**프로세스의 메모리 배치**

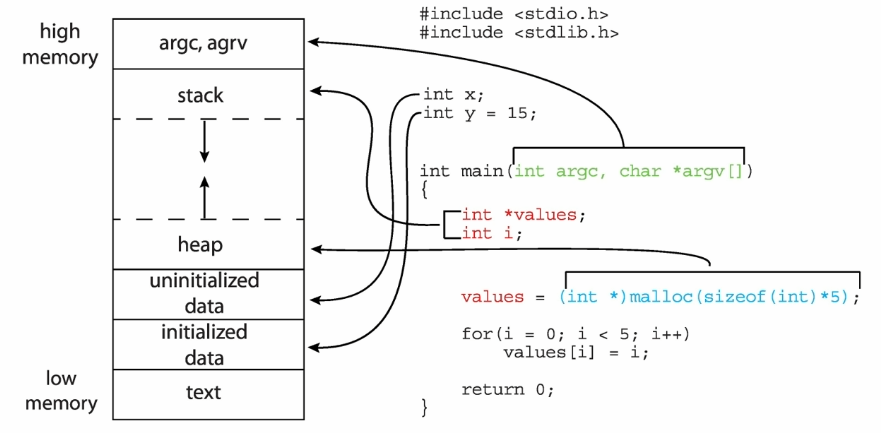
텍스트 섹션 – 실행 코드

데이터 섹션 – 전역 변수

힙 섹션 – 프로그램 실행 중에 동적으로 할당되는 메모리

스택 섹션 – 함수를 호출할 때 임시 데이터 저장장소

**C 프로그램의 메모리 배치**



**프로세스 상태**

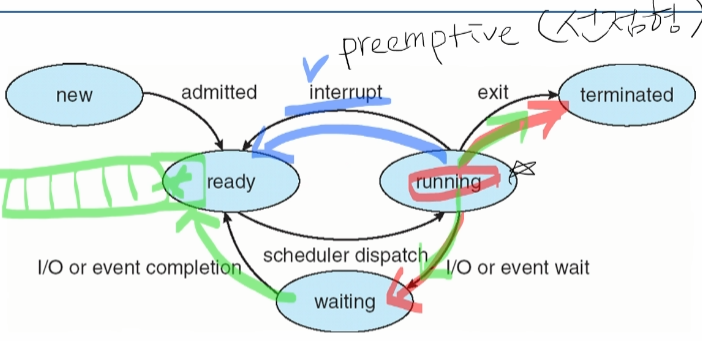
New: 프로세스가 생성 중이다

Ready: 프로세스가 CPU에 할당되기를 기다린다

Running: 명령어들이 실행되고 있다

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

Terminated: 프로세스의 실행이 종료되었다



**프로세스 제어 블록**: 특정 프로세스와 관련된 여러 정보들을 수록함

프로세스 상태

프로그램 카운터

CPU 레지스터들

CPU 스케줄링 정보

메모리 관리 정보

Accounting 정보

I/O status 정보

프로세스를 시작하거나 다시 시작시키는 데 필요한 모든 데이터를 위한 역할을 함

**다중 프로그래밍**: 현재 메모리에 있는 프로세스 수

**다중 프로그래밍의 목적**: CPU 이용을 최대화하기 위하여 항상 어떤 프로세스가 실행되도록 하는 데 있다

**I/O 바운드 프로세스**: 계산에 소비하는 것보다 I/O에 더 많은 시간을 소비하는 프로세스

**CPU 바운드 프로세스**: 계산에 더 많은 시간을 사용하여 I/O 요청을 자주 생성하지 않는다

**스케줄링 큐**

프로세스가 생성되면 ready 큐에 들어가서 준비 상태가 되어 CPU 코어에서 실행되기를 기다린다 프로세스에 CPU 코어에 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

프로세스가 종료될 때까지 이 주기를 계속한다

**문맥 교환**: CPU가 다른 프로세스로 전환할 때, 이전 프로세스의 상태를 저장하고 저장되어 있던 새 프로세스의 상태를 복원하는 작업

순수한 오버헤드이며, 문맥 교환 시간은 하드웨어의 지원에 크게 좌우된다

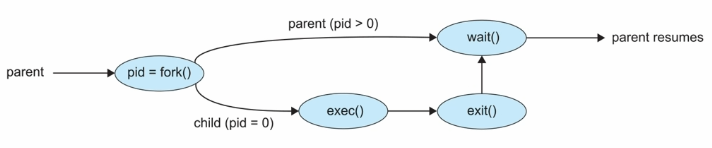
**프로세스 생성**

생성하는 프로세스를 부모 프로세스라 부르고, 새로운 프로세스는 자식 프로세스라고 부른다

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

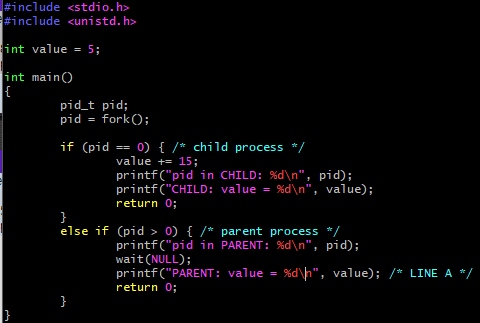
자식 프로세스에 보이는 pid 값은 0이고 부모 프로세스에게 보이는 pid 값은 0보다 큰 정수 값(사실 이 값은 자식 프로세스의 pid)이다

