CPU Scheduling 2 & Process Synchronization 1

상태	완료
생성 일시	@2023년 7월 11일 오후 11:15
완성일	@2023년 7월 12일
유형	정리

이전 시간에 소개한 개념들에서는 모두 CPU를 할당받기 위해 한 줄 서기를 했다면

오늘 설명할 내용은 여러 줄을 서서 기다리는 것임

멀티 레벨 큐 (Multi-Level Queue)

- Ready Queue를 우선 순위에 따라 여러 개로 분할한다.
  - 빠른 응답을 필요로 하는 대화형 작업은 전위 큐(interactive)에 넣는다.
    - 전위 큐에서는 주로 라운드 로빈 스케줄링 기법(사람과 상호 작용을 하기 때문에)을 사용한다.
  - 계산 위주의 작업은 후위 큐(batch-no human interaction)에 넣는다.
    - 후위 큐에서는 주로 FCFS 스케줄링 기법을 사용한다. (어차피 CPU만 오래 사용하는 프로세스들이라)

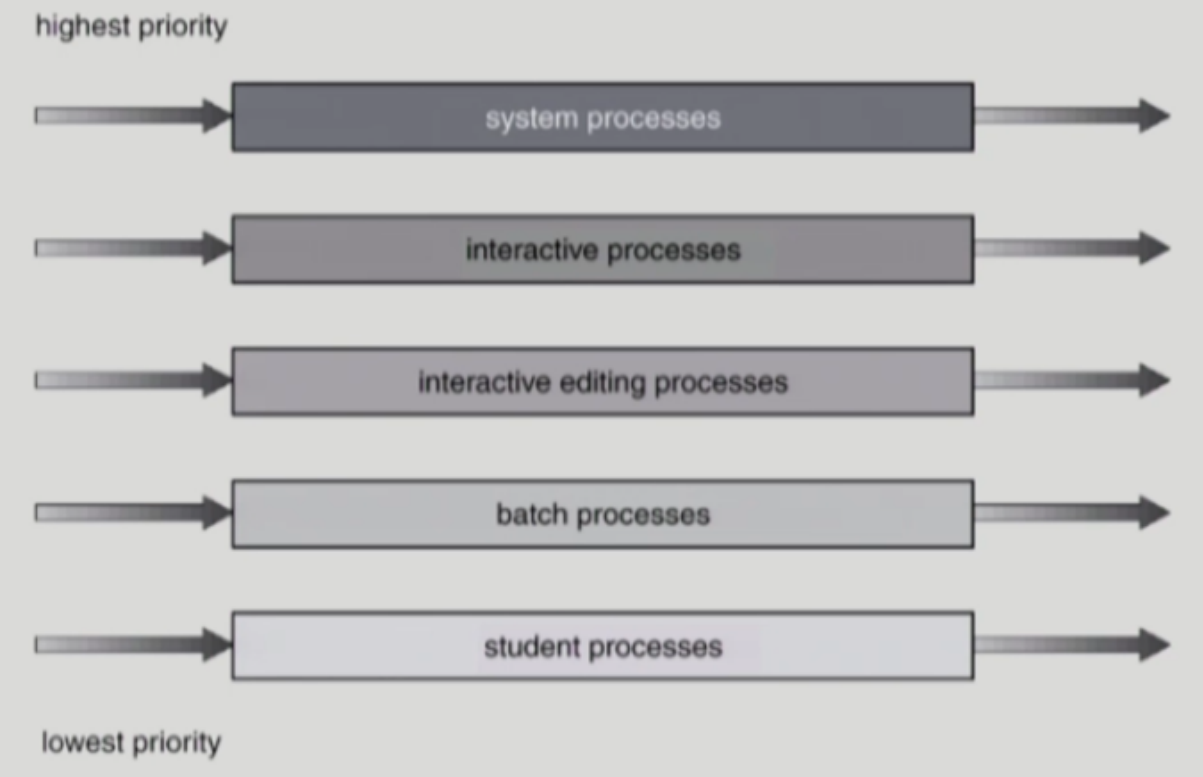
  줄의 특성에 맞는 큐별 스케줄링 방법을 채택해야 함

- 멀티 레벨 큐 자체에 대한 스케줄링이 필요하다.

  줄별로 CPU를 어떻게 할당해 줄 건지

  - 고정 우선 순위 방식 (Fixed Priority Scheduling)
    - 전위 큐에 있는 프로세스에 먼저 CPU가 할당되고, 전위 큐가 비어 있는 경우에만 후위 큐에 있는 프로세스에 CPU가 할당된다.
    - Starvation(기아) 가능성이 존재한다.
  - 타임 슬라이스 방식 (Time Slice)
    - 각 큐에 CPU time을 적절한 비율로 할당한다.

