Chapter

06

스레드 동기화

1. 스레드 동기화의 필요성
2. 상호배제
3. 멀티스레드 동기화 기법
4. 생산자 소비자 문제

강의 목표

1. 스레드 동기화의 정의와 필요 이유를 이해한다.
2. 스레드 동기화를 위한 상호배제과 임계구역에 대해 이해한다.
3. 상호배제를 구현하는 여러 방법을 이해하고 최종적으로 원자 명령에 대해 이해한다.
4. 응용프로그램에서 사용 가능한 멀티스레드 동기화 기법들을 안다.
   - 뮤텍스, 스핀락, 세마포
   - pthread 라이브러리를 이용하여 응용프로그램에서 스레드 동기화 실습
5. 생산자 소비자 문제를 이해하고 해결 방법을 안다.

1. 스레드 동기화의 필요성

스레드 동기화의 필요성

- 다수의 스레드가 동시에 공유 데이터에 쓰기를 접근하면
  - 공유데이터가 훼손되는 문제 발생 가능
    - 두 스레드가 동시에 공유 데이터 읽는 경우 -> 문제 없음
    - 한 스레드는 쓰고 한 스레드는 읽을 경우 -> 읽고 쓰는 순서에 따라 읽는 값이 달라질 수 있지만 공유데이터의 훼손은 없음
    - **두 스레드가 동시에 공유 데이터에 쓰는 경우 -> 공유 데이터 훼손 가능성**

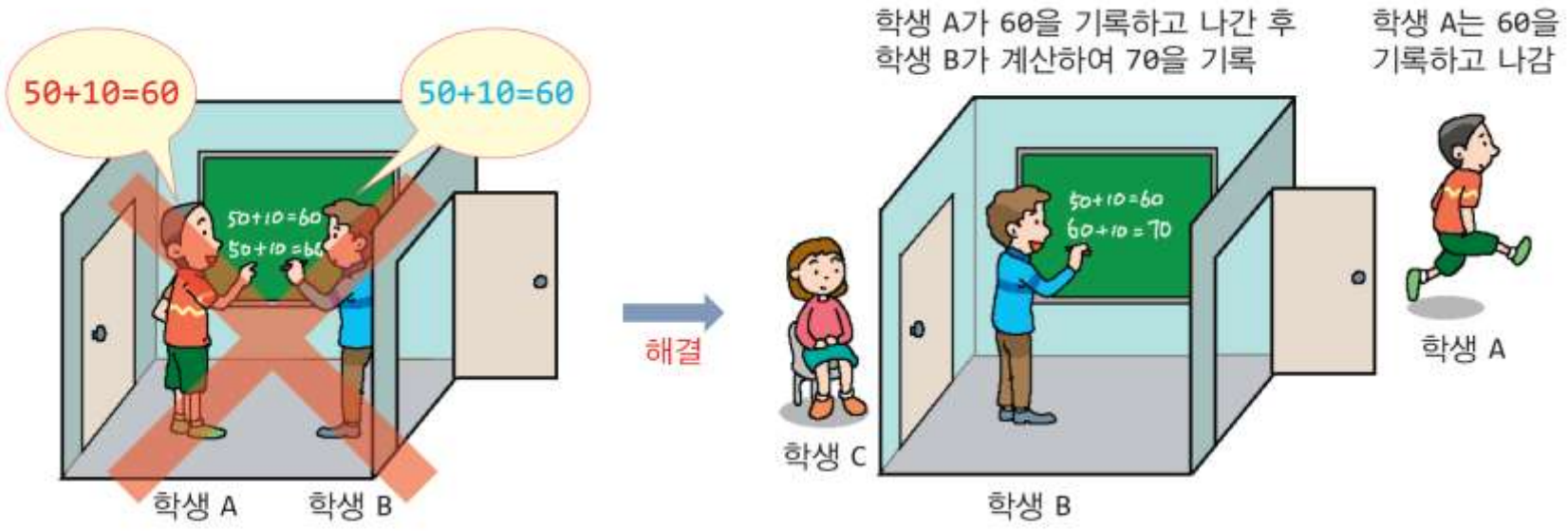
  - 예) 회원의 은행 회비 계좌에 대한 동시 접근

    은행의 서버 컴퓨터에 100만원이 있는 공동 계좌가 있을 때,
    2명이 동시에 100만원씩 입금하면,
    두 스레드가 동시에 계좌에 100만원씩 더하기 실행.
    300만원이 되어야 하는데, 만일 잔액이 200만원이 된다면!

- 스레드 동기화(thread synchronization)
  - 공유데이터에 대한 다수의 스레드가 동시에 접근할 때 공유데이터가 훼손되지 않게 하는 기법
    - 한 스레드가 공유데이터를 배타적 독점적으로 접근하도록 순서화

공유 집계판에 동시 접근하는 사례

문제 : MT가는 날, 학생들이 짐을 10개씩 옮긴 뒤 집계판에 10씩 더하기

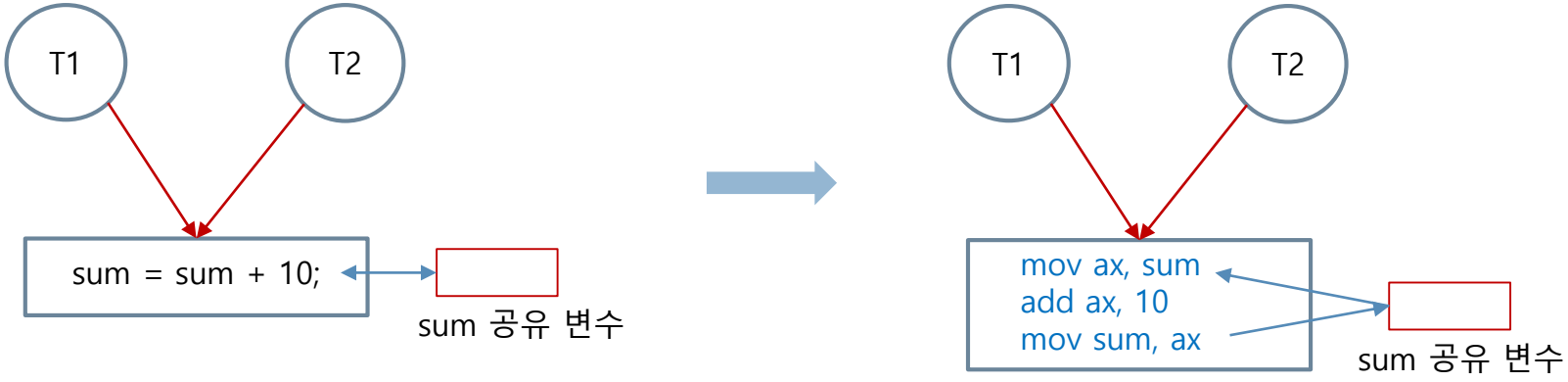


(a) 학생 A와 B가 동시에 방에 들어와서
집계판을 수정하는 경우 집계 **결과가 잘못됨**

(b) 학생 A가 집계 작업을 끝내도록
학생 B가 대기하는 경우 집계 **결과는 정상**

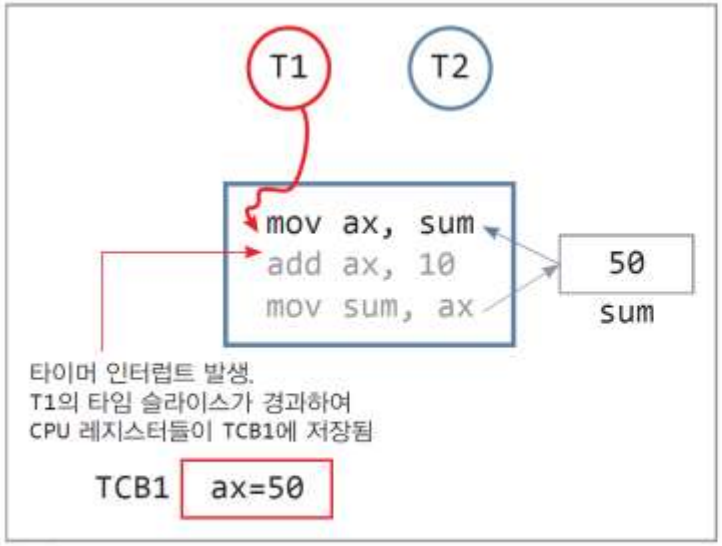
공유 집계판 문제를 프로그램으로 작성

- 공유 집계판 사례의 코드 모델
  - 두 학생 : 스레드 T1, T2
  - 공유 집계판 : 공유 변수 sum
  - 계산 식 : sum = sum + 10;

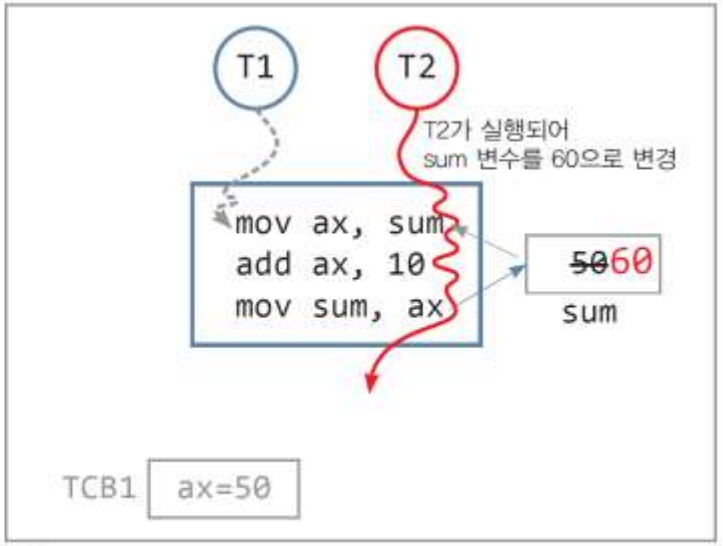



```
sum = sum + 10;   =   mov ax, sum ; sum 변수 값을 읽어 ax 레지스터에 저장
                      add ax, 10    ; ax 레지스터 값 10증가
                      mov sum, ax ; ax 레지스터 값을 sum 변수에 저장
```

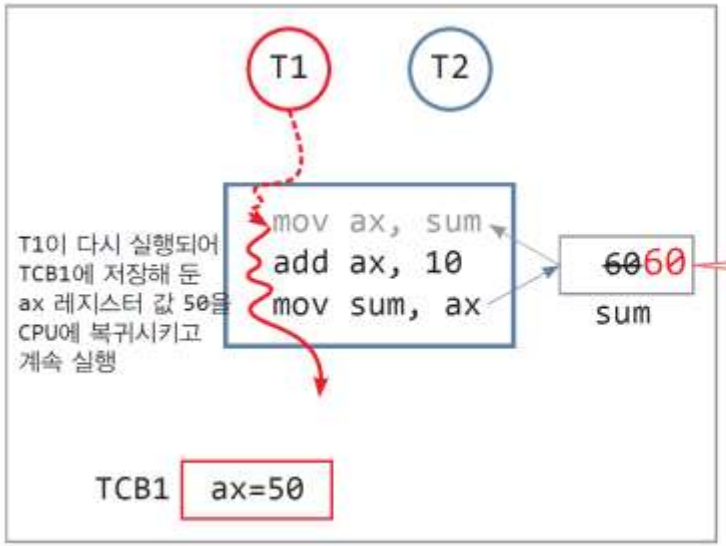
T1과 T2스레드가 공유 변수 sum에 동시에 10을 더할 때 잘못된 결과가 발생하는 경우



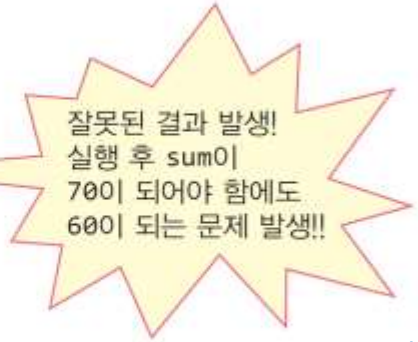
(a) T1이 sum 변수 값을 읽은 후 중단됨



(b) T2가 실행되어 sum 변수에 60 기록



(c) T1이 다시 실행되어 sum 변수에 60 기록 – 잘못된 결과 발생!



* T1이 3개의 명령 실행을 마칠 때까지 다른 스레드가 3개의 명령을 실행하지 못하게 하면...

* T1이 실행되는 동안 타이머 인터럽트를 무시하고 계속 실행하면 ...

탐구 6-1 스레드 동기화가 안 된 사례 – 공유집계판 사례를 멀티스레드로 작성

다음 프로그램은 집계판 문제를 코드로 옮긴 것이다. 이 프로그램은 잘 작동할 것인지 예측하여 보라.

```
$ gcc –o sum sum.c -lpthread
$ ./sum
sum = 13814970
$ ./sum
sum = 10352010
$ ./sum
sum = 11502410
$
```

total sum이 20000000이 되어야 정상인데, 공유 변수 sum에 대한 두 스레드의 충돌로 인해, 20000000이 안 되는 문제 발생!!

