

6장. 동기화 도구들 (Synchronization Tools)

순천향대학교 컴퓨터공학과
이 상 정

운영체제

강의 목표 및 내용

□ 목표

- 프로세스 동기화의 기본 개념
- 공유 데이터의 일관성을 보장하는데 적용되는 임계구역 (critical section) 문제를 소개
- 임계 구역 문제에 대한 소프트웨어 및 하드웨어 해결책

□ 내용

- 배경
- 임계구역 문제
- 피터슨의 해결안
- 동기화 하드웨어
- Mutex Locks
- 세마포
- 모니터

배경 (Background)

- ❑ **공유 자료**를 **병행 접근(concurrent access)**하면 **자료의 불일치(data inconsistency)**를 초래
 - 임의의 순간 **인터럽트 발생**으로 **프로세스의 부분적인 실행**
- ❑ **자료의 일관성(data consistency)**을 유지하려면 **협력적인 프로세스들(cooperating processes)**의 **바른 순서로 수행(orderly execution)**을 보장하는 메커니즘이 필요
- ❑ **경쟁 상황(race condition)**
 - 여러 개의 프로세스가 동일한 자료를 접근하여 조작하고, 그 실행 결과가 접근이 발생한 **특정 순서에 의존**하는 상황
 - 경쟁 상황으로부터 보호하기 위해 **한 순간에 하나의 프로세스만이 공유 자료를 조작**하도록 보장하도록 프로세스들을 **동기화(synchronization)**

생산자-소비자 문제 예 (1)

- ❑ **BUFFER_SIZE** 개까지 버퍼에 저장하도록 수정
 - 0으로 초기화되어 있는 **counter**라는 정수형 변수를 추가
 - 버퍼에 새 항목을 추가 시 counter 증가, 삭제 시 counter 감소

❑ 생산자 코드

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER_SIZE) ;
    /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

생산자-소비자 문제 예 (2)

❑ 소비자 코드

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    counter--;
    /* consume the item in next consumed */
}
```

생산자-소비자 문제 경쟁 상황 (1)

❑ counter++ 기계어 수준 명령 구현

```
register1 = counter
register1 = register1 + 1
counter = register1
```

❑ counter-- 기계어 수준 명령 구현

```
register2 = counter
register2 = register2 - 1
counter = register2
```

생산자-소비자 문제 경쟁 상황 (2)

- counter 초기값이 5인 경우 "counter++"와 "counter--" 문장을 병행하게 실행하면 기계어 수준 명령들을 임의의 순서로 뒤섞어 순차적으로 실행

T_0 :	<u>생산자가</u>	$register_1 = counter$ 를 수행	$\{register_1 = 5\}$
T_1 :	<u>생산자가</u>	$register_1 = register_1 + 1$ 을 수행	$\{register_1 = 6\}$
T_2 :	<u>소비자가</u>	$register_2 = counter$ 를 수행	$\{register_2 = 5\}$
T_3 :	<u>소비자가</u>	$register_2 = register_2 - 1$ 을 수행	$\{register_2 = 4\}$
T_4 :	<u>생산자가</u>	$counter = register_1$ 을 수행	$\{counter = 6\}$
T_5 :	<u>소비자가</u>	$counter = register_2$ 를 수행	$\{counter = 4\}$

• 부정확한 결과

- 실제로 5개의 버퍼가 채워져 있지만 4개의 버퍼가 채워져 있는 것을 의미하는 " $counter = 4$ "인 부정확한 상태에 도달
- T_4 와 T_5 의 문장 순서를 바꾸면, " $counter = 6$ "인 부정확한 상태에 도달

임계구역 문제 (Critical-Section Problem)

- n 개의 프로세스 $\{p_0, p_1, \dots, p_{n-1}\}$
- 각 프로세스는 임계구역(critical section)이라고 부르는 코드 부분을 포함
- 이 코드에서 다른 프로세스와 공유하는 변수의 변경, 테이블의 갱신, 파일 쓰기 등의 작업 수행
 - 한 프로세스가 자신의 임계 구역에서 수행하는 동안에는 다른 프로세스들이 그들의 임계구역에 들어 갈 수 없음
- 임계구역 문제는 이 문제를 해결하는 프로토콜을 설계하는 것
- 각 프로세스는 자신의 임계 구역으로 진입하려면 진입 허가 요청을 해야 함
- 요청을 구현한 코드의 부분을 진입 구역(entry section), 임계 구역 뒤에는 퇴출 구역(exit section), 나머지 부분은 나머지 구역(remainder section)이라고 부름.

□ 전형적인 프로세스 P_i 의 일반적인 구조