- ex) 80%는 전위 큐, 20%는 후위 큐 (계급이 낮더라도 굶어 죽지 않을 정도로만)



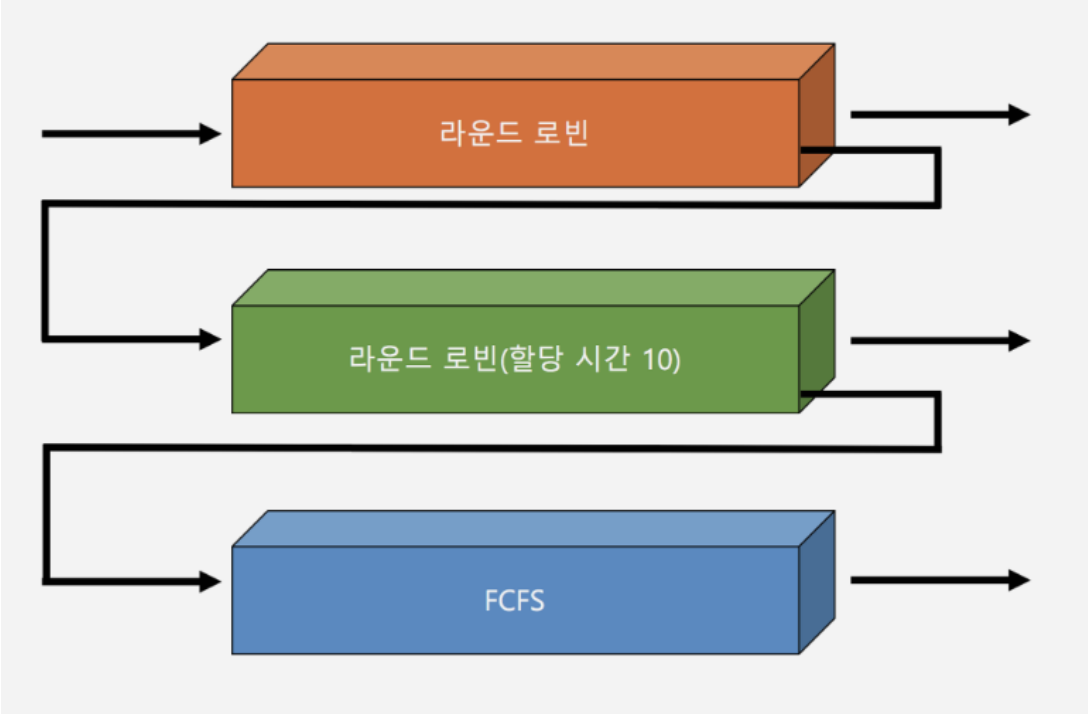
위에서 아래 순으로 우선순위를 나눠서 맞게 줄을 섬

계급사회처럼 나눠 여기에 맞게 줄을 서는 것임 그럼 CPU는 맨 위부터 할당해서 주고 더 이상 없다면 한 줄 내려와서 주고 이런 구조임 (태어난 출신에 따라서 우선순위를 영원히 극복 못 함 : 철저한 계급사회)

멀티 레벨 피드백 큐 (Multi-Level Feedback Queue)

- 프로세스가 여러 개로 분할된 Ready Queue 내에서 다른 큐로 이동이 가능하다.
- 멀티 레벨 피드백 큐를 이용하여 aging 기법으로 구현할 수 있다.
  - aging 기법은 우선 순위가 낮은 큐에서 오래 기다렸으면 우선 순위가 높은 큐로 승격하는 방식이다.
- 멀티 레벨 피드백 큐를 정의하는 요소
  - 큐의 수
  - 각 큐의 스케줄링 알고리즘

- 프로세스를 상위 큐로 승격하는 기준
  - aging 기법
- 프로세스를 하위 큐로 강등하는 기준
- 프로세스가 도착했을 때 들어갈 큐를 결정하는 기준
  - 보통 처음 들어오는 프로세스는 우선순위가 가장 높은 큐에 CPU 할당 시간을 짧게 하여 배치한다.
    - 보통 RR 할당 시간 짧게
  - 만약 주어진 할당 시간 안에 작업을 완료하지 못하면 CPU 할당 시간을 조금 더 주되, 우선순위가 한 단계 낮은 큐로 강등한다.
    - RR 할당 시간 조금 더 할당
  - 이 과정을 반복하다가 최하위 큐에 배치가 된다.
    - 보통 최하단은 FCFS 방식을 사용



현대 사회처럼 출신이 낮아도 올라갈 수 있고 출신이 높았어도 강등당할 수 있는 계급 사회

아주 CPU 버스트가 짧은 프로세스는 도착하자마자 최상위 큐(RR)에 들어가서 CPU를 얻고 짧은 시간 쓰고 바로 빠져나갈 수 있고 CPU 버스트가 좀 긴 프로세스는 맨 위 큐에서 처리가 안 끝나면 아래 큐로 점점 내려와

할당 시간은 조금 더 같게 되지만 내부 큐간의 우선순위에서 보통은 위에 큐가 비어있을 때만 아래 큐를 처리하는 방식을 사용하기 때문에 CPU를 늦게 할당받을 것임

CPU 버스트가 짧은 프로세스에 유리한 방법임

이 방식은 CPU 사용 시간이 긴지 짧은지 이런 예측이 필요 없음. 그냥 처음에 들어올 때는 누구든지 짧은 시간을 주기 때문에

지금까지 말한 CPU 스케줄링은 CPU가 하나일 때를 기준으로 말한 스케줄링 방법이었고

이제부터는 CPU가 여러 개인 경우, 스레드가 여러 개인 경우, 등 특이한 사례에 대한 스케줄링을 어떻게 하는지 살펴볼 것임

다중 처리기 스케줄링 (Multi-Processor Scheduling)

- CPU가 여러 개인 시스템 환경에서 사용하는 기법이다.
- 프로세스를 Ready Queue에 한 줄로 세워서 각 CPU가 알아서 다음 프로세스를 꺼내어 가도록 한다.
- 반드시 특정 프로세서(CPU)에서 수행되어야 하는 프로세스가 있는 경우에는 문제가 더 복잡해짐
- 일부 CPU에 작업이 편중되는 현상을 방지하기 위해 로드 밸런싱 메커니즘을 사용한다.