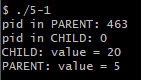


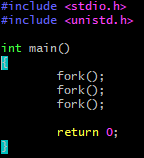
새로운 프로세스는 fork() 시스템 콜로 생성

fork() 시스템 콜 다음에 두 프로세스 중 한 프로세스가 exec() 시스템 콜을 사용하여 자신의 메모리 공간을 새로운 프로그램으로 교체

부모는 wait() 시스템 콜로 자식 프로세스가 끝나길 기다림

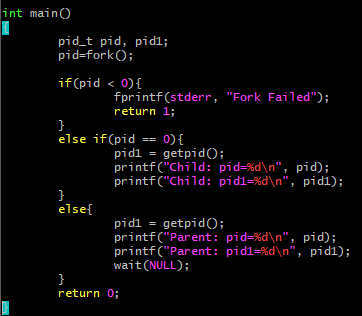




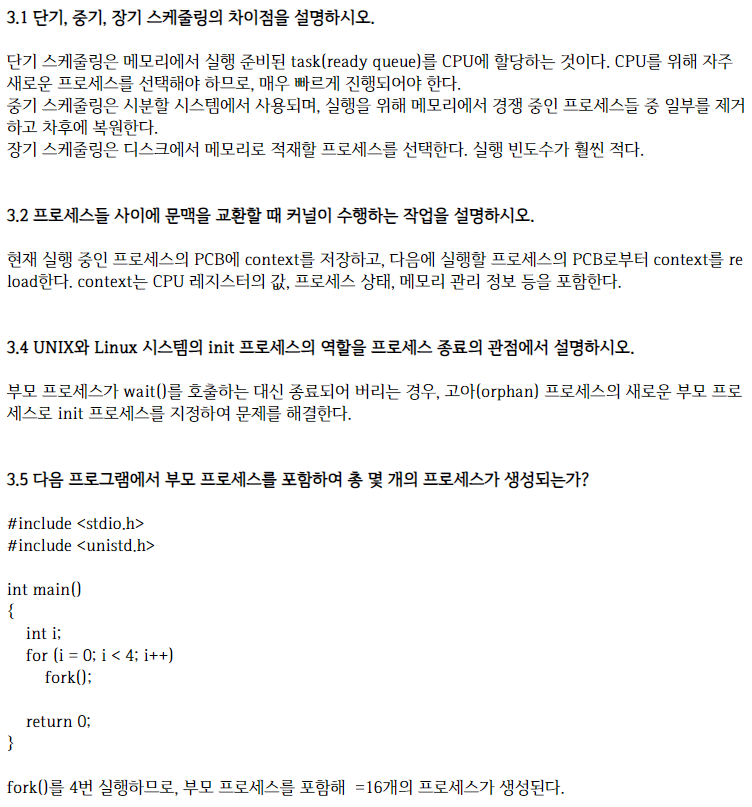


fork() 3번: 자식 프로세스 7개, 프로세스는 8번 실행

fork() 4번: 자식 프로세스 15개, 프로세스는 16번 실행







**프로세스 종료**

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

**좀비(zombie) 프로세스**: 종료되었지만 부모 프로세스가 아직 wait() 호출을 하지 않은 프로세스

종료하게 되면 모든 프로세스는 좀비 상태가 되지만 아주 짧은 시간 동안만 머무름

**고아(orphan) 프로세스**: 부모 프로세스가 wait()를 호출하는 대신 종료했을 때의 자식 프로세스

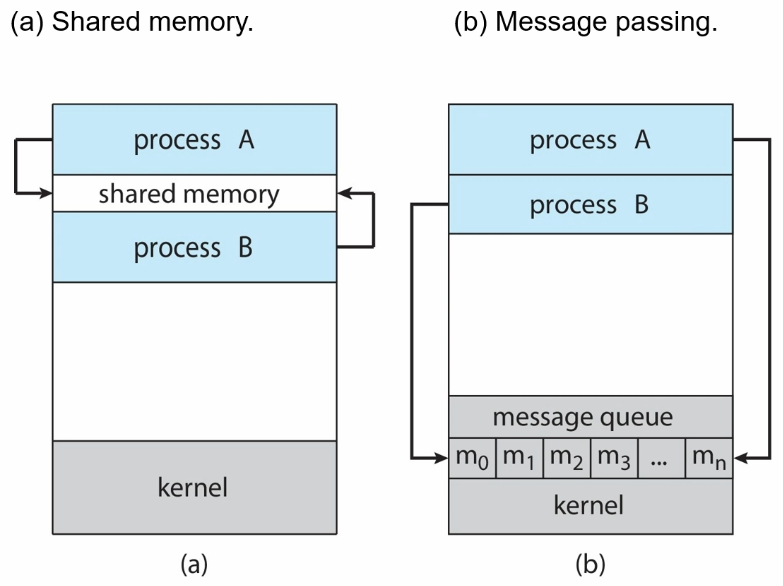
새로운 부모 프로세스로 init 프로세스를 지정함으로써 이 문제를 해결

**프로세스 간 통신(IPC)**

프로세스들은 독립적이거나 협력적인 프로세스들일 수 있다

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

프로세스 간 통신에는 공유 메모리와 메시지 전달의 두 가지 모델이 있다



**단기 스케줄러 (CPU 스케줄러)**: 다음에 실행할 프로세스를 선택하고 CPU를 할당한다

한 시스템에서 유일한 스케줄러일 수 있고 자주 호출된다(밀리초) -> 빨라야 함

**장기 스케줄러 (잡 스케줄러)**: 레디 큐로 올 프로세스를 선택한다

자조 호출되지 않는다(초, 분) -> 느려야함

장기 스케줄러는 다중 프로그래밍의 정도를 제어한다

장기 스케줄러는 우수한 프로세스 혼합을 위해 힘쓴다

**I/O 바운드 프로세스**: 계산보다 I/O를 수행하는 데 더 많은 시간을 할애함; 많고 짧은 CPU 버스트

**CPU 바운드 프로세스**: 계산하는 데 더 많은 시간을 할애한다; 드물고 매우 긴 CPU 버스트

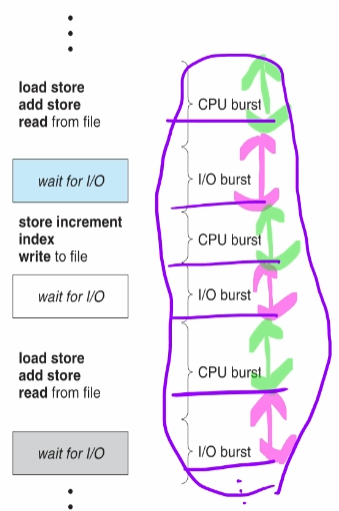
**CPU 스케줄러**

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

**CPU-I/O 버스트 사이클**



프로세스 실행은 CPU 버스트로 시작된다 뒤이어 I/O 버스트 CPU 버스트 I/O 버스트 등등으로 진행하고 결국 마지막 CPU 버스트는 또 다른 I/O 버스트 대신 실행을 종료하기 위한 시스템 요청과 함께 끝난다