```
do {
    entry section
    critical section
    exit section
    remainder section
} while (true);
```

□ 임계구역 문제 해결안은 다음의 세 가지 요구 조건을 충족해야 함

1. **상호 배제 (mutual exclusion)**: 프로세스 P_i 가 자기의 임계 구역에서 실행된다면, 다른 프로세스들은 그들 자신의 임계 구역에서 실행될 수 없음
2. **진행 (progress)**: 자기의 임계 구역에서 실행되는 프로세스가 없고, 자신의 임계 구역으로 진입하려고 하는 프로세스들이 있다면 이들 프로세스들 중 임계 구역으로 진입을 선택하고, 이 선택은 무한정 연기될 수 없음
3. **한정된 대기(bounded waiting)**: 프로세스가 자기의 임계 구역에 진입하려는 요청을 한 후부터 그 요청이 허용될때까지 다른 프로세스들이 그들 자신의 임계 구역에 진입이 허용되는 횟수에 제한이 있어야 함

운영체제의 임계구역 관리

- ❑ 임계구역을 다루는 두 가지 일반적인 접근법 사용
 - 선점형(preemptive) 커널
 - 커널 모드에서 실행되고 있는 프로세스가 선점되는 것을 허용
 - 비선점형(non-preemptive) 커널
 - 커널 모드 프로세스가 커널 모드를 종료하거나 봉쇄(block)되거나, 자발적으로 CPU의 제어를 양보할 때까지 선점되지 않고 계속 실행
 - 커널 모드에서 **경쟁 상황(race condition)** 발생하지 **않음**
- ❑ 선점형 커널은 **실시간 프로세스**가 현재 커널에서 실행 중인 프로세스를 선점할 수 있기 때문에 실시간 프로그래밍에 더 적당
 - **SMP**에서는 서로 다른 CPU에 두 프로세스가 동시에 커널 모드에 있을 수 있기 때문에 선점형 커널 설계가 더 어려움

피터슨의 해결안(Peterson's Solution)

- ❑ 고전적인 **소프트웨어 기반 해결책**
 - 두 개의 프로세스로 한정
- ❑ **load와 store 명령이 원자적(atomic)**이라고 가정, 즉 인터럽트가 될 수 없음 (현대 컴퓨터 구조에서는 보장 안됨)
- ❑ 두 프로세스가 **공유하는 자료** 항목
 - int **turn**;
 - boolean **flag[2]**
- ❑ 변수 **turn**은 임계 구역으로 진입할 순번을 나타냄
 - $turn == i$ 이면 프로세스 P_i 가 임계 구역에서 실행
- ❑ **flag 배열**은 프로세스가 임계 구역으로 진입할 준비가 되었다는 것을 나타냄
 - $flag[i]$ 가 true이라면 이 값은 P_i 가 임계 구역으로 진입할 준비가 되었다는 것을 나타냄

피터슨의 알고리즘 - 프로세스 P_i 의 알고리즘

```
do {
```

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

critical section

```
    flag[i] = false;
```

remainder section

```
} while (true);
```

```
do {
```

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

critical section

