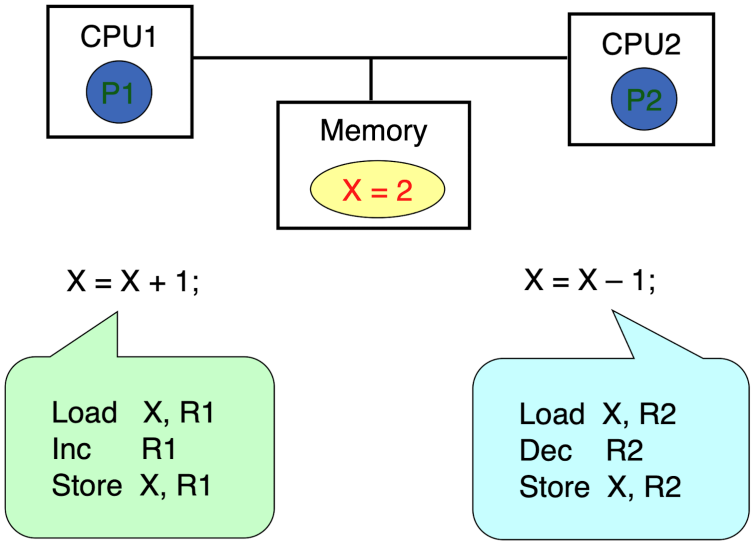
5. Process_Synchronization_1

이거 왜 하냐?

동시에 여러개의 프로세스가 공유 데이터에 동시 접근하면 consistency 보장을 못해



우리가 X++ 하는 것도 한 줄의 코드가 아니라 실제 instruction을 보면 3줄의 코드로 이뤄져 있어

프로세스 P1이 의도한 결과는 X가 3이 되는거고

P2가 의도한 결과는 X가 1이 되는거

근데 이러면 최종 결과값은 2일까? ⇒ ㄴㄴ 모름

그러면 왼쪽에 있는 코드를 순서대로 1, 2, 3

오른쪽에 있는 코드를 순서대로 4, 5, 6 이라고 할 때

실행 순서가 1 부터 순차적으로 실행되면 당연히 X는 2가 맞음

근데 1 4 2 5 3 6 이런식으로 들어오면?

1. R1에 X값 받아옴 (R1 == 2)
4. R2에 X값 받아옴(R2 == 2)
2. R1 값 1증가 (R1 == 3)
5. R2 값 1감소 (R2 == 1)
3. X에 R1 값 저장 (X == 3)
6. X에 R2 값 저장 (X == 1)

이런 경우에는 X가 1로 저장됨 ⇒ 아무렇게나 끼어들 수 있으니 경우는 다양해

그래서 이렇게 공유되는 데이터에 접근하는 코드를 critical section이라고 해

그러면 이걸 어떻게 하냐? ⇒ 특정 프로세스가 critical section을 혼자서 점유할 수 있도록 해야해

Requirements for the Solution

1. Mutual Exclusion(Mutex)

   특정 프로세스가 이미 critical section을 점유하고 있으면 다른 프로세스는 해당 코드에 접근 불가능

2. Progress

   특정 프로세스가 critical section에 진입하고 싶어하는데, 어떤 프로세스도 critical section을 실행하고 있지 않으면 이 process는 지금 들어가야 됨 (다른 프로세스한테 양보 안됨)

3. Bounded Waiting

   starvation 같은 거 방지한다고 생각하면 될듯, 특정 프로세스가 critical section에 진입하려고 할 때 무한정 대기하는 상황은 막아야됨

Initial Attempts to Solve Problem

오직 2개의 프로세스 P0과 P1만 있다고 가정

그리고 이 2개의 프로세스는 critical section에 진입하고자 하는 프로세스들

Algorithm 1

Process P_0

```
do {
    while (turn != 0) ;      /* My  turn?  */
    critical section
    turn = 1;                /* Now it's your turn  */
    remainder section
} while (1);
```

```
P1 입장에서 바라본 코드
do {
    while (turn != 1);
    critical section
    turn = 0;
    remainder section
} while (1);
```