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

CPU 중심의 프로그램은 전형적으로 긴 CPU 버스트를 가질 수 있다

I/O 버스트가 길면 I/O 바운드 프로세스, CPU 버스트가 길면 CPU 바운드 프로세스

**선점 및 비선점 스케줄링**

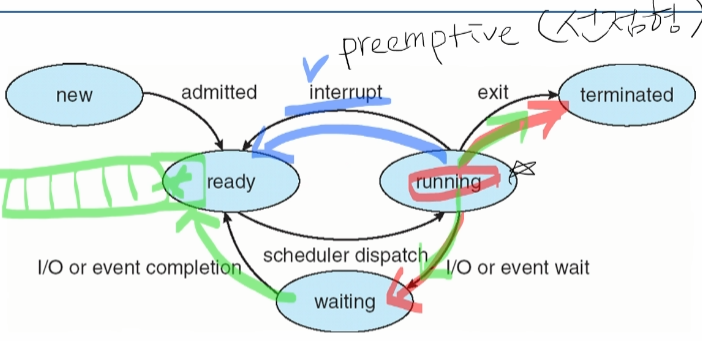
CPU 스케줄링 결정은 다음의 네 가지 상황에서 발생할 수 있다

1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때 (I/O 요청이나 자식 프로세스가 종료되기를 기다리기 위해 wait()를 호출할 때)

2. 프로세스가 실행 상태에서 준비 상태로 전환될 때 (예를 들어, 인터럽트가 발생할 때)

3. 프로세스가 대기 상태에서 준비 상태로 전환될 때 (예를 들어, I/O의 종료 시)