잘 작동하지 않는다!!!!!
이 프로그램은 2개의 worker 스레드가
공유 변수 sum에 10을 더하는 코드를
1000000번씩 실행하므로
최종 sum의 값은 20000000이 되고
main()은 20000000을 출력할 것으로 예상된다.
하지만, 결과는 20000000에 모자라는 문제발생!
이 프로그램이 잘 작동하지 않은 이유는,
sum은 공유 변수이고, 'sum = sum + 10'이
공유 변수를 액세스하는 임계구역인데,
한 스레드가 임계구역을 배타적으로 사용할 수 있도록
해주는 동기화 기능이 마련되어 있지 않기 때문

sum.c

```c
#include <stdio.h>
#include <pthread.h>

int sum = 0; // 두 스레드가 공유하는 변수

void* worker(void* arg) { // 스레드 코드
    for(int i=0; i<1000000; i++) {
        sum = sum + 10;
    }
}

int main() {
    char *name[] = {"황기태", "이찬수"};
    pthread_t tid[2]; // 2개의 스레드 ID를 담을 배열
    pthread_attr_t attr[2]; // 2개의 스레드 정보를 담을 배열

    pthread_attr_init(&attr[0]); // 디폴트 속성으로 초기화
    pthread_attr_init(&attr[1]); // 디폴트 속성으로 초기화

    pthread_create(&tid[0], &attr[0], worker, name[0]);  // 스레드 생성
    pthread_create(&tid[1], &attr[1], worker, name[1]);  // 스레드 생성

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("sum = %d₩n", sum); // 두 스레드 종료 후 sum 출력

    return 0;
}
```

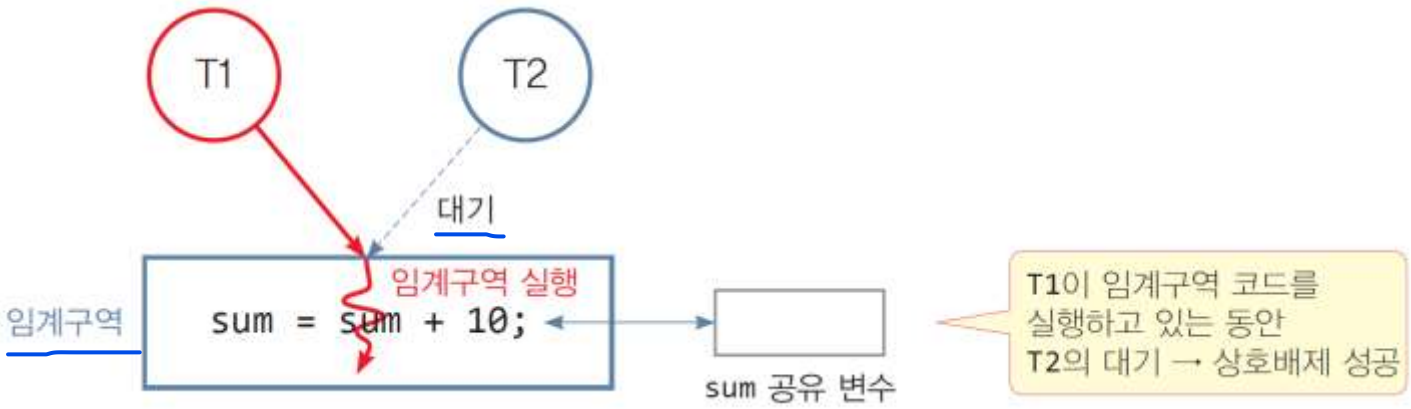
공유 데이터 접근 문제의 해결책

- 문제점
  - 여러 스레드가 공유 변수에 접근할 때, 공유 데이터 훼손
- 해결책 - 스레드 동기화
  - 한 스레드가 공유 데이터 사용을 마칠 때까지,
  - 다른 스레드가 공유 데이터에 접근하지 못하도록 제어

- 멀티스레드의 경쟁 상황이 자주 발생하는가?
  - 매우 자주 발생
    - 사용자의 멀티스레드 프로그램에서 자주 발생
    - 커널 코드에서 매우 자주 발생
      - 커널에 공유 데이터가 많기 때문
  - 다중 코어에서 더욱 조심

임계구역과 상호배제

- 스레드 동기화와 관련된 2가지 중요 개념 : 임계구역과 상호배제
- 임계구역(critical section)
  - 공유 데이터에 접근하는 프로그램 코드들
- 상호배제(mutual exclusion)
  - 임계구역을 오직 한 스레드만 배타적 독점적으로 사용하도록 하는 기술
    - 임계구역에 먼저 진입한 스레드가 임계구역의 실행을 끝낼 때까지,
    - 다른 스레드가 진입하지 못하도록 보장



공유 집계판 문제에 임계구역과 상호배제 적용

2. 상호 배제(mutual exclusion)

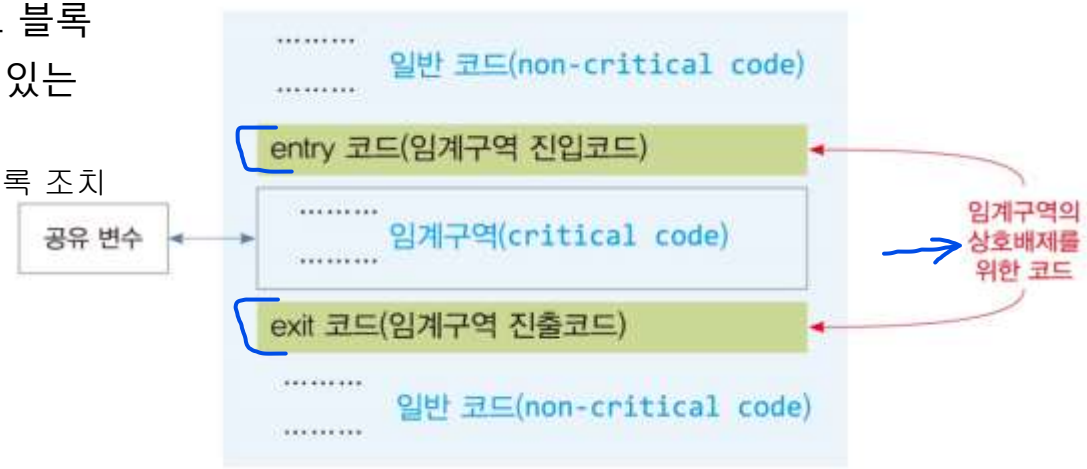
상호 배제를 포함하는 전형적인 프로그램 모습

1. 일반 코드(non-critical code)
   - 공유 데이터를 액세스하지 않는 코드

2. 임계구역 진입 코드(entry code)
   - 임계구역에 진입하기 전 필요한 코드 블록
   - 현재 임계구역을 실행 중인 스레드가 있는지 검사
     - 없다면, 다른 스레드가 들어오지 못하도록 조치
     - 있다면, 진입이 가능해질 때까지 대기

3. 임계구역 코드(critical code)

4. 임계구역 진출 코드(exit code)
   - 임계구역을 마칠 때 필요한 코드 블록
   - 대기중인 스레드가 임계구역에 진입할 수 있도록, 진입 코드에서 취한 조치 해제



상호배제 구현

- 상호배제 구현 목표
  - 임계구역에 오직 1개의 스레드만 진입

- 상호배제 구현 방법
  - 소프트웨어적 방법 - Peterson's 알고리즘 등, 설명 생략
    - 알고리즘 수준에서 제시된 것들로 구현 시 여러 문제 노출
  - 하드웨어적 방법 – 하드웨어의 도움을 받는 방법
    - 인터럽트 서비스 금지, 원자 명령 활용 등
    - 오늘날 대부분 하드웨어적 방법 사용
- 하드웨어적 방법
  - 방법 1 - 인터럽트 서비스 금지
    - 인터럽트 서비스를 금지하거나 허용하는 CPU 명령 사용
  - 방법 2 - 원자 명령(atomic instruction) 사용
    - 원자 명령은 CPU 명령임
    - 오늘날 상호배제 구현에 사용하는 방법

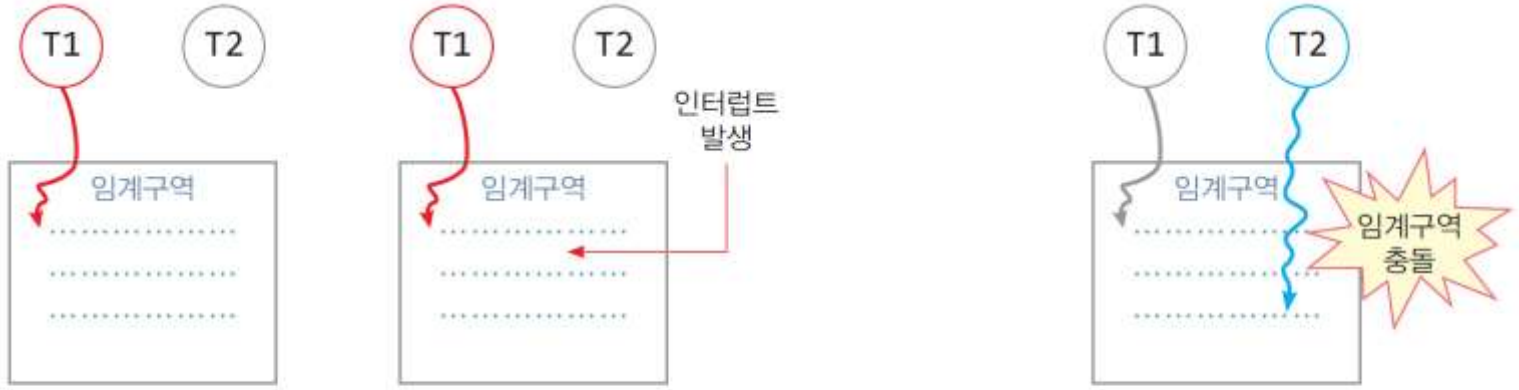
상호배제구현 방법 1 – 인터럽트 서비스 금지

- 인터럽트 서비스 금지 방법
  - entry 코드에서 인터럽트 서비스를 금지하는 명령 실행

```
cli          ; entry 코드. 인터럽트 서비스 금지 명령 cli(clear interrupt flag)
.....
임계구역 코드
...
sti          ; exit 코드. 인터럽트 서비스 허용 명령 sti(set interrupt flag)
```

    - 장치로부터 인터럽트가 발생해도, CPU가 인터럽트 발생을 무시
    - 인터럽트가 발생해도 CPU는 인터럽트 서비스 루틴을 실행하지 않음
    - 인터럽트를 무시하면 임계구역을 실행하는 스레드가 중단되지 않음
- 동작 과정 - 다음 슬라이드에서 자세히 설명
- 문제점
  - 모든 인터럽트가 무시되는 문제 발생
  - 멀티 코어 CPU나 다중 CPU를 가진 시스템에서 활용 불가
    - 한 CPU의 인터럽트 금지로 다른 CPU에게 인터럽트를 금지시킬 수 없음
    - 다른 CPU가 타이머 인터럽트 서비스 루틴을 실행하여, 임계구역을 실행중인 스레드를 컨텍스트 스위칭시키고 다른 스레드를 스케줄할 수 있음

인터럽트를 금지하지 않은 경우 : 인터럽트에 의해 두 스레드가 동시에 임계구역 실행



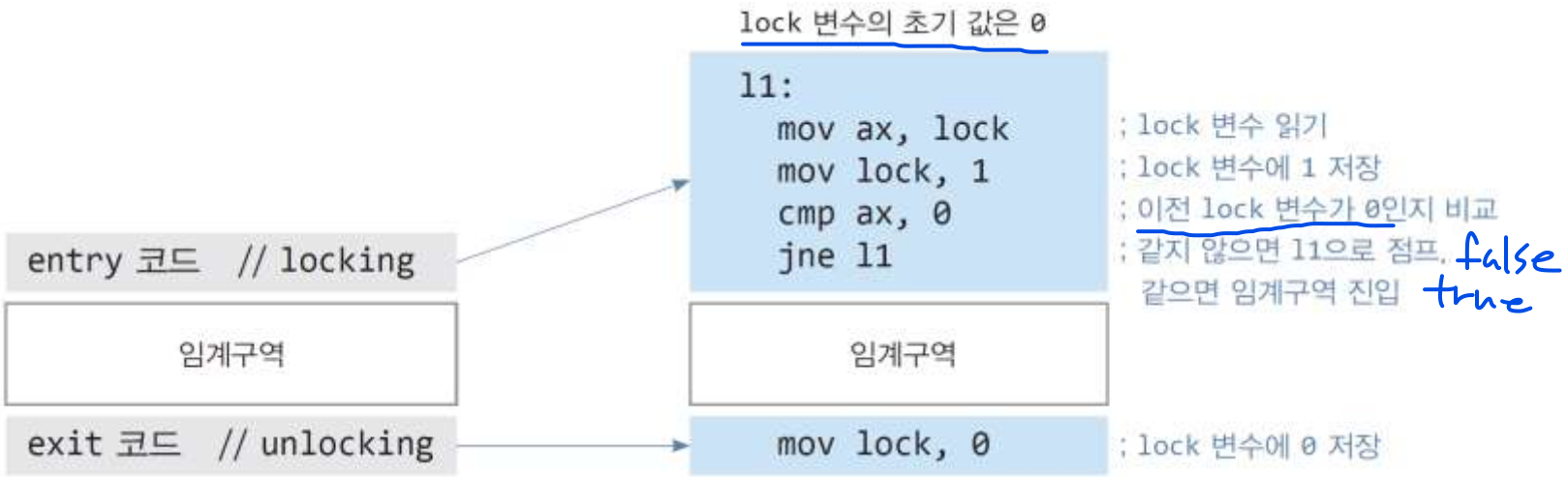
(1) T1이 임계구역 실행 중 (2) 인터럽트 발생 (3) 인터럽트 서비스 루틴이 실행되어 T1을 중단시키고, T2 스케줄 (4) 임계구역에 T2 진입

