프로세스 동기화

목차

🐿️ 프로세스 동기화

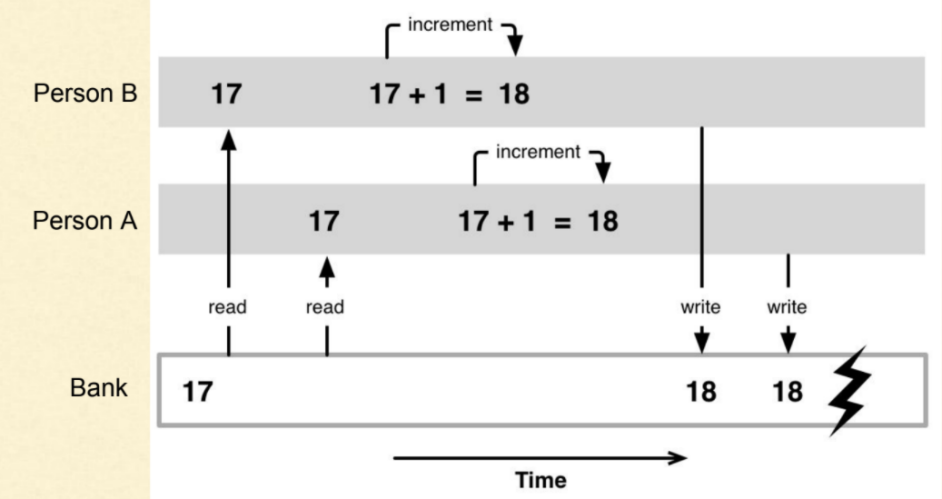
- 프로세스의 실행 순서를 제어하고 동시에 접근할 수 없는 자원에 하나의 프로세스만 접근하게 하여 데이터의 일관성을 유지하는 과정
- 프로세스는 동시에 실행되면서 서로 영향을 주고 받는데, 이 과정에서 자원의 일관성이 보장되어야 한다

💡 일관성이란?

- 자원의 상태가 동일함을 의미
- 예를 들어 A + B라는 연산을 수행한다고 할 때, A와 B에 저장된 데이터는 연산이 종료될 때까지 값이 변하면 안된다

🥊Race Condition

- 여러 프로세스들이 동시에 자원에 접근하는 상황에서는 명령어 실행 순서에 따라 결과값이 달라질 수 있다
  - 이 상황을 경쟁 상태 (Race Condition)라고 한다



- Person A와 Person B는 Bank Account에 동시에 1씩 예금을 하려고 하는데, 동시에 17이라는 자원을 읽고 예금을 하게 되어 최종 결과가 19가 아닌 18이 나오게 되었다
  - 이렇게 공유 데이터에 동시에 접근하는 것은 데이터의 불일치 문제를 발생시킬 수 있다
  - 따라서 Race Condition을 막고 일관성을 유지하기 위해서는 동기화 (Synchronization)가 필요하다

❤ Critical Section

- 코드 상에서 Race Condition이 발생할 수 있는 특정 부분
  - 즉, 공유 데이터를 접근하는 코드 부분

- Critical Section으로 인해 발생하는 문제들을 해결하기 위해서는 다음 조건을 만족해야 한다

1. Mutual Exclusion (상호 배제)

- 이미 한 Process가 Critical Section에서 작업중이면 다른 모든 Process는 Critical Section에 진입할 수 없다

2. Progress (진행)

- Critical Section에서 작업중인 Process가 없다면, Critical Section에 진입하고자 하는 프로세스가 존재하는 경우 진입할 수 있어야 한다
- 다른 Process가 기다리는 중에 한 Process가 두번 연속으로 Critical Section에 진입한다면 Progress 위배를 하게 된다

3. Bounded Waiting (한정 대기)

- Critical Section에 진입하려는 Process가 무한정 기다려서는 안된다

🛁 S/W Method

Peterson's Algorithm

```
do {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);

    // critical section
    ...
    flag = false;
    // remainder secion
} while (true);
```

- 두개의 프로세스가 존재할 때 사용하는 알고리즘
- 만약 다른 프로세스가 기다리고 있으면 turn을 바꿔주면서 먼저 Critical Section에 들어갈 수 있도록 한다
- Mutual Exclusion, Progress, Bounded Waiting 세가지를 모두 만족한다

- 그러나, Critical Section의 진입을 기다리면서 계속 CPU와 메모리를 사용하는 `Busy Waiting` 문제점이 존재한다

> 💡 Busy Waiting
>
> - OS에서 원하는 자원을 얻기 위해 기다리는 것이 아니라 **권한을 얻을 때까지 확인하는 것을 의미**
> - CPU의 자원을 쓸데없이 낭비하기 때문에 좋지 않은 Thread 동기화 방식이다
> - 자원의 권한을 얻는데 많은 시간이 소요되지 않거나, Context Switching을 위한 비용보다 Busy Waiting을 하는 것이 더 효율적인 상황에서 사용

Dekker's Algorithm

```
do {
    flag[i] = true; // P(i) wants to enter CS

    while(flag[j]) {
        if(turn == j) {
            flag[i] = false;
            while(turn == j); // P(j)가 실행을 마칠 때까지 대기
            flag[i] = true;
        }
    }
    // critical section
    ...
    turn = j;
    flag[i] = false;
    // remainder section
} while(true);
```

- 두개의 프로세스가 존재할 때 사용
- Mutual Exclusion, Progress, Bounded Waiting 세가지를 모두 만족
- Dekker's Algorithm은 Peterson's Algorithm과 달리 Critical Section을 먼저 실행한 후 상대방에게 넘겨준다
- Peterson's Algorithm은 조건 체크를 하기 전에 turn을 상대방에게 먼저 넘겨주고 시작

Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number[n]) + 1;
    choosing[i] = false;

    for(j = 0; j <= n; j++) {
        while(choosing[j]); // 번호를 뽑으려는 Process가 존재하면 대

        while(number[j] != 0 && ((number[j] < number[i]) ||
            number[j] == number[i] && j < i));
    }
    // critical section
    ...
    number[i] = 0;
    // remainder section
} while(true);
```

- N개의 프로세스가 존재할 때 사용
- 번호표를 뽑음, 각각의 Process는 고유의 번호를 가짐
- Mutual Exclusion, Progress, Bounded Waiting 세가지를 모두 만족

💀 General Method (S/W + H/W)

- 운영체제 레벨에서 구현되어 있으며 시스템 리소스를 사용
- 더 안전하고 정확한 동기화를 제공하지만, S/W적인 방법보다 더 많은 오버헤드가 발생할 수 있다

Mutex

- 한 프로세스에 의해 소유될 수 있는 Key를 기반으로 한 상호 배제 기법

```
acquire() {
    while(!available);
    available = false;
}
```

```
release() {
    available = true;
}
```

- Critical Section에 진입하기 위해 `acquire()`로 lock후 사용
- 사용이 끝나면 `release()`
- Peterson's Algorithm과 마찬가지로 Busy Waiting 단점 존재

Semaphore

- Semaphore S-integer variable → 이용가능한 총 갯수를 나타냄
  - ex) 프린터기가 총 3개 있다 → S = 3
- 두가지 atomic operation에 의해서만 접근이 가능하다
  1. `wait()`
  2. `signal()`

```
wait(S) {
    while(S <= 0);
    S--;
}
```

```
signal(S) {
    S++;
}
```

- 그렇지만 이 방식 또한 Busy Waiting이 발생하므로 비효율적이다
- 따라서 Block & Wakeup 방식을 사용
  - Critical Section으로 진입에 실패한 프로세스를 기다리게 하지 않고 Block시킨 뒤 Critical Section에 자리가 나면 다시 깨워준다

```
typedef struct {
    int value;
    struct process* list;
} semaphore;
```

- 다음과 같이 Semaphore 구조체를 정의
  - 만약 기다리는 프로세스가 존재하면 Queue에 넣고 Block시킨 뒤 자리가 나면 깨워준다

```
void wait(semaphore S) {
    S.value--;
    if(S.value < 0) {
```

```
void signal(semaphore S) {
    S.value++;
    if(S.value <= 0) {
```

```
            add this process to q
            block();
        }
    }
```

```
            remove a process P fr
            wakeup(P);
        }
    }
```

- `block()` 을 수행하면 커널은 block()을 호출한 프로세스를 중지시키고 해당 프로세스의 PCB를 wait queue에 넣어준다
- `wakeup()`을 수행하면 block된 프로세스를 P를 깨운다

Monitors

- private data와 public method를 캡슐화한다
- 절차 내의 코드로만 내부 변수에 접근이 가능하다
- thread가 리소스에 어떻게 접근하는지 모니터링하기 때문에 **Monitor**라고 불린다

구성 요소

1. entry set : 모니터 내부에 순서대로 진입하기 위해 대기하는 큐
2. wait set : 특수한 상황이 발생했을 때 모니터를 반환하고 대기하는 큐
3. condition variable : 프로세스가 모니터 안에서 기다릴 수 있게 하기 위해 사용하는 변수
    - `wait()`와 `signal()` method를 이용해서 접근이 가능하다
    - condition variable의 이름을 x라고 할 때 `x.wait()`를 호출한 thread는 다른 thread가 `x.signal()`을 호출하기 전까지 대기한다
4. shared variable : 공유 데이터, Critical Section
5. procedure : 공유 데이터에 접근하기 위한 메소드, `wait()`와 `signal()`

비유

- 모니터는 두개의 방과 하나의 복도가 있는 건물
- shared variable은 전용실(exclusive room)이다
- wait set은 대기실(wait room)이다
- entry set은 복도(hallway)이다
- thread는 전용실에 들어가고 싶은 사람들이다

동작

1. 건물에 들어가기 → 모니터에 들어가기
2. 전용실 입장 → 모니터 획득
3. 전용실에 있는 것 → 모니터를 소유하고 있는 것
4. 전용실 나가기 → 모니터 해제
5. 건물에서 나가기 → 모니터에서 나가기

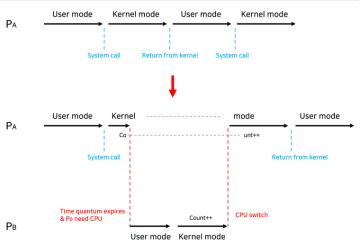
참고

▼ 링크

[운영체제(OS)] 6. 프로세스 동기화(Process Synchronization)

[목차] 1. Race Condition 2. Critical Section 3. Synchronization Algorithms 4. Synchronization Hardware 5. Mutex Locks 6. Semaphores 7. Classical Problems of Synchronization 8. Monitor

https://rebro.kr/176



뮤텍스(Mutex)와 세마포어(Semaphore)의 차이

이 글은 Medium에 개시된 글입니다. Medium에서 보시면 좀 더 유쾌한 환경에서 글을 보실 수 있습니다. 뮤텍스(Mutex)와 세마포어(Semaphore)의 차이 Toilet problem 동시성 프로그래밍의 가

https://worthpreading.tistory.com/90