**챕터3 프로세스**

**3.1 프로세스 개념**

프로세스란 실행 중인 프로그램을 말한다

하나의 시스템은 일부는 사용자 코드를 실행하고 일부는 운영체제 코드를 실행하는 프로세스의 집합체이다

초창기 컴퓨터는 작업(job)을 실행하는 일괄처리 시스템이었고, 사용자 프로그램 또는 태스크(task)를 실행하는 시분할 시스템이 뒤를 이었다

**3.1.1 프로세스**

프로세스란 실행 중인 프로그램이다

프로세스의 현재 활동의 상태는 프로그램 카운터 값과 프로세서 레지스터의 내용으로 나타낸다

프로세스의 메모리 배치는 일반적으로 여러 섹션으로 구분된다

텍스트 섹션 – 실행 코드

데이터 섹션 – 전역 변수

힙 섹션 – 프로그램 실행 중에 동적으로 할당되는 메모리

스택 섹션 – 함수를 호출할 때 임시 데이터 저장장소(예: 함수 매개변수, 복귀 주소 및 지역 변수)

프로그램 그 자체는 프로세스가 아니다

프로그램은 명령어 리스트를 내용으로 가진 디스크에 저장된 파일(실행 파일이라고 불림)과 같은 수동적인 존재(passive entity)이다

프로세스는 다음에 실행할 명령어를 지정하는 프로그램 카운터와 관련 자원의 집합을 가진 능동적인 존재(active entity)이다

실행 파일이 메모리에 적재될 때 프로그램은 프로세스가 된다

두 프로세스들이 동일한 프로그램에 연관될 수 있지만 이들은 두 개의 별도의 실행 순서로 간주한다

**3.1.2 프로세스 상태**

프로세스는 실행되면서 그 상태가 변한다

New: 프로세스가 생성 중이다

Running: 명령어들이 실행되고 있다

Waiting: 프로세스가 어떤 이벤트(입출력 완료 또는 신호의 수신 같은)가 일어나기를 기다린다

Ready: 프로세스가 처리기에 할당되기를 기다린다

Terminated: 프로세스의 실행이 종료되었다

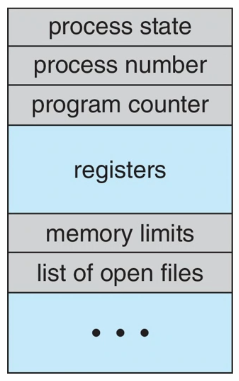
어느 한 순간에 한 CPU에서는 오직 하나의 프로세스만이 실행된다는 것을 인식하는 것이 중요하다

그렇지만, 많은 프로세스가 ready 및 waiting 상태에 있을 수 있다

**3.1.3 프로세스 제어 블록**

각 프로세스는 운영체제에서 프로세스 제어 블록(PCB, also called task control block)에 의해 표현된다

프로세스 제어 블록은 특정 프로세스와 관련된 여러 정보를 수록하며, 다음과 같은 것들을 포함한다



프로세스 상태: new, ready, running, waiting, 또는 halted(정지) 상태 등이다

프로그램 카운터: 다음에 실행할 명령어의 주소

CPU 레지스터들: 모든 프로세스 중심 레지스터들의 내용, 프로그램 카운터와 함께 이 상태 정보는 인터럽트 발생 시 저장되어야 한다

CPU 스케줄링 정보: 프로세스 우선순위, 스케줄 큐에 대한 포인터와 다른 스케줄 매개변수를 포함한다

메모리 관리 정보: 프로세스에 할당된 메모리

Accounting 정보: CPU 사용 시간과 경과된 시간, 시간 제한, 계정 번호, 잡 또는 프로세스 번호 등을 포함한다

I/O status 정보: 프로세스에 할당된 입출력 장치, 열린 파일의 목록

요약하자면 프로세스 제어 블록은 약간의 accounting 데이터와 함께 프로세스를 시작하거나 다시 시작시키는 데 필요한 모든 데이터를 위한 저장소의 역할을 한다

**3.2 프로세스 스케줄링**

다중 프로그래밍의 목적은 CPU 이용을 최대화하기 위하여 항상 어떤 프로세스가 실행되도록 하는 데 있다

시분할의 목적은 각 프로그램이 실행되는 동안 사용자가 상호 작용할 수 있도록 프로세스들 사이에서 CPU 코어를 빈번하게 교체하는 것이다

이 목적을 달성하기 위해 프로세스 스케줄러는 코어에서 실행 가능한 여러 프로세스 중에서 하나의 프로세스를 선택한다

현재 메모리에 있는 프로세스 수를 다중 프로그래밍 정도라고 한다

I/O 바운드 프로세스는 계산에 소비하는 것보다 I/O에 더 많은 시간을 소비하는 프로세스이다

반대로 CPU 바운드 프로세스는 계산에 더 많은 시간을 사용하여 I/O 요청을 자주 생성하지 않는다

**3.2.1 스케줄링 큐**

프로세스가 시스템에 들어가면 ready 큐에 들어가서 준비 상태가 되어 CPU 코어에서 실행되기를 기다린다

I/O 완료와 같은 특정 이벤트가 발생하기를 기다리는 프로세스는 wait 큐에 삽입된다

새 프로세스는 처음에 ready 큐에 놓인다. 프로세스는 실행을 위해 선택되거나 또는 디스패치 될 때까지 기다린다.

프로세스에 CPU 코어가 할당되고 running 상태가 되면, 여러 이벤트 중 하나가 발생할 수 있다

- 프로세스가 I/O 요청을 공표한 다음 I/O wait 큐에 놓일 수 있다

- 프로세스는 새 자식 프로세스를 만든 다음 자식의 종료를 기다리는 동안 wait 큐에 놓일 수 있다

- 인터럽트 또는 타임 슬라이스가 만료되어 프로세스가 코어에서 강제로 제거되어 ready 큐로 돌아갈 수 있다

**3.2.3 문맥 교환; Context Switch**

문맥 교환은 CPU가 하나의 프로세스를 다른 프로세스로 전환할 때 발생한다

CPU가 다른 프로세스로 전환할 때, 시스템은 반드시 이전 프로세스의 상태를 저장하고(state save) 문맥 교환(context switch)를 통해 저장되어 있던 새 프로세스의 상태를 불러와야 한다

