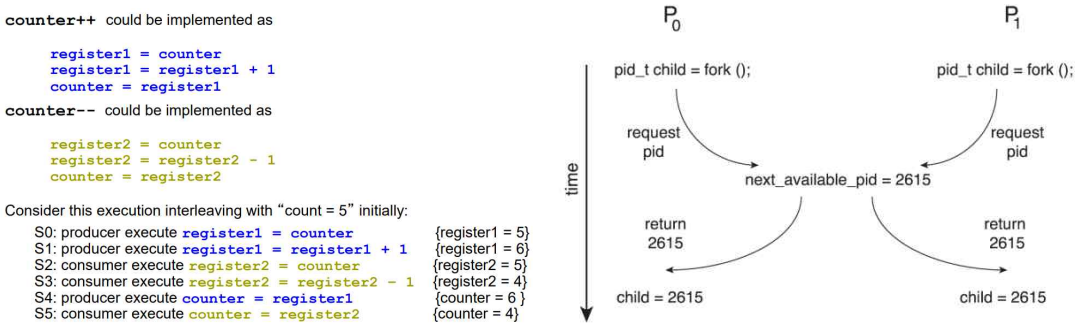
운영체제 6장
Synchronization Tools
각 프로세스, CPU, core간 동기화
0. Background
동기화 기본 개념
1. The Critical-Section Problem
동기화 개념 내 중요한 부분인 Critical-Section 소개
2. Peterson's Solution
Critical-Section 해결 방법 중 S/W만 사용하는 부분
3. Hardware Support for Synchronization
Critical-Section 해결 방법 중 H/W만 사용하는 부분(일반 AP programmer 사용 어려움)
4. Mutex Locks
5. Semaphores
H/W만 사용한 해결 방법 이용 어려움으로 인해, OS나 라이브러리 API에서 H/W 기능을 이용해 프로그래머가 쉽게 동기화 구현 가능하도록 만듦
6. Monitors
일종의 M/W, Programming에서 자동으로 Critical-Section 보호할 수 있는 쉬운방법 제공
7. Liveness
프로그램이 계속해서 진행되어야 한다는 것(동기화에서 해결해야 할 문제)
8. Evaluation

목표
-Describe the critical-section problem(임계영역) and illustrate a race condition
-Illustrate hardware solutions to the critical-section problem using (memory barriers, compare-and-swap operations, and atomic variables)
-Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
-Evaluate tools that solve the critical-section problem in low-. Moderate-, and high-contention scenarios

0. Background
Processes can execute concurrently
proc나 쓰레드가 동시에 수행할 수 있으므로(성능↑, 프로그램 모듈화) 동기화의 필요성 존재
이 동시 수행으로 인해 다양한 문제 발생
-data inconsistency
모든 노드, proc들이 어떤 Shared-data에 대해 Read-Write하는 순서에 대해 전체가 합의를 보는 것(data consistency), 이를 위해서는 Critical-Section 보호 필수
-data inconsistency에 대한 (Producer - Consumer) 예제
Producer는 queue에 data를 생산(counter 증가)하고 Consumer는 data를 소비(counter 감소)함

#Producer
while (true) {
/* produce an item in next produced */
while (counter == BUFFER_SIZE) ; /* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
#Consumer
while (true) {
while (counter == 0) ; /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--; /* consume the item in next consumed */
}
-Race Condition



될 때도 있고 안될 때도 있는 엉망인 상태가 되는 것
ex1. 파란색-노란색의 상대적인 진행 속도에 의해 기대하는 값을 벗어난 다른 값을 갖게 됨
ex2. fork() 시, Child proc에 pId 부여 과정에서 관리하는 pId 값 동시에 접근하면 같은 pId 부여하는 오류 발생 가능
*Critical Section Problem (여러 instruction, 수행 단계 등이 겹치면 엉망이 됨)
Critical Section : 공유 데이터(변수, table, file 등)에 대해 두 개 이상의 proc가 동시에 들어가 access(Write)할 때, 문제가 되는 민감한 부분
Critical Section의 독점권을 행사하게 함으로써 Race Condition 방지 가능