```
    flag[j] = false;
```

remainder section

```
} while (true);
```

1. 상호 배제(mutual exclusion) 준수
2. 진행 (progress) 요구 조건 만족
3. 대기 시간이 한 없이 길어지지 않음 (bounded waiting)

동기화 하드웨어 (Synchronization Hardware)

- 많은 시스템에서 임계 구역 코드를 지원하는 **하드웨어**를 제공
 - 대부분 **락킹(locking)**에 기반하여 임계구역 보호
- 단일 처리기 환경에서는 공유 변수가 변경되는 동안 **인터럽트 발생을 허용하지 않음**으로써 간단히 해결
 - 현재 실행되는 코드가 선점(preemption)없이 순서적으로 실행
 - 다중 처리기 환경에서는 너무 비효율적
 - 인터럽트의 불능화 메시지를 모든 프로세서에 전달되게 하기 때문에 상당한 시간을 소비
- 많은 현대 기계들은 특별한 **원자적(atomic)** 하드웨어 명령어들을 제공
 - 원자적 (atomic) = 인터럽트 되지 않음 (non-interruptable)
 - 한 워드(word)의 내용을 검사하고 변경 명령어 (test and set instruction)
 - 두 워드의 내용을 원자적으로 교환 (swap instruction)

락(Lock)을 사용한 임계 영역 문제 해결

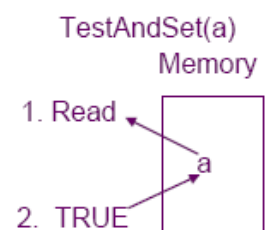
- 프로세스는 임계 영역 진입 전에 반드시 락(lock)을 획득
- 임계 영역 나올 때는 락을 방출

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```

test_and_set 명령어 정의

- 명령어가 원자적(atomicly)으로 실행
 - 전달된 인수의 원래 값을 리턴
 - false 이면 임계영역 진입
 - 인수에 새로운 true 값을 지정
 - 다른 프로세스 임계영역 진입을 방지

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```



test_and_set 명령어를 사용한 상호 배제 구현

- false로 초기화되는 **lock**이라는 공유 **Boolean** 변수를 선언하여 상호 배제를 구현

- lock이 false 이면 임계영역 진입, true이면 대기
- 임계영역 수행 후 다른 프로세스의 진입 허용 위해 lock을 false로 세트

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);

do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
} while (true);
```

compare_and_swap 명령어 정의

- 명령어가 **원자적(atomically)**으로 실행

- 인수 **value**의 **원래의 값**을 리턴
- 인수 **value**의 값이 인수 **expected**와 같은 경우에는 인수 **new_value** 값으로 지정, 즉 **value == expected**인 경우에만 **swap**이 발생

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

compare_and_swap 명령어를 사용한 상호 배제 구현

□ 공유 정수 변수 lock을 선언하고 0으로 초기화

- lock이 expected(0)인 경우 임계영역 진입
 - 다른 프로세스 진입 방지를 위해 lock을 1로 세팅
- 임계영역 수행 후 lock을 0으로 세팅하여 다른 프로세스 진입 허용

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
        /* critical section */
    lock = 0;
        /* remainder section */
} while (true);
```

한정된 대기 조건을 만족하는 상호 배제 (1)

□ test_and_set() 명령어를 사용하고, 한정된 대기 조건을 만족하는 상호 배제

- 한 프로세스 i가 임계 구역 떠날 때 waiting 배열 순회 (i+1, i+2, ..., n-1, 0, ..., i-1)
- 순회 중 waiting [j]가 true인 첫번째 대기 프로세스가 임계 영역에 진입하고 waiting[j]는 false로 지정
- 임계 영역에 진입하고자 하는 프로세스는 최대한 n-1 양보

한정된 대기 조건을 만족하는 상호 배제 (2)

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)    // 대기 중인 프로세스가 없는 경우
        lock = false;
    else          // 첫 번째 대기 중인 프로세스인 경우
        waiting[j] = false;
    /* remainder section */
} while (true);
```

뮤텍스 락 (Mutex Lock)

- ❑ 하드웨어를 기반(test_and_set과 compare_and_swap 명령어 등 사용) 해결안들은 응용 프로그래머가 사용하기에는 복잡
- ❑ 운영체제 설계자들은 임계 구역 문제를 해결하는 소프트웨어 툴을 구축
- ❑ 간장 간단한 방식이 상호 배제(mutual exclusion)를 제공하는 mutex 락
 - 먼저 락을 획득(acquire())한 후에 락을 해제(release())하여 임계 영역을 보호
 - 락의 가용 여부를 나타내는 Boolean 변수 사용
 - acquire(), release()의 호출은 원자적이어야 함
 - 하드웨어 원자적 명령어 사용하여 구현
 - 임계구역 진입 전 바쁜 대기(busy waiting) 사용
 - 이 락을 스핀락(spinlock) 이라고도 함

acquire()와 release()

```

acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);

```

세마포 (Semaphore)