CPU 코어를 다른 프로세스로 교환하려면 이전의 프로세스의 상태를 보관하고 새로운 프로세스의 상태를 복구하는 작업이 필요하다

이 작업을 문맥 교환(context switch)이라고 한다

문맥 교환이 일어나면, 커널은 과거 프로세스의 문맥을 PCB에 저장하고, 실행이 스케줄된 새로운 프로세스의 저장된 문맥을 복구한다

문맥 교환이 진행될 동안 시스템이 아무런 유용한 일을 못 하기 때문에 문맥 교환 시간은 순수한 오버헤드이다

문맥 교환 시간은 하드웨어의 지원에 크게 좌우된다

일부 하드웨어는 CPU당 여러 개의 레지스터 집합을 제공한다 -> 여러 문맥을 한 번에 불러옴

운영체제가 복잡할수록, 문맥 교환 시 해야 할 작업의 양이 더 많아진다

**3.3 프로세스에 대한 연산**

대부분 시스템 내의 프로세스들은 병행 실행될 수 있으며, 반드시 동적으로 실행되고, 제거되어야 한다

그러므로 운영체제는 프로세스 생성 및 종료를 위한 기법을 제공해야 한다

**3.3.1 프로세스 생성**

생성하는 프로세스를 부모 프로세스라 부르고, 새로운 프로세스는 자식 프로세스라고 부른다

이 새로운 프로세스들은 각각 다시 다른 프로세스들을 생성할 수 있으며, 그 결과 프로세스의 트리를 형성한다

대부분의 현대 운영체제들은 프로세스 식별자(pid)를 통해 프로세스를 식별 및 관리한다

언제나 pid가 1인 systemd 프로세스가 모든 사용자 프로세스의 루트 부모 프로세스 역할을 수행하고 시스템이 부트될 때 생성되는 첫 번째 사용자 프로세스이다

프로세스가 새로운 프로세스를 생성할 때, 두 프로세스를 실행시키는 데 두 가지 가능한 방법이 존재한다

1. 부모는 자식과 병행하게 실행을 계속한다
2. 부모는 일부 또는 모든 자식이 실행을 종료할 때까지 기다린다

새로운 프로세스들의 주소 공간 측면에서 볼 때 다음과 같은 두 가지 가능성이 있다

1. 자식 프로세스는 부모 프로세스의 복사본이다(자식 프로세스는 부모와 똑 같은 프로그램과 데이터를 가진다)
2. 자식 프로세스가 자신에게 적재될 새로운 프로그램을 가지고 있다

UNIX 예시

1. 새로운 프로세스는 fork() 시스템 콜로 생성된다  
   새로운 프로세스는 원래 프로세스의 주소 공간의 복사본으로 구성된다  
   이 기법은 부모 프로세스가 쉽게 원래 프로세스와 통신할 수 있게 한다
2. Fork() 시스템 콜 다음에 두 프로세스 중 한 프로세스가 exec() 시스템 콜을 사용하여 자신의 메모리 공간을 새로운 프로그램으로 교체한다  
   exec() 시스템 콜은 이진 파일을 메모리로 적재(load)하고 그 프로그램을 실행한다  
   이와 같은 방법으로 두 프로세스는 통신을 할 수 있으며, 그들 각자의 길을 간다
3. 부모는 wait() 시스템 콜로 자식 프로세스가 끝나길 기다린다

**3.3.2 프로세스 종료**

프로세스가 마지막 문장의 실행을 끝내고, exit() 시스템 콜을 사용하여 운영체제에 자신의 삭제를 요청하면 종료한다

1. 프로세스는 부모 프로세스에 상태 값(통상 정수 값)을 반환할 수 있다(wait() 시스템 콜을 통해)
2. 프로세스의 리소스는 할당 해제되고 운영체제로 반납된다

종료되었지만 부모 프로세스가 아직 wait() 호출을 하지 않은 프로세스를 좀비(zombie) 프로세스라고 한다

부모 프로세스가 wait()를 호출하는 대신 종료한다면 자식 프로세스는 고아(orphan) 프로세스가 된다

UNIX는 고아 프로세스의 새로운 부모 프로세스로 init 프로세스를 지정함으로써 이 문제를 해결한다

**3.4 프로세스 간 통신**

운영체제 내에서 실행되는 프로세스들은 독립적이거나 또는 협력적인 프로세스들일 수 있다

프로세스가 시스템에서 실행 중인 다른 프로세스들과 데이터를 공유하지 않는 프로세스는 독립적이다

프로세스가 시스템에서 실행 중인 다른 프로세스들에 영향을 주거나 받는다면 이는 협력적인 프로세스들이다. 데이터 공유를 포함해서

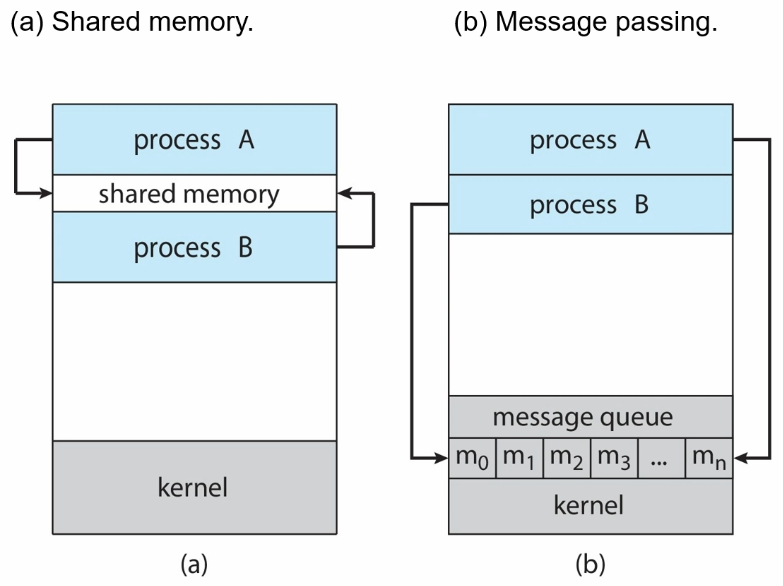
분명히 다른 프로세스들과 자료를 공유하는 프로세스는 상호 협력적인 프로세스이다

