게 해준다.

Critical Section 해결조건들

Critical Section 문제를 해결하는 알고리즘은 아래와 같은 세 가지 조건을 만족해야 함

Mutual Exclusion (상호 배제) → Mutex라는 말을 자주 사용한다.

만약 프로세스 A가 critical section에 진입해 있다면, 다른 모든 프로세스는 진입할 수 없어야 함

Progress

critical section에 진입하려는 프로세스가 존재한다면, 그중에서 한 프로세스는 critical section에 진입할 수 있어야 함

Bounded Waiting

어떤 프로세스가 Critical section에 진입할 때까지 걸리는 시간에 “limit”이 존재하여야 함

→ starvation이 없어야 한다.

→ 들어가는데 걸리는 시간을 명확히 제시해야함

두 프로세스를 위한 알고리즘

- Shared variables:

  int turn = 0; // 초기화

  // turn = 0일 때, P0가 critical section에 진입 가능

```
while (turn != 0) ; //waiting
                    //entry section
  critical section
turn = 1;           //exit xection
  remainder section
```

```
while (turn != 1) ; //waiting
                    //entry section
  critical section
turn = 0;           //exit xection
  remainder section
```

MUTEX는 지켜진다

- 불만족 조건 : progress, bounded waiting

  두 프로세스의 수행 순서가 alternate 안되면 진행 안 됨

  P0, P0, P0이면, 두 번째 P0부터 멈추게 됨

  (P1이 한번 들어가줘야 turn을 바꿔서 수행이 가능해짐

  → bounded waiting이 안지켜짐)

  그리고 이 상황은 최소 하나의 프로세스가 들어오지 못하는 상황임

  임의의 순서의 수행이 불가능하다.

동기화는 항상 시험 문제에 나오는데, 초기값을 쓰지 않으면 감점이다.

→ 초기값에 따라 작용이 달라진다.

개선된 알고리즘 – 두개의 flag 이용

- Shared variables:

  boolean flag[2]; // flag [0] = flag [1] = false 로 초기화

  // flag [0] = true일 때, P0 이 critical section에 진입

  // flag [1] = true일 때, P1 이

```
flag[0] = true;
while (flag[1]) ; //waiting
  critical section
flag[0] = false;
remainder section
```

```
flag[1] = true;
while (flag[0]) ; //waiting
  critical section
flag[1] = false;
remainder section
```

- 만족 조건 : mutual exclusion
- 불만족 조건 : progress, bounded waiting
  - 두 프로세스가 동시에 flag[] 를 true 로 할 수 있음

→ 둘 다 기다리는 것이다.

Peterson solution

- Shared variables

  int turn;

  boolean flag[2]; // flag[0] = flag[1] = false 로 초기화

```
flag [0] = true;
turn = 1;
while (flag [1] && (turn == 1));
  critical section
flag [0] = false;
  remainder section
```

```
flag [1] = true;
turn = 0;
while (flag [0] && (turn == 0));
  critical section
flag [1] = false;
  remainder section
```

- 만족 조건 : mutual exclusion, progress, bounded waiting
  - 두 프로세스가 동시에 수행되더라도, turn 값에 의하여 결정됨

critical section을 계속 점유하면 안 된다는 가정을 기본적으로 가지고 있다.
프로세스가 악의적으로 동작하면 안된다. 이는 동기화 알고리즘의 잘못이 아니다.

Peterson Solution의한계

- Peterson solution의 확장은 가능한가?
  - 3개 이상의 프로세스에서는 어떻게 구현할 것인가
  - 확장된 알고리즘의 증명은 어떻게 할 수 있나
  - 어떤 경우에도 동작함을 보여야
- 일반적으로 이러한 증명은 NP 문제임
  - 증명이 되더라도 매우 복잡함

초기해결책

- Critical section에 들어가면서 Interrupt를 disable 함

- Interleaving 방지

  Interleaving → 두개 이상의 프로세스가 막 섞여서 수행되는 것

  읽기만 하면 문제가 발생하지 않을 수도 있다.

  그러나 쓰고 읽고 하면 문제가 생가는 것이다.

  그래서 그냥 Critical section에 들어가면 Interrupt를 disable시켜버리는 것이다.

- 사용자 프로그램이 interrupt를 control하는 것은 바람직하지 않음

  → 계속 CPU를 쓰고 있을 때 그 프로세스를 끌어내리는 방법이 없다.

  → 사용자가 interrupt를 control하는 것은 위험하다.