1. The Critical-Section Problem
(1) Critical-Section Problem의 해결책(어쨌든 Critical-Section를 공유해야 함)
3가지 조건을 만족해야 함
① Mutual Exclusion - 독점권 행사(부여)
② Progress - 아무도 그 내부에 존재하지 않을 때, 들어갈 수 있어야 함
③ Bounded Waiting - Critical-Section에 누가 있는 경우 기다리지만, 그 시간에 한정이 있어야 함
✓ 이 모든 조건을 상대적 속도에 대한 고려 등의 가정 없이 적용되어야 함

(2) 이전에 배웠던 커널의 preemptive, non-preemptive 두 가지 접근법
* Preemptive : 어느 proc가 CPU를 차지하여 동작하더라도 더 우선순위 높은 것이 그 CPU 사용권 빼앗을 수 있음
* Non-preemptive : 어느 proc가 CPU 차지하여 동작 시에 그 CPU 할당 종료(커널 모드를 빠져나갈 때, block 상태일 때, 스스로 CPU 양보할 때)까지 다른 proc에서 사용권 가져올 수 없음 >> 필연적으로 커널 모드에서 race condition으로부터 자유로움(잘 보호됨)

2. Peterson's Solution
Critical-Section을 잘 보호하기 위한 순수한 S/W적인(OS, H/W 도움 X) 방법 중 하나
두 개의 변수(turn: 어떤 proc가 사용할 차례인지, flag[2]: 각 proc가 Critical-Section에 들어가고 싶은지 확인)를 공유하는 두 프로세스(or 쓰레드)가 있다고 가정

-Algorithm for Process Pi(Pj인 경우 i와 j를 뒤바꾸면 됨)
```
while (true){
flag[i] = true;
turn = j; //turn = i라 설정하면, 나중에 수행되는 j가 turn값을 overwrite하면서 j가 늦게 수행되었음에도 j가 사용할 차례가 된다는 것은 불공정(상대방 차례임을 명시하는 것이 공정)
while (flag[j] && turn == j) // i의 입장에서 j보다 한 발 늦은 것
        ;        //j의 flag가 false되거나 i turn이 되는 것을 무한 대기
// 이 조건에서 벗어났다는 것은 i가 critical section에 들어가도 된다는 것
/* critical section */
flag[i] = false; //i process가 critical section 내 활동을 모두 수행한 상태
/* remainder section */
}
```

- Peterson's Solution 문제점(CPU나 compiler 개입 시 정상 동작 불가능)
일반적인 경우, 제대로 동작하나 CPU나 compiler가 optimization을 위해 Peterson's Solution program code의 수행 순서 바꿀 수 있음(이때, multithread는 제대로 동작 X - race condition 발생)

3. Hardware Support for Synchronization
하드웨어를 통한 동기화 방법
(1) disable interrupts(Uniprocessors를 통해)
interrupts를 disable 시켰으므로, 해당 prog 수행 중 자신이 독점권을 가지므로 C.S을 계속해서 수행할 수 있음
다 수행 후에는 enable interrrupt 설정 해야 함

(2) 그 밖의 세 가지 형태의 hardware support
a. Memory barriers

b. Hardware instructions
c. Atomic variables

a. Memory Barriers
Memory model은 컴퓨터 구조가 AP을 실행할 수 있도록 보장하는 메모리
-Memory models 두 가지 존재(Multiprocessor, computer 환경에서 데이터 공유 시, 쓰는 전략)
➢ Strongly ordered
한 CPU가 Write하면, 다른 CPU가 즉시 확인할 수 있어야 함
(간단하고 문제 없겠지만, 성능에 악영향을 미침)
➢ Weakly ordered
한 CPU가 Write해도, 즉시 전파할 필요가 없음(문제가 생기지 않을 때까지 전파 X)
ex. 어떤 data를 업데이트 했는데, 다른 CPU나 컴퓨터에서 이를 사용하지 않을 경우

-memory barrier
Weakly ordered의 일종, memory barrier를 선언해놓는 그 순간까지는 업데이트를 지연시켜도 된다는 것(최대 연장 가능 지점 표기)

b. Hardware Instructions
H/W instruction은 atomically(중간에 interrupt를 받지 않으면서)하게 일어남
(Instruction Cycle, 한 Instruction의 Cycle에서는 INT가 걸리지 않음, 수행 후 INT 체크)
Test-and-Set이나 Compare-and-Swap같은 복합적인 instruction을 컴퓨터가 한 instruction으로 모아놓은 기능을 제공한다면 이런 instruction은 atomically 가능함
✓ 이것을 통해 C.S 보호 가능