4. 프로세스가 종료할 때



상황 1과 4의 경우 스케줄링 면에서는 선택의 여지가 없어 **비선점 스케줄링**이다

상황 2와 3의 경우 선택의 여지가 있어 **선점 스케줄링**이다

**스케줄링 기준**: 특정 상황에서 어떠한 알고리즘을 선택할 때 다양한 알고리즘의 서로 다른 특성을 고려해야 한다

CPU 이용률: 개념상 0%에서 100%까지 이르지만 실제 시스템에서는 40%에서 90%까지의 범위를 가져야 한다

처리량: 단위 시간당 완료된 프로세스의 개수

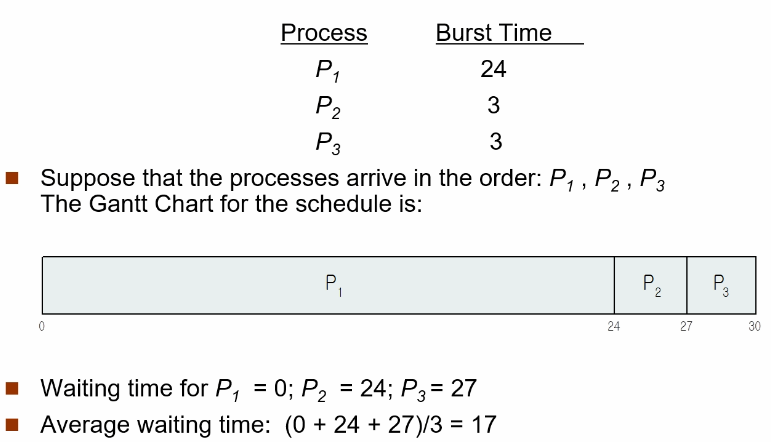
총처리 시간: 프로세스의 제출 시간과 완료 시간의 간격

대기 시간: 프로세스가 레디 큐에서 대기하는 시간

응답 시간: 하나의 요구를 제출한 후 첫 번째 응답이 나올 때까지의 시간

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

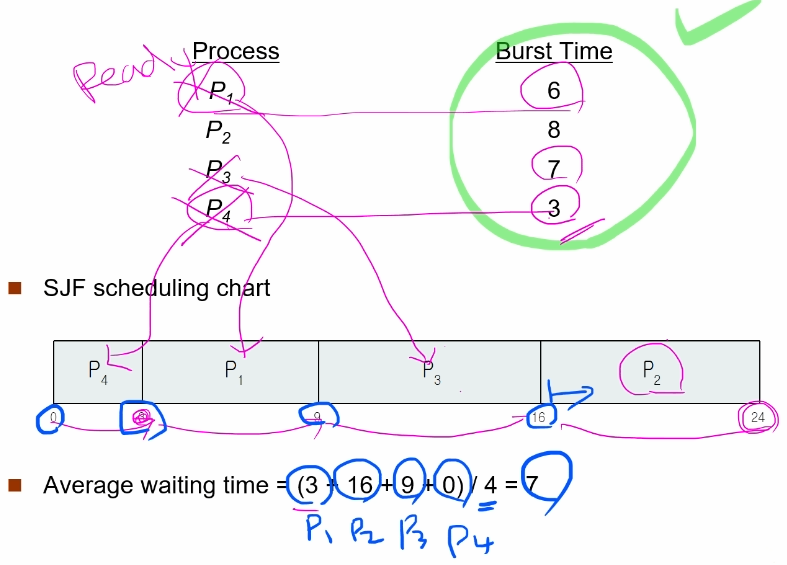
**선입 선처리 스케줄링 (FCFS)**



간단한 스케줄링 알고리즘이지만 평균대기 시간이 매우 길 수가 있다

선입 선처리 스케줄링 알고리즘은 비선점형이다

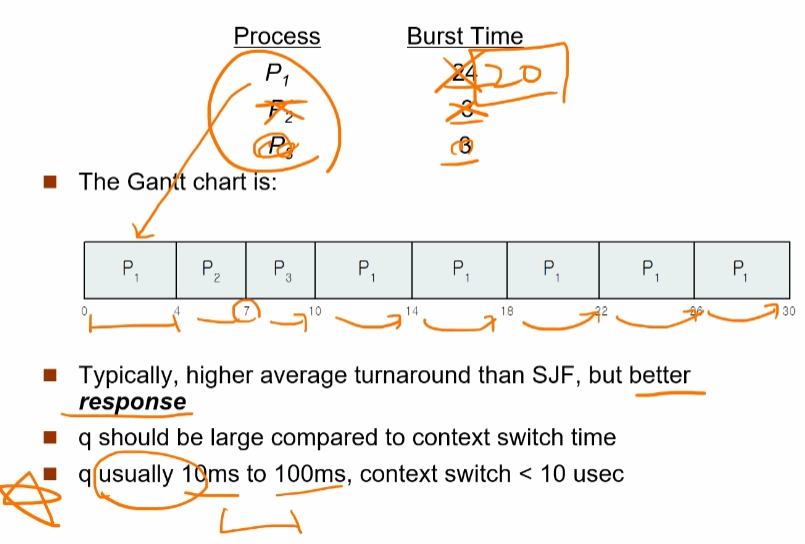
**최단 작업 우선 스케줄링 (SJF)**



평균대기 시간이 가장 짧지만 다음 CPU 버스트의 길이를 예측하기 어려우므로 구현하는 것은 어렵다

SJF 알고리즘은 선점형이거나 비선점형일 수 있다

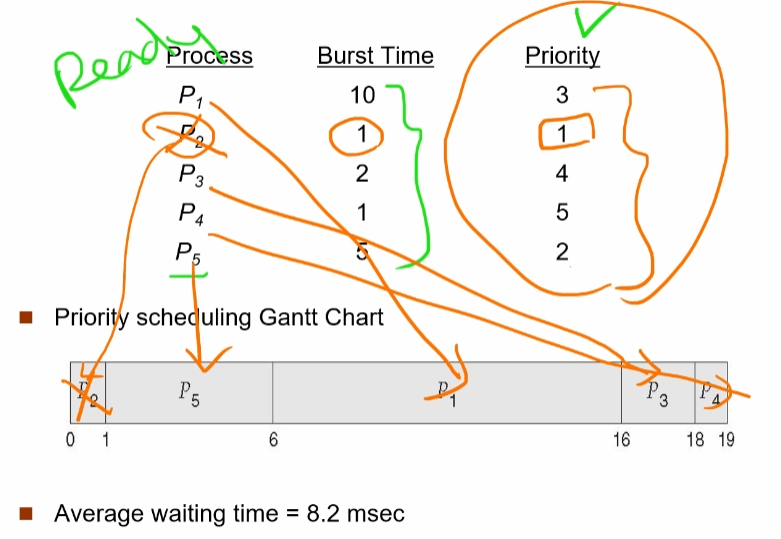
**라운드 로빈 스케줄링 (RR)**



CPU를 시간 할당량 동안 프로세스에 할당한다

RR 스케줄링은 선점형이다

**우선순위 스케줄링**



각 프로세스에 우선순위를 배정하고 CPU는 우선순위가 높은 프로세스에 할당된다

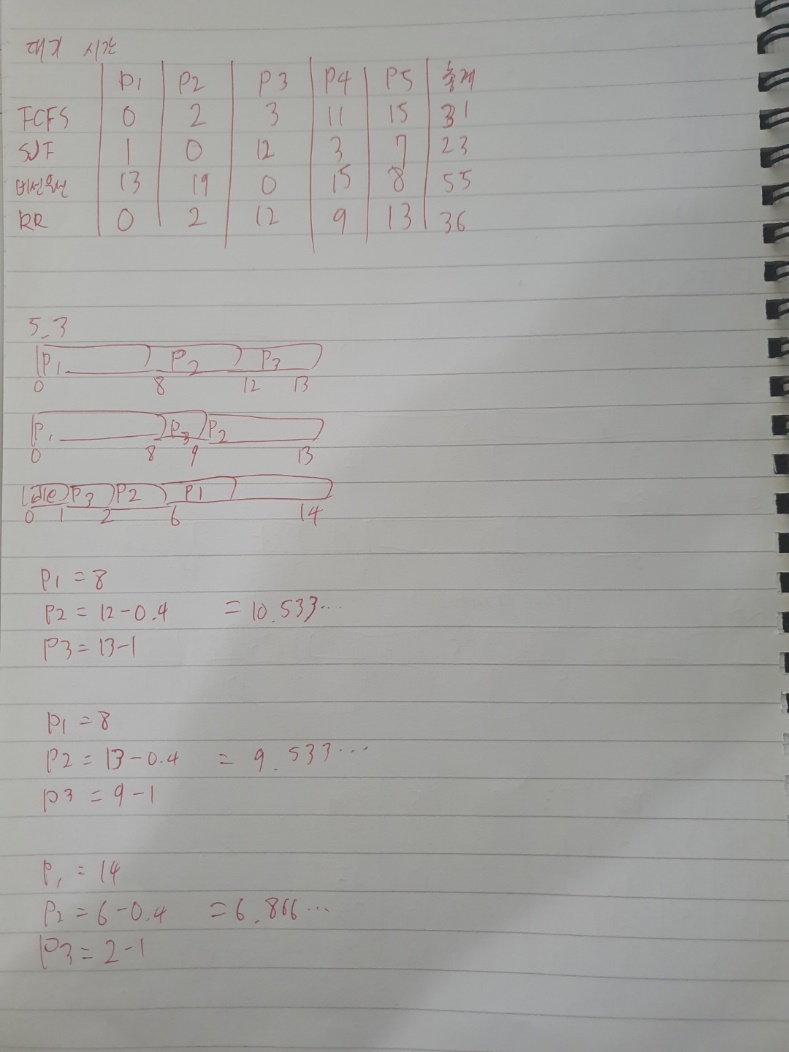
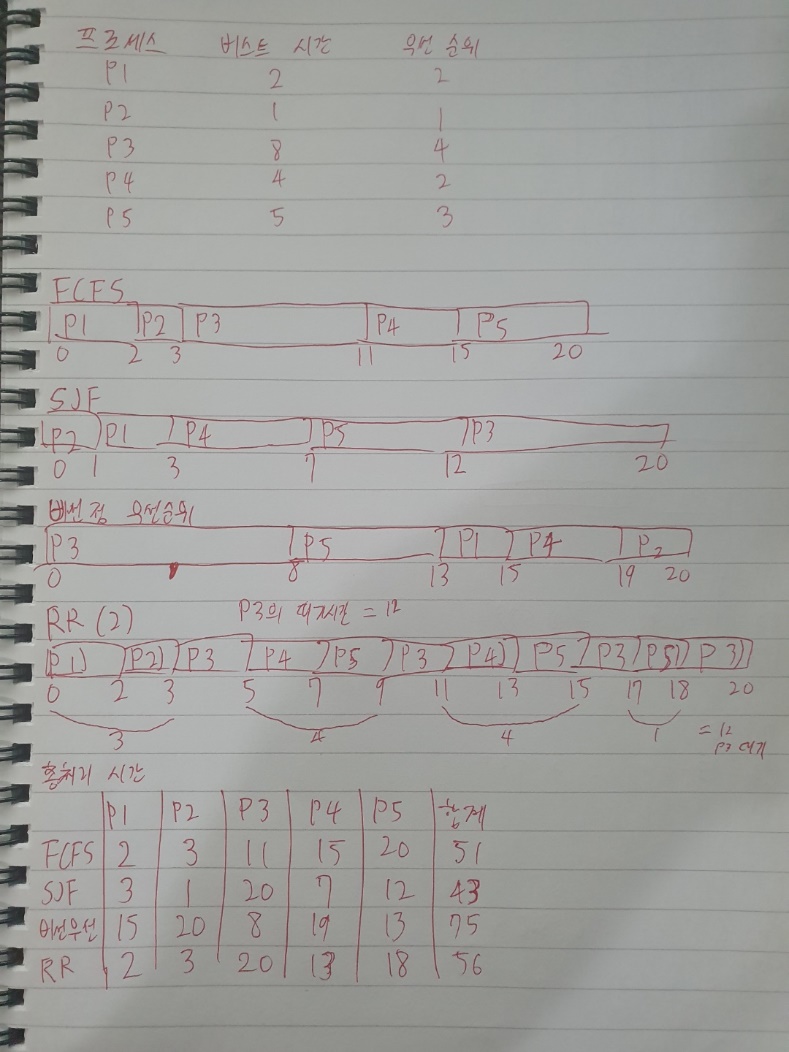
우선순위 스케줄링은 선점형이거나 비선점형이 될 수 있다