인터럽트를 금지시켜 임계구역에 대한 상호배제 구현

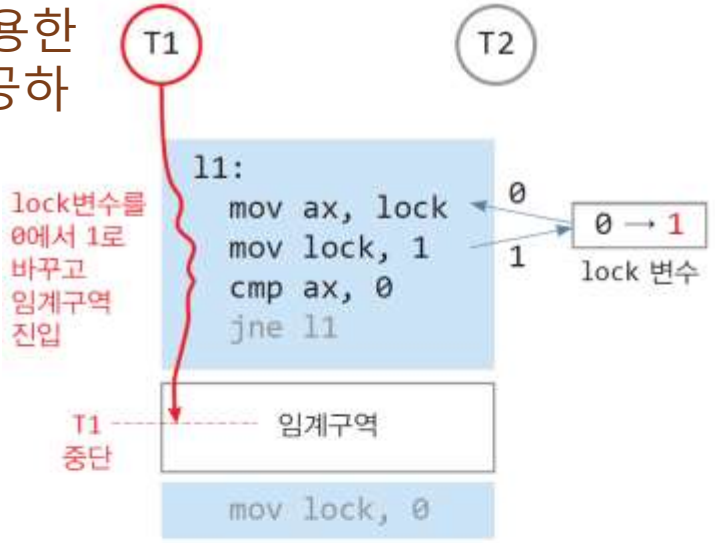
(1) T1이 임계구역 실행 중 (2) CPU는 인터럽트를 무시하고 T1을 계속 실행 (3) T1이 임계구역을 마치고 sti를 실행하고 나간 후, 인터럽트 서비스 루틴 실행됨 (4) 언젠가 T2가 실행되면 임계구역에 진입

단순 lock 변수로 상호배제 시도

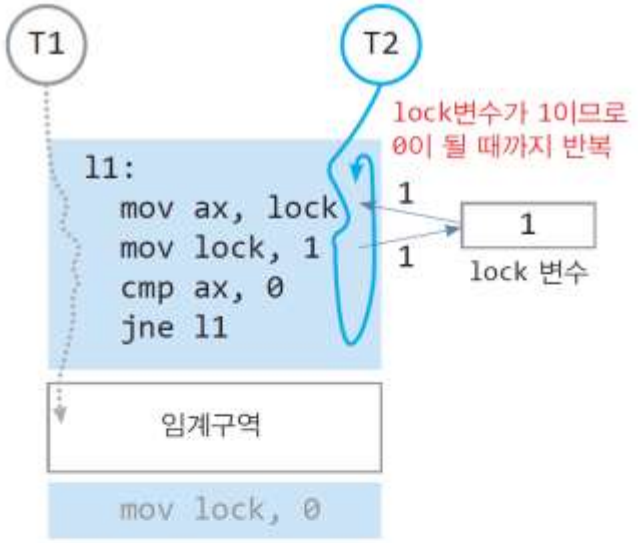
- locking/unlocking 방식으로 임계구역의 entry/exit 코드 작성하면 상호배제가 가능할까?
  - lock 변수 : 1이면 잠금 상태
  - lock 변수 : 0이면 열린 상태



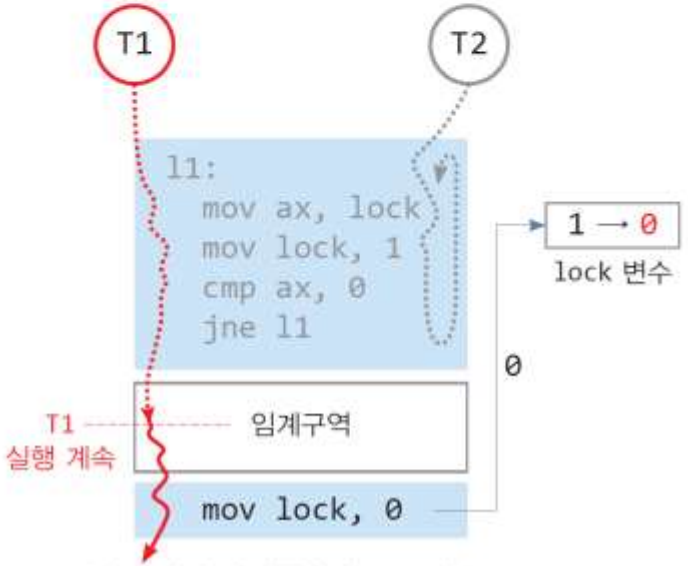
lock 변수를 사용한 상호배제가 성공하는 경우



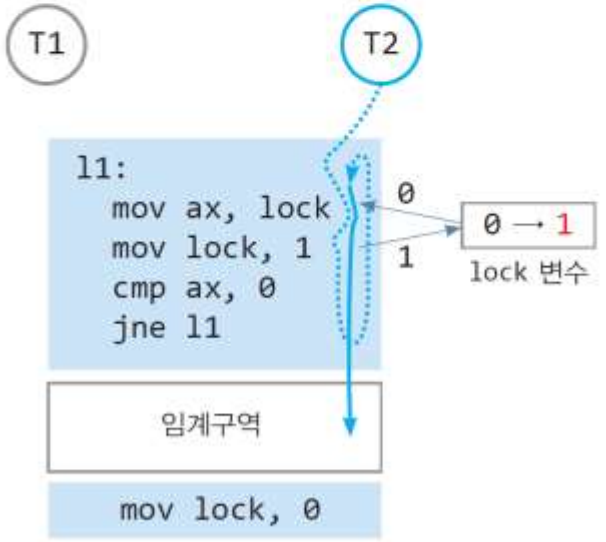
(1) T1이 실행되어 lock 변수를 1로 바꾸고 임계구역 실행. T1 중단.



(2) T2가 실행되어 lock이 0이 될 때까지 계속 lock 변수 조사. T2는 임계구역에 들어가지 못함.

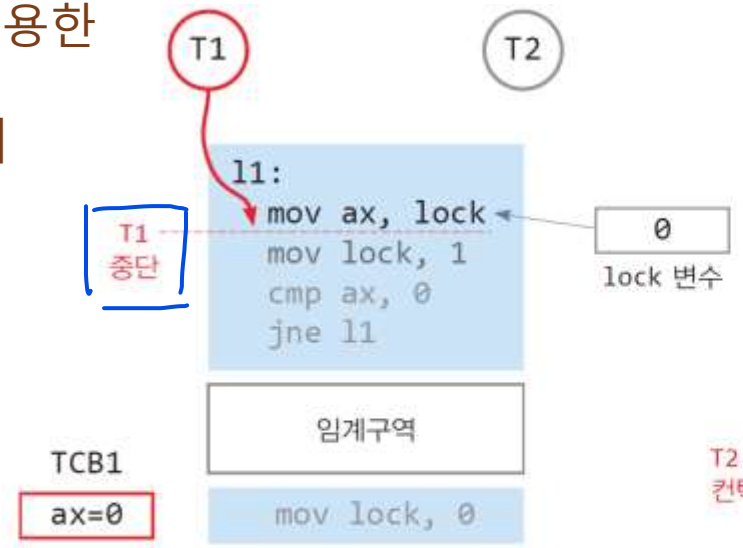


(3) T1이 다시 실행되어 lock 변수를 0으로 바꾸고 임계구역 벗어남

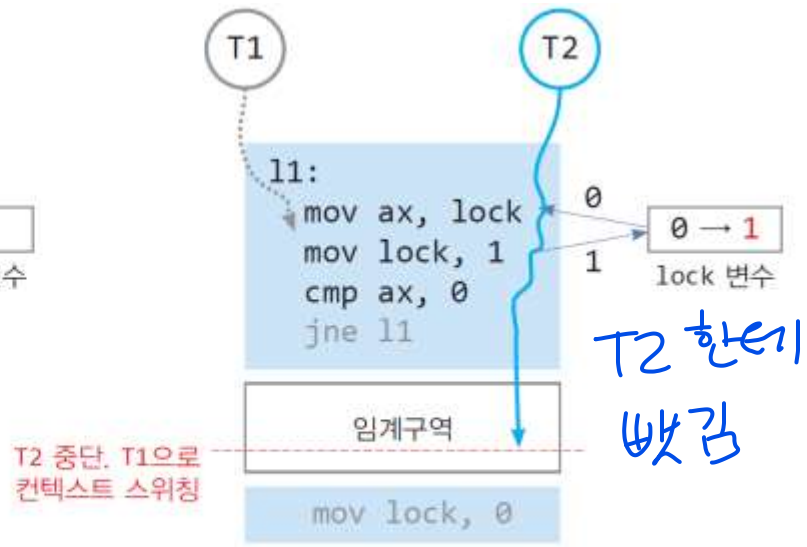


(4) T2 실행. lock 변수가 0이므로 1로 바꾸고 임계구역 진입

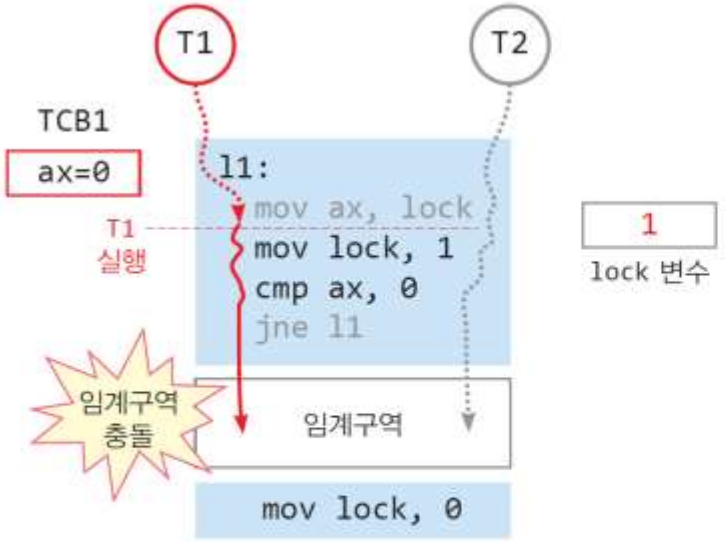
lock 변수를 사용한 상호 배제의 근본 문제 보기 – **실패 사례**



(1) T1이 lock 변수 값 0을 읽은 후 중단



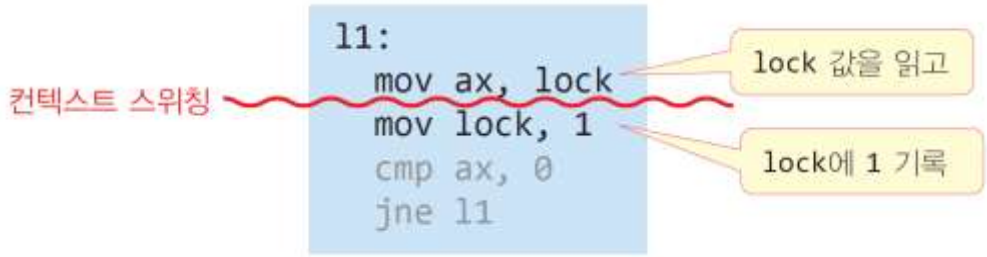
(2) T2는 lock 변수를 1로 변경하고 임계구역 진입. 그리고 중단



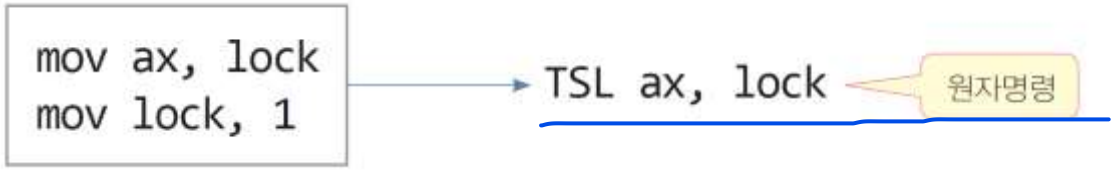
(3) T1이 다시 실행되어 lock 변수를 1로 변경.
이전에 lock 값을 읽어 놓은 ax가 0이므로 임계구역 진입.
임계구역에 T1과 T2가 동시에 들어가 있는 충돌 상황 발생

상호배제구현 방법 2 - 원자명령(atomic instruction) 사용

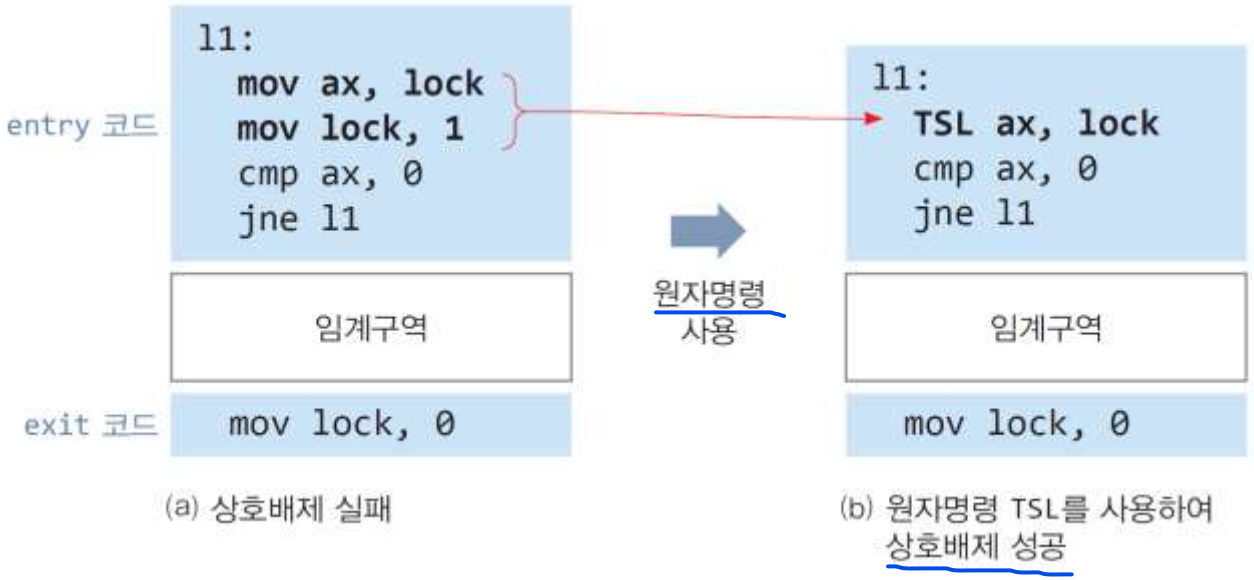
- lock 변수를 이용한 상호배제의 실패 원인?
  - 실패 원인은 entry 코드에 있음
  - lock 변수 값을 읽는 명령과 lock 변수에 1을 저장하는 2개의 명령 사이에 컨텍스트 스위칭이 될 때 문제 발생



