운영체제 7장 (Synchronization Examples)
-H/W, M/W, OS 등에서 제공하는 동기화 기능을 어떤 프로그램에서 사용할 수 있는지 예제

1. Bounded-buffer와 Readers-Writers, dining philosophers synchronization problem (=동기화 문제에 유명한 예시) 설명
2. Linux와 Windows 수준(OS level)에서 사용되는 도구들(Mutex, Semaphore 등)이 동기화 문제 해결 방법 묘사
3. POSIX나 Java 수준(API level)에서 어떻게 동기화 문제 처리하는지 설명

#3가지 AP models
1. Bounded-buffer와 Readers-Writers, dining philosophers synchronization problem
(1) Bounded-buffer Problem(=Producers-Consumers)
-사용하는 변수값
n buffers(n개의 버퍼 공간, 개당 1개의 item 보유 가능)
*Semaphore 여러 개 사용 가능하며 기능이 다양함
mutex라는 Semaphore 사용(P-C가 버퍼 사용하는 독점권 부여 위함) >> 1로 초기화 이유?
*(mutex 용도가 아닌) Condition 용도로 Semaphore 사용(full, empty)
full이라는 Semaphore 사용(특정 Condition 기다리다 Signal하는 방식)
empty라는 Semaphore 사용

-Producer Process 구조
생산한 item들을 버퍼에다 가져다 놓는 것(독점권 부여 필요)
#Producer의 wait(empty)를 Consumer의 Signal(empty)를 통해 깨움

-Consumer Process 구조
버퍼 내 존재하는 item 하나를 가져감(독점권 부여 필요) > 가져간 item은 버퍼에서 삭제
#Cosumer의 wait(full)를 Producer의 signal(full)을 통해 깨움

(2) Readers-Writers Problem
Data set이 많은 동시에 발생하는 프로세스 사이에서 공유됨
▶Readers : 오직 Data set을 읽을 수 있음(공유 Data에 여러 reader가 동시에 접근 가능)
▶Writers : 읽기와 쓰기 둘 다 가능함(공유 Data에 오직 하나만 동시 접근 가능)

-공유 데이터 종류
Data set
Semaphore **rw_mutex** (1로 초기화) : 어느 Data에 대한 R/W 독점권 표시
*처음 들어가는 Reader는 R/W mutex를 얻어 독점권을 가지고, 다른 Reader가 존재한다면 (=마지막 Reader가 나올 때까지), 이 독점권을 잡고 있어야 함.
Semaphore **mutex** (1로 초기화) >> read_count 증가/감소에 대한 access 제한(독점권)
Integer **read_count**(0로 초기화) >> 언제 독점권을 얻고 놓아야 하는지 알려줌

-Writer Process 구조
Shared Data에 대한 쓰기를 진행할 때, 독점권을 얻는 것
#Writer의 wait(rw_mutex)를 Reader의 Signal(empty)를 통해 깨움

-Reader Process 구조
Shared Data에 대한 읽기는 진행하는 것
*처음 들어가는 Reader인 경우 R/W 독점권 얻고, 마지막 빠져나오는 Reader는 독점권 놓음
*read_count에 대해 wait(mutex)와 signal(mutex)로 보호함
#Reader의 wait(empty)를 Writer의 Signal(rw_mutex)를 통해 깨움

-Readers-Writers Problem의 변수[Starvation]
① Writer가 기다리지 않는 한 어떤 Reader도 대기하지 않음
② Writer가 준비된다면 가능한 한 빨리 Write할 수 있도록 수행해야 함
▶ 이 두 가지 문제를 해결하기 위해
✓ 대부분의 OS(kernel)에서 제공하는 reader-writer locks을 통해 이 문제를 해결
(semaphore를 통해 프로그래머가 해결하는 것이 아니라)

(3) Dining philosophers synchronization problem
철학자들은 생각(thinking)하고 식사(eating)하는 두 가지 행동 반복함
젓가락 2개가 있어야 eating할 수 있음

-조건
전체 철학자는 5명이며, 젓가락 또한 5개 존재함
철학자는 자신 양 옆의 젓가락 2개가 모두 있어야 eating할 수 있음(아니면 thinking)

-공유 데이터 종류
Data set(Bowl of rice)
Semaphore chopstick [5](1로 초기화)

-Philosopher i의 Process 구조
```
while (true){
wait (chopstick[i] ); // 0번 철학자는 0번, 1번 젓가락을 집어야 함
wait (chopStick[ (i + 1) % 5] ); // 4번 철학자의 경우도, 4번. 0번 젓가락을 집어야 함
/* eat for awhile */
signal (chopstick[i] );
signal (chopstick[ (i + 1) % 5] );
/* think for awhile */
}
```

