[Chapter 6]

소프트웨어융합학과/2020105742/한지훈

1. Synchronization
   A. 개요:
      - 멀티 스레드 환경에서는 스레드 간에 IPC를 통해서 data 등을 주고받으면서 협업
      - > 스레드 간에 공유 자원(shared memory)가 존재
      - > 스레드 간에 순서가 맞지 않으면 shared memory가 왜곡될 수 있음(synchronization problem)
      - > synchronization problem은 반드시 고려하고, 해결해야 하는 문제

2. Synchronization problem examples
   A. Sharing bank account
      I. 상황: 계좌에 100만원 있는 상태에서 남자와 여자가 동시에 돈을 각각 1만원, 10만원 빼려는 상황
      II. 문제:
      - 만약 남자의 요청으로 변경된 통장 잔액(99만원)이 업데이트되기 전에 여자가 요청을 하면
      - > 여자의 요청이 끝난 후에 99만원으로 업데이트 되어 실제 값(89만원)과 차이나게 됨

      III. 특징: 특히 금융 쪽에서 일어나면 치명적인 문제이므로 반드시 해결 필요
   B. Bounded buffer
      I. 상황: producer는 buffer에 data를 하나씩 넣고, consumer는 buffer에 data를 하나씩 빼는 상황
      II. 문제:
      - 기존 count = 5, producer가 count = 6 업데이트 하기 전에 consumer가 요청을 하면
      - > consumer 요청 끝난 후에 count = 6 업데이트되어 실제 값(5)와 차이나게 됨(반대도 가능)

3. Synchronization Problem
   A. 정의: 다수의 스레드가 공유자원에 동시에 접근하려 할 때 보호를 하지 않아 data가 왜곡되는 현상
      I. Race Condition: 다수의 스레드가 공유자원에 접근하려 하는 현상
      II. Critical section: Synchronization Problem이 발생할 수 있는 code section
   B. 특징: Non deterministic
      I. 의미: 결국 동기화 문제는 생길 수도, 안 생길 수도 있음
      II. 생기는 조건: Race Condition이 발생하고, Critical section이 보호되지 않을 때

4. Requirements for Synchronization Tools
   A. Mutual Exclusion: 한 프로세스가 critical section 사용 중이면 나머지 프로세스는 쓰지 못하게 막아야 함
      I. 특징: Mutual Exclusion을 만족하면 Progress, Bounded Waiting은 부수적인 것으로, 알아서 따라옴
   B. Progress: critical section을 아무도 쓰고 있지 않으면 기다리지 말고 무조건 쓰기
   C. Bounded Waiting: 다른 누군가가 critical section 쓰고 있을 때의 기다림은 무한 X, 언젠간 반드시 사용가능

5. Lock
   A. 개요: Shared resource의 critical section 접근 전에 lock(다른 것 접근 못하게), critical section 끝나면 unlock
   B. 특징: Synchronization Tools의 3가지 조건을 모두 만족함
      - Mutual Exclusion: critical section 전후로 lock, unlock을 하여 다른 프로세스로부터 상호배제 가능
      - Progress: 접근 시 lock이 걸려있지 않다면 바로 프로세스 수행 가능
      - Bounded waiting: critical section이 끝나면 unlock으로 풀어주기 때문에 무한정 기다림은 없음

6. Low level Synchronization Tools

A. SpinLock

I. 개요: 안 잠겨있으면 잠그고, 잠겨있으면 무한 loop로 대기 > 계속 loop를 돌기 때문에 "busy wait"라 불림

II. 문제:

- Held가 공유자원이기 때문에 race condition을 만족, code 자체가 critical section이라 동기화문제 발생가능
- Busy wait는 CPU를 계속 돌리기 때문에 CPU 사용 효율이 매우 떨어짐

III. 해결법:

- Software-only algorithms: HW에 비해 SW는 느리기 때문에 커널 오버헤드가 커서 잘 쓰지 않음
- Hardware atomic instructions: lock 구현하는 critical section 부분을 Atomic하게 묶기
  - Test-and-Set: atomic하게 동작하는 함수로, lock을 구현 가능하게 해줌
  - Compare-and-Swap: Test-and-Set과 동일한 동작을 하지만 변수 하나 추가하여 안정성 높임
- Disable/re-enable interrupts: critical section 수행 중에는 interrupt 끄고, 끝나면 다시 켜기