협력적 프로세스들은 데이터를 교환할 수 있는, 즉 서로 데이터를 보내거나 받을 수 있는 프로세스 간 통신(IPC) 기법이 필요하다

프로세스 간 통신에는 기본적으로 공유 메모리(shared memory)와 메시지 전달(message passing)의 두 가지 모델이 있다

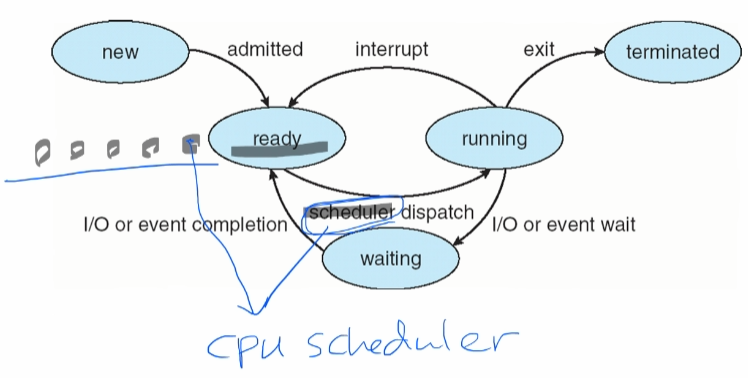
공유 메모리에 모델에서는 협력 프로세스들에 의해 공유되는 메모리의 영역이 구축된다. 프로세스들은 그 영역에 데이터를 읽고 쓰고 함으로써 정보를 교환할 수 있다

메시지 전달 모델에서는 통신이 협력 프로세스들 사이에 교환되는 메시지를 통해서 이루어진다



**챕터5 CPU 스케줄링**

**Diagram of Process State**



**Schedulers**

단기 스케줄러 (CPU 스케줄러) – 다음에 실행할 프로세스를 선택하고 CPU를 할당한다

때때로 한 시스템에서 유일한 스케줄러이다

단기 스케줄러는 자주 호출된다 (밀리초) -> 빨라야 함

장기 스케줄러 (잡 스케줄러) – 레디 큐로 올 프로세스를 선택한다

장기 스케줄러는 자주 호출되지 않는다 (초, 분) -> 느려야 함

장기 스케줄러는 다중 프로그래밍의 정도를 제어한다

프로세스는 다음 중 하나로 설명될 수 있다

I/O-bound process – 계산보다 I/O를 수행하는 데 더 많은 시간을 할애함; 많고 짧은 CPU 버스트

CPU-bound process – 계산하는 데 더 많은 시간을 할애한다; 드물고 매우 긴 CPU 버스트

장기 스케줄러는 우수한 프로세스 혼합을 위해 힘쓴다

**5.1 기본 개념**

다중 프로그래밍의 목적은 CPU 이용률을 최대화하기 위해 항상 실행 중인 프로세스를 가지게 하는 데 있다

어느 한 순간에 다수의 프로세스를 메모리 내에 유지한다

어떤 프로세스가 대기해야 할 경우, 운영체제는 CPU를 그 프로세스로부터 회수해 다른 프로세스에 할당한다

이러한 종류의 스케줄링은 운영체제의 기본적인 기능이다

**5.1.1 CPU-I/O 버스트 사이클**

프로세스 실행은 CPU 실행과 I/O 대기의 사이클로 구성된다

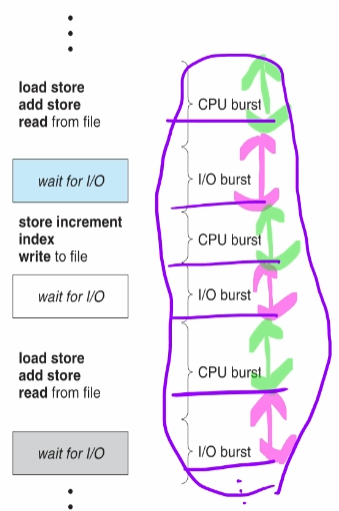
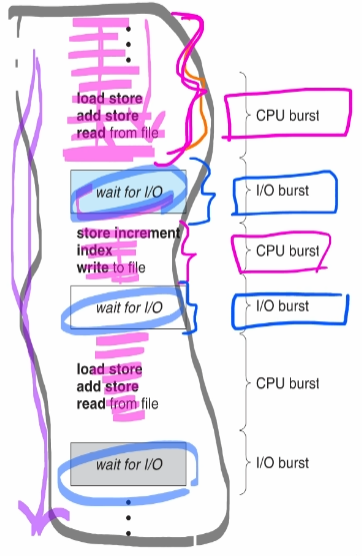
프로세스는 이 두 상태 사이를 교대로 왔다 갔다 한다

프로세스 실행은 CPU 버스트(Burst)로 시작된다. 뒤이어 I/O 버스트가 발생하고, 그 뒤를 이어 또 다른 CPU 버스트가 발생하며, 이어 또 다른 I/O 버스트 등등으로 진행된다. 결국 마지막 CPU 버스트는 또 다른 I/O 버스트가 뒤따르는 대신, 실행을 종료하기 위한 시스템 요청과 함께 끝난다

짧은 CPU 버스트가 많이 있으며, 긴 CPU 버스트는 적다

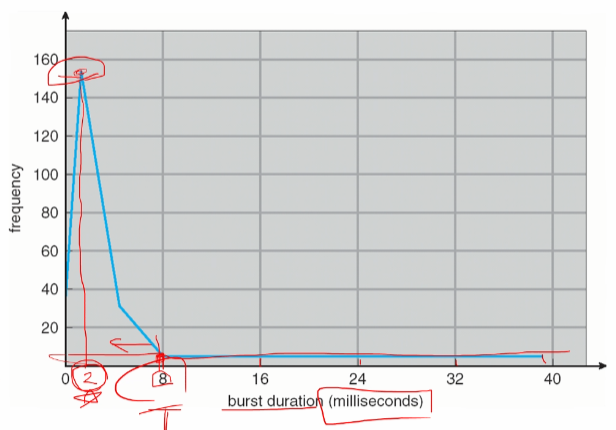
I/O 중심의 프로그램은 전형적으로 짧은 CPU 버스트를 많이 가질 것이다

CPU 중심의 프로그램은 전형적으로 다수의 긴 CPU 버스트를 가질 수 있다

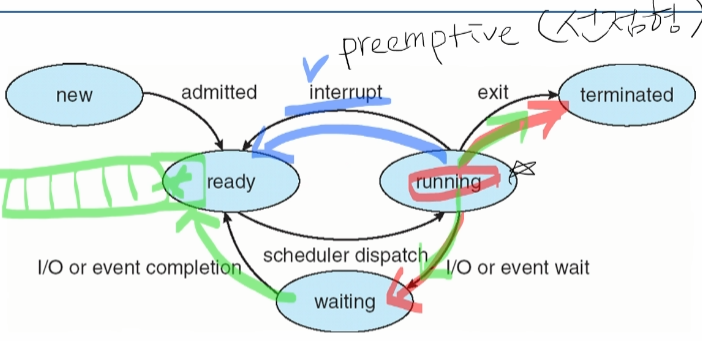


I/O 버스트가 길면 I/O-bound 프로세스, CPU 버스트가 길면 CPU-boung 프로세스

이러한 분포는 CPU 스케줄링 알고리즘을 구현할 때 매우 중요할 수 있다



**5.1.2 CPU 스케줄러**



단기 스케줄러는 레디 큐에 있는 프로세스들 중에 선택하고 그들 중 하나에 CPU를 할당한다

큐는 다양한 방법으로 명령 받을 수 있다 (선입선출 큐, 우선순위 큐, 트리, 단순히 순서가 없는 연결리스트)