- 커널을 critical section 으로 만듬
  - process as protection boundary
  - 커널에 들어갈 때 interrupt disable 함

    커널에서 수행을 마치고 돌아갈 때까지 interrupt disable 함
  - 커널 전체가 거대한 critical section으로 처리함

  공유해야 할 data를 전부 커널에 놓는 것이다.

  이것이 monoithic kernel의 구현이다.

- Scalability 문제점
  - 프로세스의 숫자가 많아 질 때 문제가 생긴다 - 대기시간
  - 커널 multithreading 도입

하드웨어 솔루션

- 인터럽트가 아닌 다른 방법이 필요함

- 명령어로 처리하면 알고리즘이 매우 간단하게 됨

  acquire lock //lock을 잡는다.

  critical section

  release lock //lock을 놓는다.

  reminder section
- 동기화를 위한 instruction이 도입됨

Synchronization Instruction

- CPU에서 원자적(atomically)으로 수행되는 명령어를 (instructions) 이용
- 원자적이란?

  명령어가 수행되는 동안 방해 받지 않음 (uninterruptible)

  → CPU에서 instructions을 정의하면,

  하드웨어와 소프트웨어의 경계를 나누고, 그 CPU의 특성을 나타낸다.

  instructions이 수행되는 중간중간에 interruption이 발생하지 않는다.

  이는 하드웨어적으로 허용이 안된다.

  각각의 명령어의 cycle이 다르다. 길고 짧음이 크게 다르다.

- Test and Set 명령어 //메모리 location을 읽어서 rv에 저장

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;   //test - read
    *target = true;         //set - write
     return rv;             //return
}
```

TestAndSet은 하나의 instruction이다 (atomic)

Test-and-Set을 사용한 Peterson 솔루션

- Shared data:

  boolean lock = false;

- Process Pi

  do{

  while (TestAndSet(&lock)); //다른 프로세스가 1을 줄 것이다.

  //0이 되는 순간 lock이 풀려들어간다.

  critical section

  lock = false; //내가 critical section을 다 썼다는 표시

  remainder section

  }

다른 명령어 – swap

- Swap 명령어

  void Swap(boolean *a, boolean *b) {

  boolean temp = *a;

  *a = *b;

  *b = temp;

  }

- Intel 80×86 instruction set
  - BTS - Bit Test and Set (386+)
  - BSWAP - Byte Swap (486+)

Swap을사용하는솔루션

- Shared data (false 로 초기화):

  boolean lock;

  boolean waiting[n];

- Process Pi

```
do {
    key := true;
    while (key == true)
        swap(&lock, &key);  //여기까지가 entry section이다.
                                //lock이 false인 순간, key가 false가 되어서 while
        을 빠져나온다.
            critical section
    lock := false;                //exit section
        remainder section
}
```

Q. 아까 설명하신 swap 부분에서 왜 while에서 swap을 하나요?

A. lock을 읽기 위함이다.

Instruction으로부족한점

- 동기화 instruction을쓰면 mutual exclusion은 해결이 되나, bounded waiting 같은 조건은 사용자 프로그램에서 제공해야 함
- Bounded waiting이주어진 문제마다 조금씩 차이가 있기 때문에, 사용자 프로그램에서 제대로 처리하는 것을 기대하기 어려움

  → 모든 프로그래머들이 시스템을 잘 알아서 리를 처리 할 수 있는 것이 아님
- 좀 더 comprehensive한 동기화 primitive가 필요함

Semaphore (세마포)

- 세마포: 두 개의 원자적 연산을 가지는 정수 변수
  - 원자적인 연산 :
    - wait() 또는 P()
    - signal() 또는 V()
  - 이 변수는 2개의 원자적인 연산(atomic operation)에 의해서만 접근됨

- P는 critical section들어가기전에, V는 나와서 수행함

- P와 V연산은 서로 독립적으로 그리고 원자적으로 수행됨
    - 하나의 프로세스가 P를 수행하여 세마포어의 값을 수정하는 동안 다른 프로세스에서 P나 V를 수행해도, 세마포어의 값을 수정하지 못한다.

Peterson 솔루션을세마포로

- 차이점
    - flag 필요 없음
    - Process: 동일한 동작
- Process:

  int s = 1;

  P(s); //entry section

  Critical Section

  V(s); //exit xection

  remainder

세마포의종류

- 세마포 값의 변화
    - P를 통과하면 decrement
    - V - increment
- 2가지의 세마포
    - Counting semaphore
        - 세마포 값의 범위가 정해져 있지 않음
        - 초기값은 가능한 자원의 수로 정해짐

          resource적인 측면이다
    - Binary semaphore
        - 세마포 value가 가질 수 있는 값은 0과 1

        - Counting semaphore보다 구현이 간단함
- Binary semaphore를 이용하여 counting semaphore를 구현할 수 있음

Original 세마포

- Busy waiting 이용
    - P(S)

      while (S < = 0) loop ; //대기

      S = S - 1;
    - V(S)

      S = S + 1;
- Busy waiting은 critical section에 진입할 조건이 될 때 까지 loop을 돌면서 기다린다.
    - 단점
        - Uniprocessor 환경에서 CPU cycle을 낭비할 수 있음

          CPU 하나를 여러프로세스가 나누어 쓰는 경우에는 도움이 안됨

          동기화와 스케줄링은 별도로 이루어지는 것임

          아무리 대기를 해봐야 critical section에서 나올 수 없음
        - 대기 중인 프로세스 중에서 누가 critical section에 진입할지 정해지지 않음

세마포with sleep queue

- Sleep queue를 이용
    - Busy waiting 방식의 CPU cycle을 낭비하는 문제를 해결
    - 세마포의 자료구조에 sleep queue를 추가하여, 대기중인 프로세스를 관리
        - 세마포의 값이 양수가 되어 critical section에 진입이 가능하게 되면, sleep queue에서 대기중인 프로세스를 깨워 실행시킴
        - 질문:
            - 누가 깨우는가
            - 누구를 깨우는가

S ≤ 0이라서 들어가는 것이 안된다.

→ 그러면 sleeping queue에 그 프로세스를 넣어서 대기하게 만듦

그리고 S>0이 되면 대기중인 프로세스를 깨움

세마포sleep queue 구현

- 세마포 자료구조 와 P/V