- ❑ 프로세스들을 동기화 시키는 (mutex 락 보다) 좀 더 복잡한 동기화 툴
- ❑ 세마포 S는 정수 변수
- ❑ 두 개의 표준 원자적 연산 wait()와 signal()로만 세마포 접근
 - 원래는 P()와 V()라고 함
- ❑ wait()와 signal() 연산 시 세마포의 정수 값을 변경하는 연산은 반드시 원자적으로(분리되지 않고) 수행



wait()와 signal()의 정의

- 세마포의 값이 0이 되면 모든 자원이 사용 중임을 나타내고 이후 자원을 사용하려는 프로세스는 세마포 값이 0보다 커질 때까지 기다림

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

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

세마포 사용

- 카운팅 세마포 (counting semaphore)
 - 제한 없는 영역(domain)을 정수값
- 이진 세마포 (binary semaphore)
 - 이진 세마포의 값은 0과 1사이의 값만 가짐
 - mutex 락과 동일
- 다양한 동기화 문제들을 해결 가능
- S_1 이 S_2 이전에 수행되어야 하는 프로세스 P_1 과 P_2 예

Create a semaphore "synch" initialized to 0

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

세마포 구현

- ❑ 어떤 두 프로세스들도 동시에 같은 세마포를 가지고 wait()와 signal()을 수행하지 않음을 보장해야 함
 - 따라서 구현은 wait, signal 코드가 임계영역에 놓여야 하는 임계 영역 문제가 된다.
- ❑ 임계영역 구현 시 바쁜 대기(busy waiting)를 사용하는 경우
 - 구현 코드가 짧음
 - 임계영역의 점유가 드물면 바쁜 대기는 거의 없게 됨
 - 많은 시간을 임계영역에 머물러 있는 응용들에는 좋은 해결 방법이 아님
 - 바쁜 대기는 다른 프로세스들이 생산적으로 사용할 수 있는 CPU 시간을 낭비
 - 프로세스가 락을 기다리는 동안 루프를 돌기 때문에 이런 유형의 세마포를 스핀락(spinlock)이라고도 함
 - 짧은 시간 락을 기다리는 경우 문맥 교환이 필요하지 않아 스핀락이 유용

바쁜 대기가 없는 세마포 구현 (1)

- ❑ 바쁜 대기 대신에 프로세스는 자신을 봉쇄하고(block) 프로세스를 세마포에 연관된 대기 큐(waiting queue)에 삽입
 - 프로세스의 상태를 대기 상태로 전환
 - 이 후에 제어가 CPU 스케줄러로 넘어가고, 스케줄러는 다른 프로세스를 실행하기 위하여 선택
- ❑ 대기 큐와 연관된 세마포의 자료
 - 세마포 값을 나타내는 정수 value
 - 대기 큐를 가리키는 포인터 리스트 list
- ❑ 세마포 관련 연산
 - block() 연산
 - 자기를 호출한 프로세스를 중지하고 대기 큐에 삽입
 - wakeup(P) 연산
 - 봉쇄된 프로세스 P의 실행을 재개
 - 대기 큐에서 프로세스를 제거하고 준비 완료 큐에 삽입

바쁜 대기가 없는 세마포 구현 (2)

```

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

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        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);
    }
}

```

교착 상태 (Deadlock)와 기아 (Starvation)

□ 교착 상태 (deadlock)

- 두 개 이상의 프로세스들이, 오로지 대기중인 프로세스들 중 하나에 의해서만 야기될 수 있는 **사건(signal) 연산 실행**을 무한정 기다리는 상황
- 두 개의 프로세스 P_0 과 P_1 에서 1로 초기화된 세마포 S와 Q를 접근하는 시스템 예

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

□ 기아(starvation)

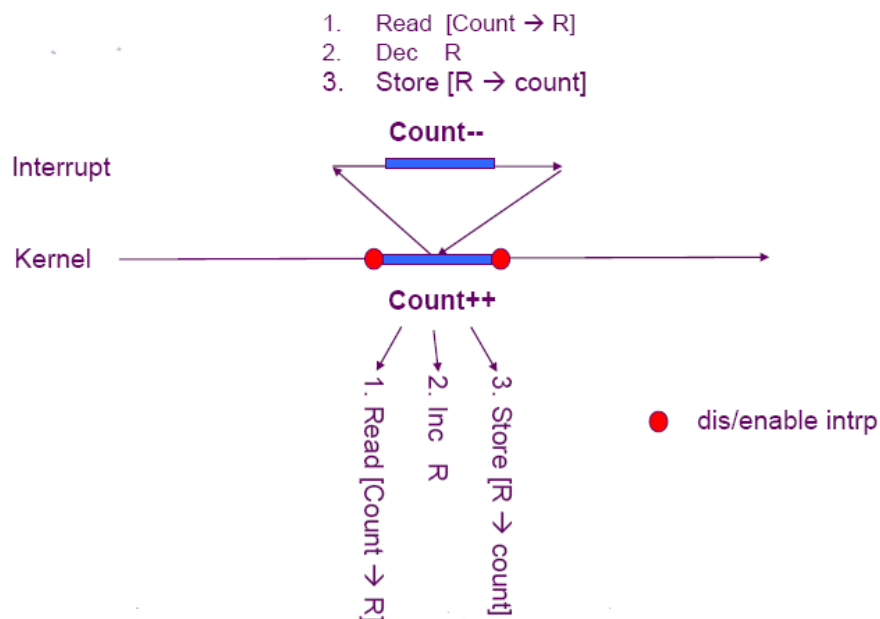
- 교착 상태와 연관된 **무한 봉쇄(indefinite blocking)**
- 프로세스들이 세마포에서 무한정 대기하는 것으로, 프로세스가 중지된 세마포 큐에서 제거되지 않음