- 해결책 – 원자명령(atomic instruction) 도입
  - lock 변수를 읽어 들이는 명령과 lock 변수에 1을 저장하는 2개의 명령을 한 번에 처리하는 원자명령 필요
  - 원자명령 : TSL(Test and Set Lock) 원자명령 = TSL 명령
    - 1970년대 Intel Pentium에서 시작, 현재 대부분의 CPU에 원자 명령 있음



임계구역의 entry/exit 코드를 원자명령으로 재 작성



(a) 상호배제 실패

(b) 원자명령 TSL를 사용하여 상호배제 성공

3. 멀티스레드 동기화 기법

멀티스레드 동기화

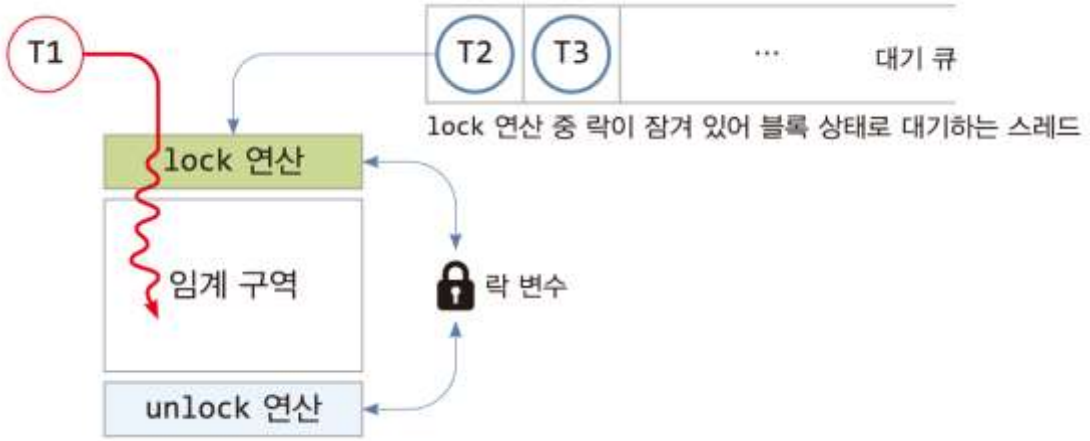
- 멀티스레드 동기화란?
  - 상호배제 기반위에,
  - 자원을 사용하려는 여러 스레드들이 자원을 원활히 공유하도록 하는 기법
  - 동기화 프리미티브(synchronization primitives)로 부름

- 대표적인 기법
  - locks 방식 : **뮤텍스**(mutex), **스핀락**(spinlock)
    - 상호배제가 되도록 만들어진 락(lock) 활용
    - 락을 소유한 스레드만이 임계구역 진입
    - 락을 소유하지 않은 스레드는 락이 풀릴 때까지 대기
  - wait-signal 방식 : **세마포**(semaphore)
    - n개 자원을 사용하려는 m개 멀티스레드의 원활한 관리
    - 자원을 소유하지 못한 스레드는 대기(wait)
    - 자원을 다 사용한 스레드는 알림(signal)

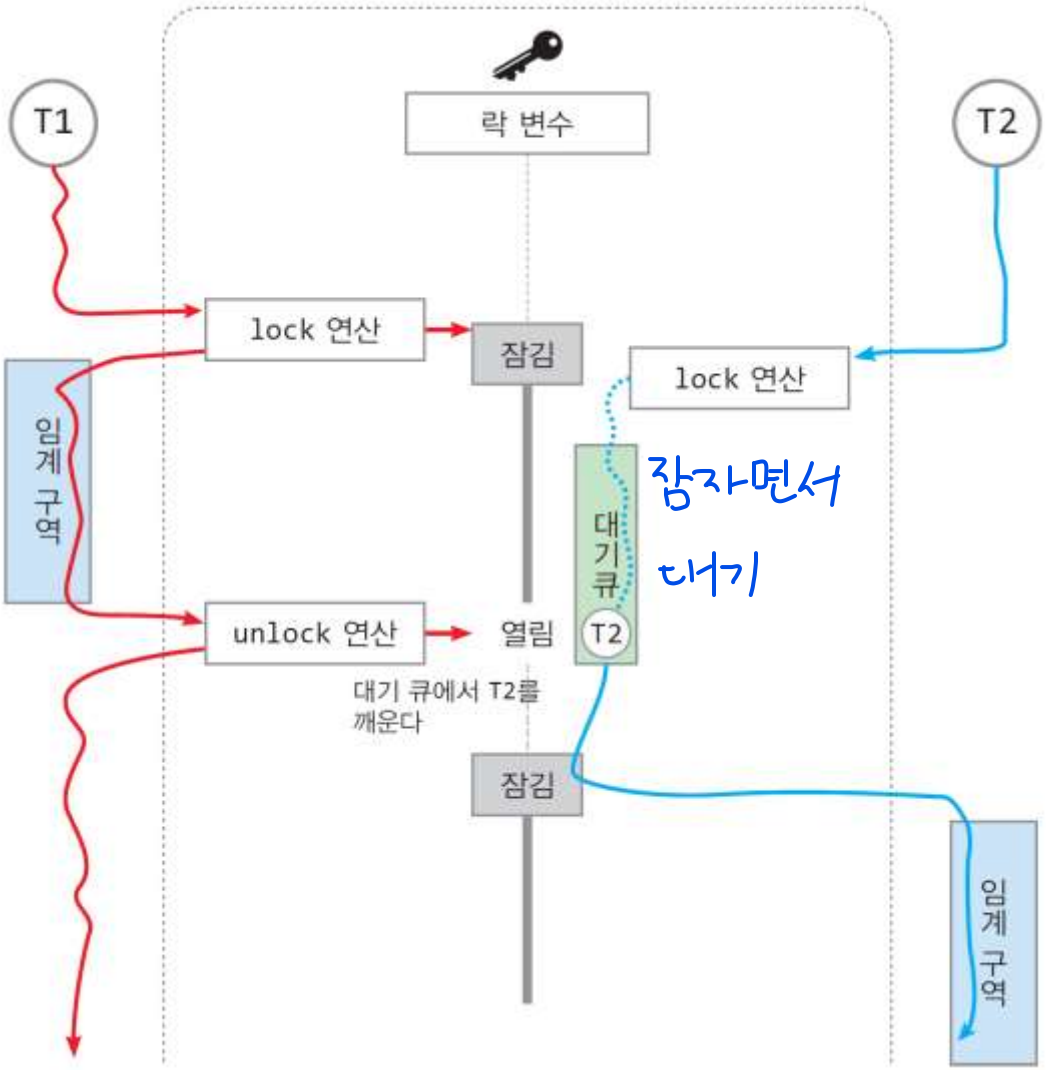
뮤텍스(뮤텍스 기법)

- 뮤텍스(mutex) 기법
  - 잠김/열림 중 한 상태를 가지는 락 변수 이용
  - 한 스레드만 임계구역에 진입시킴
  - 다른 스레드는 큐에 대기
  - sleep-waiting lock 기법
- 뮤텍스 기법의 구성 요소

1. 락 변수
   - true/false 중 한 값
   - true : 락을 잠근다. 락을 소유한
   - false : 락을 연다. 락을 해제한
2. 대기 큐
   - 락이 열리기를 기다리는 스레드
3. 연산
   - lock 연산(임계구역의 entry 코드)
     - 락이 열린 상태이면, 락을 잠그고 임계구역 진입
     - 락이 잠김 상태(lock = true)이면, 현재 스레드를 블록 상태로 만들고 대기 큐에 삽입
   - unlock 연산(임계구역의 exit 코드)
     - lock = false, 락을 열린 상태로 변경
     - 대기 큐에서 기다리는 스레드 하나 깨움



뮤텍스를 활용한 스레드 동기화 과정



뮤텍스의 특징

- 뮤텍스를 이용한 동기화 특징
  - 임계구역의 실행 시간이 짧은 경우, 비율적
    - 락이 잠겨 있으면 (컨텍스트 스위칭되어) 대기 큐에서 대기, 락이 풀리면 다시 (컨텍스트 스위칭되어) 실행
    - 락이 잠겨있는 시간보다 스레드가 잠자고 깨는 데 걸리는 시간이 상대적으로 크면 비효율적

조금만 기다리면 되는데 아까움

- 뮤텍스 동기화를 위한 POSIX 표준 라이브러리
  - 뮤텍스락 변수
    - pthread_mutex_t lock;
  - 대기큐는 pthread 라이브러리 내부에 구현되어 있기 때문에 사용자에게 보이지 않음
  - 뮤텍스 조작 함수들
    - pthread_mutex_init() – 뮤텍스락 변수 초기화
    - pthread_mutex_lock() – 뮤텍스락 잠그기
    - pthread_mutex_unlock() – 뮤텍스락 풀기
    - pthread_mutex_destroy() – 뮤텍스락 변수 사용 종료
  - pthread를 이용한 뮤텍스 동기화 코딩 사례

```
pthread_mutex_t lock;                  // 뮤텍스락 변수 생성
pthread_mutex_init(&lock, NULL);       // 뮤텍스락 변수 초기화
...
...
pthread_mutex_lock(&lock);             // 임계구역 enrty코드. 뮤텍스락 잠그기
  ... 임계구역코드 ....
pthread_mutex_unlock(&lock);           // 임계구역 exit코드. 뮤텍스락 열기
```

= 원자명령

탐구 6-2: pthread의 뮤텍스를 이용한 공유집계판의 스레드 동기화

mutex.c

```
#include <stdio.h>
#include <pthread.h>

int sum = 0; // 두 스레드가 공유하는 변수
pthread_mutex_t lock; // 뮤텍스락 변수 선언

void* worker(void* arg) { // 스레드 코드
    printf("%s 시작 ₩t %d₩n", (char*)arg, sum);

    for(int i=0; i<1000000; i++) {
        pthread_mutex_lock(&lock); // entry 코드. 뮤텍스 락 잠그기
        sum = sum + 10; // 임계구역 코드
        pthread_mutex_unlock(&lock); // exit 코드. 뮤텍스 락 열기
    }
    printf("%s 끝 ₩t %d₩n", (char*)arg, sum);
}
```

```
int main() {
    char *name[] = {"황기태", "이찬수"};
    pthread_t tid[2];
    pthread_attr_t attr[2]; // 스레드 정보를 담을 구조체

    pthread_attr_init(&attr[0]); // 디폴트 속성으로 초기화
    pthread_attr_init(&attr[1]); // 디폴트 속성으로 초기화

    pthread_mutex_init(&lock, NULL); // 뮤텍스락 변수 lock 초기화

    pthread_create(&tid[0], &attr[0], worker, name[0]);  // 스레드 생성
    pthread_create(&tid[1], &attr[1], worker, name[1]); // 스레드 생성

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("최종 sum = %d₩n", sum);

    pthread_mutex_destroy(&lock); // 뮤텍스락 lock 사용 끝

    return 0;
}
```

worker() 함수를 다음과 같이 작성하면 어떻게 될까?

```
void* worker(void* arg) { // 스레드 코드
    printf("%s 시작 ₩t %d₩n", (char*)arg, sum);
    pthread_mutex_lock(&lock); // lock
    for(int i=0; i<1000000; i++) {
        sum += 10;
    }
    pthread_mutex_unlock(&lock); // unlock
    printf("%s 끝 ₩t %d₩n", (char*)arg, sum);
}
```

바르다

```
$ gcc -o mutex mutex.c –lpthread
$ ./mutex
황기태 시작      0
이찬수 시작      0
황기태 끝        19236950
이찬수 끝        20000000
최종 sum = 20000000
$ ./mutex
황기태 시작      0
이찬수 시작      202390
이찬수 끝        14510640
황기태 끝        20000000
최종 sum = 20000000
$
```

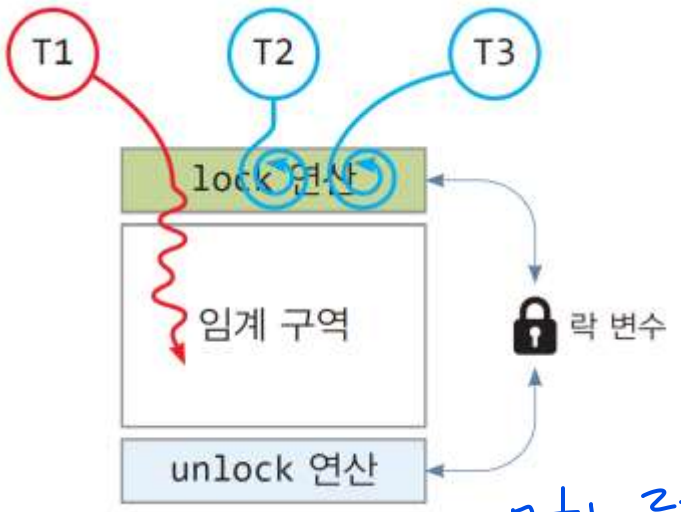
스케줄링으로 인해 두 스레드가 실행되는 순서가 달라 중간 결과는 항상 달라짐

최종 sum은 20000000으로, 실행시킬 때마다 동일한 결과

스핀락(스핀락 기법)