레디 큐에 있는 모든 프로세스는 CPU에서 실행될 기회를 기다리며 대기하고 있다

**5.1.3 선점 및 비선점 스케줄링**

CPU 스케줄링 결정은 다음의 네 가지 상황에서 발생할 수 있다

1. 한 프로세스가 러닝 상태에서 대기 상태로 전환될 때 (I/O 요청이나 자식 프로세스가 종료되기를 기다리기 위해 wait()를 호출할 때)
2. 프로세스가 러닝 상태에서 레디 상태로 전환될 때 (예를 들어, 인터럽트가 발생할 때)
3. 프로세스가 대기 상태에서 레디 상태로 전활될 때 (예를 들어, I/O의 종료 시)
4. 프로세스가 종료할 때

상황 1과 4의 경우 스케줄링 면에서는 선택의 여지가 없다

그러나 상황 2와 3을 위해서는 선택의 여지가 있다

상황 1과 4에서만 스케줄링이 발생할 경우, 우리는 이러한 스케줄링 방법을 비선점(nonpreemptive) 또는 협조적이라고 한다

그렇지 않으면, 그것은 선점(preemptive)이라고 한다

거의 모든 최신 운영체제들은 선점 스케줄링 알고리즘을 사용한다

선점은 공유 데이터에 대한 접근을 고려해야 하고

커널 모드에서 선점을 고려해야 하고

중요한 OS 활동 중에 발생하는 인터럽트를 고려해야 한다

**5.2 스케줄링 기준**

특정 상황에서 어떠한 알고리즘을 선택하려면, 우리는 다양한 알고리즘들의 서로 다른 특성을 반드시 고려해야 한다

CPU 스케줄링 알고리즘을 비교하기 위한 여러 기준이 제시되었다

비교하는 데 사용되는 특성에 따라서 최선의 알고리즘을 결정하는 데 큰 차이가 발생한다

사용되는 기준은 다음을 포함한다

- CPU 이용률(utilization): 우리는 가능한 한 CPU를 최대한 바쁘게 유지하기를 원한다

개념상으로는 CPU 이용률은 0%에서 100%까지 이르지만 실제 시스템에서는 40%에서 90%까지의 범위를 가져야 한다

- 처리량(throughput): 작업량 측정의 한 방법은 단위 시간당 완료된 프로세스의 개수로, 이것을 처리량이라고 한다

긴 프로세스의 경우에는 이 비율은 몇 초 동안 한 프로세스가 될 수 있고, 짧은 트랜잭션인 경우 처리량은 초당 수십 개의 프로세스가 될 수도 있다

- 총처리 시간(turnaround time): 특정한 프로세스의 입장에서 보면, 중요한 기준은 그 프로세스를 실행하는 데 소요된 시간일 것이다

프로세스의 제출 시간과 완료 시간의 간격을 총처리 시간이라고 한다

총처리 시간은 레디 큐에서 대기한 시간, CPU에서 실행하는 시간, 그리고 I/O 시간을 합한 시간이다

- 대기 시간(waiting time): CPU 스케줄링 알고리즘은 프로세스가 실행하거나 I/O을 하는 시간의 양에 영향을 미치지는 않는다

스케줄링 알고리즘은 단지 프로세스가 레디 큐에서 대기하는 시간의 양에만 영향을 준다

대기 시간은 레디 큐에서 대기하면서 보낸 시간의 합이다

- 응답 시간(response time): 하나의 요구를 제출한 후 첫 번째 응답이 나올 때까지의 시간, 그 응답을 출력하는 데 걸리는 시간은 아니다

CPU 이용률과 처리량을 최대화하고 총처리 시간, 대기 시간, 응답 시간을 최소화하는 것이 바람직하다

**5.3 스케줄링 알고리즘**

CPU 스케줄링은 레디 큐에 있는 어느 프로세스에 CPU 코어를 할당할 것인지를 결정하는 문제를 다룬다

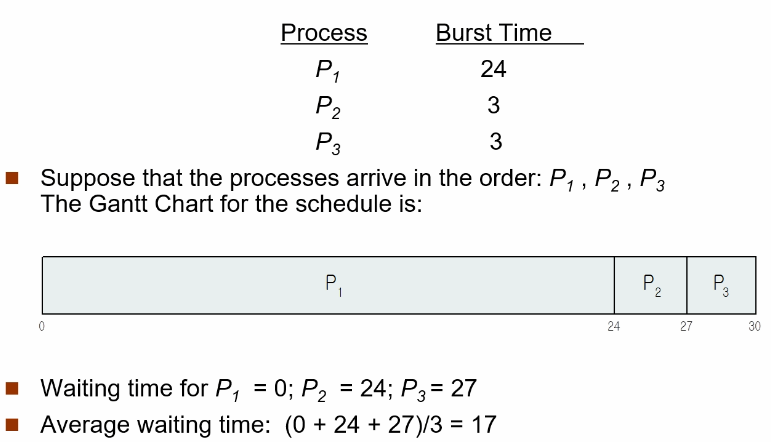
**5.3.1 선입 선처리 스케줄링 (FCFS)**

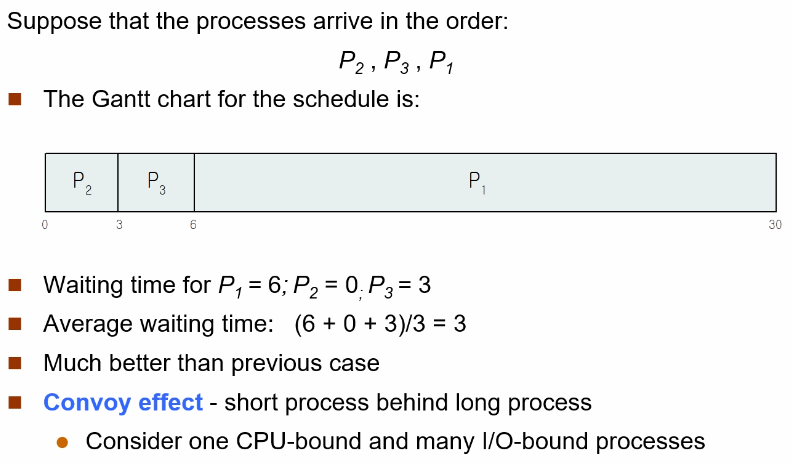
프로세스가 레디 큐에 진입하면, 이 프로세스의 프로세스 제어 블록(PCB)을 큐의 끝에 연결한다

