**OS의 정의**

User 입장에서 OS :용이성과 성능을 위한 프로그램

시스템 입장에서 OS : 자원 할당과 제어를 위한 프로그램

**기본 구조**

- CPU 및 디바이스 컨트롤러(I/O 컨트롤러)들은 공유 메모리에 엑세스하며 버스를 통해 연결됨

- I/O 디바이스와 CPU는 동시 실행이 가능

- 각 디바이스 컨트롤러는 Local Buffer를 가짐

- CPU는 메인 메모리와 Local Buffer 간의 데이터를 송/수신

- I/O란, 디바이스로부터 컨트롤러의 Local Buffer까지의 데이터 전송

- 디바이스 컨트롤러는 Interrupt를 발생시킴으로써 CPU에게 작업이 완료되었음을 알림

**I/O 컨트롤러**

- CPU를 대신하여 I/O를 관리하며 실제로 I/O 명령을 수행

- 내부에 Register와 Data Buffer가 존재.

- Bus(Address Bus / System Bus)를 통해 Read/Write 명령을 수행하여 CPU와 통신

CPU가 I/O 컨트롤러의 Data Buffer에 데이터를 채운 후 Write 명령을 수행하면, I/O 컨트롤러는 사용 가능한 상태인지 확인한 후 작업을 수행. 작업이 끝나면 Interrupt를 발생시켜 CPU에게 작업이 완료되었음을 알림.

**인터럽트의 기능**

- 일반적으로, 인터럽트는 **인터럽트 벡터**를 통해 제어를 Interrupt Service Routine 또는 Interrupt Handler에 전송

**(인터럽트 벡터**([영어](https://ko.wikipedia.org/wiki/%EC%98%81%EC%96%B4): interrupt vector)는 인터럽트가 발생했을 때, 그 인터럽트를 처리할 수 있는 서비스 루틴들의 주소를 가지고 있는 공간이다. 인텔 아키텍처에서는, 가상메모리를 사용하지 않았던 386이전에는 주로 Main Memory의 0번째에 위치하였으나, 이후부터는 IDT(Interrupt descriptor table)의 형태로 바뀌었다.)

- 인터럽트 핸들러는 OS의 일부

- 인터럽트 아키텍처는 인터럽트 된 명령어의 주소를 저장해야 함

- **Trap** (또는 예외)은 오류 또는 사용자 요청으로 인해 소프트웨어에서 생성 된 인터럽트 (0으로 나누기, 페이지 폴트 등)

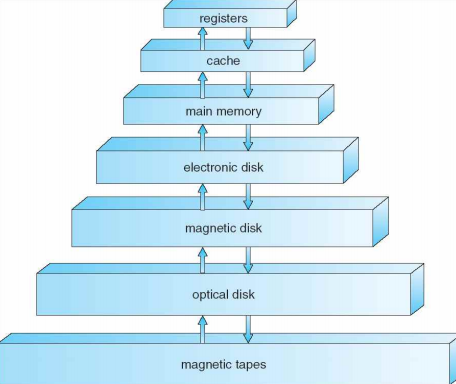
**Storage 구조**

- Main Memory : CPU가 직접 엑세스할 수 있는 대용량 저장 장치

- 2차 저장소 : 비휘발성 저장소 용량을 제공하는 Main Memory의 확장

- 자기 디스크 : 자기 기록 물질로 덮힌 단단한 금속 또는 유리 플래터. 디스크 표면은 섹터로 세분화되는 트랙으로, 논리적으로 나뉘어짐. 디스크 컨트롤러는 장치와 컴퓨터 간의 논리적 상호 작용을 결정

**메모리 계층 구조**



**CPU가 메모리에 더 빨리 접근하기 위해 나눈 구조(속도, 가격, 휘발성을 고려)**

- Register와 Cache는 CPU 내부에 존재.

- Register는 연산을 위한 저장소

- Cache는 CPU와 RAM 사이의 중간 저장소

- Main Memory는 CPU 외부에 있지만 CPU의 Cache와 연결됨.

- 그외 디스크들은 CPU가 직접 접근할 수가 없기 때문에, 데이터를 Main Memroy로 이동시키는 과정이 필요

\*CPU가 빠르게 데이터에 접근하기 위해서는 데이터를 저장하는 메모리가 작아야 한다. 실제로 레지스터는 캐시보다 더 작은 용량을, 캐시는 메모리보다, 메모리는 하드 디스크보다 더 작은 용량을 가진다. 메모리 구조 그림1에서는 위 즉 높은 속도에 있는 장치가 더 작은 크기를 차지하는 방식으로 이를 표현했다.

**메모리 엑세스 구조**

- 메모리 속도에 가까운 속도로 정보를 전송하기 위해 고속 I / O 장치에 사용됨

- 디바이스 컨트롤러는 CPU 개입없이 버퍼 블록에서 데이터 블록을 직접 주 메모리로 전송함

- 바이트 당 하나의 인터럽트보다는 블록 당 하나의 인터럽트만 생성됨

**일반적인 OS 구조**

테이블이(가) 표시된 사진

자동 생성된 설명

- 시스템에 있는 Job들은 메모리에 유지된다.

- 스케줄링을 통해 하나의 Job이 선택된다.

- I/O가 발생하면, OS는 다른 Job으로 컨텍스트 스위칭을 한다.

- **시 분할(멀티 태스킹)시스템은** CPU가 스위칭을 자주 수행하는 방식이다.

- User는 메모리에서 적어도 하나의 프로그램을 실행한다. ->**Process**

- 만약, 모든 Job들이 동시에 실행할 준비가 되었다면, 스케줄링 수행

- 만약 프로세스들이 메모리에 맞지 않으면, **Swapping**을 수행

-**가상 메모리**는 RAM이 부족한 경우에도 프로세스들을 실행할 수 있도록 해줌

OS는 어떤 프로그램에 의해 다른 프로그램이 오류가 발생하는 경우가 없도록 설계되어야 한다.