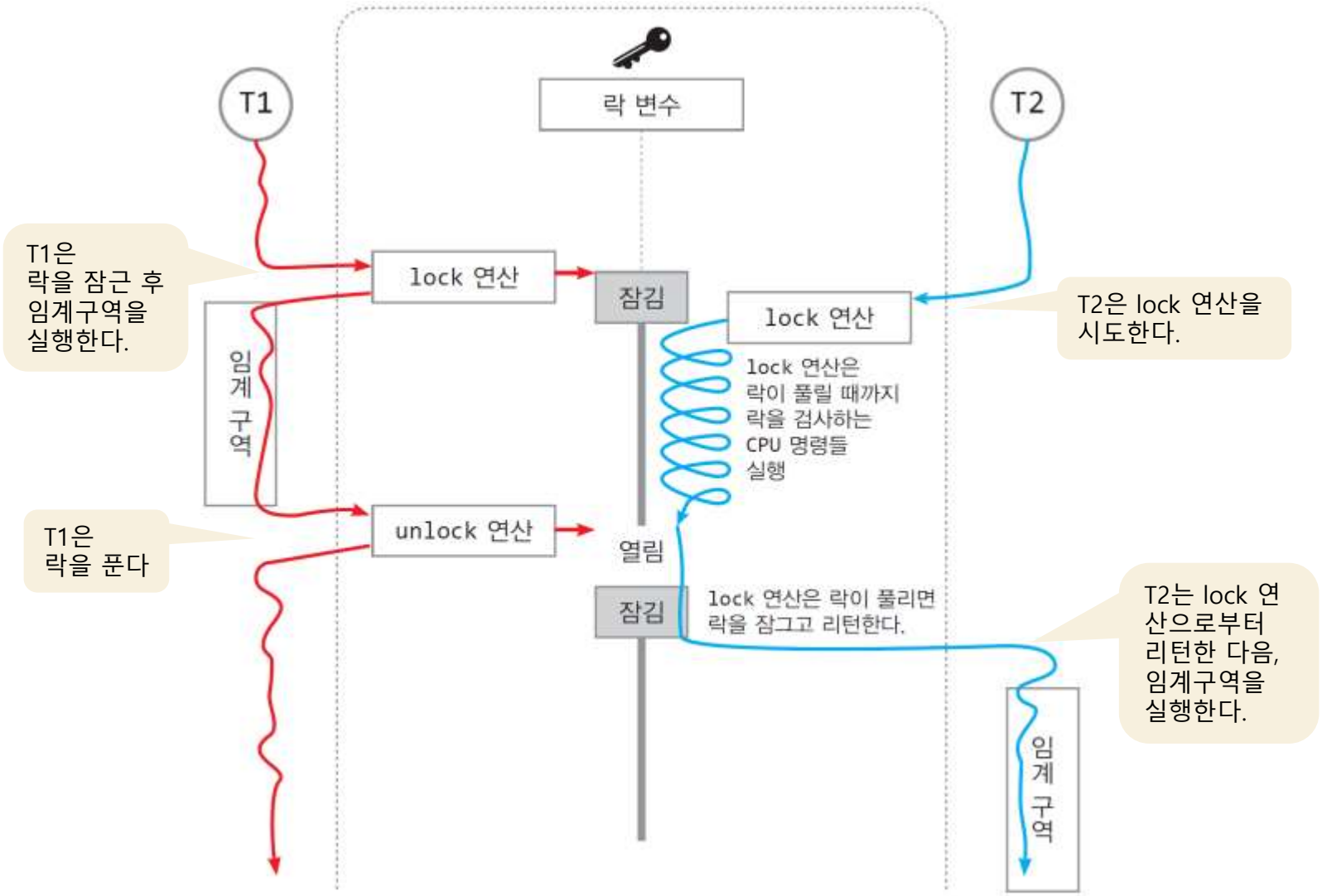
- 스핀락(spinlock) 기법
  - busy-waiting lock 기법
    - 스레드가 큐에서 대기하지 않고 락이 열릴 때까지 계속 락 변수 검사
  - 뮤텍스와 거의 같고 busy-waiting이라는 점에서만 다름
    - 대기 큐 없음
    - busy-waiting으로 인해 CPU를 계속 소모, CPU가 다른 스레드를 실행할 수 없음
  - 락을 소유한 스레드만 자원 배타적 사용, 동기화 기법
    - 공유 자원 하나 당 하나의 스핀락 사용

- 스핀락 기법의 구성 요소
  1. 락 변수
     - true/false 중 한 값
     - true : 락을 잠근다. 락을 소유한다.
     - false : 락을 연다. 락을 해제한다.
  2. 연산
     - lock 연산
       - 임계구역에 들어갈 때 실행되는 entry 코드
       - 락이 잠김 상태면, 락이 풀릴 때까지 무한 루프 돌면서 lock 연산
       - 락이 열린 상태면, 락을 잠김 상태로 바꾸고 임계구역 실행
     - unlock 연산
       - 임계구역을 나올 때 실행하는 exit 코드
       - 락을 열림 상태로 변경



잠을 안자고 무한루프
컨텍스트 스위칭 ×

스핀락을 활용한 스레드 동기화 사례



스핀락 특징

- 스핀락를 이용한 동기화 특징
  - 뮤텍스의 non-blocking 모델 - 락이 잠겨 있을 때 블록되지 않고 락이 풀릴 때까지 검사하는 코드 실행
  - 단일 CPU(단일 코어)를 가진 운영체제에서 비효율적
    - 단일 코어 CPU에서 의미 없는 CPU 시간 낭비
      - 스핀락을 검사하는 스레드의 타임 슬라이스가 끝날 때까지 다른 스레드 실행 안 됨, 다른 스레드의 실행 기회 뺏음
      - 락을 소유한 다른 스레드가 실행되어야 락이 풀림.
    - 멀티 코어에 적합
      - 락을 경쟁하는 스레드들을 서로 다른 코어에서 실행. 한 코어에서 임계구역을 실행 중일 때, 다른 코어에서 락이 풀릴 때까지 검사
  - 임계구역의 실행 시간이 짧은 경우 효과적
- 스핀락 동기화를 위한 POSIX 표준 라이브러리
  - 스핀락 변수
    - pthread_spinlock_t lock;
  - 스핀락 조작 함수들
    - pthread_spin_init() – 스핀락 변수 초기화
    - pthread_spin_lock() – 스핀락 잠그기
    - pthread_spin_unlock() – 스핀락 풀기
    - pthread_spin_destroy() – 스핀락 변수 사용 종료
  - pthread를 이용한 스핀락 동기화 코딩 사례

```
pthread_spinlock_t lock;                // 스핀락 변수 생성
pthread_spin_init(&lock, NULL);         // 스핀락 변수 초기화

pthread_spin_lock(&lock);               // 임계구역 entry 코드. 스핀락 잠그기
  ... 임계구역코드 ....
pthread_spin_unlock(&lock);             // 임계구역 exit 코드. 스핀락 열기
```

탐구 6-3 : pthread의 스핀락을 이용한 공유 집계판의 스레드 동기화

spinlock.c

```
#include <stdio.h>
#include <pthread.h>

int sum = 0; // 두 스레드가 공유하는 변수
pthread_spinlock_t lock; // 스핀락 변수 선언

void* worker(void* arg) { // 스레드 코드
    printf("%s 시작 ₩t %d₩n " , (char*)arg, sum);
    for(int i=0; i<1000000; i++) {
        pthread_spin_lock(&lock); // entry 코드
        sum = sum + 10; // 임계구역 코드
        pthread_spin_unlock(&lock); // exit 코드
    }
    printf("%s 끝 ₩t %d₩n", (char*)arg, sum);
}
```

```
int main() {
    char *name[] = {"황기태", "이찬수"};
    pthread_t tid[2]; // 두 스레드의 ID를 저장할 배열

    pthread_attr_t attr[2]; // 스레드 정보를 담을 구조체

    pthread_attr_init(&attr[0]); // 디폴트 속성으로 초기화
    pthread_attr_init(&attr[1]); // 디폴트 속성으로 초기화

    pthread_spin_init(&lock, PTHREAD_PROCESS_PRIVATE);
    // lock을 한 프로세스에 속한 스레드만이 공유하는 변수로 선언

    pthread_create(&tid[0], &attr[0], worker, name[0]);  // 스레드 생성
    pthread_create(&tid[1], &attr[1], worker, name[1]); // 스레드 생성

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("최종 sum = %d₩n", sum);

    pthread_spin_destroy(&lock);

    return 0;
}
```

```
$ gcc -o spinlock spinlock.c -lpthread
$ ./spinlock
황기태 시작      0
이찬수 시작      0
이찬수 끝        13096310
황기태 끝        20000000
최종 sum = 20000000
$ ./spinlock
황기태 시작      0
이찬수 시작      0
황기태 끝        10069990
이찬수 끝        20000000
최종 sum = 20000000
$
```

스케줄링으로 인해 두 스레드가 실행되는 순서가 달라 중간 결과는 항상 달라짐

최종 sum은 20000000으로, 실행시킬 때마다 항상 동일한 실행 결과

worker() 함수를 다음과 같이 작성하면 어떻게 될까?

```
void* worker(void* arg) { // 스레드 코드
    printf("%s 시작 ₩t %d₩n", (char*)arg, sum);
    pthread_spin_lock(&lock); // entry 코드
    for(int i=0; i<1000000; i++) {
        sum = sum + 10;
    }
    pthread_spin_unlock(&lock); // exit 코드
    printf("%s 끝 ₩t %d₩n", (char*)arg, sum);
}
```

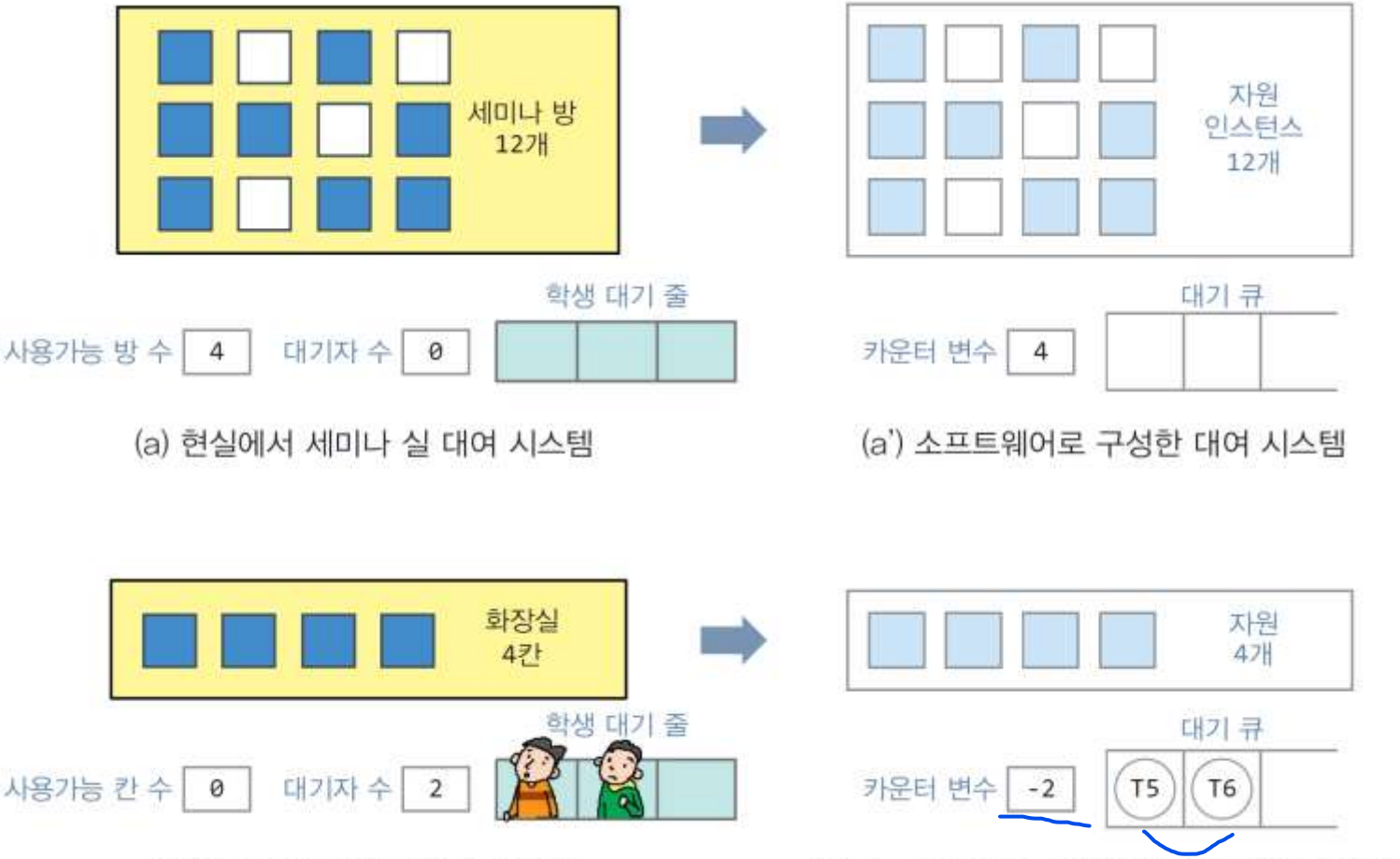
뮤텍스와 스핀락은 어떤 경우에 적합한가?

