동기화 2

Bounded-buffer problem

- N개의 버퍼에 여러 생산자와 여러 소비자가 접근
- 생산자(producer) – 한번에 하나의 아이템을 생산해 버퍼에 저장
  - 동작 조건: 버퍼 내에 저장할 공간이 있는가?
    - Buffer의 상태가 full이면 대기

  이것은 write하는 것이다.

- 소비자(consumer) – 버퍼에서 하나의 아이템을 가져옴
  - 그리고 버퍼에서 삭제
  - 동작 조건: 버퍼 내에 소비할 아이템이 있는가?
    - Buffer의 상태가 empty이면 대기

  이것은 read하고, read이후의 내용은 지워져버린다.

- 생산자와 소비자가 충돌없이 버퍼를 접근할 수 있도록
  - 버퍼 – critical section

네트워크 시스템의 패킷 같은 것이다.

network buffer를 생각해봐라

Bounded-buffer problem 구현

- Buffer ← 공유 데이터이다.

Bounded-buffer problem 세마포

세마포의 구현은 다음을 생각해야 한다

설계

- "몇 개"의 세마포를 사용할 것인가?
- "어떻게" 세마포를 사용할 것인가?

→ 어떤 데이터를 보호할 것인가?

- 문제 해결을 위한 세마포
  - Empty: 버퍼 내에 저장할 공간
    - 생산자의 진입을 관리
  - Full: 버퍼 내에 소비할 아이템
    - 소비자의 진입을 관리
  - Mutex: 버퍼에 대한 접근을 관리
    - 생산자, 소비자가 empty, full 세마포를 진입한 경우,
      buffer의 상태 값을 변경하기 위한 세마포어
- 세마포어 value의 초기값
  - full = 0 //state를 알리는 것이다.
  - empty = n //state를 알리는 것이다. buffer에 empty인 entry가 n개
  - mutex = 1

buffer의 state를 나타내 보여야 한다.

→ 3개의 세마포를 사용하는 것이다.

생산자 프로세스

1) 문제에서 말하는 목적을 보고

2) 생산자와 소비자의 프로토콜을 정의 (생산자와 소비자가 어떻게 움직이는지를 정의)

3) 세마포 설계

4) 동기화 알고리즘 구현 (presudo code)

5) validation (MUTEX, deadlock은 없나?)

do {

```
    ...
    ...
    wait(empty); //버퍼에 적어도 하나의 공간이 생기기를 기다림
                 //empty >1이어야 통과한다.
    wait(mutex); //critical section에 진입하기 위해 기다림
                 //binary 세마포
                 //mutex = 1이면 write
                 //mutex = 0이면 누군가가 이미 들어가 있다는 뜻
    ...
    produce an item in nextp
    nextp++
    ...
    signal(mutex); //critical section에서 빠져 나왔음을 알림
                   //mutex++의 의미인데, binary이므로 0이 된다.
    signal(full); //버퍼에 아이템이 있음을 알림
                  // full++이 된다.
} while (1);
```

소비자프로세스

```
do {
    wait(full); //버퍼에 적어도 하나의 아이템이 채워지기를 기다림
                //초기값이 0이었다.
                //full ≥ 1이면 통과가 된다.
    wait(mutex);
    ...
    remove an item from buffer;
    nextc++;
    ...
    signal(mutex);
```

```
    signal(empty); //버퍼에 하나의 빈 공간이 생겼음을 알림 ...
    ...
} while (1);
```

mutex는 mutex 변수로 만족시키는 것이 가능하다.

생산자 프로세스는 empty가 0이 될 때까지 계속 움직이는 것이 가능하다.

이 구현은 하나의 프로세스만이 버퍼에 들어갈 수 있는 구현이다.

concurrency 문제이다.

buffer에 하나의 mutex만을 사용하는 것이다.

이는 덩어리가 너무 크다는 의미이다.

동시에 모든 버퍼들을 쓸 수도 없다.

그래서 이것을 두고 concurrency 가 낮다라는 표현을 사용하는 것이다.

concurrency를 높히려면 각각의 버퍼마다 세마포를 두고, 세마포 설계를 다시하는 것이다.