- Dual Mode Operation (커널 / 유저)

- I/O 보호

- Memory 보호

- Timer

**Dual Mode Operation**은 운영체제가 자체적으로 시스템 구성을 보호할 수 있도록 Dual Mode로 작동하도록 한다.

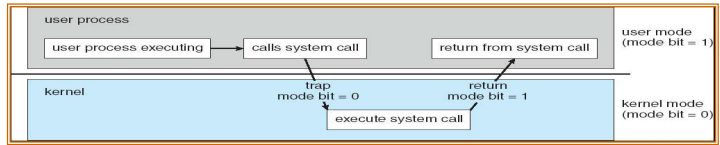
- User Mode / Kernel Mode로 나뉘어짐

- 하드웨어에 의해 제공되는 Mode Bit

- User Code / Kernel Code가 실행될 때를 구분하는 기능 제공

- 일부 명령들은 커널 모드에서만 실행 가능하도록 하는 권한을 가진다.

- System Call은 User Mode를 Kernel Mode로 바꾸고, 작업이 종료될 때 유저 모드로 다시 전환한다.



**I/O Protection**

 유저가 잘못된 I/O 명령을 직접 시도할 수 있다면(I/O register를 User Mode에서 직접 다룰 수 있다면) 오류가 발생할 가능성이 높아지고, 이는 OS의 정상적인 동작을 방해할 수 있다. 그래서 유저가 직접 수행하지 못하도록 I/O 명령을 모두 Privileged Instruction으로 만들어서 Kernel Mode에서만(커널이 대신) 수행할 수 있도록 한다.

- 모든 I/O 명령은 권한이 존재한다.

- 유저모드 실행 불가 -> 커널에 요청하여 커널 모드에서 작동

**Memory Protection**

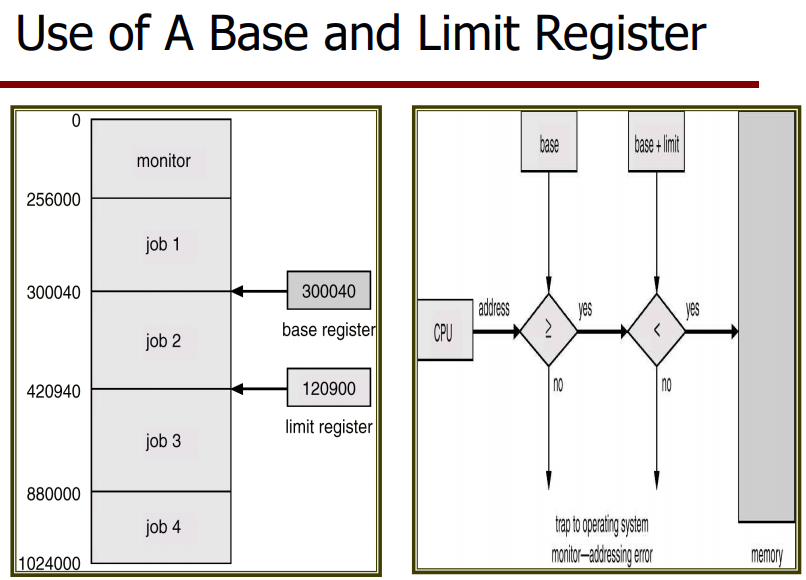
- 인터럽트 벡터와 인터럽트 서비스 루틴에 대해 메모리 보호를 제공해야 한다.

- 메모리 보호를 위해 프로그램이 엑세스 가능한 주소 범위를 결정하는 두 개의 Register를 추가한다.

- Base Register : 유효한 최소 물리 메모리 주소

- Limit Register : 크기

이렇게 함으로써, 정의된 범위를 벗어난 메모리가 보호된다.



해당 범위 밖이라면 운영체제에게 오류를 알린다(Trap)

**Timer**

- 지정된 기간이 지나면 인터럽트를 발생시킴

- 매 Clock Tick마다 Timer가 감소

- Timer가 0이 되면 인터럽트 발생

- 보통 시간 공유를 구현하는 데에 사용됨

- 현재 시간을 계산하는 데에도 사용

- Load-Timer는 Privileged Instruction

**System Call**

- OS에 의해 제공되는 서비스에 대한 프로그래밍 인터페이스

- 응용프로그램에서 운영체제에게 커널 기능 수행을 요청하는 방법. 즉, 유저 프로세스가 소프트웨어 인터럽트를 통해 커널의 기능을 이용하기 위한 서비스를 요청하는 방법.

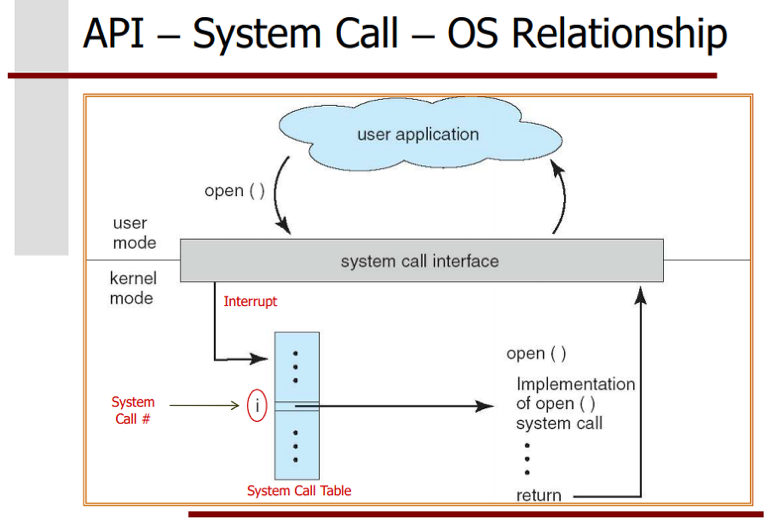
- 일반적으로 High-Level Language로 구현됨

- 어플리케이션을 통해 프로그램에서 엑세스(Windows API, Posix API 어플리케이션 혹은 JVM 등등)

**System Call Interface**

- 각 시스템 콜에는 번호가 할당된다.

- 시스템 콜 인터페이스는 이 번호에 대해 인덱스되는 테이블을 가지고 있는데, 다음과 같이 동작한다.



어떤 프로세스가 실행되고 있다면, 기본적으로 User Mode로 동작한다. 이 때, I/O Instruction을 사용하려 한다면(Privileged Intstruction이므로) 소프트웨어 레벨에서 인터럽트를 걸어, Kernel Mode로 변경해 주어야 한다.

종종 System Call이 발생할 때 많은 정보가 필요할 수 있는데, 일반적으로 3가지 방법이 존재한다.