Swap-Turn Problem

이 상황은 2개의 프로세스가 CPU를 활용해가면서 대기하고 있는거

while 문을 돌면서 자기 turn이 오는지 확인하고 자기 turn이라면 진입

근데 이거 문제 있어 ⇒ 뭐가 문제냐? ⇒ swap-turn

실행되고 있는 프로세스가 다른 프로세스의 turn으로 돌려주기 전까지 다른 프로세스가 critical section으로 진입하지 못해

1. turn == 0이라서 P0 가 CS 로 진입
2. CS 를 다 수행하고 turn 1로 변경
3. p0가 remainder section 진행 후 다시 entry section에 와서 CS 진입하려고 시도
4. 근데 P1이 turn을 넘겨 받았는데 CS에 진입하려 하지 않으면? (즉 do while문에 아직 진입하지 않았으면)
5. 그럼 CS가 놀고 있는 거지
6. 이 상태에서 P0은 계속 while문 돌면서 자기 turn인지 확인

Algorithm 2

이전에는 프로세스가 turn을 넘겨줘야만 critical section에 진입할 수 있는 문제가 있었어

⇒ 그러다보니 critical section에 어떤 프로세스도 진입하지 못하는 문제가 발생

이제는 turn을 직접 넘겨주지 말고 내가 들어갈건지를 결정하도록 만들어

그래서 사용한게 배열을 활용한 진입

- Shared variables
  - **boolean** **flag[2]**;
    initially **flag [i] = flag [j] = false;** **/*no one is in CS*/**
  - "P_i ready to enter its critical section" if (**flag [i] == true**)
- Process P_i

critical section

```
do {
    flag[i] = true;          /* Pretend  I am in */
    while (flag[j]) ;        /* Is he also in? then wait*/

    flag [i] = false;        /* I am out now*/

    remainder section

} while (1);
```

flag[i]: process i가 critical section에 진입하고자 하는 의지 표현

- true: CS에 진입할 의지가 있다 (or 이미 실제로 진입해 있을 수 있음)

entry section에서 현 프로세스가 진입할 의지를 나타냄

⇒ while loop을 통해 다른 프로세스가 진입했는지 확인(busy waiting)

만약 다른 프로세스가 없다면 그대로 CS에 진입

exit section에서 빠져나옴

그래서 이건 swap turn 문제는 해결했어 그럼 뭐가 문젤까?

⇒ deadlock

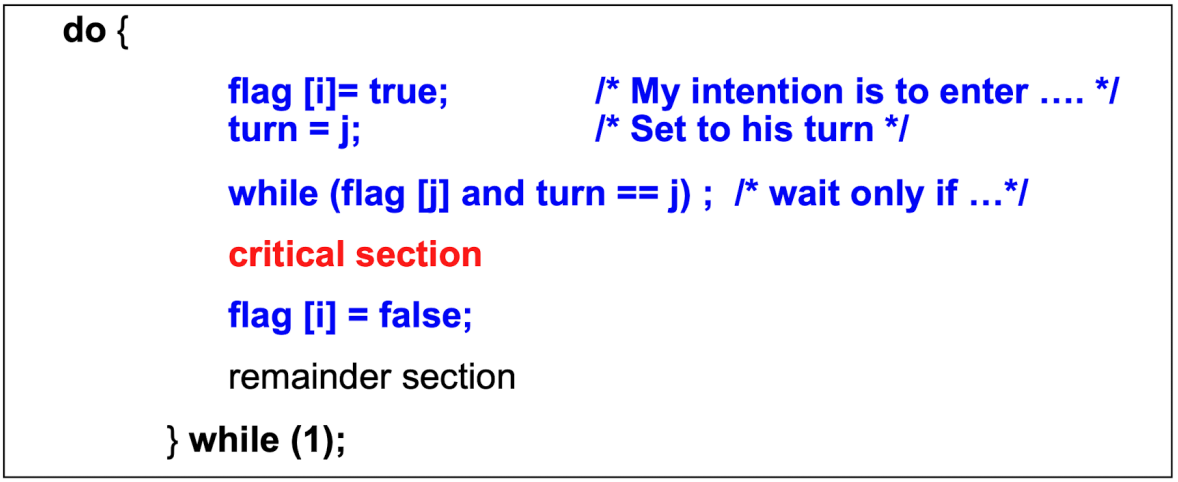
만약 모든 프로세스가 동시에 진입하려고 시도한다면?