CPU가 가용 상태가 되면, 줍니 큐의 앞부분에 있는 프로세스에 할당된다

선입 선처리를 위한 코드는 작성하기 쉽고 이해하기 쉽다

부정적인 측면으로 선입 선처리 정책하에서 평균대기 시간은 종종 대단히 길 수 있다





선입 선처리 스케줄링 알고리즘은 비선점형

일단 CPU가 한 프로세스에 할당되면, 그 프로세스가 종료하든지 또는 I/O 처리를 요구하든지 하여 CPU를 방출할 때까지 CPU를 점유한다

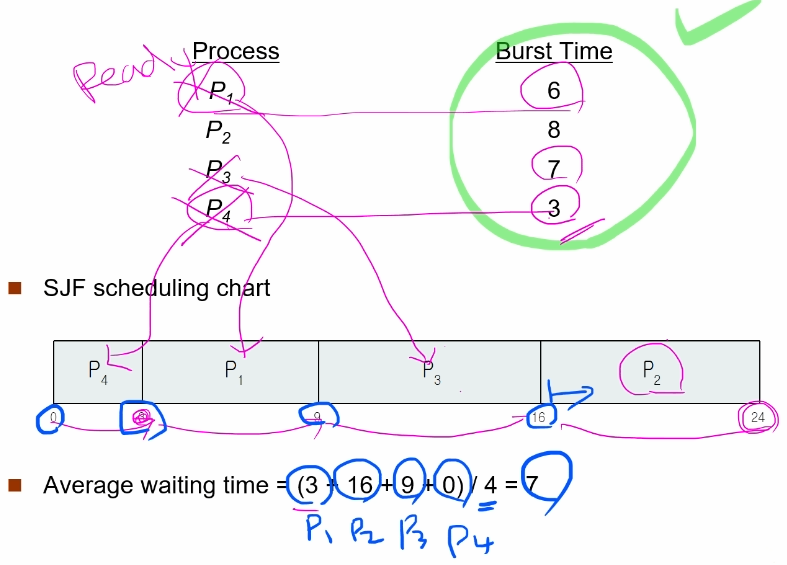
선입 선처리 알고리즘은 특히 대화형 시스템에서 문제가 되는데, 대화영 시스템에서는 각 프로세스가 규칙적인 간격으로 CPU의 몫을 얻는 것이 매우 중요하기 때문이다

**5.3.2 최단 작업 우선 스케줄링 (SJF)**

이 알고리즘은 각 프로세스에 다음 CPU 버스트 길이를 연관시킨다

CPU가 이용 가능해지면, 가장 작은 다음 CPU 버스트를 가진 프로세스에 할당한다

두 프로세스가 동일한 길이의 다음 CPU 버스트를 가지면, 순위를 정하기 위해 선입 선처리 스케줄링을 적용한다



SJF 알고리즘은 주어진 프로세스 집합에 대해 최소의 평균대기 시간을 가진다는 점에서 최적임을 증명할 수 있다

짧은 프로세스를 긴 프로세스의 앞으로 이동함으로써, 짧은 프로세스의 대기 시간을 긴 프로세스의 대기 시간이 증가하는 것보다 더 많이 줄일 수 있다

SJF 알고리즘이 최적이긴 하지만, 다음 CPU 버스트의 길이를 알 방법이 없기 때문에 CPU 스케줄링 수준에서는 구현할 수 없다

한 가지 접근 방식은 SJF 스케줄링과 근사한 방법을 사용하는 것이다

다음 CPU 버스트의 길이를 알 수는 없지만, 그 값을 예측할 수는 있을 것이다

우리는 다음 CPU 버스트가 이전의 버스트와 길이가 비슷하다고 기대한다

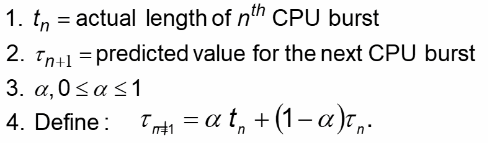
그러므로 다음 CPU 버스트 길이의 근삿값을 계산해, 가장 짧은 예상 CPU 버스트를 가진 프로세스를 선택한다

다음 CPU 버스트의 길이 결정

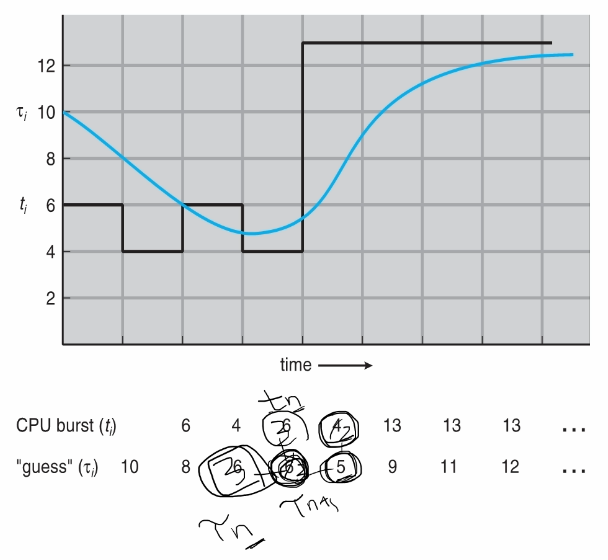
길이만 추정 가능 – 이전의 길이와 유사해야 함

그리고 다음 CPU 버스트 예측이 가장 짧은 프로세스를 선택한다

다음 CPU 버스트는 일반적으로 측정된 이전의 CPU 버스트들의 길이를 지수 평균한 것으로 예측한다



일반적으로 a는 1/2

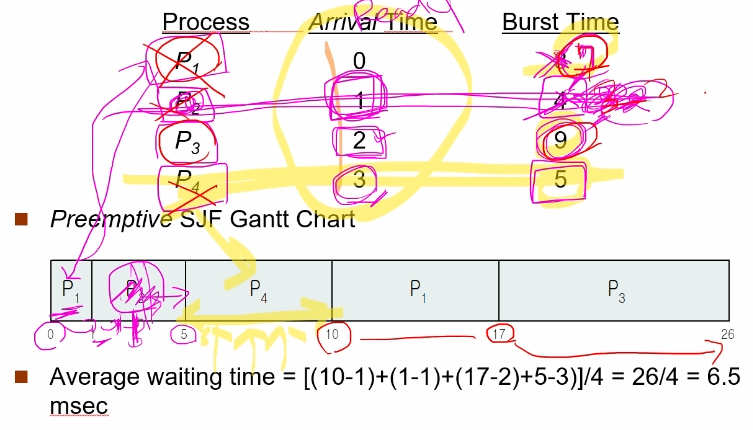


SJF 알고리즘은 선점형이거나 비선점형일 수 있다

앞의 프로세스가 실행되는 동안 새로운 프로세스가 준비 큐에 도착하면 선택이 발생한다

선점형 SJF 알고리즘은 현재 실행하는 프로세스를 선점할 것이고, 반면에 비선점형 SJF 알고리즘은 현재 실행하고 있는 프로세스가 자신의 CPU 버스트가 끝나도록 허용한다