임계 구역 문제 발생

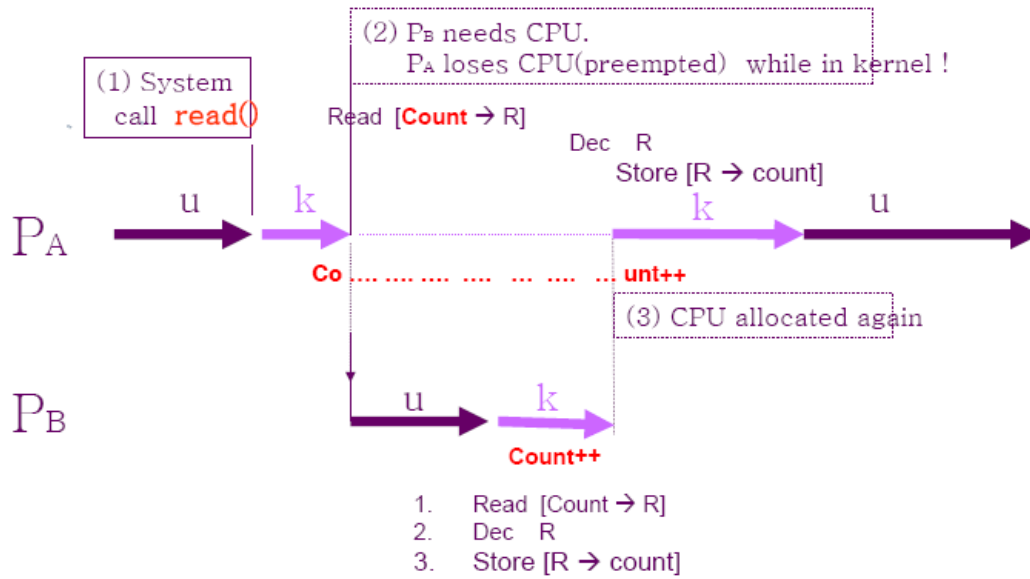
□ 운영체제에서 임계 구역 문제가 언제 발생하는가?

1. 커널에서 인터럽트 루틴 처리
2. 프로세스가 커널에서 선점
 - 시스템 호출 중에 선점
3. 다중 처리기(multiprocessor)
 - 공유 메모리에서 커널 데이터