```
typedef struct {
    int value; // 세마포어 값
    struct process *list; // 세마포어를 대기하는 프로세스의 sleep queue
    }
semaphore;

P(semaphore *S)
S → value--;
if ( S→value < 0) {
    add this process to S→list; //프로세스를 sleep queue에 삽입
    block();                    //자발적으로 CPU를 놓는 것이다.
    }

V(semaphore *S)
S → value++;
if (S→value <= 0) {
    remove a process P from S→list; // 프로세스를 sleep queue에서 꺼내온 후,
    wakeup(P); // 실행
}
```

sleep을 했는데 누가 누구를 깨우느냐의 문제가 있다.

이에 대해서 V가 sleep queue의 하나를 깨운다.

그런데 사람들이 이를 좋은 solution이라고 생각하지 않았다.

그 이유는 sleep queue에서 깨우는 것은 scheduling정책이다.

그리고 동기화와 스케줄링이 얽히게 구성이 된 것이다.

그에 대한 방법으로, wakeup(P); 대신에 wakeupall(list);를 쓴다

다 깨워버리는 것이다.

그 깨워진 것을 ready queue에 자동으로 보내지고, 그것을 cpu가 스케줄링하는 것이다.

그리고 다른 것들은 자동적으로 wait()에 들어갈 것이다.

Binary Semaphore의 구현

- Test and set의 명령어를 이용하여 binary semaphore를 구현
- Semaphore S
  - critical section에 진입한 프로세스가 있는지 나타냄 (0 or 1)
- P(S)
  - while(testandset(&S));
  - S의 값이 리턴되고, S는 1로 바뀐다.
- V(S)
  - S = 0;
  - 진입한 프로세스가 없음을 나타내어 wait()에서 대기 중인 프로세스를 진입 가능하도록 한다.

Counting Semaphore의구현

- Binary semaphore를 이용하여 counting semaphore 구현
  - Counting semaphore를 CS라 하고, counting semaphore의 value는 C
  - 2개의 binary semaphore S1, S2 를 이용
  - S1 = 1, S2 = 0으로 초기화 (C가 양수라고 가정)

```
wait operation
P(S1);          //세마포를 잡는다.
C--;
if (C < 0){
  V(S1);       //세마포를 놓는다.
  P(S2);       //sleep
}
else
  V(S1);
```