선점형 SJF 알고리즘은 때때로 최소 잔여 시간 우선 스케줄링이라고 불린다



**5.3.3 라운드 로빈 스케줄링 (RR)**

라운드 로빈 스케줄링 알고리즘은 선입 선처리 스케줄링과 유사하지만 시스템이 프로세스들 사이를 옮겨 다닐 수 있도록 선점이 추가된다

시간 할당량 (time quantum) 또는 타임슬라이스 (time slice)라고 하는 작은 단위의 시간을 정의한다

시간 할당량은 일반적으로 10에서 100밀리초 동안이다

CPU 스케줄러는 레디 큐를 돌면서 한 번에 한 프로세스에 한 번의 시간 할당량 동안 CPU를 할당한다

CPU 버스트가 한 번의 시간 할당량을 초과하면, 프로세스는 선점되고 레디 큐로 되돌아간다

따라서 RR 스케줄링 알고리즘은 선점형이다

레디 큐에 n개의 프로세스가 있고 시간 할당량이 q이면, 각 프로세스는 최대 q 시간 단위의 덩러리로 CPU 시간 1/n을 얻는다

각 프로세스는 자신의 다음 시간 할당량이 할당될 때까지 (n-1)\*q 시간 이상을 기다리지는 않는다

RR 알고리즘의 성능은 시간 할당량의 크기에 매우 많은 영향을 받는다

극단적인 경우, 시간 할당량이 매우 크면, RR 정책은 선입 선처리 정책과 같다

반대로 시간 할당량이 매우 적다면 (예를 들면, 1마이크로초) RR 정책은 매우 많은 문맥 교환을 야기한다

우리는 시간 할당량이 문맥 교환 시간과 비교해 더 클 것을 원한다

실제로 대부분의 현대 운영체제들은 10에서 100밀리초 범위의 시간 할당량을 가지고 있다

문맥 교환을 하는 데 걸리는 시간은 보통 10마이크로초 미만이다 따라서 문맥 교환 시간은 시간 할당량의 작은 부분을 차지한다

총처리 시간 또한 시간 할당량의 크기에 좌우된다

한 프로세스 집합의 평균 총처리 시간은 시간 할당량의 크기가 증가하더라도 반드시 개선되지는 않는다

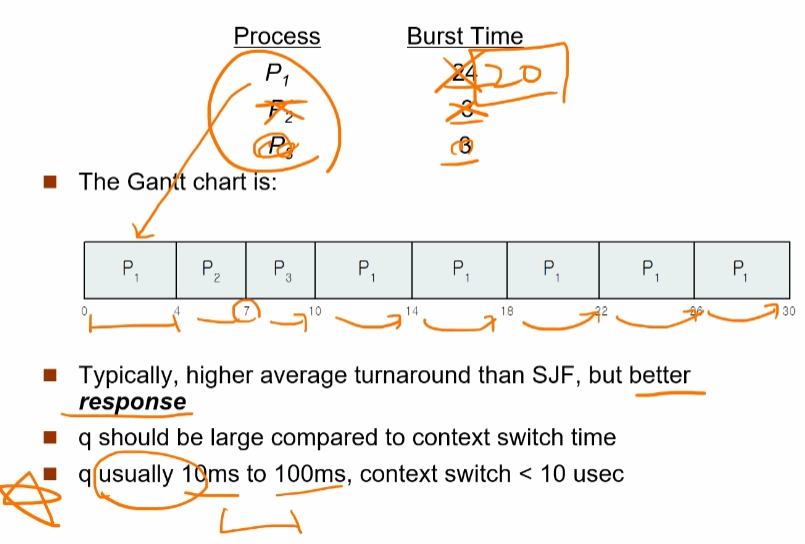
일반적으로, 대부분의 프로세스가 단일 시간 할당량 안에 다음 CPU 버스트를 끝낸다면, 평균 총처리 시간은 개선된다

시간 할당량이 문맥 교환 시간에 비해 커야 하지만 너무 커서는 안 된다

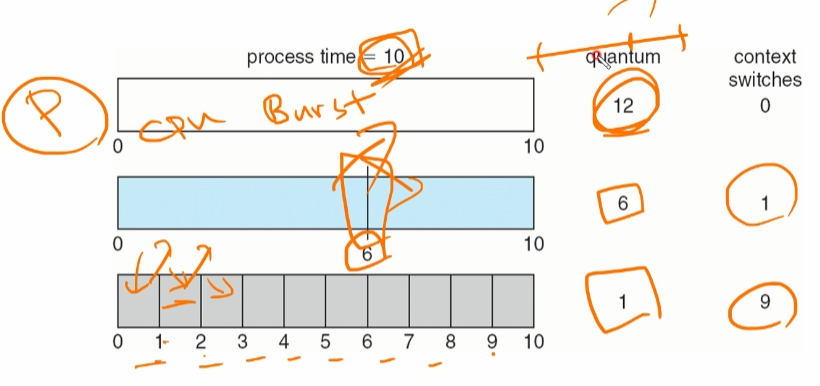
앞서 지적한 것처럼 시간 할당량이 너무 크다면 RR 스케줄링은 선입 선처리 정책으로 퇴보한다