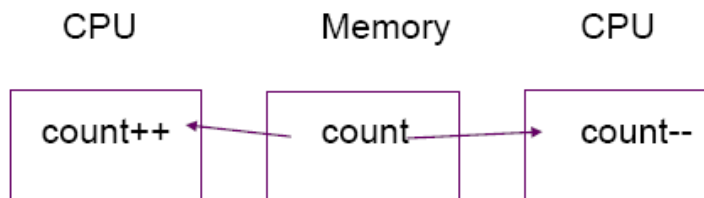
커널에서 인터럽트 루틴 처리



프로세스가 커널에서 선점



다중 처리기



세마포의 문제점

- 세마포를 이용하여 임계 구역 문제를 해결할 때 프로그래머가 **세마포를 잘못 사용하면 다양한 유형의 오류**가 너무나도 쉽게 발생

- 상호 배제 요구 조건을 위반하거나, 교착 상태가 발생
- 아래의 경우

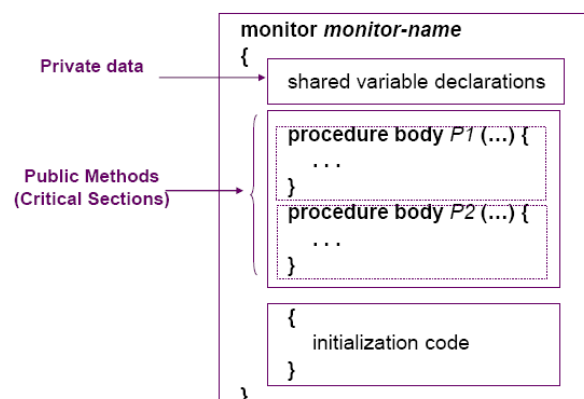
signal(mutex);	wait(mutex);
...	...
임계 구역	임계 구역
...	...
wait(mutex);	wait(mutex);

- 프로세스에서 wait(mutex)나 signal(mutex) 또는 둘 다 생략된 경우

모니터 (Monitor)

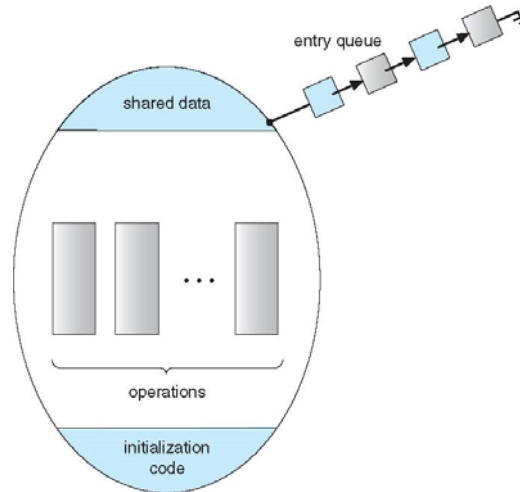
- **모니터**는 쉽고 효율적인 프로세스 동기화 수단을 제공하는 **고급 언어수준의 동기화 구조물(high-level language synchronization construct)**

- 추상화된 데이터 형(**abstract data type**)을 안전하게 공유
 - 예) 자바의 객체
- 데이터와 데이터를 조작하는 함수들을 하나의 단위로 묶어서 보호



모니터 구조

- 모니터 구조물은 모니터 안에 **항상 하나의 프로세스만이** 활성화되도록 보장
 - 프로그래머가 모니터 내부에 상호 배제가 보장되는 연산자들의 집합을 정의

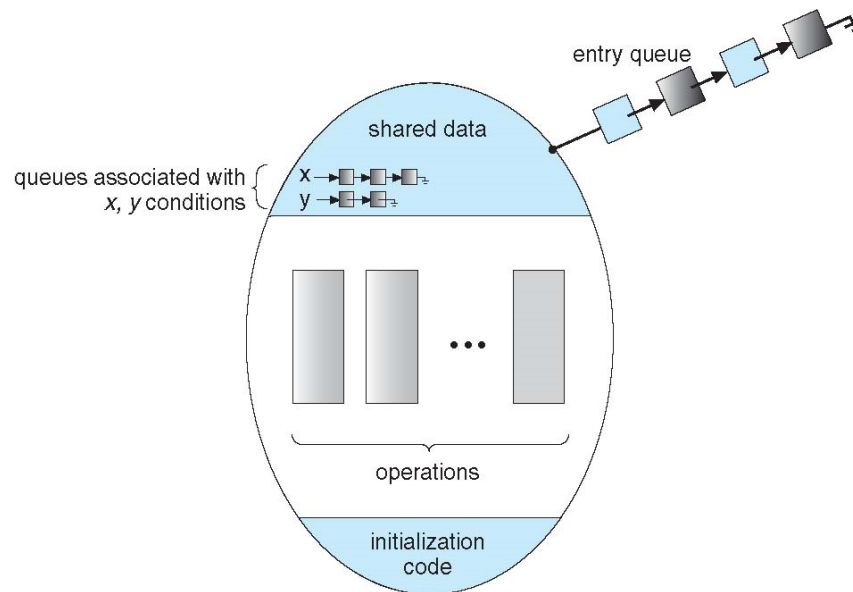


condition 형의 변수

- 모니터 내의 프로세스(연산자)들의 **동기화**를 위해 **condition 변수 선언**
 - **condition x, y ;**
- condition 형 변수에 호출될 수 있는 연산은 오직 **wait()**와 **signal()**
 - **$x.wait()$** : 연산을 호출한 프로세스는 다른 프로세스가 $x.signal()$ 을 호출할 때까지 일시중단
 - **$x.signal()$** : 연산은 정확히 하나의 일시 중단 프로세스를 재개. 만약 일시 중단된 프로세스가 없으면, $signal()$ 연산은 아무런 효과가 없음