1. 락이 잠기는 시간이 긴(임계구역이 긴) 응용 : 뮤텍스
   - 락을 얻지 못했을 때, CPU를 다른 스레드에게 양보하는 것이 효율적
   - 락이 잠기는 시간이 짧은 경우 : 스핀락이 효율적
2. 단일 CPU를 가진 시스템 : 뮤텍스
   - 단일 CPU에서 스핀락은 크게 의미 없음
3. 멀티 코어(멀티 CPU)를 가진 시스템 : 스핀락
   - 임계구역은 보통 짧게 작성되므로,
   - 잠자고 깨는 컨텍스트 스위칭 없이 바로 자원 사용
4. 사용자 응용프로그램 : 뮤텍스, 커널 코드 : 스핀락
   - 커널 코드나 인터럽트 서비스 루틴은 빨리 실행되어야 하고,
   - 인터럽트 서비스 루틴 내에서 잠잘 수 없기 때문
5. 스핀락을 사용하면 기아 발생 가능
   - 스핀락은 무한 경쟁 방식이어서 기아 발생 가능
   - 락을 소유한 스레드가 락을 풀지 않고 종료한 경우나 코딩이 잘못된 경우에도 기아 발생 가능

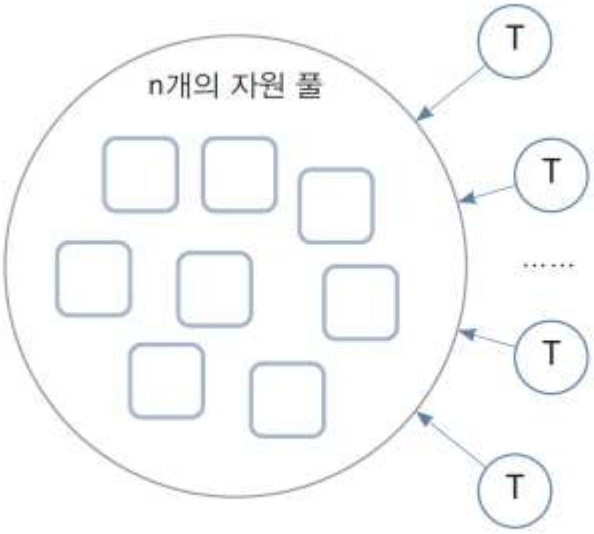
뮤텍스와 스핀락 비교

	뮤텍스	스핀락
대기 큐	있음	없음
블록 가능 여부	락이 잠겨 있으면 블록됨 (blocking)	락이 잠겨 있어도 블록되지 않고 계속 락 검사 (non-blocking)
lock/unlock 연산 비용	저비용	CPU를 계속 사용하므로 고비용
하드웨어 관련	단일 CPU에서 적합	멀티코어 CPU에서 적합
주 사용처	사용자 응용 프로그램	커널 코드, 인터럽트 서비스 루틴

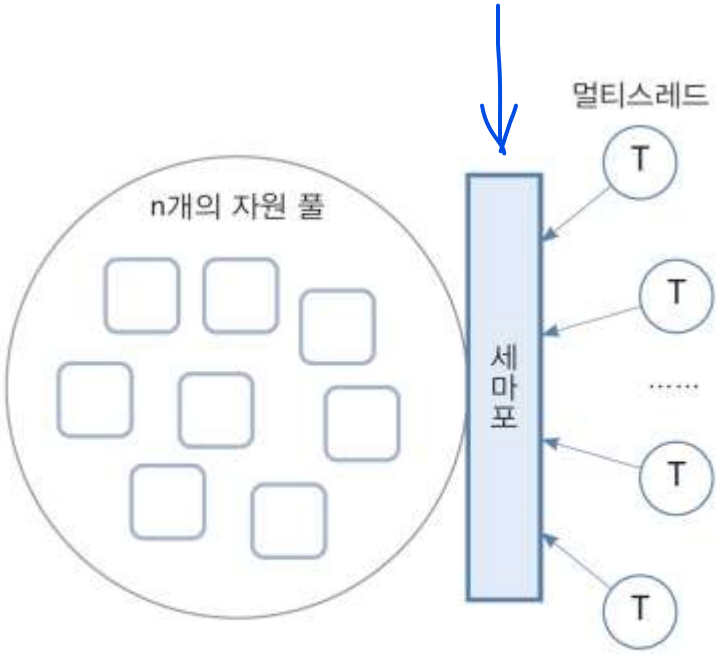
세마포의 필요성을 이해하기 위한 대여 시스템 사례 – 여러 자원을 여러 명이 사용하는 경우



세마포가 필요한 상황



(a) 멀티스레드가 n개의 자원을 활용하려는 상황



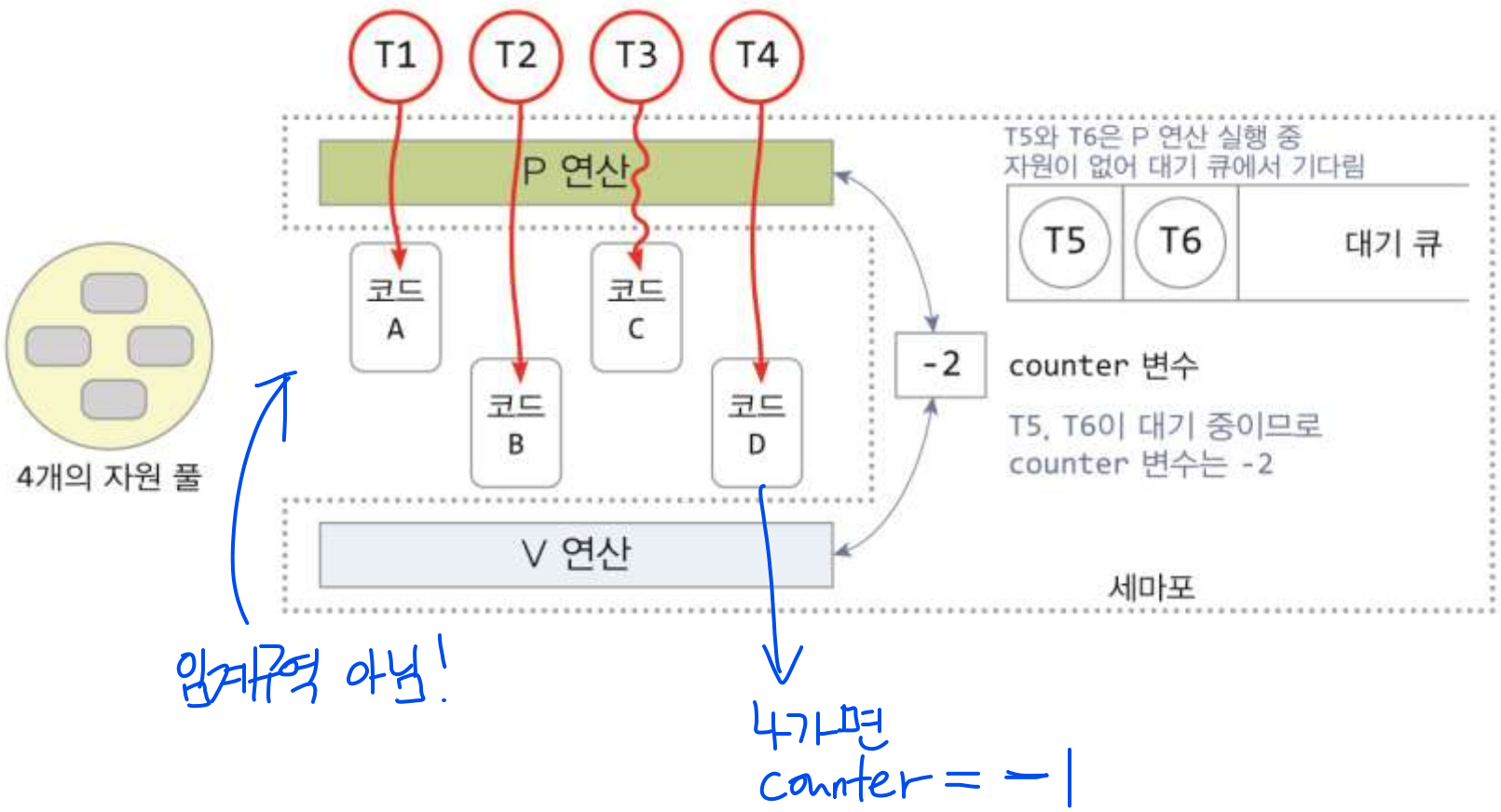
(b) 세마포를 이용하여 멀티스레드가 n개의 자원을 원활히 사용할 수 있도록 관리

세마포

- 세마포(semaphore) 정의
  - 멀티스레드 사이의 자원 관리 기법
    - n개의 공유 자원을 다수 스레드가 공유하여 사용하도록 돕는 자원 관리 기법
      - n개의 프린터가 있는 경우, 프린터를 사용하고자 하는 다수 스레드의 프린터 사용 관리
- 구성 요소
  1. 자원 : n 개
  2. 대기 큐 : 자원을 할당받지 못한 스레드들이 대기하는 큐
  3. counter 변수
     - 사용 가능한 자원의 개수를 나타내는 정수형 전역 변수
     - n으로 초기화(counter = n)
  4. P/V 연산
     - P연산(wait 연산) – 자원 요청 시 실행하는 연산
       - 자원 사용 허가를 얻는 과정
     - V 연산(signal 연산) – 자원 반환 시 실행하는 연산
       - 자원 사용이 끝났음을 알리는 과정

세마포를 이용한 멀티스레드 자원 관리의 구조

4개의 인스턴스를 가진 자원에 대해, 4개의 스레드(T1~T4)가 할당 받아 사용,
2개의 스레드 T5와 T6는 자원을 기다리고 있는 상태
counter 변수는 사용 가능한 자원의 개수를 나타내지만 음수이면 대기 중인 스레드의 수를 나타냄



P 연산과 V 연산

- 세마포 종류 2가지 – sleep-wait 세마포와 busy-wait 세마포
  - 자원을 할당받지 못한 경우의 행동에 따라 구분
- sleep-wait 세마포
  - P연산 : counter--, **대기 큐에서 잠자기**
  - V연산: counter++, 사용가능 자원이 있으면 잠자는 스레드 깨우기
- busy-wait 세마포
  - P연산 : 사용 가능 자원이 생길 때까지 **무한 루프** 후 자원이 생기면 counter—
  - V연산: counter++;

```
P 연산 { // wait
  counter--; // 자원 요청
  if counter < 0 {
    ... 현재 스레드를 대기 큐에 삽입 ... // sleep-wait
  }
}
```

```
V 연산 { // signal
  counter++; // 자원 반환
  if counter <= 0 { // 기다리는 스레드 있는 경우
    ... 대기 큐에서 한 스레드 깨움 ...
  }
}
```

(a) 수면 대기(sleep–wait) 세마포 기법

```
P 연산 { // wait
  while counter <= 0; // busy-wait
  counter--;
}
```

```
V 연산 { // signal
  counter++;
}
```

(b) 바쁜 대기(busy–wait) 세마포 기법

(b)는 counter가 0보다 작을 일 없음

세마포 활용을 위한 POSIX 표준 라이브러리

- 세마포 구조체
  - sem_t s; // counter 변수 등을 가진 세마포 구조체
- 세마포 조작 함수들
  - sem_init() - 세마포 초기화
  - sem_destroy() - 세마포 기능 소멸
  - **sem_wait()**
    - P 연산을 수행하는 함수(blocking call)
    - sleep-wait 방식으로, 가용 자원이 없으면 대기 큐에서 잠을 잠
  - sem_trywait()
    - P 연산을 수행하는 함수(non-blocking call)
    - 가용 자원이 있으면, counter 값을 감소시키고 0리턴
    - 없으면, counter 값을 감소시키지 않고 -1 리턴
  - **sem_post()**
    - V 연산을 수행하는 함수
  - sem_getvalue()
    - 세마포의 현재 counter 값을 리턴하는 함수

```
sem_t sem;                  // 세마포 구조체 생성

sem_wait(&sem);             // P 연산. 자원 사용 요청
... 할당받은 자원 활용 ...
sem_post(&sem);             // V 연산. 자원 사용 끝
```

탐구 6-4 : 세마포 활용 사례

3개의 칸이 있는 화장실에
5명의 고객이 사용하고자 할 때.
세마포를 이용하여
3칸의 화장실을 5명의 고객 스레드가
활용할 수 있게 관리하는 예

sem.c

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t toiletsem; // POSIX 세마포 구조체로 모든 스레드에 의해 공유

void* guestThread(void* arg) { // 고객의 행동을 묘사하는 스레드 코드
    int cnt = -1;

    sem_wait(&toiletsem); // P 연산. 자원 사용 요청. 세마포의 counter 값 1 감소
    sem_getvalue(&toiletsem, &cnt); //  세마포의 counter 을 cnt 변수로 읽어오기
    printf("고객%s 화장실에 들어간다... 세마포 counter = %d₩n", (char*)arg, cnt);
    sleep(1); // 1초 동안 화장실을 사용한다.
    sem_post(&toiletsem); // V 연산. 화장실 사용을 끝냈음을 알림
    sem_getvalue(&toiletsem, &cnt); //  세마포의 counter 값을 cnt 변수로 읽어오기
    printf("고객%s 화장실에서 나온다.세마포 counter = %d₩n", (char*)arg, cnt);
}

#define NO 0                    // 자식 프로세스와 세마포 공유하지 않음
#define MAX_COUNTER 3 // 자원의 개수, 동시에 들어갈 수 있는 스레드의 개수

int main() {
    int counter = -1;
    char *name[] = {"1", "2", "3", "4", "5" };
    pthread_t t[5]; // 스레드 구조체

    // 세마포 초기화 : MAX_COUNTER 명이 동시에 사용
    sem_init(&toiletsem, NO, MAX_COUNTER);
    sem_getvalue(&toiletsem, &counter); //  세마포의 현재 counter 값 읽기
    printf("세마포 counter = %d₩n", counter);

    for(int i=0; i<5; i++) pthread_create(&t[i], NULL, guestThread, (void*)name[i]); // 5명의 고객(스레드) >생성

    for(int i=0; i<5; i++) pthread_join(t[i],NULL); // 모든 고객이 소멸할 때까지 대기

    sem_getvalue(&toiletsem, &counter); //  세마포의 현재 counter 값 읽기
    printf("세마포 counter = %d₩n", counter);
    sem_destroy(&toiletsem); // 세마포 기능 소멸

    return 0;
}
```

이 예제는 이름없는 세마포 사용하고 있음.
맥에서는 현재 이름있는 세마포만 지원하므로
실행 안됨. 맥에서는 sem_init() 대신 sem_open()을
사용하세요.