그러면 서로 못들어가 (둘 다 들어가고자 하는 의지가 있는데 다른 프로세스가 들어가려고 하면 둘 다 진입하지 않으니까)

Algorithm 3

그래서 나온게 Peterson Algorithm

Algorithm 1에 나온 swap turn을 적용하면서 Algorithm 2에 나온 진입 의지를 동시에 사용



Algorithm 1은 심각한 문제는 아니지만, 성능 저하 문제가 있어 (CS에 진입하지 않으니까)

Algorithm 2는 프로세스가 정지해버리는 문제가 생겨 (서로 미뤄버려서), 근데 2가 1의 문제를 해결

그러면 애초에 Algorithm 2의 문제만 해결해버리면 싹 다 해결되겠네?

근데 이 해결방법이 Algorithm 1에 swap turn에 있어

CS에 진입할 때 프로세스가 진입하려는 의지와 현재 turn이 자신인 경우에만 진입하도록 한다면?

그러면 대부분의 프로세스가 entry section에 대기할 수 있게 되고 (throughput 증가) 자기 차례만 된다면 바로 CS에 진입할 수 있으니까

또한 deadlock 문제도 발생하지 않는게 turn값이 무조건 하나는 보장되기 때문에 서로 대기할 일도 없어

그러면 이 방법의 문제는 뭐냐? ⇒ **Busy Waiting**

결국 대기하는 데에도 자원을 먹게 되는게 가장 큰 문제

Solution to Critical-section Problem using Locks

```
pthread_mutex_t mutex;
do {
    pthread_mutex_lock(mutex);
    // critical section
    pthread_mutex_unlock(mutex);
    // remainder section
} while (true);
```

코드 상으로 이렇게 쓸 수 있어

그러면 lock은 반드시 하나의 프로세스만 점유한다는 걸 어떻게 보장할까?

⇒ Hardware를 통해

Synchronization Hardware

1. 단일 프로세서의 경우

   일단 이거는 신경 안 써도 돼 ⇒ 생각해보면 하나의 CPU밖에 없기 때문에 다른 프로세스가 끼어들 일이 크게 없어 물론 예외가 있지 ⇒ timer interrupt

   근데 인터럽트는 프로세스 실행 중에 꺼버릴 수 있어서 애초에 critical section에 진입하면 interrupt 끄고 CS 진행하고 CS 나오면서 interrupt 켜주면 됨

2. 다중 프로세서의 경우

   이제 이게 문젠데 multiprocessor에서 interrupt 끈다? 운영체제 돌릴 생각이 없단 소리지 뭐

   그래서 가장 단순한 방법 ⇒ hardware의 도움을 받아서 데이터가 atomic하게 실행되는걸 보장하면 돼

가장 대표적인 게 Test-and-Set, Swap

이제 2번의 경우에 대해 알아봅시다

Test-and-Set (TAS)

```
bool TestAndSet(bool &target) {
    bool rv = target;
    target = true;
    return rv;
}
```

돌아가는 로직은 parameter로 들어온 target을 true로 바꾸고 target의 기존 값을 반환하도록 하는거

하드웨어 회로를 구워버려서 더이상 동시성에 대해 고려할 필요가 없어짐 ⇒ 저 3개 line의 코드가 한번에 실행되는 게 보장 돼

```
do {
    while (TestAndSet(lock));
    // critical section
    lock = false;    // TAS에서 lock = true로 변경했기 때문에 다시 false로
    // remainder section
} while (true);
```