IV. Primitive

- 무한 loop로 인한 busy waiting을 피할 수 없음, 최소화 필요
  - 내부 보호 용도로만 lock 쓰고, 사용자가 임의로 lock 걸 수 없게 하자
  - Lock은 OS(low level)에서만 사용 가능, high-level synchronization 구현을 위해 사용됨

B. Disabling Interrupts

I. 개요: critical section을 수행하는 동안에는 interrupt 끄고, 끝나면 다시 켜기

II. 장점: Busy waiting을 하지 않음(무한 loop가 없음)

III. 문제:

- Interrupt를 쌓아두는 것이 아니고 버려버리기 때문에 중요한 interrupt를 놓칠 수 있음
- critical section 길면 다른 프로세스 스케줄링 불가

IV. 사용처: Batch system, Embedded system

V. Primitive: Disabling Interrupts도 lock과 마찬가지로 OS primitive, high-level synchronization 구현을 위해 사용

7. High-level Synchronization Tools

A. Semaphore

I. 개요: 양의 정숫값(critical section에 들어갈 수 있는 최대 프로세스 수)인 티겟을 가짐

II. 동작 방식:

- Wait: critical section에 들어가면서, 티켓 수 1 감소
- Signal: critical section 수행 종료 후, 티켓을 반납하면서 티켓 수 1 증가
  - 티켓 = 0이면 wait가 불가능해지는 방식

III. 종류:

- Binary/Mutex Semaphore: 초기 티켓값이 0 또는 1
  - 초기 티켓값 = 0: 프로세스 간의 시간적 동기화(General synchronization)에 적용 가능
    A 이후 signal 걸어주고 B 이전에 wait 걸어주면 B는 반드시 A가 실행된 직후에만 실행 가능
  - 초기 티켓값 = 1: Critical section을 보호하는 목적으로 사용 가능
    공동계좌 예제: 한 프로세스가 wait를 걸면, 그 프로세스가 signal 걸기 전엔 다른 프로세스 접근 불가
- Counting Semaphore: 초기 티켓값이 2 이상
  - 2개의 프로세스 간의 race condition만 고려하는 경우만 있는 것은 아님
    EX) 출력가능 프린터가 5개이고, 최대 사용가능 사용자를 5명으로 제한하고 싶을 때 티겟값 = 5

IV. Binary Semaphore VS Mutex lock(Block)

- Synchronization Tool이 만들어졌을 초기 시절에는 lock을 user level에서도 자유롭게 걸 수 있었음
  - Binary semaphore이 나왔지만 사람들이 불편하다고 하여 lock의 형태(Mutex lock)로도 구현
  - 결국 Binary Semaphore, Mutex lock는 같은 역할을 한다고 볼 수 있음

V. 구현:

- 구조체 구성: 티켓 수(value), 대기 프로세스들의 PCB 담아놓은 포인터(*L)
- 특징: 구현 코드를 보면 OS Primitive 한 lock이 Semaphore에 사용되었음을 알 수 있음

VI. 문제:

- Deadlock: 2개의 Semaphore S, Q에 대해 P1은 S, P2는 Q에 wait를 건 후에, 다시 반대로 wait를 거는 상황
  - S, Q 티켓 수 = 0이기 때문에 둘 다 기다릴 수밖에 없고, 이 기다림은 절대 풀리지 않음
  - Starvation 또는 indefinite blocking로 이어질 가능성 있음
- OS에서 제공하기 때문에 사용하기 어렵고 버그도 존재

B. Monitors

I. 개요: JAVA에서 처음 나온 개념으로, tool을 OS가 아니라 프로그래밍 언어로 제공하자

II. 동작 방식:

- 모든 공유자원을 선언하고, 공유자원에 접근하는 모든 함수들(critical section) 나열
  - 이 함수들 중 하나라도 수행되면 다른 함수들은 수행 안되게 막을 수 있음(Block)

III. 문제: Cin 입력시간이 길어지면 다른 함수들은 그 시간동안 수행되지 못하고 막힘

IV. 해결법: Conditional variable(특정 조건이 만족되면 block 풀어 다른 함수들 수행될 수 있게 함

C. Monitors VS Semaphores

- Condition variable: block이 이미 풀려 있는 상황에서 Condition variable로 block 풀려고 하는 상황
  - 해당 신호는 버려짐(History 없음)
- Semaphore: block이 이미 풀려 있는 상황에서 Signal로 block 풀려고 하는 상황
  - Signal 날린 수만큼 티켓 수 증가(History 있음)