1. 레지스터에 그대로 매개변수를 전달 : 레지스터 수보다 매개변수가 더 많을 수 있음

2. 매개변수들을 메모리 블록에 저장하고, 해당 블록의 주소를 레지스터에 전달 : Linux와 Solaris가 사용하는 방법

3. OS에 의해 Stack에서 Pop되거나 프로그램에 의해 Stack에 Push되는 방식

2, 3번째 방법은 매개변수의 수가 제한되지 않는다.

**Types of System Calls**

- 프로세스 제어

- end / abort

- load / excute

- create / terminate process

- get / set process attributes

- wait for time

- wait / signal event

- allocate / free memory

- 파일 관리

- create / delete

- open / close

- read / write / reposition

- get / set file attributes

- 장치 관리

- request / release device

- read / write / reposition

- get / set device attributes

- attach / detach devices

- 정보 관리

- get / set [time / date / process / file / device] attributes

- 통신

- create / delete communication connection

- send / receive messages

- transfer status info

- attach / detach remote devices

**커널에서 사용되는 자료구조**

- Stack

- Queue

- List (Singly / doubly / circularly)

- Tree

- Hash Function

- Bitmap

**System Boot (간략)**

1. 전원

2. BootStrap Loader가 저장되어 있는, 지정된 메모리(ROM에서)를 읽고 코드를 실행(부트스트랩에 실행 코드가 작성되어 있음)

-> 커널 로딩

**운영체제 기능**

- User Interface

- CLI 및 GUI 제공

- 프로그램 실행

- 프로그램을 메모리에 로드하고, 실행하며, 정상/비정상적으로 종료할 수 있음

- I/O 연산

- 실행된 프로그램은 I/O가 필요할 수 있는데, 이는 파일 또는 I/O 장치도 포함됨

- File 조작

**통신**

- 프로세스는 네트워크를 통해 컴퓨터 혹은 프로세스간에 정보를 교환할 수 있음

- Shared Memory 혹은 메시지 전달을 통해 이루어짐

- 오류 감지

- CPU, 메모리, I/O 장치, 유저 프로그램 등에서 발생

- 각 오류에 대해 정확하고 일관된 처리를 보장

**자원 관리**

- 자원 할당

- 여러 유저들 또는 여러 Job들이 동시에 실행되고 있을 때, 각각 자원을 할당시켜 주어야 함

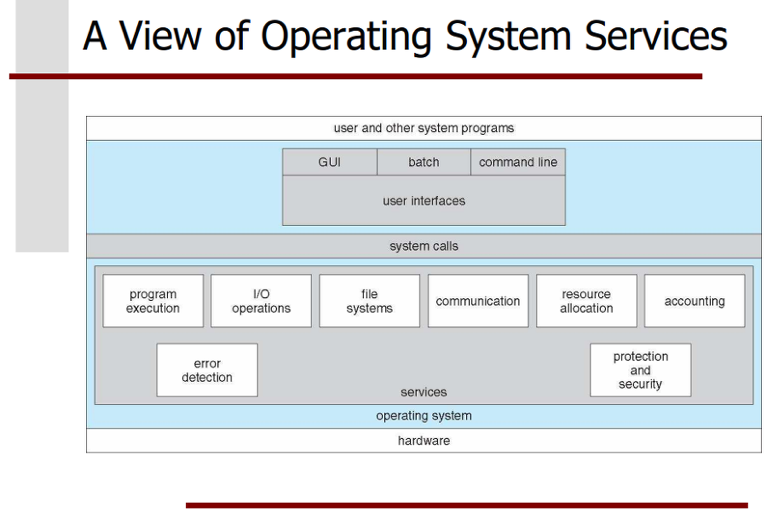
- Accounting

- 어떤 유저가 얼마나 많은 자원을 사용하는지 추적해야 함

- Protection / Security

- 프로세스 간에 서로 간섭하지 못하도록 해야 함

- 자원에 대한 모든 접근 제어를 보장하는 것을 수반해야 함



**OS의 구성 요소**

- 프로세스 / 스레드 관리

- 프로세스 동기화

- 메모리 관리

- 가상 메모리 관리

- 파일 시스템 관리

- 저장소 관리

- I/O 관리

- Protection / Security

**Unix는 시스템 프로그램 / 커널로 구성됨.**

커널

- 물리적 하드웨어부터 시스템 콜 인터페이스까지

- 파일 시스템, CPU 스케줄링, 메모리 관리 등을 제공

## [OS](https://teraphonia.tistory.com/category/OS)

[**[OS] 정리2**](https://teraphonia.tistory.com/786)

2018. 9. 23. 14:46

**Program**이란, 디스크에 존재하며 명령들의 리스트를 포함하고 있는 파일

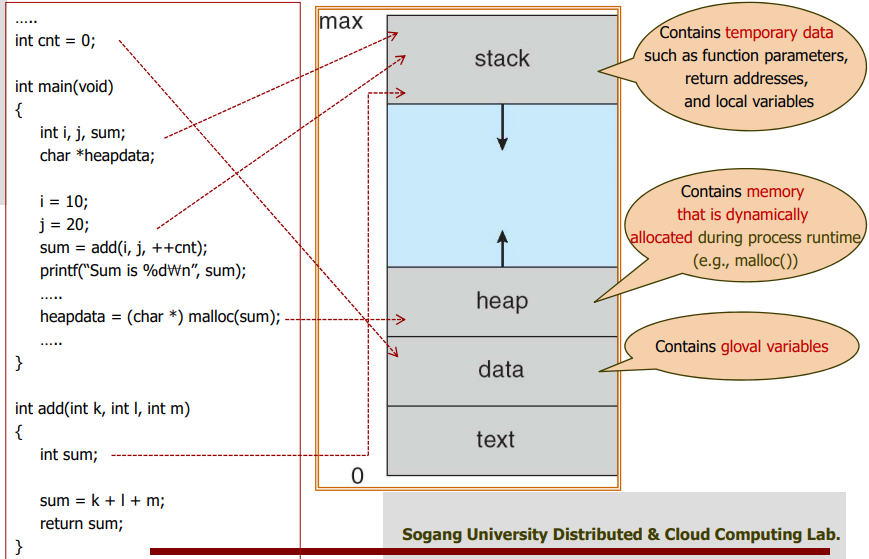
**Process**

- Process란 Program과 Program Counter를 의미한다. 프로그램이 메모리에 올라가면 프로세스가 된다.