  특정 CPU만 일하고 나머지 CPU는 놀면 안됨
  - 대칭형 다중 처리
    - 각 CPU가 각자 알아서 스케줄링을 결정
  - 비대칭형 다중 처리
    - 하나의 CPU(전체적인 컨트롤을 담당)가 다른 모든 CPU의 스케줄링 및 데이터 접근을 책임지고 나머지 CPU는 거기에 따라 움직이는 방식

실시간 스케줄링 (Real-time Scheduling)

보통은 그때그때 스케줄링하기보다는 realtime JOB 들이 주어져 있으면 미리 스케줄링하여 적재적소에 배치되도록 함

- 정해진 시간 안에 반드시 실행이 되어야 하는, 즉 deadline이 있는 프로세스일 때 사용하는 기법이다.
- 경성 실시간 시스템 (hard real-time system)

    - 정해진 시간 안에 반드시 작업이 끝나도록 스케줄링해야 한다.
- 연성 실시간 시스템 (soft real-time system)
    - 데드라인이 존재하기는 하지만 지키지 못했다고 해서 위험한 상황이 생기지는 않는다.
    - 일반 프로세스에 비해 높은 우선 순위를 갖도록 구현한다.

스레드 스케줄링 (Thread Scheduling)

- Local Scheduling
    - 유저 레벨 스레드의 경우 운영체제가 해당 스레드의 존재를 모른다.
    - OS가 아닌 사용자 프로세스가 직접 어느 스레드한테 CPU를 줄 것인지 결정한다.

    운영체제는 스레드의 유무를 모르기 때문에 해당 프로세스에게 CPU를 줄지 말지만 정하는 것이고

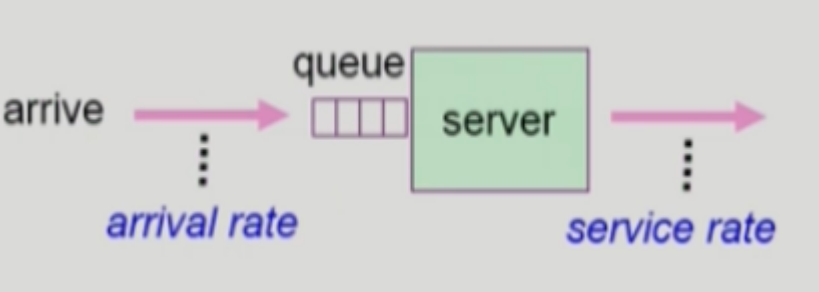
    해당 프로세스가 CPU를 잡았을때 어느 스레드에게 CPU를 줄지는 프로세스 내부에서 정하는 것임 ex 코드의 따라
- Global Scheduling
    - 커널 레벨 스레드의 경우 일반 프로세스처럼 커널의 단기 스케줄러가 어떤 스레드를 스케줄 할지 결정한다.

어떤 알고리즘이 좋은가 평가 방법

스케줄링 알고리즘의 평가

큐잉 모델

- 주로 이론가들이 수행하는 방식이다.
- 확률 분포를 통해 프로세스들의 도착률과 CPU의 처리율을 입력값으로 주면, 복잡한 수학적 계산을 통해 각종 성능 지표인 CPU의 처리량, 프로세스의 평균 대기 시간 등을 구하게 된다.



구현 및 실축

- 주로 구현가들이 수행하는 방식이다.
- 운영 체제 커널의 소스 코드 중 CPU 스케줄링을 수행하는 코드를 수정해서 커널을 컴파일한 후 시스템에 설치한다. 그런 다음 동일한 프로그램을 원래 커널과 CPU 스케줄러를 수정한 커널에서 수행해 보고 실행 시간을 측정한다. (쉽게 내가 구현한 스케줄링 코드와 기존의 것을 각각 컴퓨터에 설치하여 비교하는 방법)