[Chapter 7]

소프트웨어융합학과/2020105742/한지훈

8. Bounded Buffer Problem

A. 개요: Producer는 item을 원형 큐에 1개씩 넣고, Consumer는 1개씩 빼는 작업을 하는 상황

B. Implementation with semaphores

I. Semaphore 구성

- Mutex: 초기값 1, 상호배제(Mutal exclusion) 구현을 위해 사용하는 Semaphore
- Empty: 초기값 N, 큐가 비어 있을 때 더 가져가지 못하게 사용하는 Semaphore
- Full: 초기값 0, 큐가 꽉 차 있을 때 더 못 채우게 사용하는 Semaphore

II. Producer 동작

wait (empty); empty의 티켓 1개 사용(-1), 만약 꽉 차 있어 empty=0이면 실행 불가

- wait (mutex); Producer가 item 넣는 동안 consumer은 접근하지 못하게 막음(-1)(상호 배제)
- critical section 수행(item 1개 넣기)
- signal (mutex); 티켓 반환하여(+1) 다시 Consumer가 접근할 수 있게 함
- signal (full); item 1개 넣었으니 티켓 1장 추가(+1)

III. Consumer 동작

- Full, empty에 대해 wait와 signal의 순서에만 차이가 있고, Producer와 동일하게 동작함

C. Implementation with mutex lock and condition variables

I. 특징

- mutex lock: binary semaphore과 같은 동작을 하는 코드라고 할 수 있음
- CondVar: history가 없기 때문에 lock을 계속 풀어도 누적되지 않아 counter 개념 X > int count 선언
- Producer의 "while (count==N)"을 빠져나가기 위해서는 Consumer의 "count--;"가 수행되어야 함
  - But Producer에 의해 lock(mutex)로 잠겨있어 수행할 수 없음
  - Mutex 일시적으로 풀어주는 conditional variable을 input parameter로 추가 선언 필요

9. Readers-Writers problem

A. 개요: 문서를 쓸 수 있는 사람은 한번에 최대 1명(아무도 안 읽을 때), 읽는 사람은 여러 명 가능(쓰지 않을 때)

I. 필요 요소:

- Readcount: 문서를 일고 있는 사람의 수 확인 필요(공유자원의 관리 위해)
- Mutex: 스레드 간에 Mutal exclusion을 구현하기 위해 필요
- Wrt: 누가 쓰고 있을 때는 읽는 사람이 없게 하고, 누가 읽고 있을 때는 쓰는 사람이 없게 하기 위함

B. Implementation with semaphores

I. 구현:

- Wrt: write/read 안하고 있으면 1, 하고 있으면 0
- 만약 read 하기 전 readcount == 1이라면, 최초로 읽으러 온 사람임(읽기 작업 시작)
  - wait(wrt)를 걸어 write를 못하게 막아야 함
- 만약 read 한 이후 readcount == 0이라면, 마지막으로 읽은 사람임(읽기 작업 종료)
  - Signal(wrt)를 통해 다시 write 할 수 있게 해야 함

II. 특징: conditional variable로 단순하게 구현 가능

III. 문제: reader가 많으면 writer starvation 가능, 하지만 이는 code 때문이 아니고 스케줄러 정책 때문임

10. Dining philosopher problem

A. 개요: 철학자들은 Thinking - Getting hungry – Eating의 순환을 하는데, 먹기 위해서 양옆의 젓가락을 집어야 함

B. simple solution

I. 공유자원: 젓가락(양옆의 철학자가 모두 접근 가능) > 접근을 보호하자

II. 방식: 좌/우 젓가락을 순서대로 wait 건 후 먹고, 좌/우 젓가락을 순서대로 signal 걸기

III. 문제: Deadlock(모든 철학자가 동시에 왼쪽 젓가락을 들면, 오른쪽 젓가락을 아무도 들 수 없음

- starvation으로 이어짐

IV. 해결법:

- 좌/우 순서가 아니라 랜덤으로 좌/우 중 한 개를 먼저 집는 방식
  - 해결 안됨(운 나빠서 랜덤으로 모두 왼쪽/오른쪽 집게 될 수 있음)
  - 공유자원을 젓가락으로 두면 해결하지 못함