- Code(TexT) / Data / Stack / Heap / Program Counter로 구성되어 있다.

- Data : 전역변수 / Static 변수가 저장됨

- Stack : 임시 변수 - 지역변수 및 함수 매개변수, 리턴값



**Process State**

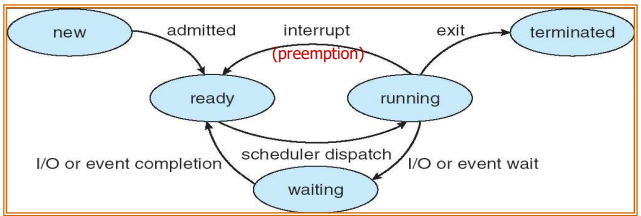
- new : 프로세스가 생성됨

- ready : CPU에 할당되기를 기다림

- running : 명령들 실행중

- waiting : Event 발생 기다림

- terminated : 실행 종료됨



**PCB(Process Control Block)**

- 프로세스 관련 정보를 포함

- 프로세스 상태

- 프로세스 번호

- PC

- CPU Registers

- CPU 스케줄링 정보

- 메모리 관리 정보

- I/O 상태 정보

테이블이(가) 표시된 사진

자동 생성된 설명

**프로세스 스케줄링**

- CPU를 최대한 효율적으로 사용하기 위한 전략

- Time Sharing : 사용자가 각 프로그램을 실행하는 동안 상호작용을 할 수 있도록 프로세스간에 CPU를 전환하는 것

- Uni-Processor 시스템은 프로세스를 하나만 가질 수 있음

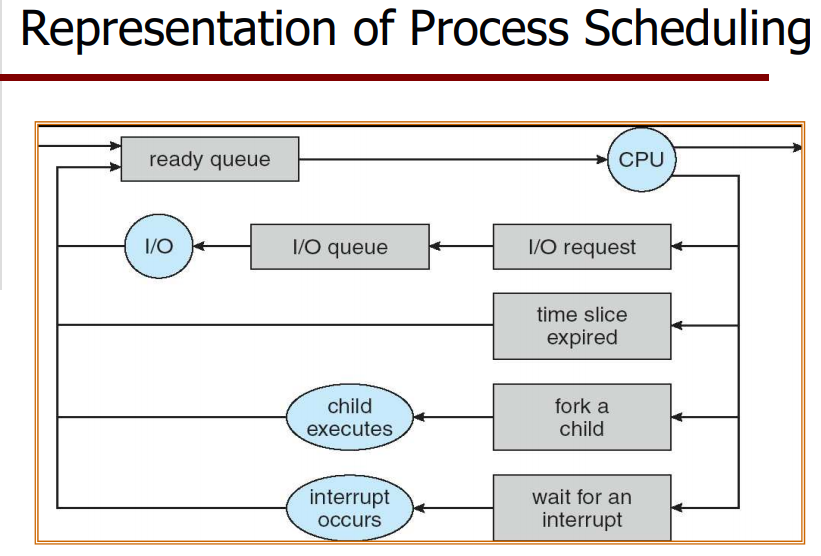
- 프로세스가 여러개라면, 실행 중인 프로세스를 제외한 나머지는 CPU를 사용 가능할 때까지 대기해야 하며, Reschedule될 수 있음

**프로세스 스케줄링 Queue**

- Job queue : 시스템의 모든 프로세스 집합

- Ready queue : RAM에 상주하며, waiting, ready 상태의 프로세스 집합

- Device queue(I/O Queue) : I/O Device를 기다리는 프로세스 집합



**Schedulers**

- Long-Term 스케줄러(Job 스케줄러) : 디스크로부터 Main Memory 상의 ready\_queue로 옮길 프로세스 선택(디스크 -> 메인메모리)

- Short-Term 스케줄러(CPU 스케줄러) : ready\_queue의 프로세스 중 다음에 실행할 프로세스를 선택하고 CPU 할당(메인메모리 -> CPU)

- Long-Term 스케줄러는 드물게 수행된다(초 or 분 단위)

- Short-Term 스케줄러는 자주 호출된다(밀리 초)

CPU Burst time : CPU 작업수행시간

I/O Burst time : I/O 요청에 대한 응답 대기시간

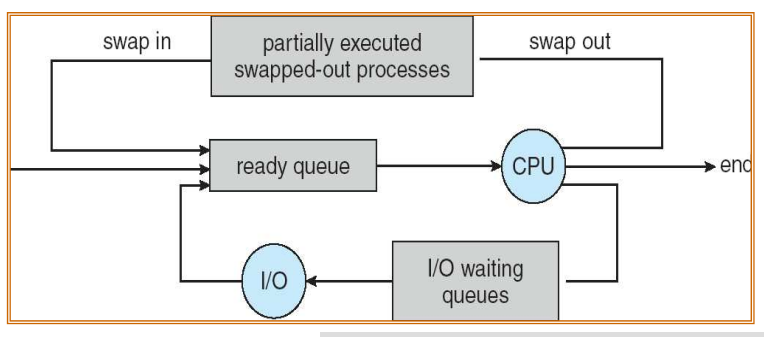
**Long-Term 스케줄러**

- I/O Bound Process : 수행 속도가 I/O 시스템의 속도에 의해 제한되는 프로세스 / 짧은 CPU 버스트를 가짐

- CPU Bound Process : 수행 속도가 CPU 속도에 의해 제한되는 프로세스 / 적은 수의 매우 긴 CPU 버스트를 가짐

**Medium-Term 스케줄러**

- Swapping을 이용 : 때로는 사용하지 않는 프로세스를 메인 메모리에서 디스크로 옮기고(Swap-Out), 나중에 해당 프로세스를 디스크로부터 메인 메모리에 다시 올려서(Swap-In) 실행을 지속할 수 있다.