시뮬레이션

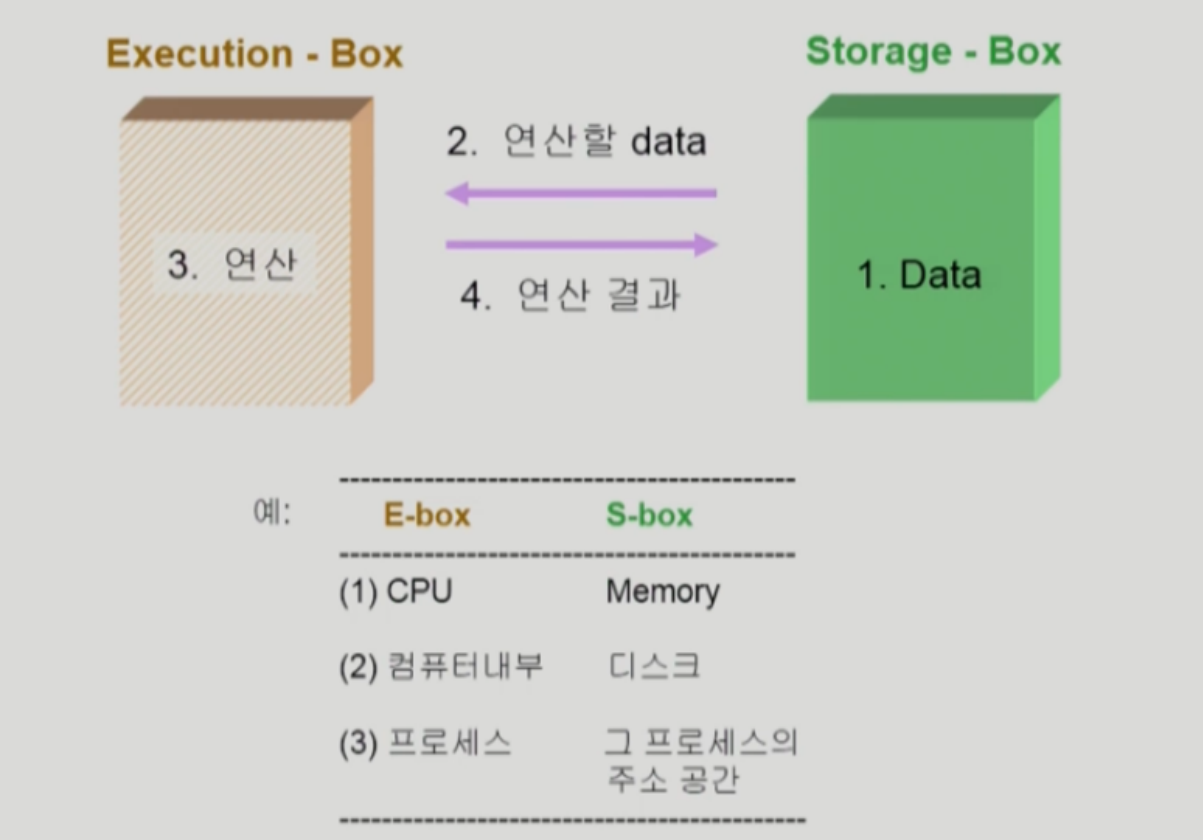
- 가상으로 CPU 스케줄링 프로그램을 작성한 후 프로그램의 CPU 요청을 입력값으로 넣어 어떠한 결과가 나오는지 확인한다.
- 입력값은 가상으로 생성할 수도 있고 실제 시스템에서의 CPU 요청 내역을 추출해 사용할 수도 있다.
    - 실제 시스템에서 추출한 입력값을 트레이스 (trace)라고 부른다.
    - 트레이스는 몇 초에 어떤 프로세스가 도착하고, 각각 CPU 버스트 시간을 얼마로 하는지에 대한 정보를 시간 순서대로 적어 놓은 파일을 말한다.

Process Synchronization(동기화)1

해당 개념을 설명하기에 앞서 아래 개념을 알고 가야 함

데이터 접근

컴퓨터 시스템 안에 어떤 데이터에 접근하는 것은 아래와 같은 과정(패턴)을 통하게 되어있음



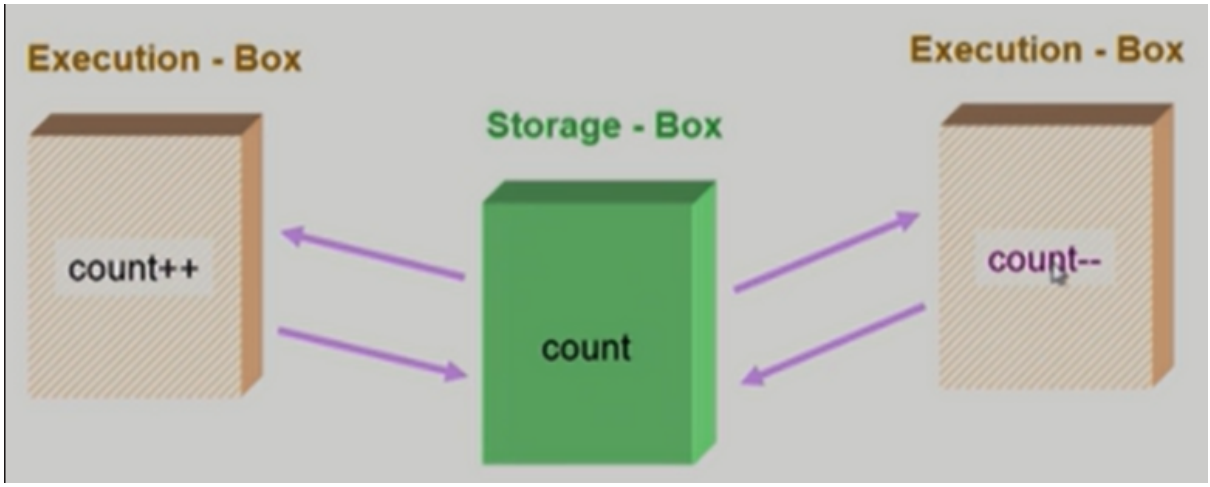
컴퓨터 시스템에서 데이터 연산은 저장 공간과 실행 공간이 아래와 같은 흐름으로 동작하면서 이루어진다.

1. 저장 공간에 데이터가 있다.
2. 연산할 데이터를 실행 공간으로 가져온다.
3. 실행 공간에서 연산한다.
4. 연산 결과를 저장 공간에 반영한다.

저장 공간은 메모리나 해당 프로세스의 주소 공간, 디스크 등이 있고, 실행 공간은 CPU나 프로세스, 컴퓨터 내부 등이 있다.

이런 식의 방식에서는 누가 먼저 읽었느냐에 따라 결과가 달라질 수 있고 이러한 문제를 Synchronization(동기화)문제라고 함

Race Condition(경쟁조건)



저장 공간을 공유하는 실행 공간이 여러 개 있는 경우 Race Condition의 가능성이 있다. 예를 들어 메모리에 count 변수가 있고 서로 다른 CPU가 각각 증가 연산, 감소 연산을 한다고 가정해 보자.