```
$ gcc –o sem sem.c -lpthread
$ ./sem
세마포 counter = 3
고객2 화장실에 들어간다... 세마포 counter = 2
고객3 화장실에 들어간다... 세마포 counter = 1
고객1 화장실에 들어간다... 세마포 counter = 0
고객2 화장실에서 나온다.세마포 counter = 1
고객4 화장실에 들어간다... 세마포 counter = 0
고객3 화장실에서 나온다.세마포 counter = 1
고객1 화장실에서 나온다.세마포 counter = 2
고객5 화장실에 들어간다... 세마포 counter = 1
고객4 화장실에서 나온다.세마포 counter = 2
고객5 화장실에서 나온다.세마포 counter = 3
세마포 counter = 3
$
```

카운터 세마포와 이진 세마포

- 카운터 세마포(counter semaphore)
  - 여러 개의 자원을 관리하는 세마포 (앞서 설명)
- 이진 세마포(binary semaphore)
  - 한 개의 자원을 관리하는 세마포
    - 1개의 자원에 대해 1개의 스레드만 액세스할 수 있도록 보호
    - 뮤텍스와 매우 유사
- 이진 세마포 구성 요소
  1. 세마포 변수 S
     - 0 과 1 중 하나를 가지는 전역 변수, S는 1로 초기화
  2. 대기 큐
     - 사용 가능한 자원이 생길 때까지 스레드들이 대기하는 큐
     - 스레드 스케줄링 알고리즘 필요
  3. 2개의 원자 연산
     - wait 연산(P 연산) – 자원 사용 허가를 얻는 과정
       - S를 1 감소 시키고, 0보다 작으면 대기 큐에서 잠듦. 0보다 크거나 같으면, 자원 사용하는 코드 실행
     - signal 연산(V 연산) – 자원 사용이 끝났음을 알리는 과정
       - S를 1 증가시키고, 0보다 크면 그냥 리턴. 0보다 작거나 같으면 대기 큐에 있는 스레드 중 하나를 깨움

동기화 이슈 : 우선순위 역전

- 우선순위 역전(priority inversion)
  - 스레드의 동기화로 인해 높은 순위의 스레드가 낮은 스레드보다 늦게 스케줄링되는 현상
    - 우선순위를 기반으로 스케줄링하는 실시간 시스템에서,
    - 스레드 동기화로 인해 발생
- 우선 순위 역전 현상 사례
  - 다음 슬라이드
- 우선 순위 역전의 문제점
  - 실시간 시스템의 근본 붕괴
    - 우선순위가 높다는 것은 중요한 일을 할 가능성이 높은데, 높은 순위의 스레드(T3)가 늦게 실행되면 심각한 문제 발생 가능
    - 낮은 순위의 스레드(T2)가 길어지면 더욱 심각한 문제 발생

우선 순위 역전 사례

*** 3가지 가정**

1) 3개의 스레드
   - T3 : 높은 순위의 스레드
   - T2 : 중간 순위의 스레드
   - T1 : 낮은 순위의 스레드
2) T1과 T3가 공유 변수 사용
   - 세마포로 동기화
3) T2는 공유 변수 사용하지 않음

*** 상황 발생**

1) T1이 먼저 도착, P 연산 후 자원 할당
2) 그 다음 T3 도착, T1 중단시키고 T3 실행, T3는 P 연산 내에서 잠듦
3) T1 다시 실행
4) T2 도착, T2는 T1보다 순위가 높고 공유 변수 사용하지 않기 때문에 실행
   -> **우선순위 역전 발생!**
5) T2 종료 후 T1 실행
6) T1의 V 연산 후 T3 실행

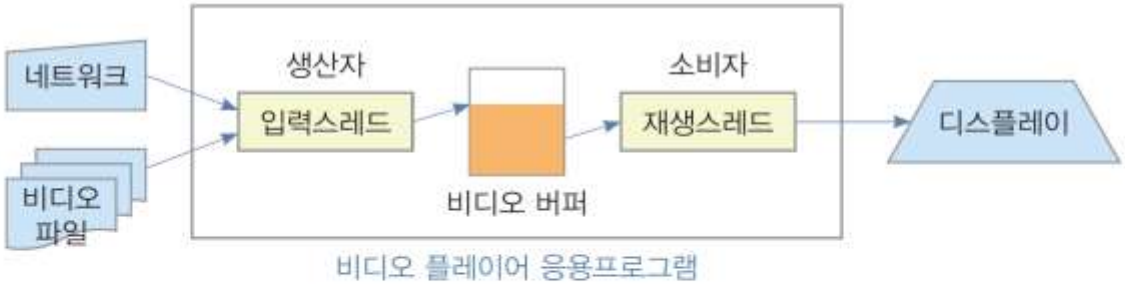
우선순위 역전 해결책

- 2가지 해결 책
  - 우선순위 올림(priority ceiling)
    - 스레드(T1)가 공유 자원을 소유하게 될 때, 스레드의 우선순위를 미리 정해진 높은 우선순위로 일시적으로 올림 T2를 막음
    - 선점되지 않고 빨리 실행되도록 유도
  - 우선순위 상속(priority inheritance)
    - 낮은 순위의 스레드(T1)가 공유 자원을 가지고 있는 동안,
    - 높은 순위의 스레드(T3)가 공유 자원을 요청하면,
    - 공유 자원을 가진 스레드(T1)의 우선순위를 요청한 스레드(T3)보다 높게 설정하여 빨리 실행시킴 //

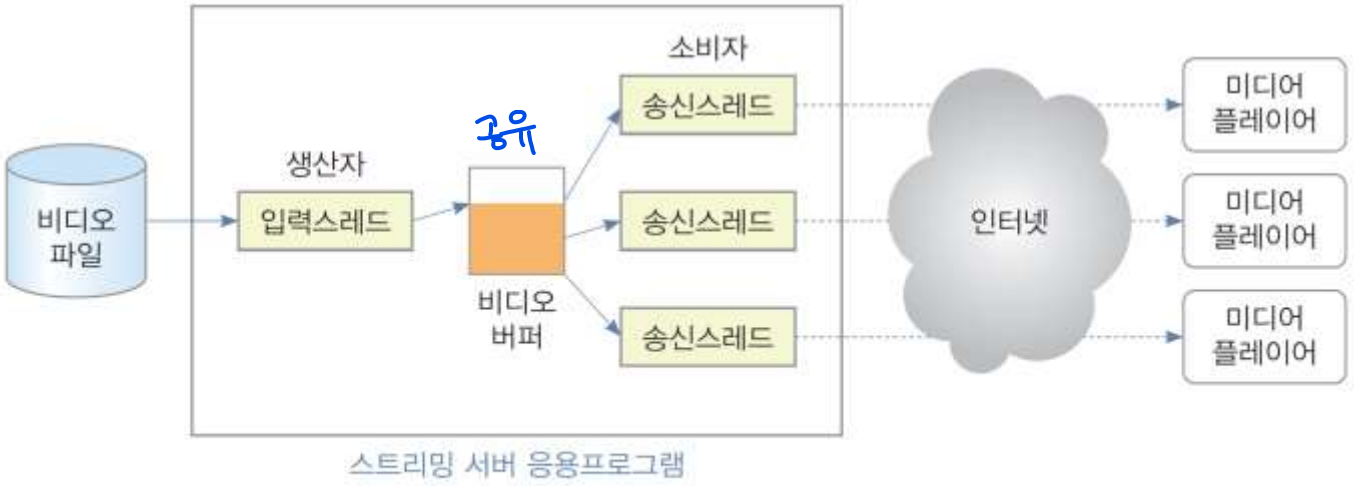
4. 생산자 소비자 문제

응용프로그램에 존재하는 생산자 소비자 문제 사례

* 생산자 소비자 문제는 많은 멀티스레드 응용프로그램에서 발생하는 전형적인 동기화 문제



(a) 미디어 플레이어의 구조(1:1 생산자 소비자 관계)

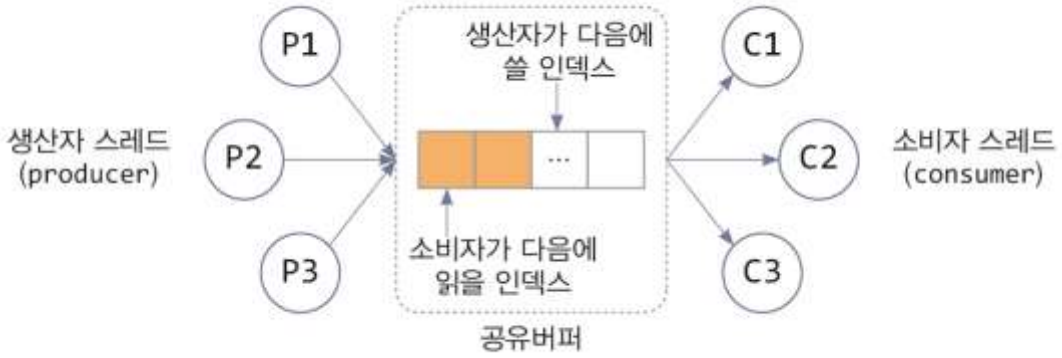


(b) 스트리밍 서버의 구조(1:N 생산자 소비자 관계)

생산자 소비자 문제의 정의

- 생산자 소비자 문제란?
  - 공유버퍼를 사이에 두고, 공유버퍼에 데이터를 공급하는 생산자들과,
  - 공유버퍼에서 데이터 읽고 소비하는 소비자들이,
  - 공유버퍼를 문제 없이 사용하도록 생산자와 소비자를 동기화시키는 문제

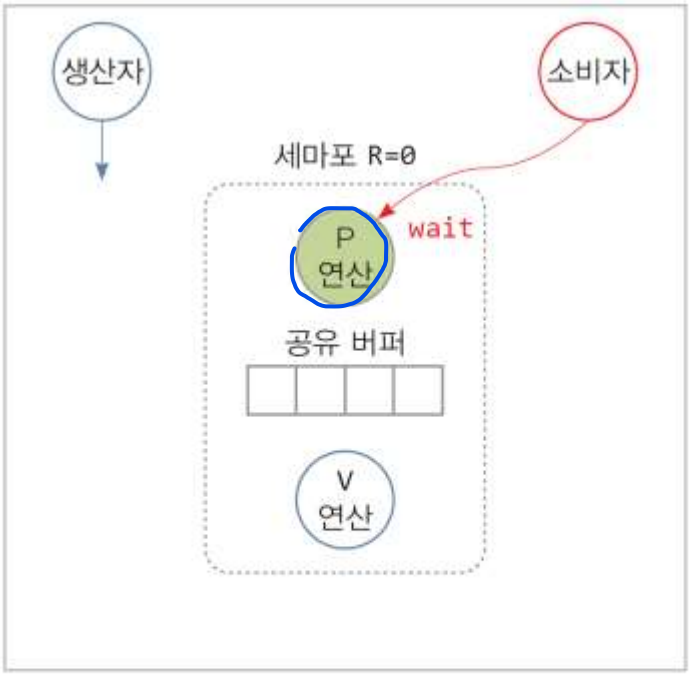
- 생산자 소비자 문제의 구체적인 3가지 문제
  - 문제 1 - **상호 배제 해결**
    - 생산자들과 소비자들의 공유 버퍼에 대한 상호배제
  - 문제 2 - **비어 있는 공유 버퍼 문제**(비어 있는 공유버퍼를 소비자가 읽을 때)
  - 문제 3 - **꽉 찬 공유버퍼 문제**(꽉 찬 공유버퍼에 생산자가 쓸 때)



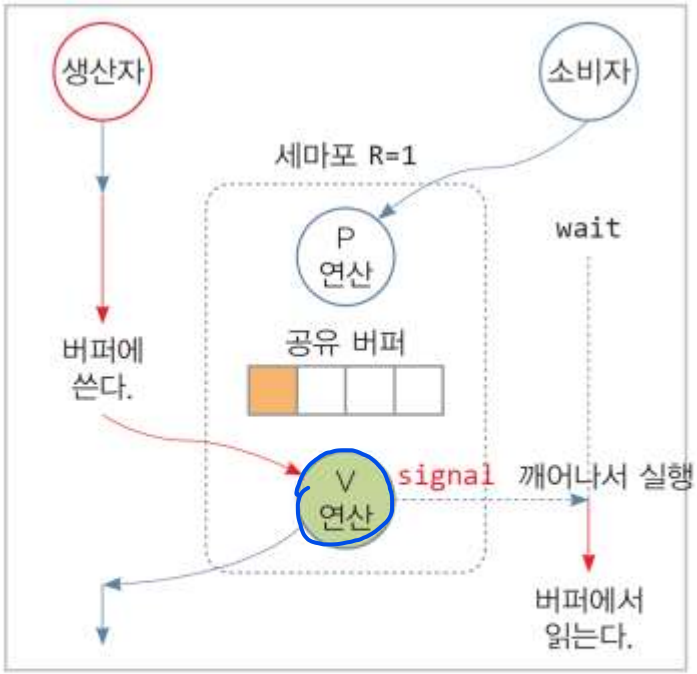
비어 있는 버퍼 문제 해결

- 세마포 R 활용(읽기 가능한 버퍼 개수) : 버퍼가 비어 있는지 살피는 P/V 연산으로 해결

(1) 버퍼가 비어 있는 상태에서
소비자가 읽으려고 할 때



(2) 빈 버퍼에 생산자가 쓸 때



소비자 : 버퍼에서 읽기 전 P 연산 실행

P 연산 : 버퍼가 빈 경우(R=0),
소비자가 잠을 자면서 대기하도록 작성

생산자 : 버퍼에 데이터를 기록하고 V 연산 실행

V 연산 : 세마포 변수 R을 1증가(R=1),
대기 중인 소비자를 깨우도록 작성

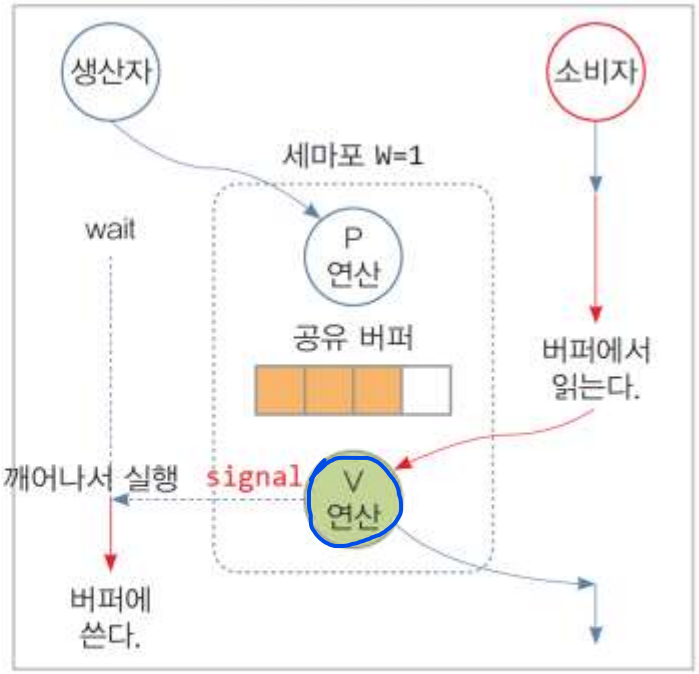
소비자 : 깨어나면 P 연산을 마치고 공유버퍼에서
읽는다. P 연산에서 세마포 R을 1감소(R=0)

꽉 찬 공유 버퍼 문제 해결

- 세마포 W(쓰기 가능한 버퍼 개수) 활용 : 버퍼가 꽉 차 있을 때 처리하는 P/V 연산으로 해결

(1) 공유 버퍼가 찬 상태에서
생산자가 쓰려고 할 때

(2) 버퍼에 데이터가 있는 경우
소비자가 읽을 때



생산자 : 버퍼에 쓰기 전 P 연산 실행

P 연산 : 버퍼가 꽉 찬 상태(W=0)이면,
생산자가 잠을 자면서 대기하도록 작성

소비자 : 버퍼에서 데이터를 읽은 후 V 연산 실행

V 연산 : 세마포 변수 W를 1증가시키고(W=1),
대기 중인 생산자를 깨우도록 작성

생산자 : 깨어나면 P 연산을 마치고 공유버퍼에
쓴다. P 연산에서 세마포 W를 1감소(W=0)

생산자와 소비자 알고리즘

R : 버퍼에 읽기 가능한 버퍼의 개수. 0이면(비어있는 경우) 대기
W : 버퍼에 있는 쓰기 가능한 버퍼의 개수. 0이면(꽉 차있는 경우) 대기
M : 뮤텍스. 생산자 소비자 모두 사용