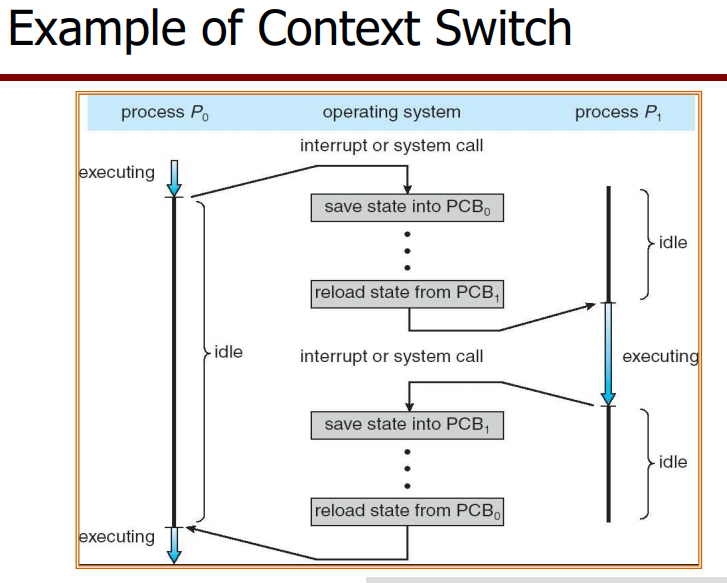


**Context Switch**

- CPU가 다른 프로세스로 전환할 때, 이전 프로세스의 상태를 저장하고 새 프로세스의 저장된 상태를 로딩해야 한다.

- Process의 Context(CPU Registers, 프로세스 상태, 메모리 관리 정보 등)는 PCB에 저장되어 있다.

- 스위칭은 부하가 있는 작업이다. 일반적으로 스위칭하는 데에 몇 밀리초가 걸리지만, 이는 하드웨어에 따라 다르다.



**Process 생성**

- 부모 프로세스가 자식 프로세스를 만드는 형태이고, 이는 Tree로 구현된다.

- process id로 관리됨

- 자원 공유 옵션

- 부모 / 자식 프로세스는 모든 자원을 공유한다.

- 부모 자원의 일부만 공유한다

- 공유하지 않는다.

- 실행 옵션

- 부모 / 자식 프로세스는 동시에 실행된다.

- 부모는 자식 프로세스가 종료될 때까지 대기한다.

- 주소공간 옵션

- 자식 프로세스는 부모 프로세스의 복사본이다

- 로드될 프로그램을 가지고 있다

**Zombie 프로세스**

- 종료되었지만 Process Table에 남아 있는 프로세스. 보통 exit() 함수를 호출하면서 프로세스를 종료시키고, 모든 자원을 해제시킨다. 그러나 프로세스의 exit 상태와 pid가 여전히 남게 된다. 이렇게 남겨 두는 이유는, 부모 프로세스에서 자식 프로세스의 종료 상태까지 확인할 수 있도록 하기 위함이다.

(IPC는 따로 공부하자)

**IPC(InterProcess Communication) : 프로세스 간 통신**

- Cooperating 프로세스

- Independent 프로세스는 다른 프로세스에게 영향을 주거나 받을 수 없는 프로세스를 말한다. 그 반대를 바로 Cooperating 프로세스라 한다.

- 정보 공유, 계산속도 향상, 편리함.

- 공유 메모리 : 자원 공유

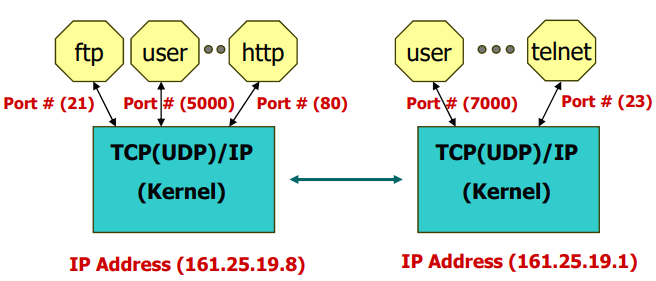
- Message Passing : 동기화 목적

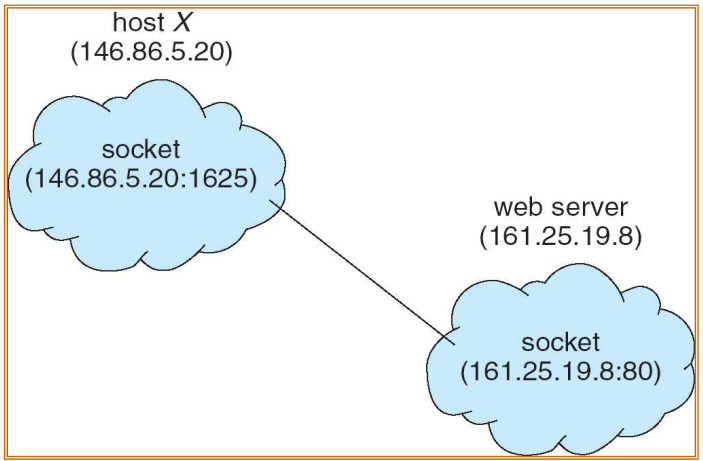
**Socket**

- 통신을 위한 Endpoint

- IP 주소와 Port 번호를 붙여 사용

- 통신은 두 소켓 사이의 작용





**MultiThread Model**

- 프로세스의 생성 및 컨텍스트 스위칭 부하가 크므로 Thread 사용

**Thread**

- CPU 스케줄링의 기본 단위이자 프로세스 내의 제어의 흐름이라고 할 수 있다.

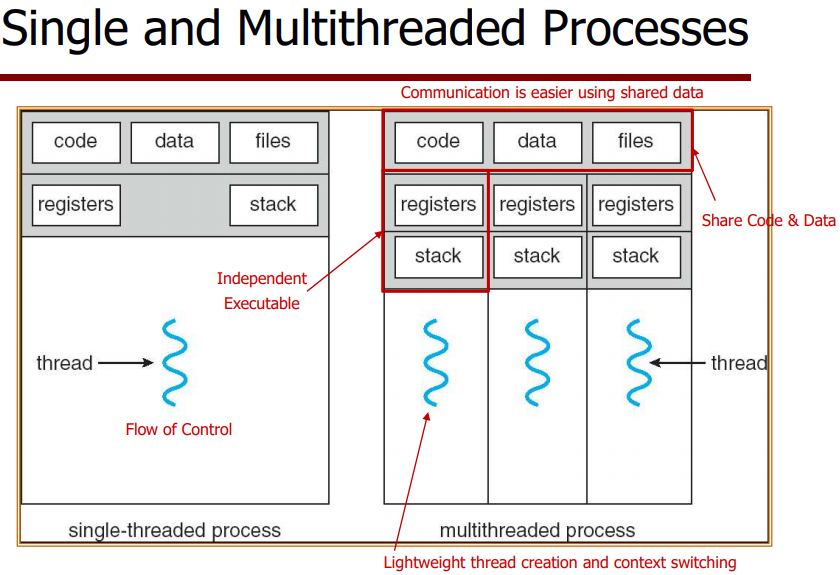
- PC / Registers / Stack 으로 이루어져 있다.