정상적인 동작을 한다면 CPU A가 count 변수를 가져와 1 증가하여 메모리에 반영하고, CPU B가 count 변수를 가져와 1 감소하고 다시 메모리에 반영하여 count 변수 값의 변화가 없을 것이다.

하지만 CPU A에서 증가 연산을 하는 동안 CPU B가 메모리에 있는 count 변수를 가져가서 연산한다면 A가 연산한 것을 저장하고 B가 저장한 것을 저장 다시 저장할때 A가 연산한 결과는 skip 되버려 B 결과 `(count - 1)`이 저장될 것이다.

이처럼 공유하는 하나의 주체에 여러 주체가 마치 경쟁하듯 접근하려 하는 것을 경쟁 상태, 즉 race condition이라고 한다.

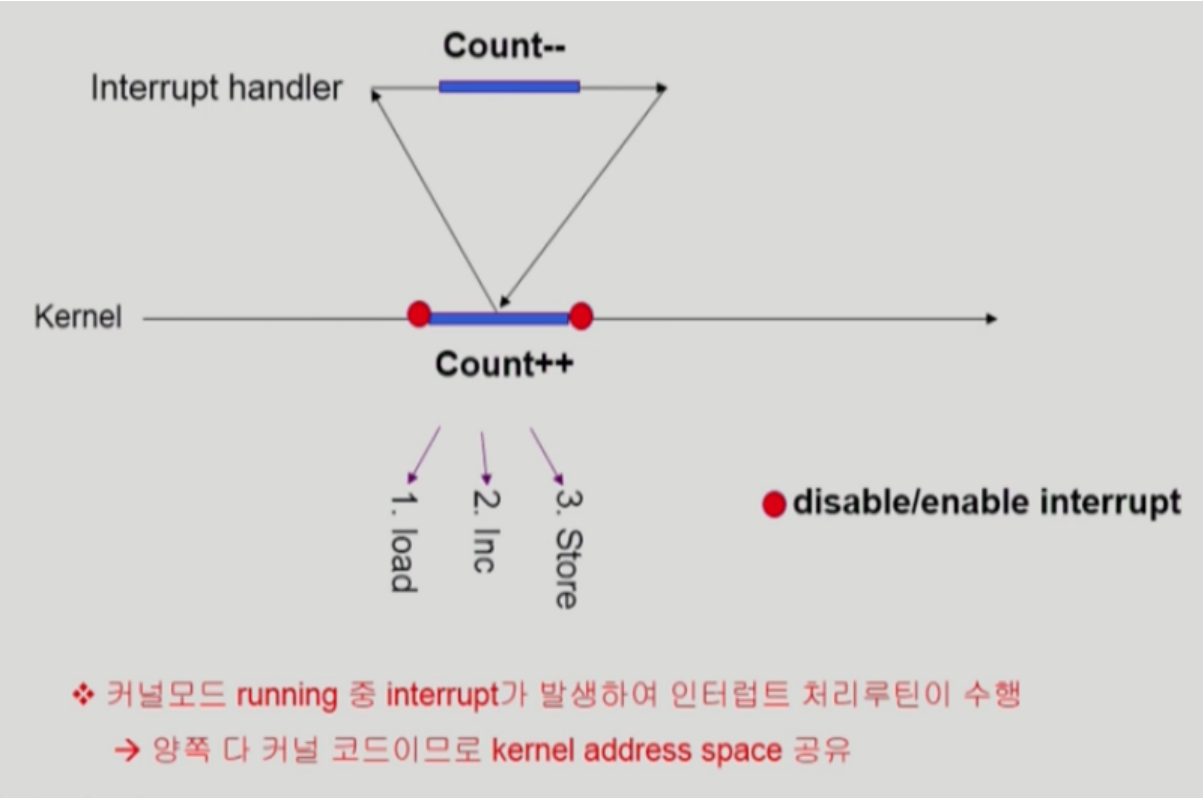
- 운영 체제에서 race condition이 발생하는 상황
  - 커널 수행 중 인터럽트 발생 시
  - 프로세스가 시스템 콜을 호출하여 커널 모드로 수행 중인 가운데 context switch가 일어나는 경우
  - 멀티 프로세서에서 공유 메모리 내의 커널 데이터

> 유저 레벨에서는 문제가 생기지 않던 것(프로세스별 각자의 공간을 가지니까)이 커널로 들어가게 되면 커널의 데이터는 공유되기 때문에 즉 여러 프로세스가 동시에 사용할 수 있는 데이터라 문제가 될 수 있는 것임

*운영 체제에서의 race condition (*발생하는 상황*)*

인터럽트 핸들러 vs 커널

커널 모드가 수행 중인 상태에서 인터럽트가 발생하여 인터럽트 처리 루틴이 실행되는 상황이다.