✓ 모든 철학자는 왼쪽 젓가락 집은 후, 오른쪽 젓가락을 집으려 할 때 Deadlock 발생 가능

#해결 방법

① 철학자 수 조정

② 철학자가 젓가락을 잡는 매커니즘 변경

ex. 한 철학자는 왼쪽 먼저 잡고, 다른 철학자는 오른쪽 먼저 잡도록

ex. 한쪽만 젓가락을 잡고 대기하지 못하도록 설정

-Monitor 이용한 Dining Philosophers Process 구조

젓가락이 아닌 철학자들의 상태(State)에 집중하여 문제 해결

pickup(), putdown() 함수 작성을 통해 해결

Deadlock 문제는 발생하지 않으나, Starvation은 발생 가능함

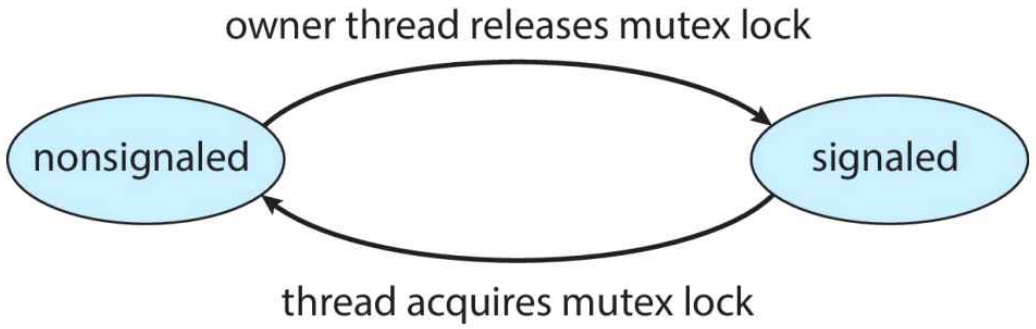
```
monitor DiningPhilosophers
{
        enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5]; //5명의 철학자가 지금 기다리는 상태인지 아닌지 판단

        void pickup (int i) {
                state[i] = HUNGRY;
                test(i);   //젓가락이라는 리소스를 사용할 수 있는 상태인지 판단
                if (state[i] != EATING) self[i].wait;
        }
        void putdown (int i) {
                state[i] = THINKING;
                // test left and right neighbors
                test((i + 4) % 5);          //왼쪽, 오른쪽 철학자들의 상태 체크
                test((i + 1) % 5);
        }
        void test (int i) {
                if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) &&
                (state[(i + 1) % 5] != EATING) ) {
                        state[i] = EATING;
                        self[i].signal (); //i가 그동안 wait 중이었으면 신호 보냄
                }
        }
        initialization_code() {
                for (int i = 0; i < 5; i++)
                        state[i] = THINKING;
        }
}
```

2. Linux와 Windows 수준(OS level)에서 사용되는 도구들(Mutex, Semaphore 등)이 동기화 문제 해결 방법 묘사

(1) Kernel Synchronization - Windows
-interrupt masks : 단일 프로세서 시스템에서 글로벌 리소스에 대한 액세스를 보호
*단일 프로세서 시스템에서 유용하게 사용, 멀티 프로세서 시스템에서는 interrupt disable 진행할 때, 다른 CPU까지 이를 적용시켜야 하므로 확장성에 문제 생김

-spinlocks : 싱글/멀티프로세서 시스템에서 사용, 스핀 잠금 스레드는 절대 우선되지 않음

-dispatcher objects : 뮤텍스, 세마포어, 이벤트 및 타이머를 수행 가능
Events : 앞서 이야기했던 Condition 변수처럼 작동함
Timer : timer 만료 시, 하나 이상의 thread에게 통지함
signaled-state(자원이 이용가능함), non-signaled state(자원이 이용 불가하여 대기해야 함)



(2) Linux Synchronization
2.6과 그 후부터, fully preemptive 사용함 <C.S 보호는 진행해야 하나 나머지 부분에 대해>
(non-preemptive일 때면, C.S 부분을 다 수행할 때까지 interrupt를 받지 않으므로, 동기화 문제가 자연스럽게 해결됨 but 더 우선순위 높은 것이 기다려야 함)
-리눅스가 제공하는 것
Semaphores
atomic integers // 이 값을 변경할 때, C.S으로 보호를 해줌
spinlocks
reader-writer versions of both

-단일 CPU 시스템에서, spinlock은 커널 선점 활성화 및 비활성화으로 대체됨

(3) POSIX(API의 표준, 조금 더 상위 level) Synchronization
-POSIX API가 제공하는 것(UNIX, Linux, macOS에서 사용됨)
mutex locks
semaphores
condition variable

-POSIX Mutex Locks
pthread mutex init(&mutex, NULL); - mutex 초기화
pthread mutex lock(&mutex); - mutex에 대한 사용권 얻음
pthread mutex unlock(&mutex); - mutex 사용권 해제