→ 그것에 맞는 알고리즘을 설계하고 validation하는 것이다.

덩어리가 작아지면 어떤 문제가 생기는가?

→ deadlock이 발생할 가능성이 증가한다.

concurrency와 deadlock은 반비례 관계이다.

학생질문

full 이나 empty를 일반변수로 관리하면 누가 읽는 중에 쓴다는 등의 문제가 생길 수 있다.

그래서 동기확 깨진다.

따라서 세마포를 사용하는 것이다.

사실 보면 producer나 consumer나 둘 다 쓰고, 읽고 지우기 때문에

둘다 writer, writer이다.

Readers-Writers Problem

Reader는 진짜 읽기만 한다.

Readers-Writers Problem 도식

Writers는 soly only하다.

Readers-Writers Problem Solution 1

- 문제 해결을 위한 자료구조와 세마포어
  - int readcount : 버퍼에 현재 접근하는 Reader 숫자
  - semaphore wrt : Writers 간 동기화를 위한 세마포
  - semaphore mutex : Readcount와 wrt 의 접근이 원자적으로 수행됨을 보장하기 위한 세마포
- 초기값
  - mutex = 1, wrt = 1, readcount = 0

Solution 1 -writer

wait(wrt); // entry section

…

writing is performed

…

signal(wrt); //exit section

Solution 1 -reader

```
wait(mutex);      //binary이다.
    readcount++;
    if (readcount == 1)
        wait(wrt); // writer 들어오지 못하게 함
                   //writer의 초기값이 1이기에 writer가 동작하지 못한다.
```

```
                    // 이걸 통과하면 wrt = 0이 된다.
signal(mutex);
...
reading is performed
...
wait(mutex);
    readcount--;
    if (readcount == 0) //이제 reader가 아무도 없다는 이야기이다.
        signal(wrt); // 이걸 통과하면 wrt = 1이 된다.
signal(mutex);
```

교수님 질문

Q. 하나의 writer, 여러개의 reader가 동작한다는 조건을 만족하는가?

A. 만족을 한다.

문제점

R1 W R2가 들어오면,

R1, R2, ...???? W는 언제 수행될지를 모르는 것이다.

이 알고리즘은 starvation이 존재한다.

- Writer의 starvation을 예방하는 solution이 존재함

deadlock과 starvation이 존재하면 안된다.

교수님 질문

deadlock과 starvation의 차이점이 무엇인가?

deadlock은 구조적으로 진행이 안되는 것이다. (시간이 무한대로 흘러도 안됨)

starvation은 시간이 충분히 지나면 해결가능성이 있기는 하다.

starvation은 preference를 의미한다.

solution1의 concurrency가 좋은가 안 좋은가?

안 좋다.

그 이유는 버퍼별로 따로 동작을 하지 못한다.

버퍼별로 세마포를 만들어야 한다.

이것이 시험에 나올 가능성이 있다.

숙제

Writer의 starvation을 예방하는 solution을 만들어 봐라, 찾아봐라

Dining-Philosophers Problem

최대 2사람만 식사가능

5사람이 어떻게 사이 좋게 밥을 먹을 수 있는가?

모두들 자신의 오른쪽 젓가락 을 동시에 집었을 경우 deadlock

누군가가 계속 밥을 못먹음 : starvation

Dining-Philosophers Problem Solution 1

- 젓가락을 집을 때 동기화를 하는 방법
  - 자신의 왼쪽 또는 오른쪽 젓가락부터 집도록 한다.
- 세마포 chopstick[5] : 각 젓가락에 대한 동기화
- 초기값은 모두 1

```
do {
    ...
    think
    ...
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    ...
    eat
```

```
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
} while (1);
```

- 동시에 chopstick[i]를잡으면 deadlock 발생 → 이건 시간이 지나도 해결되지 않는다.

Dining-Philosophers Problem 개선안

- Deadlock의 해결 방안들

  1) 한 번에 최대 4명의 철학자만 식탁에 앉도록 한다.

  2) 젓가락의 상태를 미리 검사하여 양 쪽의 젓가락이 모두 사용 가능할 때만 젓가락을 집도록 한다.