조건 변수를 갖는 모니터



자바 모니터 - 스레드 동기화

- ❑ 자바의 기본 스레드 동기화 모델은 **모니터와 같은 병행성 기법** 제공 (7.4.1절 참조)
 - 모든 객체들은 **하나의 락(lock)**과 연관
- ❑ 메서드 또는 객체의 블록에 **synchronized** 선언
 - 공유 데이터에 락을 설정
 - **synchronized 메서드(블록)**가 종료하면 락이 해제되어 다른 스레드가 접근 가능

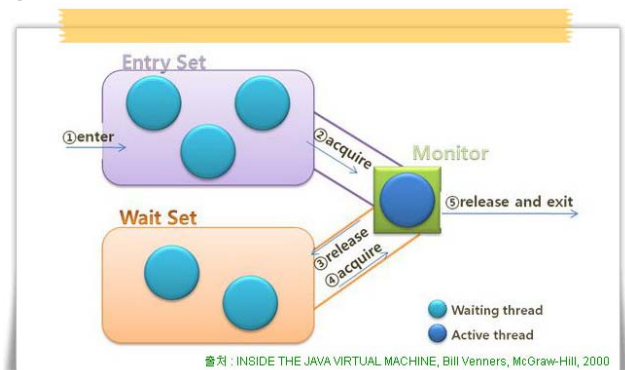
```
public synchronized void method() {
    .....
}
```

```
synchronized(참조하는 객체 변수) {
    .....
}
```

자바 스레드 동기화 모델

□ 하나의 데이터(객체)마다 하나의 모니터를 결합

- 결합된 데이터(객체)가 동시에 두 개 이상의 스레드에 의해 접근할 수 없도록 **락(lock)** 기능 제공
- **synchronized** 메서드를 호출한 스레드는 봉쇄되어 객체의 락에 설정된 **진입 집합(Entry Set)**에 추가
 - 진입 집합은 락이 가용해지기를 기다리는 스레드의 집합
- 락이 가용한 경우 호출 스레드는 객체 락의 소유자가 되어 메서드로 진입
- 스레드가 메서드를 종료하면 락이 해제
 - 락이 해제될 때 진입 집합이 비어 있지 않으면 JVM은 집합에서 스레드를 임의로 선택하여 락을 부여

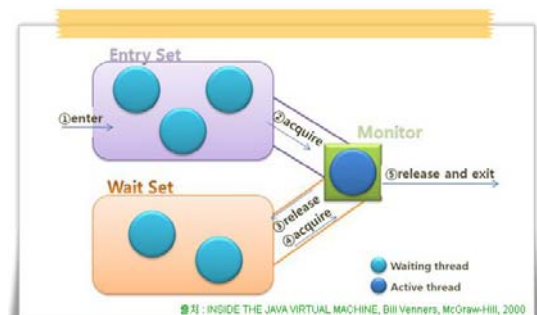


자바 조건 변수

□ 언어 수준에서 자바는 기명 조건 변수를 지원하지 않음

□ 자바의 모니터는 **무명 조건 변수** 하나에만 연결

- **wait()**와 **notify()** 연산이 하나의 조건 변수에만 적용
- 스레드가 **wait()** 메서드 호출 시 동작
 - 스레드가 객체의 락을 해제
 - 스레드 상태가 봉쇄(blocking)로 설정
 - 스레드가 **대기 집합(Wait Set)**에 삽입되어 대기
- 스레드가 **notify()** 메서드 호출 시 동작
 - 대기 집합에서 임의의 스레드 T 선택
 - 스레드 T를 진입 집합으로 이동
 - T의 상태를 봉쇄에서 실행 가능으로 설정



은행 입출금 예 - 동기화 비적용

```
class monitorTest0 {
    public static void main(String[] args)
        throws InterruptedException {
        BankAccount b = new BankAccount();
        BankDeposit d = new BankDeposit(b);    // 입금 스레드
        BankWithdraw w = new BankWithdraw(b);  // 출금 스레드

        d.start(); w.start();
        d.join(); w.join();
        System.out.println( "\nbalance = " + b.getBalance());
    }
}

class BankAccount {
    int balance;
    void deposit(int amt) {    // 입금 메서드
        int temp = balance + amt;

        System.out.print("+");
        balance = temp;
    }
}
```

```
// 출금 메서드
void withdraw(int amt) {
    int temp = balance - amt;

    System.out.print("-");
    balance = temp;
}

int getBalance() {
    return balance;
}
}
```