```
signal operation
P(S1);
C++;
if (C <= 0) {
  V(S2);       //누군가가 기다리고 있다.
}
  V(S1);
```

교수님 질문

Q. Binary semaphore를 이용하여 counting semaphore 구현한다는 의미가 무엇인가?

다시 말하자면 중간 중간에 interrupt가 발생가능하다는 것이다.

A. P(S1); 을 수행할 때는 interrupt가 발생하지 않는다.

semaphore라는 것은 각각의 동작에서 interrupt가 발생해도,

semaphore 시멘틱에서는 문제가 없다는 것을 의미한다.

P(S1), V(S1)같은 Binary semaphore만 atomic하게 되면,

그걸 통해 만든 wait, signal은 얼마든지 interrupt가 발생가능하다.

그리고 커널이 multi thread가 되는게, 이러한 원리로 구성되어 있다.

즉, key가 되는 것들을 구현해 놓고 structure를 짜면, atomicIty를 지킬 수 있다.

즉, mechine level에서 atomic하지 않더라도, P의 동작이 atomicIty를 제공할 수 있다.

semaphore가 1 → 들어간다

semaphore가 0 → block을 한다.

P(S1)에서, S1 = 0이면, wait한다.

V(S1)은 기다리고 있는 것을 떼어주는 것이다.

Semaphore의 단점

- Deadlock이 발생할 가능성이 존재한다.
- P와 V의 연산이 분리 되어 있기 때문에, 잘못 사용할 경우에 대한 대책이 없음
  - P(); → critical section → P();

Critical section 후에 lock을 풀어주는 V(); 가 없으므로 한 프로세스가 critical section에 진입하면 나올 수 없음 (다른 프로세스들은 critical section에 진입 못하므로 deadlock이 발생)

- V(); → critical section → P();

  Critical section 전에 lock을 걸어주는 P(); 가 없으므로 여러 프로세스가 동시에 critical section에 진입 할 수 있으므로 mutual exclusion을 보장 못함

- High-level 언어에서 동기화를 제공하는 방법이 필요해짐

Deadlock

- Deadlock
  - 두 개 이상의 프로세스들이 끝없이 기다리고 있는 상황

프로세스가 많아지면 deadlock이 있는지를 파악하기도 어렵다.

Partial order

- Deadlock 을 해결하는 방법
- 집합 P에 대한 binary relation(≤)으로 reflexive, antisymmetric, transitive 조건을 만족한다.
- 집합 P 의 모든 원소 a,b,c에 대해 다음 조건을 만족한다.
  - a ≤ a (reflexivity)
  - if a ≤ b and b ≤ a then a = b (antisymmetry)
  - if a ≤ b and b ≤ c then a ≤ c (transitivity)

집합의 원소들 중 일부가 위에서 제시된 조건을 만족하는 순서를 가진다는 것이 partial order이다.

lock을 잡는 순서를 정할 수 있게 된다.

inode에 잡아야 하는 lock이 있다.

X의 lock을 잡으려면, Z(Y(X()))의 순서대로 잡아서 deadlock을 방지하는 것이다.

이것의 실제 구현이 쉽지가 않기에 monitor가 나온다.

Monitor

- 세마포는 사용자가 P/V를 제대로 사용해야 한다
  - 명시적으로 코드에 넣어야 함
  - Error-prone (어떤 리소스를 먼저 잡고, 나중에 잡아야 하는 지를 알아야 한다.)

- High-level 언어에서의 동기화 방법

  High-level 언어에서는 criticla section만 지정하는 것이다.

  나머지는 알아서 동기화 해줘라! 라는 것이다.

  - 예: Java의 thread에서 동기화를 위해 Monitor가 사용됨
  - 한 순간에 하나의 프로세스만 monitor에서 활동 하도록 보장

- Application은 procedure를 호출 하는 것 만으로 동기화를 해결 할 수 있음
  - 프로그래머는 동기화(P와 V연산)를 명시적으로 코딩할 필요가 없음

Monitor 사례: “synchronized” in Java

- “synchronized” 블럭을 통해 동기화되는 영역을 지정 가능

synchronized 내부동작

- Entry queue에서 대기하고 있는 쓰레드들은 increaseBalance() 함수에 진입하기 위해 기다리고 있음
- 쓰레드 T1이 increaseBalance() 함수에서 나간 후에 다른 쓰레드가 함수에 진입할 수 있음

Monitor 사례: Transaction in Database

- Transaction

- Transaction으로 처리해야 하는 명령문들을 모아놓은 작업 단위
- 모든 작업단위가 반드시 완전히 수행되면, 성공
    - 만약 어느 하나라도 실패한다면 transaction은 취소됨

Transaction의 특성?

ALL or Nothing이다.

→ 이것이 동기화를 표현하는 방법이다.

→ 어느 하나라도 실패한다면 transaction은 취소

Begin_transaction과 End_transaction으로 critical section을 지정하면 DB에서 알아서 처리한다.