①test_and_set Instruction (이 역할을 하는 Machine Instruction이 따로 존재한다는 의미)
Definition:

```
boolean test_and_set (boolean *target) {
        boolean rv = *target;   //target value를 return하는 함수
        *target = true; //target에다 true값을 항상 집어넣음 - 0이면 C.S이 비어있음
        return rv:
}
```

>> 따라서 atomically하게 수행 가능하다

```
// Test_and_set을 통해 C.S 보호 가능
do {
        while (test_and_set(&lock)) //lock == 0 이면 C.S이 비어있음
                ; /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
```

} while (true);

②compare_and_swap Instruction
Definition:
//value == 0 이면, C.S 비어있음
int compare _and_swap(int *value, int expected, int new_value) {
int temp = *value;
//expected value가 자신이 원하는 값이면 C.S내에 들어갈 수 있으며, value 값에 new_value 채워 넣을 수 있음
if (*value == expected)
*value = new_value;
return temp;
}
>> 이것 또한 한 instruction으로 되어 있어, atomically하게 수행 가능하다

// compare_and_swap을 통해 C.S 보호 가능
while (true){
while (compare_and_swap(&lock, 0, 1) != 0)
//lock == 0이면, C.S 비어있고 C.S 내 들어갔음을 표시하기 위해 lock = 1 넣음
; /* do nothing */
/* critical section */
lock = 0;
/* remainder section */
}

+ compare_and_swap을 Bounded-waiting Mutual Exclusion에 사용할 수 있음
여러 개의 노드들이 존재하는데, 사용권을 하나씩 돌아가면서 C.S에 들어갈 수 있도록(Token passing 같은 느낌)

while (true) {
waiting[i] = true;
key = 1; //어떤 노드가 들어가 있다고 가정한 것
while (waiting[i] && key == 1)
  key = compare_and_swap(&lock,0,1);
waiting[i] = false;
/* critical section */
j = (i + 1) % n; //우선 순위 낮은 노드가 수행하지 못할 것을 방지하여 넘어가도록 함
while ((j != i) && !waiting[j])
  j = (j + 1) % n;
if (j == I) //waiting하는 j를 발견했으면, lock을 0으로 풀고 j에게 사용권 줌
  lock = 0;

```
else
        waiting[j] = false;
        /* remainder section */
}
```

c. Atomic Variables
어떤 값에 대해서 변화를 시키는 동작이 atomic하게 이루어지는 변수

```
#Atomic Variables
void increment(atomic_int *v)
{
        int temp;
        do {
                temp = *v;
        }
        //v값을 temp에 넣고 v == temp면, temp를 하나 증가시켜 v에다 넣음
        (v == temp가 아니면 그 사이에 누군가가 끼여 값에 변화를 일으켰다는 것)
        while (temp != (compare_and_swap(v,temp,temp+1));
}
```

4~5. Mutex Locks (목적 :자원에 대한 접근 동기화)
H/W에서 제공하는 것을 이용한 쉬운 API(H/W에서 제공하는 것이 속도가 빠르지만, 이용 복잡하고 실수하였을 경우, 시스템에 문제 생길 수 있음)

(1) Solution to Critical-section Problem Using Locks(OS에서 제공)
-Mutex Lock
가장 간단한 software tool, Locking 메커니즘으로 락을 걸은 스레드만이 C.S(임계영역)를 나갈 때 락 해제 가능(C.S 진입 시 acquire(), 나갈 때 release(), 호출은 atomic 해야 함)
*acquire(), release()은 test-and-set, compare-and-swap 등의 H/W의 atomic instruction을 이용해서 Mutex(자원 독점권 부여 용도) 구현 가능

-Spin Lock
C.S에 진입 불가능할 때 진입이 가능할 때까지 루프를 돌면서 재시도하는 방식으로 구현된 락(임계 구역 진입 전까지 루프를 계속 돌고 있기 때문에 busy waiting 발생)