상황을 좀 가정해보자면

1. P0 이 CS에 진입 (어떠한 프로세스도 대기중이지 않다고 가정)
2. rv = false, lock = true로 변경
3. P1 이 진입
4. rv = true, lock = true (P1 진입 불가능, P0이 lock을 점유 중에 있어서)
5. P0 이 CS를 끝내고 나오면서 lock = false로 변경
6. P1이 바로 CS에 진입

Compare and Swap(여기서는 편의상 swap code 사용)

```
lock = false;
do {
    key = true;
    while (lock == true) Swap(lock, key);
    // critical section
    lock = false;
    // remainder section
} while (true);
```

key: 내(현재 프로세스)가 들어가고 싶어

lock: 현재 critical section에 프로세스가 있는지 여부

key, lock을 계속 바꿔가면서 확인

만약 lock이 true이면 진입하지 못하게 대기

아 ㅇ ㅓ ㄴ제 끝나냐

Semaphores

지금까지 봐왔던것도 semaphore긴 한데 이게 종류가 2개로 갈려

1. binary semaphore (mutex)

2. semaphore

1의 경우는 프로세스 2개끼리만 resource 접근에 대해 다루는 거

2의 경우가 이제 여러개의 프로세스를 경쟁 구도로 밀어 넣은거

semaphore는 P, V 로 볼건데

1. P: wait function
2. V: signal function

일단 P가 진입하기 위해 확인하는 코드 일거고, V가 CS 점유 끝났을 때 쓰는 코드

```
mutex = 1;
do {
    P(mutex);    // mutex--;
    // critical section
    V(mutex);    // mutex++;
    // remainder section
} while (1);
```

이렇게만 보면 mutex랑 다를게 없는데 가장 큰 장점은 throughput을 향상할 수 있어

즉 entry section에 대기 중인 프로세스의 수를 늘릴 수 있어

```
// initial S = 1
P(S):    while (S <= 0) wait;
S--;

V(S): S++;
```

S: 대기 중인 프로세스의 수

즉 S--는 대기 중인 프로세스의 수가 감소 했다는 소리고 S가 0이면 CS에 접근 할 수 있는 resource가 없음

S++은 CS에서 빠져나오면서 사용할 수 있는 resource의 개수를 증가 시킴

⇒ 지금이야 S = 1로 했지만 만약 사용할 수 있는 resource의 수가 많다면 S = 2, 3, 4, ... 이렇게 더 커지겠지

그래서 여기서 알 수 있는 것

1. Semaphore가 양수 ⇒ 현재 사용가능한 자원의 수
2. Semaphore가 음수 ⇒ 현재 대기중인 프로세스의 수

그러면 binary semaphore는 대기 중인 프로세스가 무조건 1개 일거니까 S의 초기값이 0일거고

나머진 동일

Semaphore Implementation

semaphore를 구현할 수도 있는데 이걸 구현하려면 multiprocessor 상황에서 문제가 생겨

⇒ S값에 접근하는게 critical section이 되어버려

그럼 왜 uniprocessor에서는 문제가 안 생길까? uniprocessor는 interrupt disable이 가능하니까

그래서 S값에 대한 접근을 atomic하게 만드는 방법이 뭘까?

⇒ Peterson Algorithm

그럼 개발자들이 모든 process의 resource 접근에 대해 직접 코드로 구현할 수 있겠네?