  3) 철학자에게 번호를 부여하여 홀수인 철학자는 왼쪽의 젓가락을, 짝수인 철학자는 오른쪽의 젓가락을 먼저 집도록 한다.

- 위의 해결방안들은 starvation까지 해결해주지는 못함
  - 각각의 방안에 대해 starvation에 대한 고려를 추가할 수 있다.
    - 한 차례 굶은 철학자에게 우선권을 주는 방안

Dining-Philosophers Problem Solution 2

- 양쪽의 젓가락을 한꺼번에 집는 방법
  - 젓가락의 상태를 미리 검사하여 양 쪽의 젓가락이 모두 사용 가능할 때 만 젓가락을 집도록 하는 방법
- 사용되는 자료구조 State[5] : 각 철학자의 상태 (THINKING, HUNGRY, EATING)
- 문제 해결을 위한 세마포
  - mutex : 젓가락을 집거나 놓는 수행을 critical section으로 관리하기 위한 세마포 - 초기값: 1
  - self[5] : 철학자 각각 젓가락 두 개를 잡기를 기다리는 세마포어
    - 초기값: 0

- self[i] 의미는 철학자 i가 HUNGRY 상태이더라도, 다른 젓가락 하나를 사용할 수 없을 경우 waiting을 하기 위한 세마포어

Solution 2

```
take_chopsticks(int i)
{
    wait(mutex);
    state[i] = HUNGRY;
    test(i);                    //자신이 먹을 수 있는지의 여부를 판단
    signal(mutex);
    wait(self[i]);
}
```

Mutex를 통해 진입하여, test(i) 를 통해 양쪽의 철학자 상태를 검사한 후,
자신이 먹을 차례를 기다린다.

```
put_chopsticks(int i)
{
    wait(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    signal(mutex);
}
```

- Mutex를 통해 진입하여, test(LEFT), test(RIGHT)를 통해 양쪽의 철학자 상태를 검사한 후,
  먹을 차례를 기다리는 철학자에게 signal을 보내준다. ▶ test(i)에서 수행

```
test(int i) {
```

```
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;
        signal(self[i]);
    }
}
```

- Test를 수행한 철학자(i) 가 HUNGRY 상태이고, 양쪽의 철학자가 모두 젓가락을 집지 않은 상태(NOT EATING)이면 take_chopsticks()에서 wait 했던 자신의 세마포 self[i]에 signal 을 보내어 eating 상태로 진행하도록 한다.

해설

- Solution2 철학자 좌우 젓가락이 사용 가능할 때만 critical section에 진입
    - 즉, i 번째 철학자가 식사를 하기 위해서는 i-1(LEFT) 과 i+1(RIGHT) 철학자가 EATING 상태가 아니어야 한다.

- take_chopstick()과 put_chopstick()의동작을 구체적으로 살펴보자.

put_chopstick()

put_chopstick에는 preference에 대한 개념이 없다.

양쪽의 젓가락을 모두 사용가능할 때 집으므로,

젓가락 1개를 집지 않으므로, deadlock도 없다.

교수님의 질문

concurrence는 어떠한가?

실질적으로 동기화되는 세마포는 mutex뿐이다.

이 말은 이 전체를 critical section으로 보는 것이다.

하나를 test하는 동안에는 다른 녀석들을 test하지 못한다.

한번에 하나만 test가 가능하다는 의미이다.

이론적으로보면 5개의 젓가락이므로,

2명까지 test가 가능한데, 1명만 test가 가능하므로 Concurrency가 떨어진다고 볼 수 있다.

동기화 알고리즘 체크 요소들

- Concurrency를 얼마나 제공하는지
    - Global lock을 쓰면 문제는 간단하게 풀린다

Global lock을 쓰는 것이 그렇게 나쁘지 않다.

문제를 풀 때 Global lock을 우선 이용하여 풀어본 뒤에,

복잡성을 더하여 Concurrency를 높히는 것이 좋다.

Global lock은 하나의 세마포만을 사용하는 것이다.

Concurrency를 maximize하는 것은 매우 어렵다.