-Semaphore[Semaphore S - integer variable]
Mutex보다 상위 개념, 음수가 아닌 정수값을 통해 쓰레드 간 공유되는 변수(가용 자원 개수 설정)이고 C.S 문제 해결 및 동기화 구현에 사용됨 (wait 연산과 signal 연산 가지고 있음)
signaling 메커니즘으로 락을 걸지 않은 스레드도 signal을 사용해 락을 해제할 수 있음
@유형
a. 개수 세마포어 (Counting semaphore) - 세마포어의 일반적인 기능

도메인이 0이상인 임의 정수값인 세마포어(세마포어 값을 증감하여 자원 개수 표현 가능)
여러 개 자원을 가질 수 있으며 제한된 자원으로 액세스 작업 시, 사용
b. 이진 세마포어 (Binary semaphore)
0 또는 1 값만 가질 수 있는 세마포어(Mutex와 동일한 개념 -C.S 사용하는지 안하는지 확인)
자원이 하나이기 때문에 뮤텍스로도 사용 가능
@구현
init(), wait(), signal() 함수 구현[semaphore는 busy-waiting, no-busy-waiting버전 존재]

★Couting semaphore가 동기화 응용에 사용할 수 있는 사례

#자원이 없을 경우 대기하는 방식
① Busy waiting 방식
원하는 자원을 얻기 위해 기다리는 것이 아니라 권한을 얻을 때까지 확인하는 것(C.S 들어가려고 경쟁, CPU 사용권 스스로 놓지 않음)
-장점
자원의 권한 획득에 많은 시간이 소요되지 않는 상황인 경우(C.S로 진입하려는 proc가 별로 없을 때)나 Context Switching 비용보다 성능적으로 더 우수한 상황인 경우, 효과적임
-단점
권한 획득을 위해 많은 CPU 낭비

② Sleeping 방식 (no-busy-waiting 방식, CPU 사용권 스스로 놓음)
권한 얻기 위해 기다리는 시간을 wait queue에 실행 중인 Thread 정보 담고 다른 Thread에게 CPU를 양보하는 것 (본인 C.S에 진입 못할 시, queue나 list에 넣어 wait() 상태 빠짐)
커널은 권한 이벤트가 발생하면 wait queue에 담긴 Thread를 깨워 CPU를 부여함
@연산
block() : 할당받을 자원이 없어 어느 자료구조에 들어가 sleep()에 빠지는 것
wakeup() : 할당 가능 자원이 생기면, block()상태에 들어간 것을 깨움
-장점
기다리는 시간이 예측이 불가능한 상황인 경우, 효과적임
-단점
wait queue에 넣는 비용 + Context Switching 비용이 듦

#Semaphore with no Busy waiting 구현

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) { // ex.S->value == -3이면, 3개의 proc가 리스트에서 대기 중
                add this process to S->list;
                block();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Semaphores의 문제점

-AP 프로그래머의 잘못된 사용 가능

✓ 더 쉬운 무엇인가가 M/W, language 등에서 제공되는 것이 좋을 듯

#M/W, language 등에서 제공하는 더 쉬운 것

6. Monitors

M/W에서 제공하며, monitor를 선언하고 C.S이 필요한 prog를 그 내부에서 작성하는 것

하나의 proc만 monitor 내 함수를 수행할 수 있음<여러 개가 대기 가능>

동기화 되고 있는 데이터, lock(=monitor lock), 몇 개의 condition variable로 구성

(1) 대기 대상

-Monitor 사용권 기다림

-Monitor 내 x,y condition 변수 기다림

(2)Condition Variables(condition x, y;)

특정 조건을 만족하기를 기다리는 변수이며, 우리가 C.S나 monitor에 진입하려고 할 때 확인하는 조건뿐만 아니라, 다양한 조건을 기다릴 수 있는데 이를 위해 사용

ex. P-C 관계에서 생산자가 공유 자원 내에 공간이 꽉 차있으면 더 생성 불가

-연산

wait(), signal() 두 연산만이 condition 변수에 허용됨

x.wait() : 이 함수를 호출한 프로세스는 대기

x.signal() : 대기 중인 프로세스가 있다면 x.wait()를 호출한 프로세스 중 하나가 실행됨. 대기 중인 프로세스가 없다면 아무 일도 일어나지 않음