ㅇㅇ 가능, 근데 상식적으로 저거 일일이 구현하는 건 말이 안되잖아

그래서 라이브러리에서 저 내용이 다 구현되어 있고 개발자는 편리하게 P, V만 호출하는 거

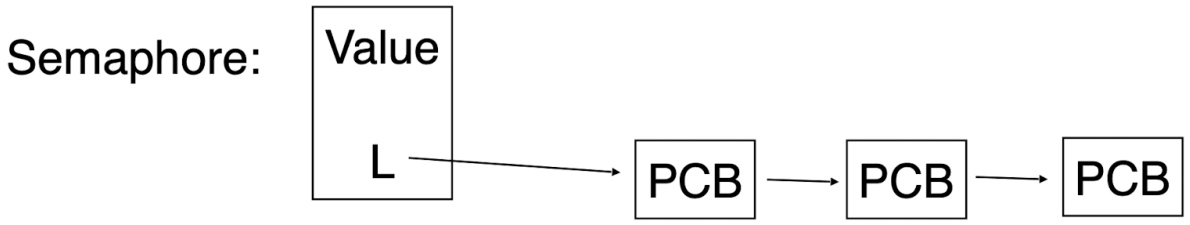
그러면 Peterson Algorithm으로 다 구현하면 되지 않을까?

ㄴㄴ, 이건 busy waiting을 하기 때문에 resource를 쓸데 없이 잡아먹는 문제가 있어

Block / Wakeup Implementation

바로 위에서 busy waiting 이야기 했으니 당연히 얘는 busy waiting에 대한 solution 이겠지

이거는 PCB를 뒤에 주렁주렁 매달아서 실행되고 있는 프로세스가 끝나면 다음 프로세스 깨우는 방식으로 진행



block: kernel이 process를 CS에 진입하지 못하게 막고 저 wait queue 뒤에 붙여버려

wakeup(P): 맨 앞에 있는 프로세스를 ready queue에 집어넣어

이게 busy waiting이랑 뭐가 다르냐?

⇒ thread가 sleep에 들어가(자원 소모를 하지 않아)

그러면 이게 만능 아냐?

⇒ ㄴㄴ... 이건 critical section이 길 때 좋아 반대로 critical section이 짧으면 busy waiting이 더 좋아

이게 생각해보면 당연한게 block-wakeup은 context-switch 비용이 있는데

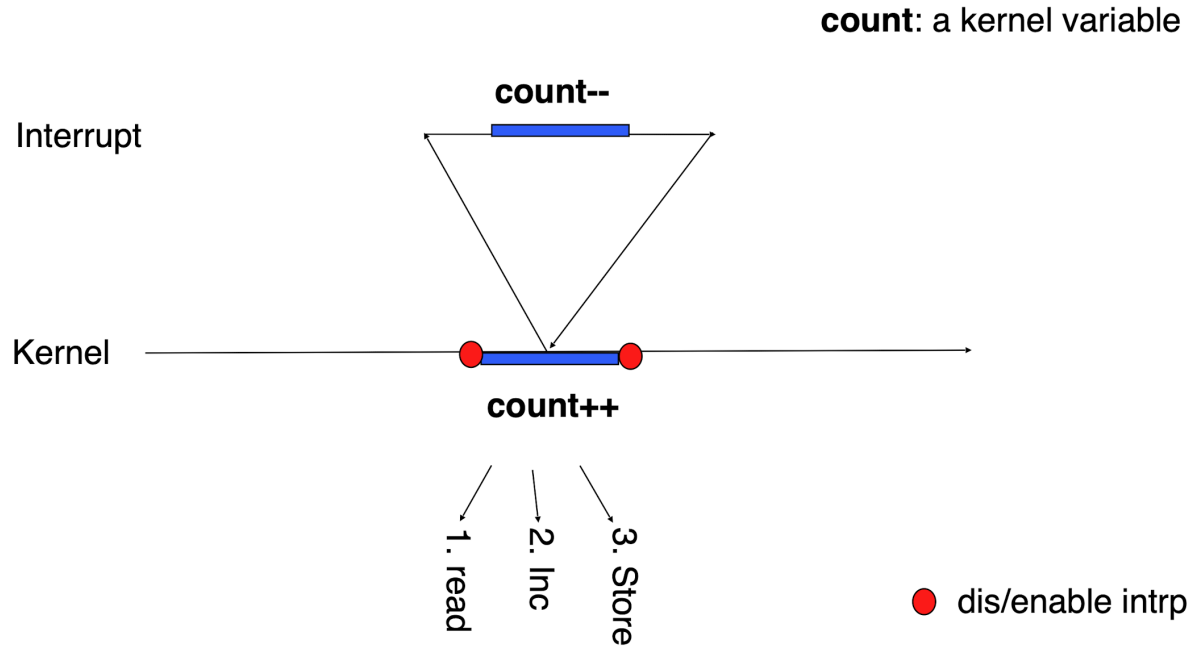
busy waiting의 경우에는 저 비용이 없기 때문에

CS가 짧아서 프로세스 교체가 빈번하게 발생하는 경우에는 busy waiting이 더 유리해

When the CS problem occurs in Operating System?