- 각 Thread는 레지스터 상태와 Stack을 가진다.

- 프로세스 내부에서 여러 스레드가 생성될 수 있고, Code / 주소공간 / 운영체제 자원을 공유한다.

- Thread 실행 환경을 Task라고 하며, 프로세스는 기본적으로 하나의 스레드를 가진다.

- Thread의 생성 및 Switching은 프로세스에 비해 매우 부하가 적다



**멀티 프로세서 혹은 멀티코어 환경**

- 스레드는 다른 프로세서 혹은 멀티코어 환경에서 병렬로 실행될 수 있다.

**싱글 프로세서 환경**

- 한 번에 하나만 수행하고, 병렬로 실행되는 것처럼 보이도록 함

- 그렇다고 성능이 무조건 느린 것은 아니다. I/O Bound 스레드가 Block 상태가 되면, 커널이 현재 프로세스 내의 다른 스레드로 전환을 하기 때문에, 계산 속도가 향상된다.

**User Level vs Kernel Level Thread**

- User Level Thread

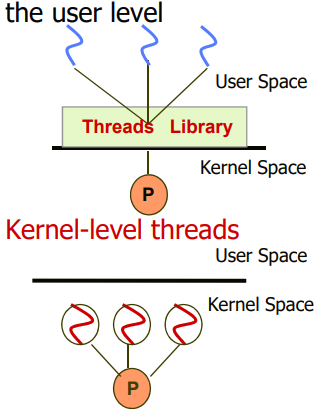
- Thread 관리가 유저레벨에서 수행됨 -> 커널은 스레드가 몇개인지도 모름

- 커널 모드 전환을 하지 않기 때문에 빠름.

- 멀티프로세서 혹은 멀티코어 환경 사용 불가

- Kernel Level Thread

- Thread가 커널레벨에서 관리됨 -> Context Switching 수행



**Signal**

- UNIX 시스템에서 특정 이벤트가 발생했음을 프로세스에 알리는 데 사용

- Synchronous signals

- signal을 유발하는 작업을 수행한 동일한 프로세스로 전달됨(불법 메모리 엑세스, 0으로 나누기 등)

- Asynchronous signals

- 외부 이벤트에 의해 발생(프로세스 종료(Ctrl + C같은), 등)

Thread 취소 : 스레드 실행이 정상 완료되기 전에 종료해 버리는 방법

- 비동기 취소 : 해당 스레드를 즉시 종료

- 지연 취소 : 취소해야 하는지, 그 상태를 주기적으로 체크하는 방식

**Thread Pool**

- 여러개의 대기 스레드를 만들어 둔다

- 장점

- 스레드를 매번 새로 만드는 것보다 이미 만들어 둔 스레드를 사용하는 것이 빠르다

- 스레드풀의 미리 만들 스레드 수를 지정할 수 있다.

- **Thread Local Storage(TLS)**

- 각각의 스레드는 고유한 스택을 가진다. 그래서 스택과 매개변수같은 경우 스레드 별로 고유한 메모리를 가진다. 하지만, 전역변수와 정적 변수같은 경우 프로세스 내의 모든 스레드가 공유하게 된다.

- 일반 전역 변수의 경우, 모든 스레드가 공유하므로 접근 시 race condition이 발생할 수 있다.

- TLS는 정적/전역 변수를 스레드 각각 독립적으로 메모리 공간을 만들어 주고 싶을 때 사용한다.

- 물론, 자료구조를 스레드 수만큼 생성하고 각각의 구역을 나누어 관리하는 식으로 구현해도 되겠지만, TLS를 쓰면 각각의 스레드에 귀속되므로 더 깔끔한 구조가 된다.

프로세스의 실행은 **CPU 실행 사이클**과**I/O 대기**로 구성된다. 일반적으로 짧은 CPU 버스트가 많고, 적은 수로 긴 CPU 버스트가 발생한다.

**CPU 스케줄러**

- 메인 메모리에 있는 Ready 상태의 프로세스들 중 하나에게 CPU를 할당시킨다.

- 스케줄링이 발생하는 경우

1. running에서 wait 상태로 변경될 때 (I/O 요청하는 경우)

2. running에서 ready 상태로 변경될 때 (Interrupt 발생)

3. wait 상태에서 ready 상태로 변경될 때 (I/O 요청 완료)

4. 종료될 때

- 1, 4일 경우에만 스케줄링하는 방식이 비선점 스케줄링이다.

- 1~4 모든 경우에 스케줄링하는 방식이 선점형 스케줄링이다.

**Dispatcher**

- Short-Term 스케줄러에 의해 선택된 프로세스에게 CPU 제어를 넘겨주는 모듈

- Context Switching

- 유저 모드로 전환

- 유저 프로그램을 재시작하기 위해 해당 주소로 이동

Dispatch Latency

- Dispatcher가 한 프로세스를 중지하고 다른 프로세스를 시작하기까지 걸리는 시간

**스케줄링의 척도(Scheduling Criteria)**

- CPU 사용률 극대화(CPU utilization) : CPU를 최대한으로 놀지 않도록 효율적으로 이용

- 처리 능력 극대화(Throughput) : 주어진 시간에 최대한 많은 작업을 처리 (처리한 프로세스 수 / 시간)

- 경과 시간(반환 시간) 최소화(Turnaround time) : ready\_queue에서 대기한 시간 ~ 작업을 마치고 반환하기까지 걸린 시간을 최소화

- 대기 시간(Waiting time) 최소화 : ready\_queue에서 대기한 시간의 총합

- 응답 시간(Response time) 최소화 : reqdy\_queue에 들어와서 처음으로 CPU를 할당받기까지 대기한 시간

**최적화 척도(Optimization Criteria)**

- 최대 CPU 사용량

- 최대 처리 시간

- 최소 반환 시간

- 최소 대기 시간

- 최소 응답 시간

**스케줄링 알고리즘**

- FCFS(First-Come, First-Served)

- SJF(Shortest Job First)

- Priority

- Round Robin(RR)