우선순위 스케줄링 알고리즘의 주요 문제는 **기아 상태(Stravation)**이다

우선순위가 낮은 프로세스들이 CPU를 무한히 대기하는 경우가 발생한다

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

노화는 오랫동안 시스템에서 대기하는 프로세스들의 우선순위를 점진적으로 증가시킨다



**동기화**

공유 데이터를 동시에 접근하면 데이터의 일관성을 망칠 수 있다

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

**Producer**

while(true) {

/\* produce an item in next\_produced \*/

while(count == BUFFER\_SIZE) // 버퍼가 꽉 차 있으면

; /\* do nothing \*/

buffer[in] = next\_produced;

in = (in + 1) % BUFFER\_SIZE;

count++;

}

**Consumer**

while (true) {

while (count == 0)

; /\* do nothing \*/ // 버퍼가 비어 있으면

next\_consumed = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

count--;

/\* consume the item in next\_consumed \*/

}

두 개의 프로세스가 동시에 변수 count를 조작하도록 허용한다면

**경쟁 상황(Race Condition)**이 일어날 수 있다

T0: producer execute register1 = count [register1 = 5]

T1: producer execute register1 = register1 + 1 [register1 = 6]

T2: consumer execute register2 = count [register2 = 5]

T3: consumer execute register2 = register2 – 1 [register2 = 4]

T4: producer execute count = register1 [count = 6]

T5: consumer execute count = register2 [count = 4]

위와 같은 race condition으로부터 보호하기 위해, 우리는 한 순간에 하나의 프로세스만이 변수 count를 조작하도록 보장해야 한다

어떤 형태로든 프로세스들이 동기화되도록 할 필요가 있다

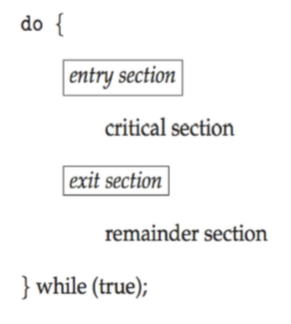
**critical section 문제**

각 프로세스는 **critical section**이라고 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)

각 프로세스는 자신의 critical section으로 진입하려면 진입 허가를 요청해야 한다. 이러한 요청을 구현하는 코드 부분을 **entry section**이라고 부른다

critical section 뒤에는 **exit section**이 따라올 수 있다

코드의 나머지 부분들을 총칭하여 **remainder section**이라고 부른다



**critical section 문제에 대한 해결안은 다음의 세 가지 요구 조건을 충족해야 한다**

1. **상호 배제(mutual exclusion)**: 프로세스 Pi가 자신의 critical section에서 실행된다면, 다른 프로세스들은 그들 자신의 critical section에서 실행될 수 없다

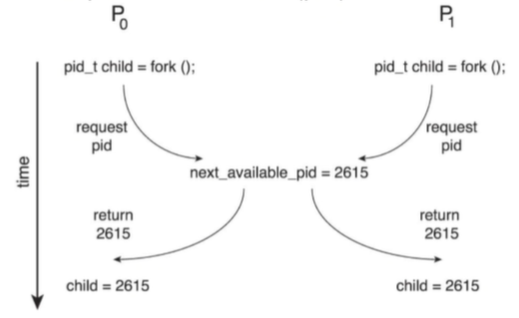
2. **진행(progress)**: 자기의 critical section에서 실행되는 프로세스가 없고 그들 자신의 critical section으로 진입하려는 프로세스들이 있다면, remainder section에서 실행 중이지 않은 프로세스들만 다음에 누가 그 critical section으로 진입할 수 있는지를 결정하는 데 참여할 수 있으며, 이 선택은 무한정 연기될 수 없다

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

각 프로세스가 0이 아닌 속도로 실행되는 것을 가정한다

그러나 n개의 프로세스 간의 상대적인 속도에 대한 가정은 하지 않는다

pid를 배정할 때의 race condition



**Peterson의 해결안**

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

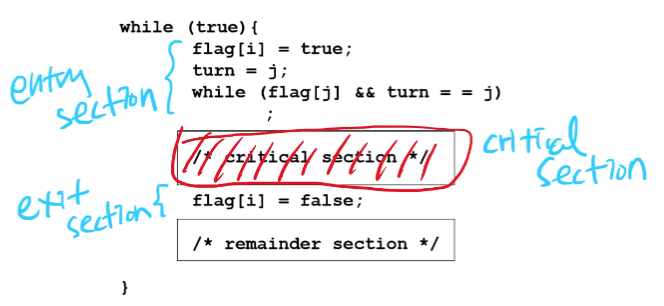
두 프로세스는 두 개의 데이터 항목을 공유한다

int turn;

boolean flag[2];

turn은 critical seciton으로 진입할 순번을 나타낸다

flag 배열은 프로세스가 critical section으로 진입할 준비가 되었다는 것을 나타낸다



while (true) {

flag[j] = true; // j가 들어갈 준비 O

turn = i; // 만약 i가 기다리고 있다면 너 실행해

while (flag[i] && turn == i) // i가 실행 중이면 대기

;

/\* critical section \*/

flag[j] = false;