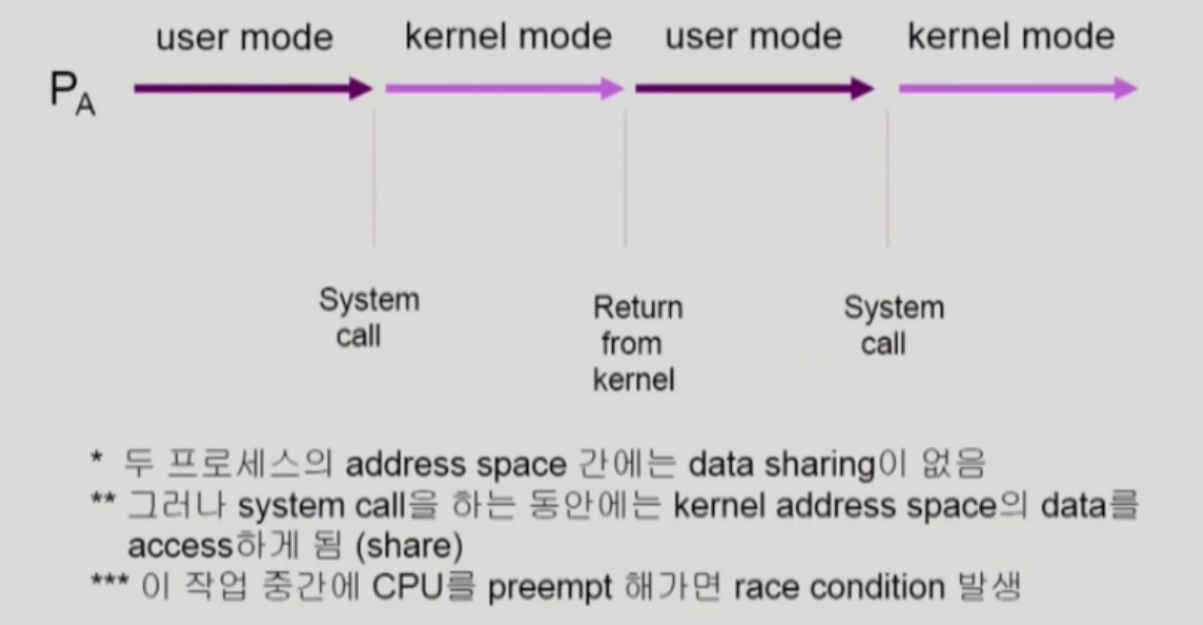


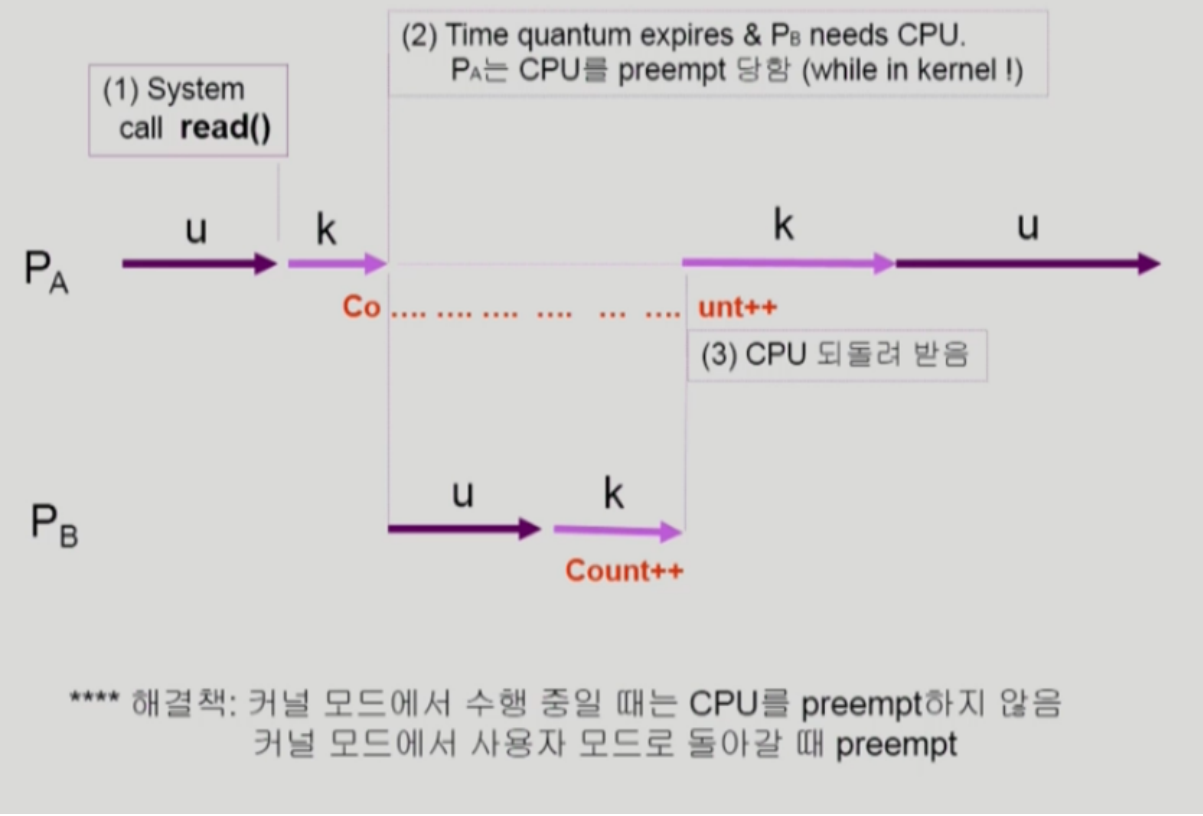
커널이 증감 연산을 할 때 메모리에서 count 변수를 load하고, count 변수를 증감하고 count 변수를 메모리에 store하는 3가지 연산이 발생한다.