1. interrupt handler 와 커널 사이
2. kernel이 system call을 수행중일 때 context switch 발생
3. Multiprocessor - kernel data in the shared memory

1. Interrupt handler vs Kernel



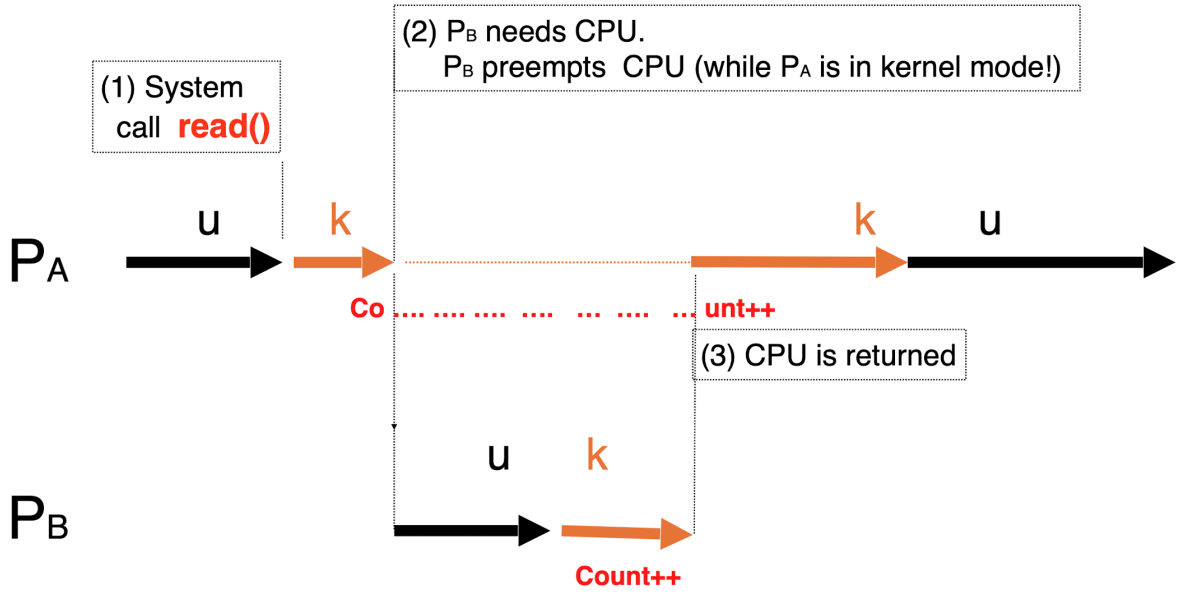
kernel로 진입할 때는 무조건 interrupt disable하고 들어가야 돼

1 과정 이후에 interrupt가 발생한다고 가정해보면

1. register에 count 값 불러옴 (10)
2. interrupt 로 넘어가서 count 1 빼버림 (9)
3. 다시 돌아와서 count값 1 증가 (10)
4. Store (10)

의도했던 값은 11이었는데 이렇게 하니까 10이 저장됨

2. Preempt a process running in kernel



이렇게 돼버리면 count++ 과정에서 다른 프로세스가 접근

다른 프로세스에서도 count++을 호출하면 원래 의도한 값은 +2 인데 결과를 장담 못해

그래서 UNIX에서는 system call이 호출되면 무조건 대기하도록 만들어 (wait)

3. multiprocessor

이거 지금까지 우리가 했던거 (semaphore 로 막았잖아)