/\* remainder section \*/

}

1. 상호 배제가 제대로 지켜짐

2. 진행에 대한 요구 조건을 만족함

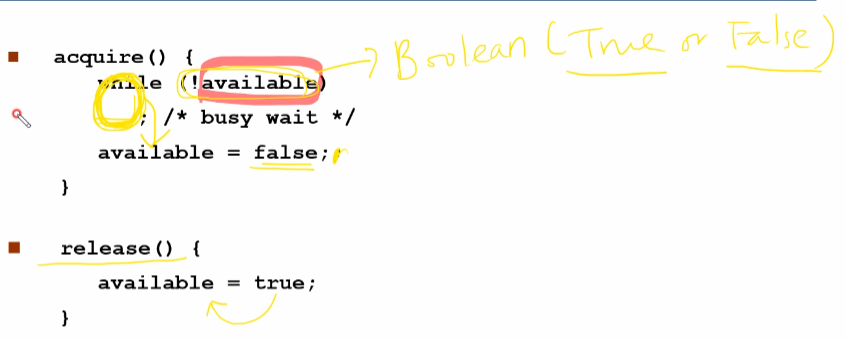
3. 대기 시간이 한없이 길어지지 않음

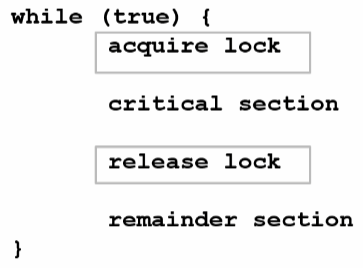
**Mutex Locks**

운영체제 설계자들은 critical section 문제를 해결하기 위한 상위 수준 소프트웨어 도구들을 개발한다

가장 간단한 도구가 바로 mutex lock이다

프로세스는 critical section에 들어가기 전에 반드시 lock을 획득해야 하고 critical section을 빠져나올 때 lock을 반환해야 한다





acquire() {

while (!available)

; /\* busy wait \*/

available = false;

}

release() {

available = true;

}

acquire() 또는 release() 함수 호출은 원자적으로 수행되어야 한다

이 솔루션의 문제는 **busy waiting**을 해야 한다는 것이다

프로세스가 critical section에 있는 동안 critical section으로 들어가기를 원하는 다른 프로세스들은 acquire() 함수를 호출하는 반복문을 계속 실행해야 한다

이런 mutex lock과 같은 유형을 **스핀락(spinlock)**이라고도 한다

락을 사용할 수 있을 때까지 프로세스가 “회전”하기 때문이다

그러나 문맥 교환이 필요하지 않다는 장점이 있다

최신 다중 코어 컴퓨팅 시스템에서 스핀락은 많은 운영체제에서 널리 사용된다

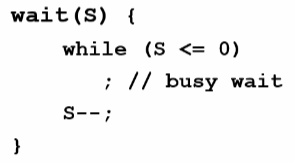
**세마포**

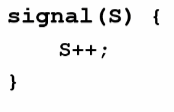
mutex lock은 일반적으로 동기화 도구의 가장 간단한 형태로 생각된다

세마포는 mutex와 유사하게 동작하지만 프로세스들이 자신들의 행동을 더 정교하게 동기화할 수 있는 방법을 제공하는 강력한 도구이다

세마포 S는 정수 변수로서, 초기화를 제외하고는 단지 두 개의 표준 원자적 연산 wait()와 signal()로만 접근할 수 있다

wait() 연산은 검사하다, signal() 연산은 증가하다라는 의미로 지어졌다





**세마포 사용법**

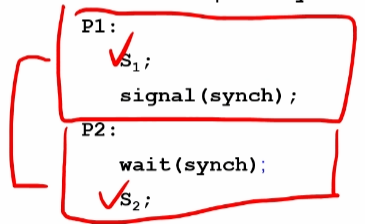
운영체제는 종종 카운팅(counting)과 이진(binary) 세마포를 구분한다

**카운팅 세마포**의 값은 제한 없는 영역(domain)을 갖는다

**이진 세마포**의 값은 0과 1 사이의 값만 가능하다

따라서 이진 세마포는 mutex lock과 유사하게 동작한다

카운팅 세마포는 유한한 개수를 가진 자원에 대해 접근을 제어하는 데 사용될 수 있다



P1의 S1 부분이 반드시 P2의 S2 부분보다 먼저 실행되기를 바라는 경우

S1이 끝난 후

signal(synch) {

synch++;

}

wait(synch) {

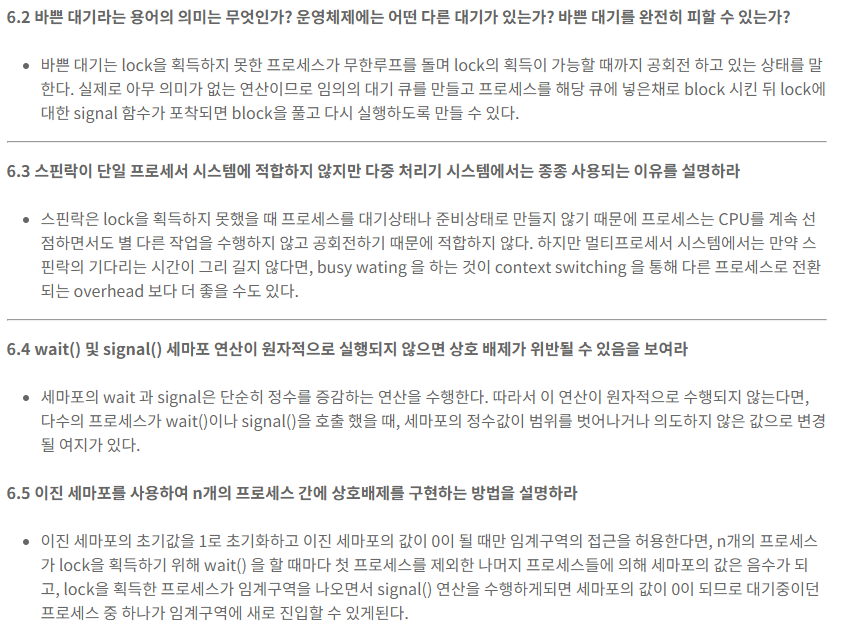
while (synch <= 0) // signal(synch)가 호출되어 synch가 1이 되고 while문에서 빠져나감

;

synch--;