```
Consumer { // 소비자 스레드
   while(true) {
      P(R); // 세마포 R에 P/wait 연산을 수행하여
            // 버퍼가 비어 있으면(읽기 가능한 버퍼 수=0) 대기한다.

      뮤텍스(M)를 잠근다.
      공유버퍼에서 데이터를 읽는다. // 임계구역 코드
      뮤텍스(M)를 연다.

      V(W); // 세마포 W에 대해 V/signal 연산을 수행하여
            // 버퍼가 비기를 기다리는 Producer를 깨운다.
   }
}
```

```
Producer { // 생산자 스레드
   while(true) {
      P(W); // 세마포 W에 P/wait 연산을 수행하여
            // 버퍼가 꽉 차 있으면(쓰기 가능한 버퍼 수=0) 대기

      뮤텍스(M)를 잠근다.
      공유버퍼에 데이터를 저장한다. // 임계구역 코드
      뮤텍스(M)를 연다.

      V(R); // 세마포 R에 대해 V/signal 연산을 수행하여
            // 버퍼에 데이터가 저장되기를 기다리는 Consumer를 깨운다.
   }
}
```

탐구 6–5 생산자-소비자로 구성된 응용프로그램 만들기

1개의 생산자 스레드와 1개의 소비자 스레드로 구성되는 간단한 응용프로그램을 작성하라.

- 생산자 스레드
  - 0~9까지 10개의 정수를, 랜덤한 시간 간격으로, 공유버퍼에 쓴다.
- 소비자 스레드
  - 공유버퍼로부터 랜덤한 시간 간격으로, 10개의 정수를 읽어 출력한다.
- 공유버퍼
  - 4개의 정수를 저장하는 원형 큐로 작성
  - 원형 규는 배열로 작성
- 2개의 세마포 사용
  - semWrite : 공유버퍼에 쓰기 가능한 공간(빈 공간)의 개수를 나타냄
    - 초기값이 4인 counter 소유
  - semRead : 공유버퍼에 읽기 가능한 공간(값이 들어 있는 공간)의 개수를 나타냄
    - 초기값이 0인 counter 소유
- 1개의 뮤텍스 사용
  - pthread_mutex_t critical_section
  - 공유버퍼에서 읽는 코드와 쓰는 코드를 임계구역으로 설정
  - 뮤텍스를 이용하여 상호배제

탐구 6-5 정답

procon.c

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>

#define N_COUNTER 4 // 공유 버퍼에 저장할 정수 공간의 개수
#define MILLI 1000

void mywrite(int n);
int myread();

pthread_mutex_t critical_section;
sem_t semWrite, semRead; // POSIX 세마포
int queue[N_COUNTER]; // 공유 버퍼
int wptr; // queue[]에 저장할 다음 인덱스
int rptr; // queue[]에서 읽을 다음 인덱스

void* producer(void* arg) { // 생산자 스레드 함수
    for(int i=0; i<10; i++) {
        mywrite(i); // 정수 i를 공유 버퍼에 저장
        printf("producer : wrote %d₩n", i);

        // m 밀리초 동안 잠을 잔다.
        int m = rand()%10; // 0~9 사이의 랜덤한 정수
        usleep(MILLI*m*10); // m*10 밀리초동안 잠자기
    }
    return NULL;
}

void* consumer(void* arg) { // 소비자 스레드 함수
    for(int i=0; i<10; i++) {
        int n = myread(); // 공유 버퍼의 맨 앞에 있는 정수 읽어 리턴
        printf("₩tconsumer : read %d₩n", n);

        // m 밀리초 동안 잠을 잔다.
        int m = rand()%10; // 0~9 사이의 랜덤한 정수
        usleep(MILLI*m*10); // m*10 밀리초 동안 잠자기
    }
    return NULL;
}
```

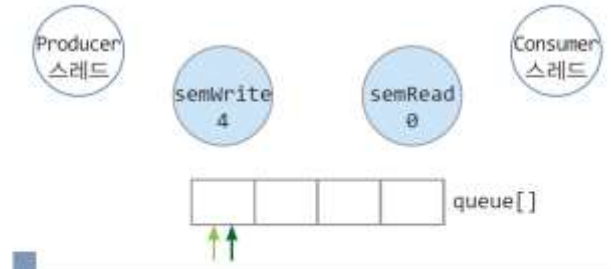
```c
void mywrite(int n) { // 정수 n을 queue[]에 삽입
        sem_wait(&semWrite); // queue[]에 쓸 수 있는지 요청

        pthread_mutex_lock(&critical_section); // 뮤텍스 락 잠그기
        queue[wptr] = n; // 버퍼에 정수 n을 삽입한다.
        wptr++;
        wptr %= N_COUNTER;
        pthread_mutex_unlock(&critical_section); // 뮤텍스 락 열기

        sem_post(&semRead); // consumer 스레드 깨우기
}

int myread() { // queue[]의 맨 앞에 있는 정수를 읽어 리턴
        sem_wait(&semRead); // queue[]에서 읽을 수 있는지 요청

        pthread_mutex_lock(&critical_section); // 뮤텍스 락 잠그기
        int n = queue[rptr]; // 버퍼에서 정수를 읽는다.
        rptr++;
        rptr %= N_COUNTER;
        pthread_mutex_unlock(&critical_section); // 뮤텍스 락 열기

        sem_post(&semWrite); // producer 스레드 깨우기
        return n;
}

int main() {
    pthread_t t[2]; // 스레드구조체

    srand(time(NULL)); // 난수 발생을 위한 seed 생성

    pthread_mutex_init(&critical_section, NULL); // 뮤텍스 락 초기화

   // 세마포 초기화 : N_COUNTER 개의 자원으로 초기화
    sem_init(&semWrite, 0, N_COUNTER); // 가용버퍼의 개수를 N_COUNTER로 초기화
    sem_init(&semRead, 0, 0); // 가용버퍼의 개수를 0으로 초기화

    // producer와 consumer 스레드 생성
    pthread_create(&t[0], NULL, producer, NULL); // 생산자 스레드 생성
    pthread_create(&t[1], NULL, consumer, NULL); // 소비자 스레드 생성

    for(int i=0; i<2; i++)
        pthread_join(t[i],NULL); // 모든 스레드가 소멸할 때까지 대기

    sem_destroy(&semRead); // 세마포 기능 소멸
    sem_destroy(&semWrite); // 세마포 기능 소멸
    pthread_mutex_destroy(&critical_section); // 뮤텍스락 소멸
    return 0;
}
```

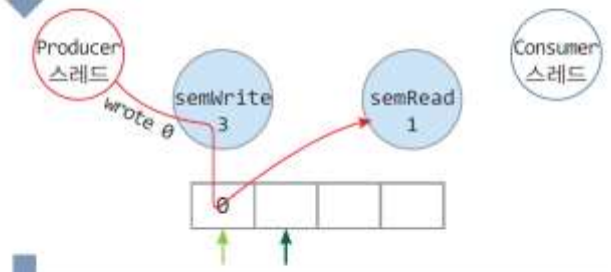
임계구역

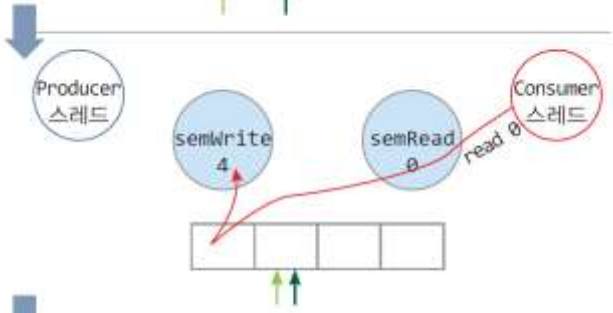
임계구역

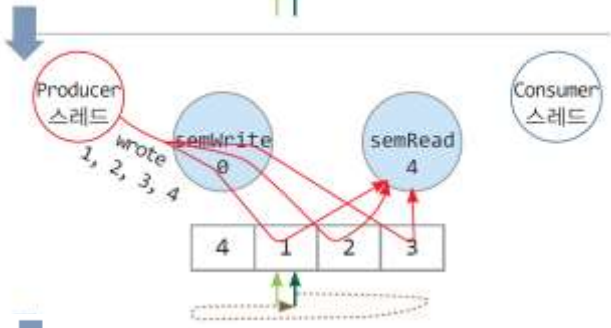
탐구 6-5 컴파일 및 실행

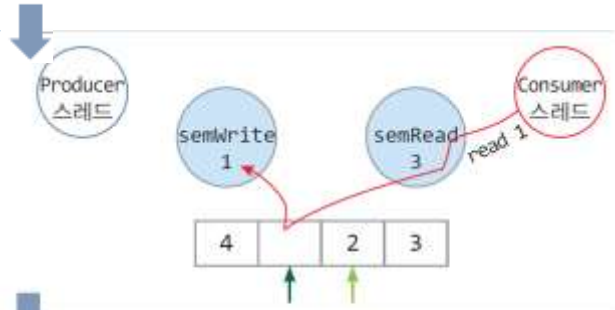

```
$ gcc –o procon procon.c -lpthread
$ ./procon
producer : wrote 0
        consumer : read 0
producer : wrote 1
        consumer : read 1
producer : wrote 2
producer : wrote 3
        consumer : read 2
        consumer : read 3
producer : wrote 4
producer : wrote 5
        consumer : read 4
producer : wrote 6
        consumer : read 5
producer : wrote 7
        consumer : read 6
producer : wrote 8
        consumer : read 7
        consumer : read 8
producer : wrote 9
        consumer : read 9
$
```

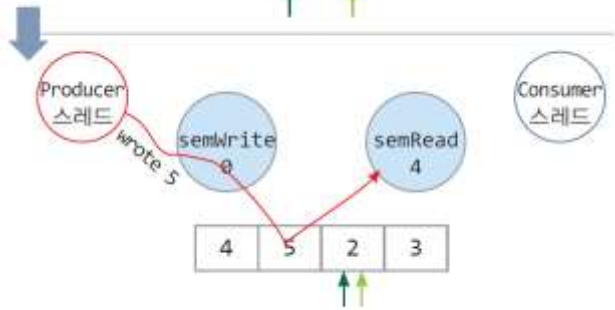












: rptr(다음에 읽을 위치)

: wptr(다음에 쓸 위치)

```
$ ./procon
producer : wrote 0
   consumer : read 0
producer : wrote 1
   consumer : read 1
producer : wrote 2
producer : wrote 3
   consumer : read 2
   consumer : read 3
producer : wrote 4
producer : wrote 5
   consumer : read 4
producer : wrote 6
   consumer : read 5
producer : wrote 7
   consumer : read 6
producer : wrote 8
   consumer : read 7
   consumer : read 8
producer : wrote 9
   consumer : read 9
$
```