6. 동기화 도구들

```
// 입금 스레드 클래스
class BankDeposit extends Thread {
    BankAccount b;

    BankDeposit(BankAccount b) {
        this.b = b;
    }

    public void run() {
        for (int i=0; i<100; i++)
            b.deposit(1000);
    }
}
```

```
// 출금 스레드 클래스
class BankWithdraw extends Thread {
    BankAccount b;

    BankWithdraw(BankAccount b) {
        this.b = b;
    }

    public void run() {
        for (int i=0; i<100; i++)
            b.withdraw(1000);
    }
}
```

```
bigdata@master:~/os$ javac monitorTest0.java
bigdata@master:~/os$ java monitorTest0
-----
-----
-----++-----
balance = -71000
bigdata@master:~/os$
```

은행 입출금 예 - 동기화 적용

```

class monitorTest1 {
    public static void main(String[] args)
        throws InterruptedException {
        BankAccount b = new BankAccount();
        BankDeposit d = new BankDeposit(b); // 입금 스레드
        BankWithdraw w = new BankWithdraw(b); // 출금 스레드

        d.start(); w.start();
        d.join(); w.join();
        System.out.println("balance = " + b.getBalance());
    }
}

class BankAccount {
    int balance;
    synchronized void deposit(int amt) { // 입금 메서드
        int temp = balance + amt;

        System.out.print("+");
        balance = temp;
    }
}

```

```

// 출금 메서드
synchronized void withdraw(int amt) {
    int temp = balance - amt;

    System.out.print("-");
    balance = temp;
}

int getBalance() {
    return balance;
}

```

6. 동기화 도구들

```

// 입금 스레드 클래스
class BankDeposit extends Thread {
    BankAccount b;

    BankDeposit(BankAccount b) {
        this.b = b;
    }

    public void run() {
        for (int i=0; i<100; i++)
            b.deposit(1000);
    }
}

```

```

// 출금 스레드 클래스
class BankWithdraw extends Thread {
    BankAccount b;

    BankWithdraw(BankAccount b) {
        this.b = b;
    }

    public void run() {
        for (int i=0; i<100; i++)
            b.withdraw(1000);
    }
}

```

```

bigdata@master:~/os$ javac monitorTest1.java
bigdata@master:~/os$ java monitorTest1

```

```

+++++
-----
balance = 0
bigdata@master:~/os$ █

```

은행 입출금 예 - 조건 변수 동기화 적용

 입금/출금을 반복 수행하도록 동기화

```
class monitorTest2 {
    .....
}

class BankAccount {
    int balance;
    boolean d_turn = true; // 입금/출금 순서 플래그

    synchronized void deposit(int amt) { // 입금 메서드
        int temp = balance + amt;

        System.out.print("+");
        balance = temp;

        notify(); // 조건대기 출금 스레드 깨움
        d_turn = false; // 출금 차례 지정
        try { // 조건 대기
            wait();
        } catch (InterruptedException e) {}
    }
}
```

순천향대학교

6. 동기화 도구들

```
synchronized void withdraw(int amt) {
    while (d_turn) { // 입금차레이면 조건대기
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    int temp = balance - amt;
    System.out.print("-");
    balance = temp;
    notify(); // 조건대기 입금 스레드 깨움
    d_turn = true; // 입금 차례 지정
}

int getBalance() {
    return balance;
}

class BankDeposit extends Thread { // 입금 스레드 클래스
    .....

class BankWithdraw extends Thread { // 출금 스레드 클래스
    .....
}
```

```
bigdata@master:~/os$ javac monitorTest2.java
bigdata@master:~/os$ java monitorTest2
+++++
-+++++
balance = 0
```

순천향대학교 컴퓨터공학과

실습 과제 - 자바 모니터, 은행 입출금 예

- ❑ 앞의 자바 동기화의 은행 입출금 예 monitorTest0, 1, 2 프로그램들을 작성하고 실행 분석
- ❑ 자바 모니터를 사용한 임의의 프로그램 작성