}

S2 실행



- 경쟁 조건은 프로세스가 공유 데이터에 병행하게 접근할 때 발생하며 최종 결과는 병행 접근이 발생항 특정 순서에 따라 다르다. 경쟁 조건으로 인해 공유 데이터 값이 손상될 수 있다

- 임계구역은 공유 데이터가 조작될 수 있으며, 경쟁 조건이 발생할 수 있는 코드 영역이다. 임계구역 문제는 데이터를 협력적으로 공유하기 위하여 자신의 활동을 동기화하는 프로토콜을 설계하는 것이다

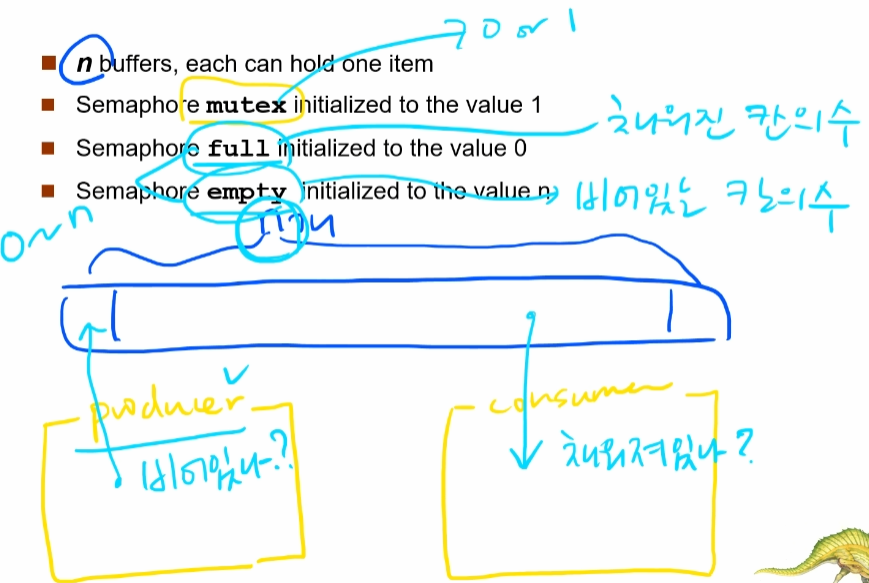
- 임계구역 문제에 대한 해결책은 (1) 상호 배제, (2) 진행 및 (3) 한정된 대기의 세 가지 요구 사항을 충족해야 한다. 상호 배제를 통해 한 번에 하나의 프로세스만 임계구역에서 활성화된다. 진행은 프로그램들이 다음에 어떤 프로세스가 임계구역에 들어갈 것인지 협력적으로 결정하리라는 것을 보장한다. 한정된 대기는 프로그램이 자신의 임계구역에 들어가기 전에 대기하는 시간을 제한한다.

- Peterson의 해결안과 같은 임계구역 문제에 대한 소프트웨어 해결책은 최신 컴퓨터 아키텍처에서 제대로 작동하지 않는다.

- mutex lock은 프로세스가 임계구역에 들어가기 전에 락을 획득하고 임계구역에서 나올 때 락을 해제할 것을 요구함으로써 상호 배제를 제공한다

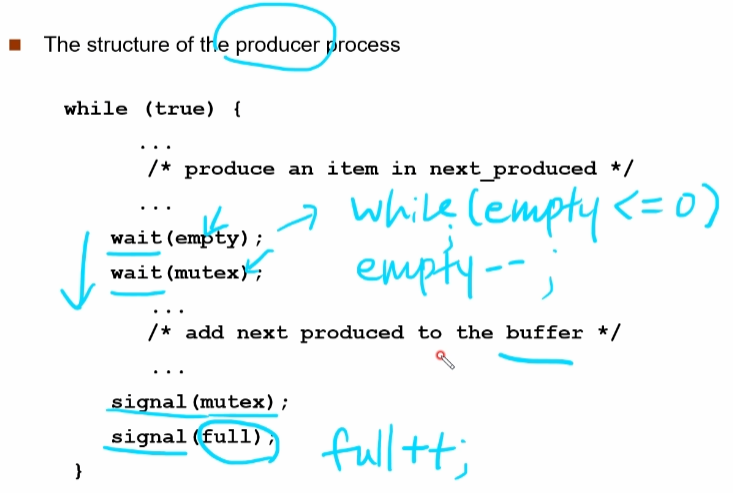
- mutex lock과 같이 세마포를 사용하여 상호 배제를 제공할 수도 있다. 그러나 mutex 락은 락의 사용 여부를 나타내는 이진 값을 가지지만, 세마포는 정수 값을 가지므로 다양한 동기화 문제를 해결하는 데 사용할 수 있다.

**Bounded-Buffer 문제**



mutex 이진 세마포

full, empty 카운팅 세마포



wait(empty) {

while (empty <= 0) // 빈 칸이 있을 때까지

;

empty--;

}

while(mutex) {

while (mutex <= 0) // 열쇠를 가져옴

;

mutex--;

}

/\* 버퍼에 작성, 크리티컬 섹션 \*/

signal(mutex) {

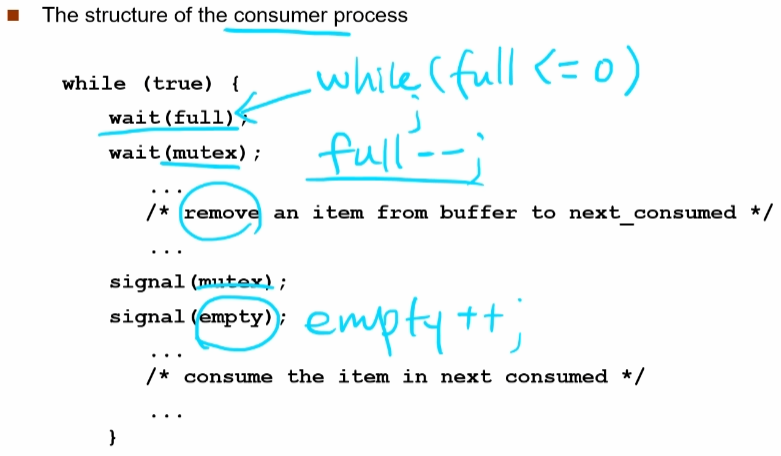
mutex++; // 키 반납

}

signal(full) {

full++; // 버퍼 1 증가

}



wait(full) {

while (full <= 0) // full 버퍼에 값이 있기를

;

full--;