만약 load 연산을 수행한 후 증감 연산을 수행하기 전에 인터럽트가 들어왔다고 가정하자. 그러면 인터럽트 핸들러를 통해 인터럽트를 수행할 것이다. 이때 하필 인터럽트가 공유 변수인 count를 1 감소하는 연산이었고, 인터럽트가 끝난 이후 커널은 이전 연산 과정인 load 이후부터 수행하므로 count는 감소하지 않은 상태이다. 결과적으로 count는 기존 값에서 1이 증가된 수치가 된다.

이처럼 중요한 변수의 값을 건드리는 동안에는 인터럽트가 발생해도 연산이 끝나고 수행될 수 있게끔 disable 처리를 해 준다. 즉 인터럽트 들어와도 내가 하는 연산이 완료될 때까지는 처리 안 하는 것임

커널 내에서 실행 중인 프로세스를 선점하는 경우





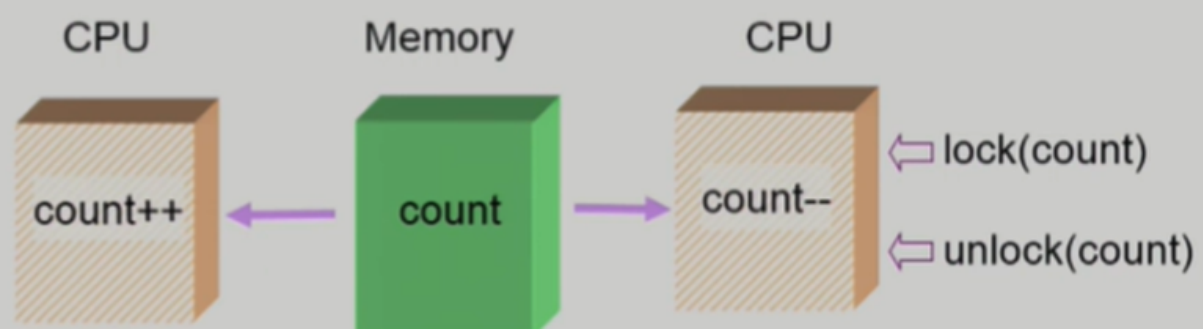
프로세스 A가 count 변수를 load하고, 1 증가하기 위해 시스템 콜을 호출하여 커널 모드에서 작업을 처리하는 도중에 CPU 할당 시간이 끝나 context switch가 발생하였다. 이후 프로세스 B가 CPU를 할당받아 A와 동일하게 시스템 콜을 호출했고, count 변수를 1 증가하였다. 그리고 다시 context switch가 발생하여 프로세스가 A가 CPU를 잡아 count 변수를 1 증가하였다.

count 변수 값이 2 증가해야 하는데 결과는 그렇지 않다. 프로세스 A는 프로세스 B가 증가 연산을 하기 전 count 값을 갖고 있었고, CPU를 다시 할당받았을 때 그 시점의 context를 가지고 값을 증가하였기 때문에 결과적으로 1만 증가하였다.

이를 해결하기 위해 커널 모드에서 수행 중일 때는 CPU 할당 시간이 끝나도 선점하지 않고, 사용자 모드로 돌아갔을 때 선점한다. (시분할 시스템은, 실시간 처리 시스템(Real Time Processing System)이 아님 조금 시간이 더 지났다고 해서 시스템의 큼큰 문제가 생기지는 않는 시스템임)

멀티 프로세서

이 상황에서는 앞에서 설명한 어떠한 방법으로도 해결하지 못함 == interrupt enable/disable



CPU가 메모리에서 데이터를 가져오기 전에 lock을 걸어 다른 CPU가 같은 데이터에 접근하는 것을 막아 준다. 연산이 끝난 후 데이터를 다시 메모리에 저장할 때 lock을 풀어 줌으로써 다른 CPU가 접근할 수 있게 해 준다.

- 방법 1
  - 한 번에 하나의 CPU만 커널에 들어갈 수 있게 하는 방법
  - 커널 전체에 lock을 걸기 때문에 비효율적이다.
    - (CPU 하나만 커널 들어가서 작업하고 나머지는 놀고 있으니까)
- 방법 2
  - 커널 내부에 있는 각 공유 데이터에 접근할 때마다 해당 데이터에 lock을 거는 방법

프로세스 동기화 문제

- 공유 데이터의 동시 접근은 데이터의 불일치 문제를 유발할 수 있다.

- 일관성 유지를 위해서는 협력 프로세스 간의 실행 순서를 정해 주는 메커니즘이 필요하다.
    - 하나가 읽어서 수정하는 동안에는 다른 프로세 접근을 막는
- Race condition
    - 여러 프로세스가 동시에 공유 데이터에 접근하는 상황
    - 데이터의 최종 연산 결과는 마지막에 그 데이터를 다룬 프로세스에 따라 달라짐
- race condition을 막기 위해서는 병행 (concurrent) process는 동기화되어야 한다.

▼ 병행(concurrent) 프로세스란?

병행(concurrent) 프로세스에 대해 설명하면, 여러 개의 프로세스가 동시에 실행되는 것을 의미합니다. 이때 동기화(synchronization)가 필요한 이유는 다음과 같습니다.

병행 프로세스들은 서로 다른 메모리 주소 공간에서 작동합니다. 그래서 어떤 작업을 처리한 프로세스의 결과물을 다른 프로세스가 바로 알 수 없으며, 그 결과 다른 프로세스가 이전에 처리했던 작업의 결과물을 기반으로 작업을 수행한다면, 위험한 결과를 초래할 수 있습니다.