*named와 unnamed 세마포어 제공
-POSIX Named Semaphores
서로 상관관계 없는 proc끼리도 사용 가능
#sem = sem open("SEM", O_CREAT, 0666, 1 ←세마포어 초기값);
//다른 proc 이용가능(semaphore 이름), SEM semaphore 없을 시, 새로 생성(O_CREAT)

#sem wait(sem);, sem post(sem);를 C.S 위 아래에 집어넣음으로써 C.S 보호함

-POSIX Unnamed Semaphores
서로 상관관계 있는 proc끼리 사용 가능
#sem init(&sem, 0, 1);

#sem wait(sem);, sem post(sem);// 마찬가지로 C.S 보호 위해 이를 사용

-POSIX Condition Variables
mutex lock과 condition variables을 동시에 사용할 수 있음
#pthread mutex init(&mutex, NULL);

#pthread cond init(&cond var, NULL);

#실제 Condition Variables 사용 사례 p.26
condition a == b 까지 대기하다,
다른 thread가 pthread_mutex_lock(&mutex);과 pthread_mutex_unlock(&mutex); 사이
a=b 로 초기화하고 signaling해줄 때 작동함

(4) Java Synchronization(특히, 모니터 기능을 추가로 제공함)
Java provides rich set of synchronization features:
-Java monitors
메소드를 synchronized로 선언하면, 그 함수는 마치 모니터에 선언해놓은 함수처럼 이용 O
(=synchronized 메소드를 Thread가 call하면 전체 obejct의 다른 thread는 그것 이용 불가)

#대기 형태
①Monitor 밖에서 대기하는 entry set(Object 사용권 기다림)
②Wait Condition을 대기하는 wait set(Object의 메소드 call하다가 기다림, Object 내 존재)
✓ wait() - notify() 함수를 통해 대기 및 깨우기 가능

```
#Bounded Buffer 구현 예시
public BoundedBuffer() {
        count = 0;      //item의 개수
        in = 0; //producer 나 Consumer 가 놓거나 가져가야할 데이터
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE]; //처음에 비워져 있음
}
/*Producers call this method*/
public synchronized void insert(E item){ //Producer의 역할
        while (count == BUFFER SIZE){ //pool이 꽉 차있다
                try{
                        wait();
                }
                catch (InterruptedException ie) { }
        }
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        notify(); //remove()의 wait()에게 item이 생성되었음을 전달함
}

/*Consumers call this method*/
public synchronized E remove() { //Consumer의 역할
        E item;
        while (count == 0) {     //buffer가 비어있음
                try {
                        wait();
                }
                catch(InterruptedException ie) { }
        }
        item = buffer[out]; //out point에서 item 하나 들고옴
        out = (out + 1) % BUFFER_SIZE;
        count--;
        notify(); //빈공간 만들어, insert()의 wait()에게 버퍼 공간 하나 비워졌음을 전달함
        return item;
}
```

-Reentrant locks (mutex lock과 유사한 기능)
Lock key = new ReentrantLock();을 통해 key를 생성하고, lock(), unlock()을 통해 그 사이 C.S 보호 가능(lock을 한 이후, 오류가 나도 unlock 수행하도록 finally 이용함)

-Semaphores
#생성자
Semaphore(int value);
#사용방법
Semaphore sem = new Semaphore(1);
acquire(), release()를 통해 C.S 보호 가능(finally 이용)

-Condition variables
#Java Condition Variables 사용
Condition variables는 ReentrantLock과 관련되어 있음.
newCondition() 사용하여 condition variable 생성함
	Lock key = new ReentrantLock(); //키 생성
	Condition condVar = key.newCondition(); //키에다 새로운 조건 생성하여 추가함
쓰레드는 **await()과 signal()을** 통해 대기하고 신호 줌

#예시 코드 (doWork())
```
public void doWork(int threadNember){
	lock.lock();
	try {
		if(threadNumber != turn)
			condVars[threadNumber].await();
		/*Do some work for awhile...*/
		/*Now signal to the next thread.*/
		turn = (turn + 1) % 5;
		condVars[turn].signal();
	}
	catch (InterruptedException ie) { }
	finally {
		lock.unlock();
	}
}
```

(5) Alternative Approaches - 다른 대안
-Transactional Memory
이를 선언해놓으면, 그 안에 있는 작업을 Race Condition 없이 Atomic 하게 사용 가능
(atomic { /*내부에 선언한 것들은 atomic하게 수행됨*/ })

-OpenMP (Transactional Memory와 유사한 형태)
(#pragma omp **critical** { /*①*/})
① 내부에 선언한 것들은 atomic하게 수행됨
② #pragma omp **parallel**하면, 내부 선언을 core 수만큼 thread 만들어 병렬처리도 가능

-Functional Programming Languages
Recursion을 기반으로 하는 프로그래밍 기법
특징 : 공유 데이터가 존재하지 않으며, 함수의 입력만 가지고 프로그래밍을 수행함
✓ C.S을 자동으로 보호 가능(공유 데이터가 없으니)