**FCFS**: 비선점 / CPU를 먼저 요청한 프로세스가 먼저 처리됨

- ready\_queue에 도착한 순서에 따라 CPU 할당

- FIFO Queue로 구현됨

- **Convoy Effect** : 수행시간이 긴 프로세스가 CPU를 독점하게 되어, 다른 프로세스들이 오랫동안 해당 프로세스가 종료되기를 기다리는 경우 발생

- 평균 대기 시간이 긴 경우가 종종 발생

테이블이(가) 표시된 사진

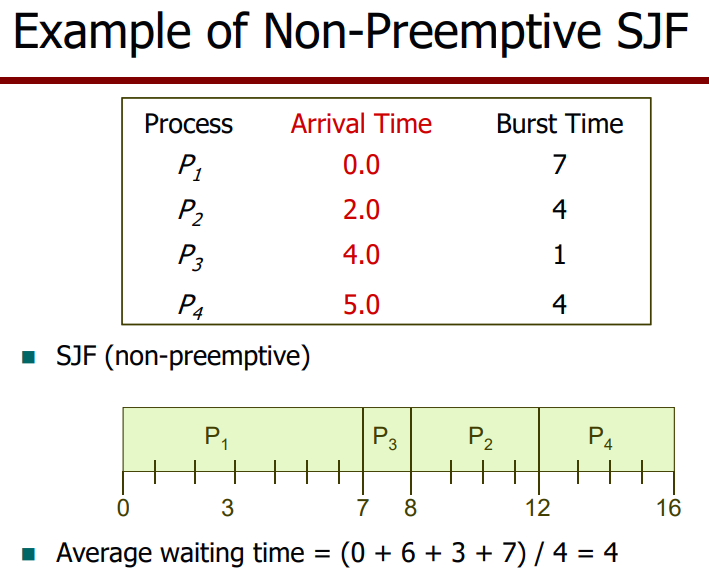
자동 생성된 설명

**SJF** : 선점 or 비선점 / ready\_queue 내의 프로세스 중 수행시간 짧다고 판단되는 것을 먼저 수행

- 각 프로세스의 CPU 버스트 길이를 비교하여 CPU가 이용 가능해지면 가장 작은 CPU 버스트를 가진 프로세스에게 할당해 줌

- 주어진 프로세스들에 대해 평균 대기시간이 최소인 최적 알고리즘

일단 먼저 도착한 프로세스를 실행할 테니, P1로 시작



테이블이(가) 표시된 사진

자동 생성된 설명

**Timeline**

0 P1(6)

1 P1(5)

2 P1(5) > P2(4) 이므로 P2실행. -> P2(3)

3 P2(2)

4 P1(5) > P2(2) > P3(1) 이므로 P3 실행 -> P3(0)

5 P1(5) > P4(4) > P2(2) 이므로 P2 실행 -> P2(1)

6 P2(0)

7 P1(5) > P4(4) 이므로 P4 실행 -> P4(3)

8 P4(2)

9 P4(1)

10 P4(0)

11 P1(4)

12 P1(3)

13 P1(2)

14 P1(1)

15 P1(0)

다음 CPU Burst 길이는

1. 이전 CPU Burst 길이

2. Exponential 평균

을 이용하여 추정한다.

**Priority Scheduling** : 선점 or 비선점 / 각 프로세스에 우선 순위 번호를 부여하고, 우선 순위가 높은 프로세스부터 CPU를 할당해 줌

- 일반적으로 낮은 번호가 높은 우선순위를 나타냄

- 어떻게 보면 SJF도 다음 CPU 버스트 타임을 이용한 우선 순위 스케줄링이라고 할 수 있음

- Starvation 문제 : 낮은 우선순위의 프로세스가 영원이 수행되지 않는 경우가 생김

- Aging : Starvation 문제를 해결하는 방법으로, 시간이 지날수록 우선 순위를 높이는 방식

**Round Robin(RR)** : 선점 / 각 프로세스는 동일한 크기의 Time Slice(CPU 시간 : 일반적으로 10 ~ 100ms로 설정한다)을 할당받음

- Time Slice동안 처리하고 완료하지 못했다면 ready\_queue의 맨 뒤로 들어간다. CPU 제어는 다음 프로세스로 넘어간다.

- 할당 시간이 크면 FCFS 스케줄링과 같아지고, 할당 시간 너무 작으면 Context Switching이 자주 발생하여 성능저하가 큼

테이블이(가) 표시된 사진

자동 생성된 설명

일반적으로 평균 처리시간이 길지만 응답 시간은 짧다.

**Multiple Processor Scheduling**

- 스케줄링은 CPU를 여러 개 사용할 수 있을 때 복잡해진다.

**Symmetric Multiprocessing(SMP)**

- 둘 이상의 프로세서

- 모든 프로세서는 운영체제에서 실행되며, 관리된다.

- 모든 프로세서는 공유 메모리를 사용하여 통신한다.

- 모든 프로세서가 메인 메모리, 네트워크, I/O 장치 등 하드웨어에 대한 엑세스 권한을 가진다.

- 프로세서는 동일한 프로세스를 실행하지 않는다.

- 각 프로세서는 각각 ready\_queue를 가지거나, 하나의 ready\_queue를 공유하여 사용할 수 있다.

**Asymmetric Multiprocessing**

- SMP가 가능했던 시기 이전의 방식이다.

- Master 프로세서만이 운영체제에서 실행되며, 관리된다.

- 프로세서들은 Master-Slave 관계

- Master 프로세서는 Slave 프로세서들에게 프로세스를 할당한다.

- Master 프로세서가 자료구조를 다룬다.

- Master 프로세서만이 프로세스 스케줄링, I/O, 등의 처리를 한다.

**Load Balancing**

- SMP 시스템의 모든 프로세서가 작업 부하를 고르게 받도록 분산시킨다.

- 각 프로세서가 개별적으로 ready\_queue를 가지는 시스템에서만 필요한 기법

- Push Migration : 특정 Task가 프로세서들의 부하를 주기적으로 체크하고, 고르게 분배한다.

- Pull Migration : 프로세서가 스스로 작업이 많은 프로세서로부터 프로세스를 가져온다.