}

wait(mutex) {

while (mutex <= 0) // 키를 가져옴

;

mutex--;

}

/\* 소비 항목을 버퍼에서 제거, 크리티컬 섹션 \*/

signal(mutex) {

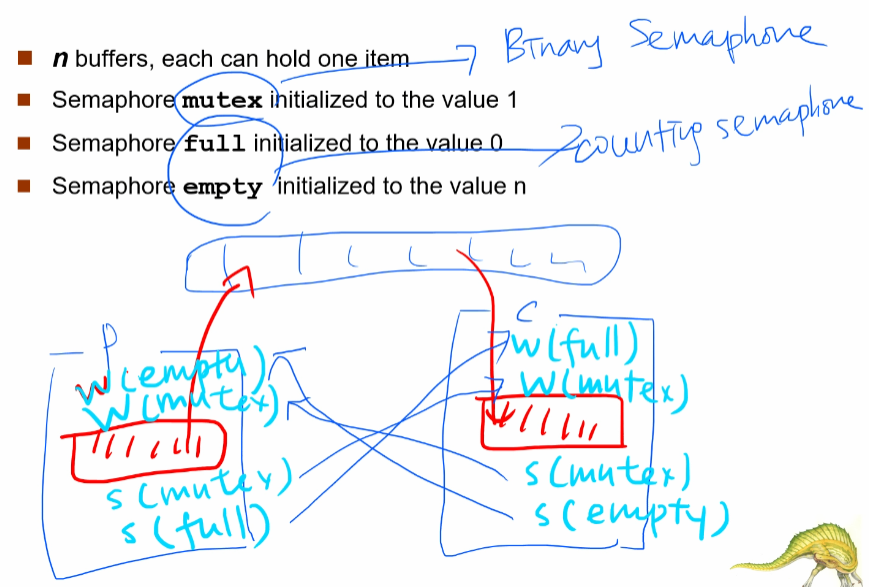
mutex++; // 키 반납

}

signal(empty);

empty++; // 빈 칸 증가

}



**Reader-Writer 문제**

reader – 데이터베이스의 내용을 읽기만 함

writer – 갱신(읽고 쓰기)을 함

만약 두 reader가 동시에 공유 데이터에 접근하더라도 불행한 결과가 발생하지는 않는다

하지만 하나의 writer와 어떤 쓰레드(reader or writer)가 동시에 데이터베이스에 접근하면, 혼란이 야기될 수 있다

semaphore rw\_mutex = 1;

semaphore mutex = 1;

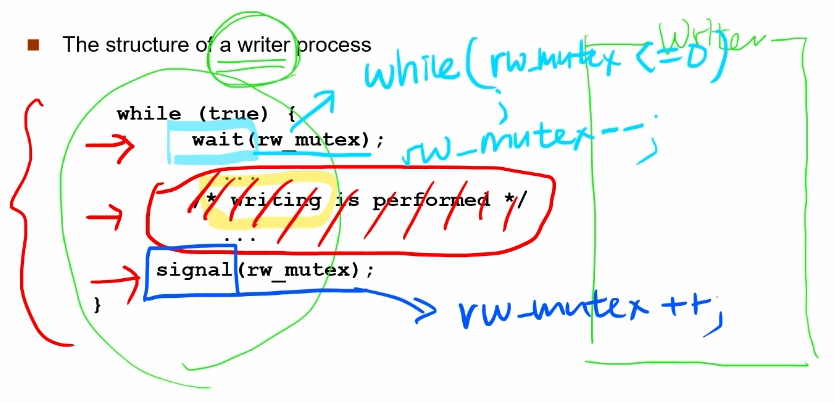
int read\_count = 0;

mutex와 rw\_mutex 세마포는 각각 1로 초기화되고 read\_count는 0으로 초기화된다

mutex 세마포는 read\_count를 갱신할 때 상호 배제를 보장하기 위해 사용된다

read\_count는 현재 몇 개의 프로세스들이 객체를 읽고 있는지 알려준다

rw\_mutex 세마포는 writer들을 위한 상호 배제 세마포이다



wait(rw\_mutex) {

while (rw\_mutex <= 0) {

;

rw\_mutex--;

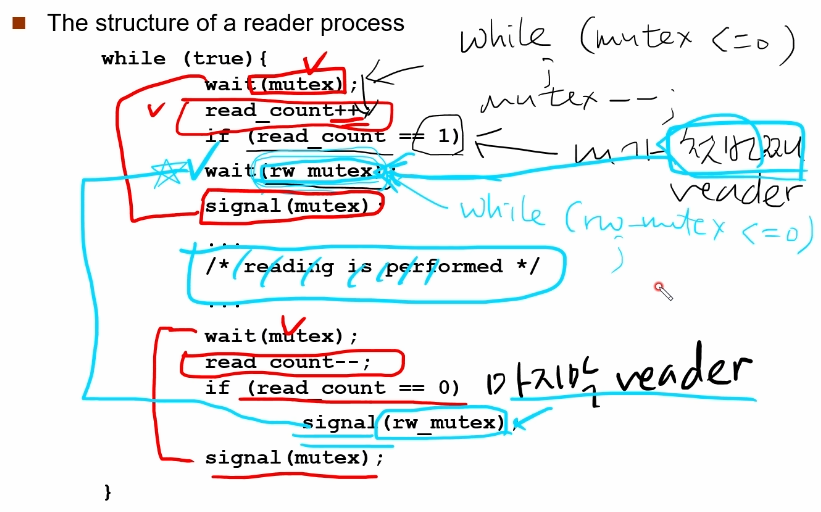
}

/\* writing is performed, critical section \*/

signal(rw\_mutex) {

rw\_mutex++;

}



wait(mutex) {

while (mutex <= 0) { // read\_count를 갱신할 때 상호 배제를 보장하기 위함

;

mutex--;

read\_count++;

if (read\_count == 1) // 만약 첫 번째 reader라면 writer가 못 들어오게 막는 일을 해야 함

wait(rw\_mutex) {

while(rw\_mutex <= 0)

;

rw\_mutex--;

}

signal(mutex) {

mutex++;

}

/\* reading is performed \*/

wait(mutex) {

while (mutex <= 0)

;

mutex--;

read\_count--;

if (read\_count == 0) // 마지막 reader라면 writer가 들어올 수 있게 키를 걸어주는 일을 해야 함

signal(rw\_mutex) {

rw\_mutex++;

}

signal(mutex) {

mutex++;

}