경험으로 CPU 버스트의 80%는 시간 할당량보다 짧아야 한다

시간 할당량 4인 RR의 예



시간 할당량과 문맥 교환



**5.3.4 우선순위 스케줄링**

SJF 알고리즘은 일반적인 우선순위 스케줄링 알고리즘의 특별한 경우이다

우선순위가 각 프로세스들에 연관되어 있으며, CPU는 가장 높은 우선순위를 가진 프로세스에 할당된다

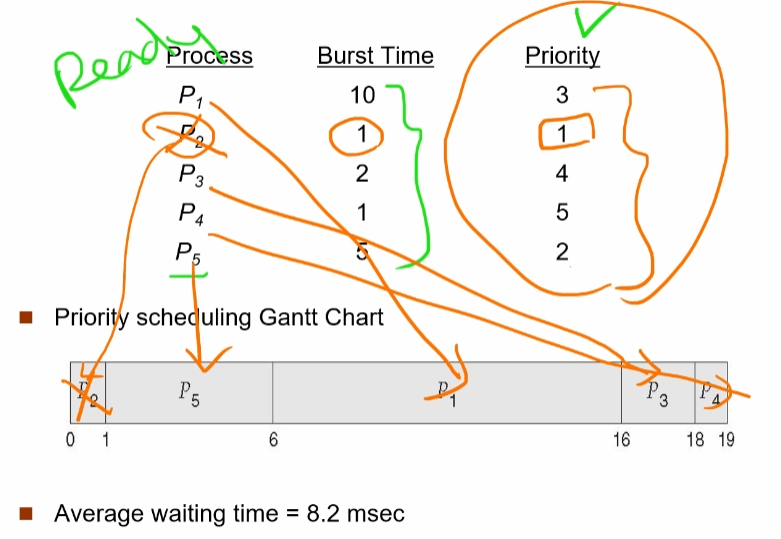
SJF 알고리즘은 우선순위(p)가(예상되는) 다음 CPU 버스트의 역인 단순한 우선순위 알고리즘이다

CPU 버스트가 클수록 우선순위가 낮으며, 그 역도 성립한다

우선순위가 같은 프로세스들은 선입 선처리 (FCFS) 순서로 스케줄 된다

우선순위는 일반적으로 0에서 7 또는 0에서 4,095까지 같은 일정 범위의 수가 사용된다

그러나 0이 최상위 또는 최하위 우선순위인가에 대해서 일반적인 합의는 없다



우선순위는 내부적 또는 외부적으로 정의될 수 있다

내부적으로 정의된 우선순위는 프로세스의 우선순위를 계산하기 위해 어떤 측정 가능한 양들을 사용한다 (시간 제한, 메모리 요구, 열린 파일의 수, 평균 I/O 버스트의 평균 CPU 버스트에 대한 비율 등)

외부적 우선순위는 프로세스의 중요성, 컴퓨터 사용을 위해 지불되는 비용의 유형과 양, 그 작업을 후원하는 부서 그리고 정치적인 요인 등과 같은 운영체제 외부적 기준에 의해 결정된다