C. Deadlock-free version

I. 공유자원: 철학자의 상태값(본인 + 양쪽 사람들의 상태값 확인을 통한 구현)

II. 방식:

음식을 먹으려 할 때 pickup()을 수행하여 나의 상태를 HUNGRY로 바꾸고 나에 대한 test를 함

- 만약 양옆의 사람이 모두 EATING 상태가 아니라면, 내 상태를 EATING으로 바꿔 음식을 먹음
- 만약 양옆의 사람 중 한명이라도 EATING 상태라면, test는 아무것도 하지 않고 종료됨
- 다 먹고 나서는 putdown()을 수행하여 나의 상태를 THINKING으로 바꿈
- 다 먹었다는 것을 양옆에 알리기 위해서 test를 내 양옆의 사람에 대해 수행
- 나는 EATING 상태가 아니라서 test의 나머지 조건만 충족 시 배고픈 사람은 음식을 먹을 수 있음

III. 특징:

- 각 함수의 critical section은 wait (mutex)/signal (mutex)를 통해 보호됨
- Counting semaphore s[N]을 통해 공유자원 동시 접근 수를 N(철학자 수)로 제한

[Chapter 8]

소프트웨어융합학과/2020105742/한지훈

11. Deadlock
   A. 개요: 풀리지 않는 교착 상태로, 공유자원 오염 위험이 있음
   B. 원인: 동기화 문제를 해결하려는 과정에서 발생하며, 심각한 문제이기 때문에 해결이 필요함
   C. 발생 조건: 아래 4가지 조건이 동시에 만족되면 Dead lock이 "발생할 수도" 있음
      I. Mutual exclusion: 동기화 문제 해결이 전제가 되기 때문에 대부분 참이라고 볼 수 있음
      II. Hold and wait: 철학자 예제에서, 왼쪽 젓가락을 hold하는 상태에서 오른쪽 젓가락을 wait하는 상황
      III. No preemption: Hold 하는 상황에서 다른 프로세스가 원할 때 안 내려놓고 있는 상황
      IV. Circular wait: 철학자 예제에서, 1번도 오른쪽 기다리고 2, 3,…n번째가 모두 오른쪽 기다리는 상황
   D. 해결법: 4가지 조건 중 1개라도 만족하지 않으면 deadlock 절대 발생하지 않음

12. Handling Deadlocks
   A. Deadlock prevention: 개발자가 설계 단계에서 4개중 1개를 거짓으로 만들어 사전에 방지
      I. 특징: 가장 바람직한 방법이며, 4가지 조건이 존재하는 이유라고 할 수 있음
   B. Deadlock avoidance: 어플이 돌아갈 때 OS가 사전에 deadlock 발생할 수 있는지 모니터링
      I. 특징: 커널이 계속 모니터링을 하기 때문에 커널의 오버헤드가 커서 잘 쓰지 않음
   C. Deadlock detection and recovery: 모니터 오버헤드 줄이기 위해 일단 돌리고, 사후에 deadlock 발생하면 풀기
      I. 특징: detection, recovery 작업이 어렵기 때문에 결코 커널의 오버헤드가 작지 않아 잘 쓰지 않음
   D. Deadlock ignorance: Deadlock 그냥 무시하기
      I. 특징: 제일 안 좋은 방식이며, 시스템의 무결성이 깨지기 때문에 말이 안됨

13. Dining Philosopher – 공유자원: 젓가락
   A. Deadlock 조건 만족 여부
      I. Mutual exclusion: 존재(왼쪽/오른쪽 젓가락에 대해 wait/signal을 걸어주기 때문에)
      II. Hold and wait: 존재(왼쪽 젓가락을 쥔 상태에서 오른쪽 젓가락을 기다리기 때문에)
      III. No preemption: 존재(젓가락을 내려놓는 코드가 존재하지 않기 때문에)
      IV. Circular wait: 존재(모든 철학자가 오른쪽 젓가락 기다리는 상황이 발생 가능하기 때문에)
         - 4가지 조건이 모두 만족되었기 때문에 deadlock이 "발생할 수도" 있음 > 배제할 수 없음

14. Dining Philosopher – 공유자원: 철학자들의 상태값
   A. 기존 방식과 비교
      I. 젓가락을 hold하고 wait하는 것이 아니라 test를 통해 3명의 상태를 논리연산으로 동시에 확인
         - 즉 공유자원을 젓가락이 아닌 철학자의 상태값으로 설정했기 때문에 Hold and wait가 거짓이 됨
         - 1개 이상의 조건이 만족되지 않았기 때문에 Deadlock이 절대 발생하지 않음