따라서, 프로세스 간 데이터를 공유해야 하는 경우, 즉 둘 이상의 프로세스가 동시에 하나의 자원(예: 파일, 메모리 등)을 사용할 경우에는 이를 동기화하여 작업을 수행해야 합니다. 이렇게 동기화를 수행하면 각각의 프로세스를 동작하면서 발생할 수 있는 문제를 충분히 방지할 수 있습니다.

▼ 동기화란?

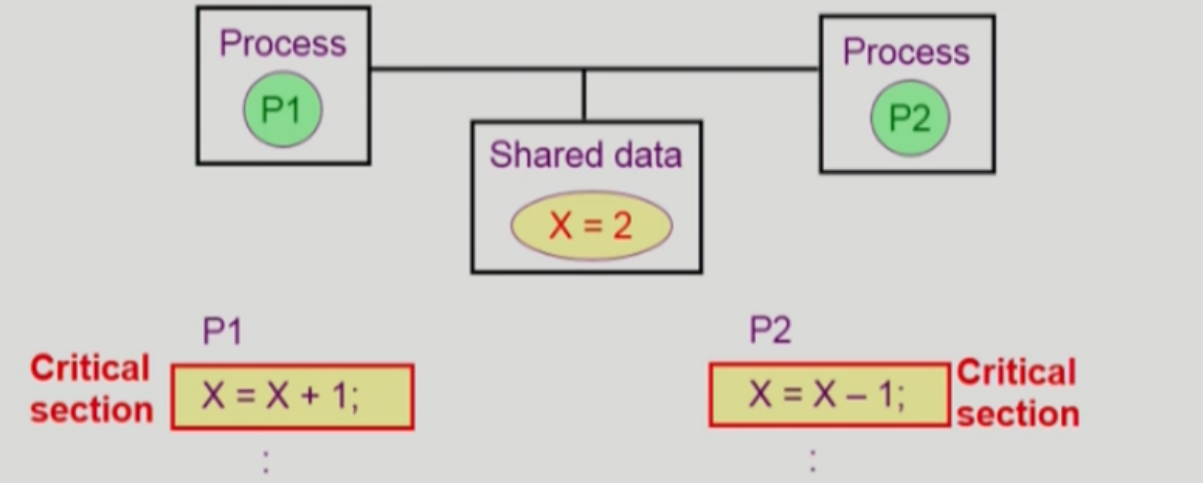
동기화(synchronization)란, 멀티 스레드, 멀티 프로세스 등에서 공유 자원에 대한 접근을 조절하는 것입니다. 여러 개의 스레드나 프로세스에서 동일한 자원에 접근하면, 데이터의 무결성과 일관성을 보장할 수 없습니다. 이런 문제를 해결하기 위해, 공유 자원의 접근을 동기화하는 것이 필요합니다.

동기화의 기본 개념은 단일 스레드처럼 동작하게 만드는 것입니다. 각 스레드(혹은 프로세스)는 자원에 접근할 때, 다른 것이 이를 접근하지 못하도록 차단하고, 자원을 사용한 후 다른 스레드의 접근이 가능하도록 해줍니다.

일반적으로, 동기화를 구현하는 방법에는 뮤텍스(Mutex), 세마포어(Semaphore) 등의 도구를 사용합니다. Mutex는 공유 자원에 대한 단독 액세스를 보장하는 락(lock)입니다. Semaphore는 공유 자원에 대한 접근을 n개의 스레드까지 허용하는 카운팅 락(counting lock)입니다. 이러한 락들을 사용하여 여러 스레드, 프로세스 간의 공유 자원에 대한 안전한 접근을 보장할 수 있습니다.

임계 구역 문제 (The Critical-Section Problem)

- n 개의 프로세스가 공유 데이터를 동시에 사용하기를 원하는 경우
- 각 프로세스의 code segment에는 공유 데이터를 접근하는 코드인 임계 구역 (critical section)이 존재한다.
- 하나의 프로세스가 임계 구역에 있을 때 다른 모든 프로세스는 임계 구역에 들어갈 수 없어야 한다.



Shared data가 critical section이 아님

즉 p1이 공유 중인 데이터(critical section)를 실행중이면 CPU를 뺏겨서 다른 프로세스에 CPU가 넘어가더라도 공유 데이터에 접근하는 critical section에 들어가지 못하는 것임 p1이 critical section을 빠져나왔을 때 즉 공유 데이터로부터 자유로워졌을 때 p2가 접근할 수 있음 (p1 끝날 때 까지 기다려야 함)

▼ 임계구역이란?

임계 구역(critical section)은 프로그램에서 공유 자원을 변경하거나, 여러 스레드나 프로세스에서 함께 접근하는 코드 부분을 말합니다. 이 부분이 잘못 구현되면, 다른 스레드나 프로세스에서 같은 자원에 접근할 때 충돌이 발생하여 데이터의 무결성과 일관성, 그리고 안정성을 보장할 수 없습니다.

따라서, 임계 구역에 대한 동기화 처리를 제대로 처리해야 합니다. 스레드나 프로세스들이 공유하는 자원에 대한 접근을 제어하는 것이 핵심입니다. 이를 위해 뮤텍스, 세마포어 같은 동기화 도구를 사용하여 상호배제(mutual exclusion)를 구현합니다.

즉, 임계 구역에서 문제가 발생하지 않도록, 여러 스레드나 프로세스가 동시에 접근할 수 없도록 보호하고, 접근 권한을 획득한 단 하나의 스레드나 프로세스만이 해당 자원을 사용하도록 해줍니다.