*Semaphore의 signal()이 항상 S++을 하여 그 상태에 영향을 주는 것과는 다름

>> Signal 보낸다는 것은 해당 프로세스가 Lock을 얻었음을 의미

실행 중인 P proc와 대기하고 있는 Q proc 가 있다고 하자

어떤 Condition x에 대해 Q는 wait하고 있고 p는 signal을 주는 관계임

#proc P가 x.signal() 호출해 프로세스 Q 깨우면, P와 Q는 모니터 안에서 동시에 활성화됨

(3)signal 준 후 Monitor 내 존재하는 proc 중 누가 수행될 것인가? 하는 옵션 두 개 존재

>> 대기 대상 하나에게만 Monitor 사용권을 물려주어야 함

-Signal and wait : signal 주고, 내가 대기함으로써 기다리던 다른 프로세스 먼저 수행

-Signal and continue : signal 주고, 내가 먼저 실행된 후, sleep()하여 다른 프로세스 수행

#모니터 내 프로세스 재개 순서(signal() 호출 시 어떤 프로세스가 실행될지는 구현에 따름)
>> 대기 proc 중 누구를 깨우느냐에 대한 여러 방법 존재
-FCFS 전략 : 종종 타당하지 않을 수 있음
-conditional-wait (x.wait(c)으로 구성됨) : 프로세스의 우선순위(Priority) 고려 전략
등

#Semaphore을 사용한 Monitor 구현(Monitor 사용권 기다림 예시)
변수[semaphore mutex; , semaphore next; , int next_count = 0;]

```
wait(mutex);      //monitor 사용권 잡는 동작
…
body of F(Monitor에 있는 함수;
…
if (next_count > 0)       //어떤 proc가 monitor에 들어오기를 기다림
        signal(next)      //어떤 proc에게 signal을 주면서 모니터 사용권 넘김
else
        signal(mutex);    //monitor 사용권을 기다리는 것이 없을 때, 그냥 사용권 놓음
```

#Condition Variable을 사용한 Monitor 구현
변수[semaphore x_sem;, int x_count = 0;]
@The operation x.wait()(이것 호출 - 이미 사용권 가지고 있었다는 것)는 이렇게 구현 가능

```
x_count++;
if (next_count > 0)
        signal(next);     //monitor 기다리는 것이 O면, 자신 wait하고 그에게 사용권 넘김
else
        signal(mutex);    //monitor 기다리는 것이 없다면 그냥 사용권 놓음
wait(x_sem);      //그리고 자신은 x,y condition 변수 기다리는 장소에서 기다림
//기다리고 있으면 signal(x_sem)이 이 장소에 있는 것들을 깨움
x_count--;
```

@The operation x.signal()

```
if (x_count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);       //signal 받는 proc에게 monitor 사용권 넘김
        next_count--;
```

#Single Resoure allocation
자신이 어떤 자원을 사용하려고 할 때, Maximum 시간을 조정해서 이 시간 동안 자신이 계획된 자원을 t라는 시간 동안 점유하겠다고 지정하는 것
-acquire(), release()

7. Liveness
계속해서 program의 진행(progress)이 수행되어야 한다는 것
동기화를 위해 mutex lock이나 semaphore 통해 대기하고 깨어나는 동안 bounded-waiting(기다리는 시간의 한계 존재) 기준을 위반하지 않아야 함
무한정 대기하는 것은 liveness 조건 위반

(1) Liveness 위반 예시
-Deadlock : 프로그램 더 이상 진행 불가
서로가 서로를 기다리는 것(서로가 상대방이 원하는 다른 자원을 잡고 대기하는 것)

#deadlock의 또 다른 형태
Starvation : 무한정 blocking 당하는 것
우선 순위 낮은 것이 계속해서 우선 순위 높은 것이 도착함으로 인해 지속적으로 기다리는 것

Priority Inversion : CPU 사용권에 대한 우선순위 높은 것이, 접근하고자 하는 자원(ex. C.S 자원 이용, CPU만 이용 등)이 달라 더 낮은 우선순위를 가진 것에게 밀려나는 현상
✓ priority-inheritance(우선 순위 상속) protocol을 통해 해결 가능
우선순위 낮은 것이 C.S같은 자원 사용하여 높은 우선순위 가진 proc 대기하도록 했을 때, 그것의 우선순위를 상속받는다는 것(중간 우선순위 proc가 높은 우선순위 것을 역전 막도록)