우선순위 스케줄링은 선점형이거나 비선점형이 될 수 있다

프로세스가 레디 큐에 도착하면, 새로 도착한 프로세스의 우선순위를 현재 실행 중인 프로세스의 우선순위와 비교한다

선점형 우선순위 스케줄링 알고리즘은 새로 도착한 프로세스의 우선순위가 현재 실행되는 프로세스의 우선순위보다 높다면 CPU를 선점한다

비선점형 우선운위 스케줄링 알고리즘은 단순히 레디완료 큐의 머리 부분에 새로운 프로세스를 넣는다

우선순위 스케줄링 알고리즘의 주요 문제는 기아 상태(Starvation)이다

우선순위 스케줄링 알고리즘을 사용할 경우 낮은 우선순위 프로세스들이 CPU를 무한히 대기하는 경우가 발생한다

부하가 과중한 컴퓨터 시스템에서는 높은 우선순위의 프로세스들이 꾸준히 들어와서 낮은 우선순위의 프로세스들이 CPU를 얻지 못하게 될 수도 있다

낮은 우선순위의 프로세스들이 무한히 봉쇄되는 문제에 대한 한 가지 해결 방안은 노화(aging)이다

노화는 오랫동안 시스템에서 대기하는 프로세스들의 우선순위를 점진적으로 증가시킨다

**챕터6 동기화**

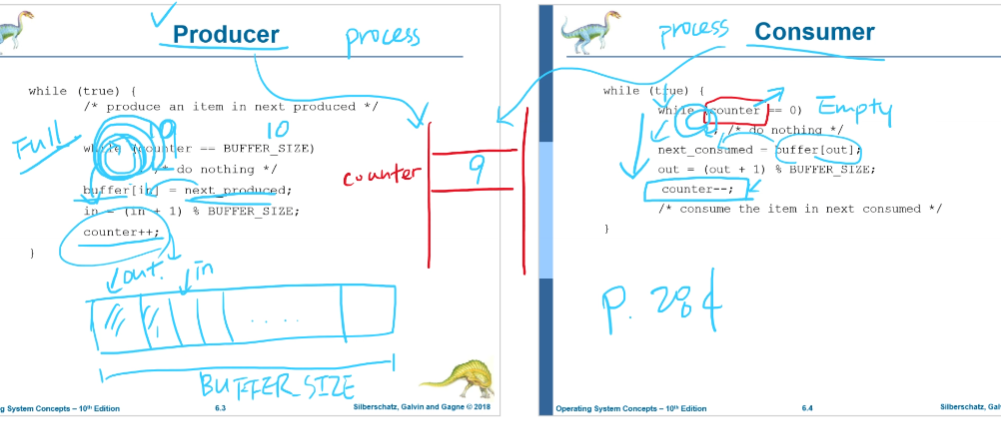
**6.1 배경**

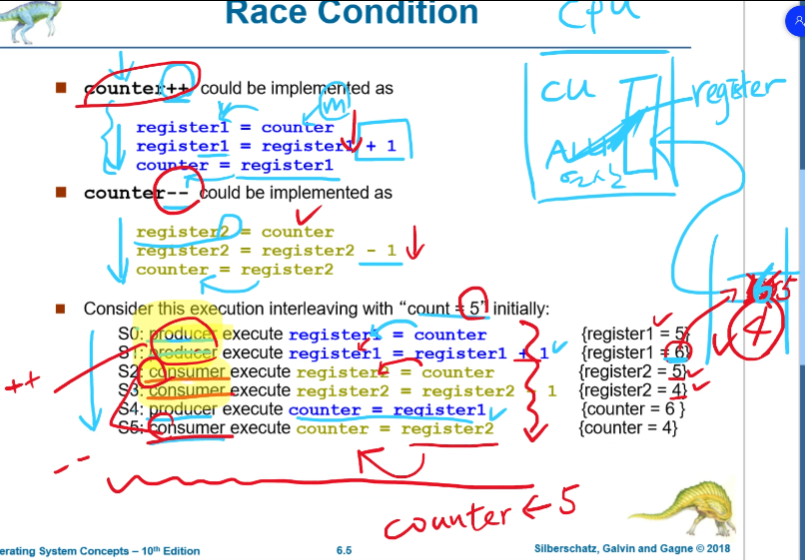
프로세스는 병행하게 또는 병렬로 실행될 수 있다

프로세스는 언제든지 중단될 수 있으며, 일부분만 진행될 수 있다

공유 데이터를 동시에 접근하면 데이터의 일관성을 망칠 수 있다

데이터의 일관성을 유지하기 위해서는 협럭 프로세스의 질서 있는 실행을 보장하는 메커니즘이 필요하다





이러한 부정확한 상태에 도달하는 것은 두 개의 프로세스가 동시에 변수 count를 조작하도록 허용했기 때문이다

이처럼 동시에 여러 개의 프로세스가 동일한 자료를 접근하여 조작하고, 그 실행 결과가 접근이 발생한 특정 순서에 의존하는 상황을

Race Condition이라고 한다

위와 같은 Race Condition으로부터 보호하기 위해, 우리는 한순간에 하나의 프로세스만이 변수 count를 조작하도록 보장해야 한다

이러한 보장을 위해, 우리는 어떤 형태로든 프로세스들이 동기화되도록 할 필요가 있다

운영체제의 역할 중 하나가 Race Condition을 방지하는 것이다

**6.2 critical section 문제**

각 프로세스는 critical section이라고 부르는 코드 부분을 포함하고 있고, 그 안에서는 적어도 하나 이상의 다른 프로세스와 공유하는 데이터에 접근하고 갱신할 수 있다 (공통변수 변경, 표 갱신, 파일 작성 등)

이 시스템의 중요한 특징은 한 프로세스가 자신의 critical section에서 수행하는 동안에는 다른 프로세스들은 그들의 critical section에 들어갈 수 없다는 사실이다

즉 동시에 두 프로세스는 그들의 critical section 안에서 실행할 수 없다

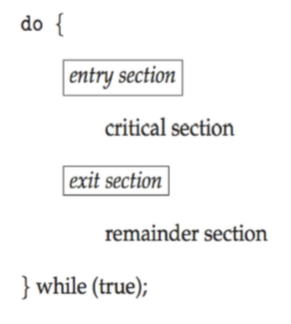
critical section 문제는 프로세스들이 데이터를 협력적으로 공유하기 위하여 자신들의 활동을 동기화할 때 사용할 수 있는 프로토콜을 설계하는 것이다

각 프로세스는 자신의 critical section으로 진입하려면 진입 허가를 요청해야 한다

이러한 요청을 구현하는 코드 부분을 entry section이라고 부른다

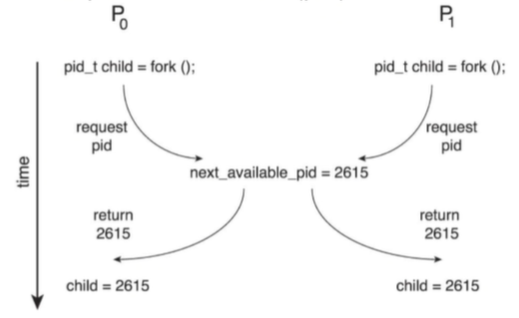
critical section 뒤에는 exit section이 따라올 수 있다

코드의 나머지 부분들은 총칭하여 remainder section이라고 부른다



critical section 문제에 대한 해결한은 다음의 세 가지 요구 조건을 충족해야 한다

1. 상호 배제(mutual excliusion): 프로세스 Pi가 자신의 critical section에서 실행된다면, 다른 프로세스들은 그들 자신의 critical section에서 실행될 수 없다
2. 진행(progress): 자기의 critical section에서 실행되는 프로세스가 없고 그들 자신의 ciritical section으로 진입하려는 프로세스들이 있다면, remainder section에서 실행중이지 않은 프로세스들만 다음에 누가 그 critical section으로 진입할 수 있는지를 결정하는 데 참여할 수 있으며, 이 선택은 무한정 연기될 수 없다
3. 한정된 대기(bounded waiting): 프로세스가 자신의 critical section에 진입하려는 요청을 한 후부터 그 요청이 허용될 때까지 다른 프로세스들이 그들 자신의 critical section에 진입하도록 허용되는 횟수에 한계가 있어야 한다
   1. 각 프로세스가 0이 아닌 속도로 실행되는 것을 가정한다
   2. 그러나 n개의 프로세스 간의 상대적인 속도에 대한 가정은 하지 않는다



P0 및 P1의 두 프로세스는 fork() 시스템 콜을 사용하여 자식 프로세스를 생성한다

이 예에서, 커널 변수 next\_available\_pid에 race condition이 있으며, 이 변수는 다음 사용 가능한 프로세스 식별자의 값을 나타낸다

상호 배제가 제공되지 않으면 동일한 프로세스 식별자 번호가 두 개의 다른 프로세스에 배정될 수 있다

race condition이 발생하기 쉬운 다른 커널 자료구조로는 메모리 할당을 관리하는 자료구조, 프로세스 리스트를 유지하는 자료구조, 인터럽트 처리를 위한 자료구조 등이 있다

운영체제에서 이러한 race conditon이 발생하지 않도록 보장하는 것은 커널 개발자의 책임이다

**6.3 Peterson의 해결안**

현대 컴퓨터 구조가 load와 store 같은 기본적인 기계어를 수행하는 방식 때문에 Peterson의 해결안이 이러한 구조에서 올바르게 실행된다고 보장할 수는 없다

Peterson의 해결안은 critical section과 나머지 구역을 번갈아 가며 실행하는 두 개의 프로세스로 한정된다

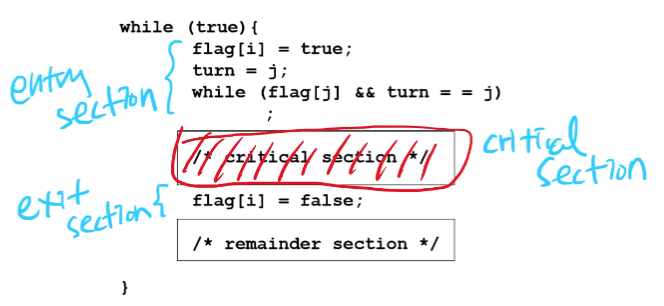
두 프로세스는 두 개의 데이터 항목을 공유한다

int turn;

boolean flag[2];

turn은 critical section으로 진입할 순번을 나타낸다

flag 배열은 프로세스가 critical section으로 진입할 준비가 되었다는 것을 나타낸다



flag[i] = true; // 들어갈 준비가 됐다

turn = j; // 만약 j가 기다리고 있다면 실행해