**Process Affinity (친화성)**: CPU Cache와 프로세스간의 관계를 고려해서 성능 향상

- 만약 시스템에서 Process Migration을 허용해서 새로 스케줄링이 되어 다른 프로세스가 해당 CPU를 할당받게 되면, Cache 메모리의 내용도 비우고 새로운 프로세스의 정보로 채워야 한다. 이런 Migration에 대한 Cost가 높기 때문에 대부분의 SMP 시스템들은 프로세스의 Migration을 피하려 한다.

- 한 마디로 프로세스가 같은 프로세서에서 실행되도록 하는 방식이다.

- Soft Affinity(Migration 허용) : 동일한 프로세서에서 현재 실행중인 프로세슬르 유지하려고 하지만, 이를 100% 보장하지는 않음

- Hard Affinity(Migration 불가) : Migration될 수 없는 프로세서를 지정하는 System Call 제공

**Symmetric Multithreading (SMT)**

- Memory Stall : 메모리에 엑세스할 때 CPU와 직접 연결된 메모리는 자신의 register와 RAM이다. register는 CPU 내부에 존재하며, 접근이 빠르지만, RAM으로의 접근은 조금 느리다. 그 이유는 Memory Bus를 통해 데이터가 전달되기 때문에, 메모리로부터 데이터를 기다리는 현상이 생기게 되는데, 이를 Memory Stall이라고 한다.

 예를 들면, Cache Miss가 발생할 경우, 데이터가 사용가능한 상태로 될 때까지 기다리는 데에 많은 시간을 들인다. 이 문제를 해결하기 위해 HW 설계자는 2개 이상의 HW Thread가 각 Core에 할당되는 멀티 프로세서 코어를 개발했다. intel에서는 **Hyper Threading**이라는 이름으로 다룬다.

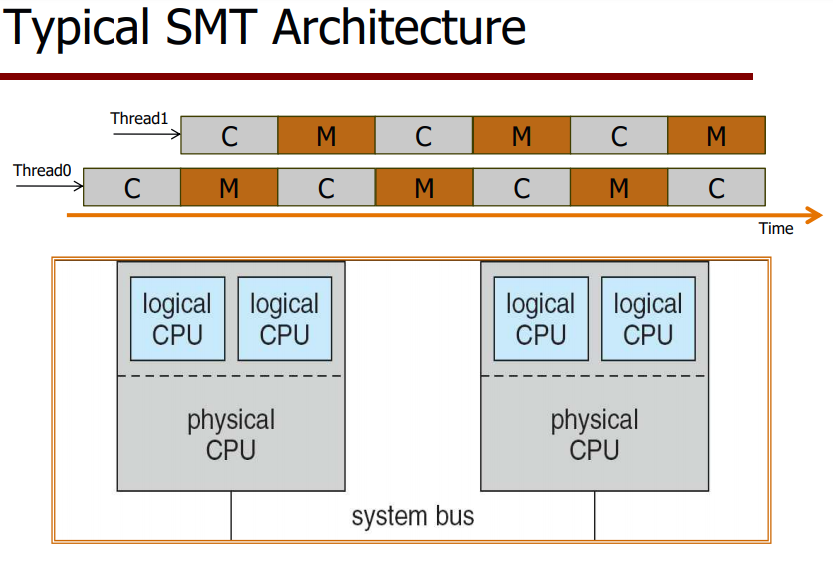
- OS 관점에서, 각 HW Thread는 SW Thread를 실행할 수 있는 논리적 프로세서로 정의한다.

**Memory Stall 문제**

텍스트, 시계, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**SMT**



**MultiCore Processor**

- 최근 경향은 동일한 물리적 칩에 여러 개의 코어를 배치하는 것이다. 이를 MultiCore Processor라고 한다.

- 2 단계의 스케줄링이 필요하다.

Level 1. SW Thread를 HW Thread로 매핑

Level 2. 각 코어가 실행할 HW Thread를 결정

**프로세스 동기화**

공유 메모리에 동시 접근하면 문제가 생긴다.

다음 코드를 보자.

텍스트이(가) 표시된 사진

자동 생성된 설명

counter++과  counter--의 구현이 다음과 같을 것이다

counter++

register1 = counter

register1 = register1 + 1

counter = register1

counter--

register2 = counter

register2 = register1 - 1

counter = register2

이제 다음과 같은 순서로 실행이 된다고 생각해 보자.

counter는 5로 시작한다.

텍스트, 병, 실내이(가) 표시된 사진

자동 생성된 설명

결과는 counter가 5여야 하지만, 각각 4, 6이 된다.

**Race Condition**

- 여러 프로세스가 공유 데이터를 동시에 엑세스하고 조작하는 상황. 공유 데이터의 최종적인 값이 마지막으로 완료되는 프로세스에 의해 결정되는 상황.

- 이 때, 프로세스 동기화가 필요하다.

- 위에서 살펴본 ++, --연산은 Atomic하게 수행되어야 한다.

Atomic Operation : 프로세스가 수행 중에 중단되지 않고 완료되는 동작

**Critical Section**

- N개의 프로세스들이 공유 데이터를 사용하기 위해 경쟁하는 상황을 해결하기 위한 동기화 방법

- 각 프로세스는 공유 데이터에 엑세스하게 해 주는 Critical Section이라는 코드를 가진다.

- 한 프로세스가 Critical Section에서 실행 중일 때, 해당 코드를 다른 프로세스가 실행하는 것을 허용하지 않는 방식.

**Critical Section의 구현 방법**

- 동기화 하드웨어 사용

- TestAndSet 명령

- Swap 명령

- Semaphore 사용

- High-Level 동기화 구성을 이용

- Monitor

**크리티컬 섹션의 속성**

- Mutual Exclusion(상호 배제) : 프로세스 Pi가 어떤 크리티컬 섹션 안에서 실행중일 때, 다른 프로세스들이 해당 크리티컬 섹션에서 실행될 수 없다.

- Progress(진행) : 크리티컬 섹션에 진입한 프로세스가 없는 상태에서, 진입하려는 프로세스가 여러 개 있다면 어느 것이 들어갈지를 결정해 준다.

- Bounded Waiting(한정 대기) : Thread의 Starvation상태를 막기 위해 크리티컬 섹션에 대한 접근 대기 시간은 한정되어야 한다.

**크리티컬 섹션 구현 알고리즘**

기본적으로 다음과 같은 구조를 가진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Entry / Exit을 통해 크리티컬 섹션에 진입하고 나올수 있다.

**알고리즘 1.**

텍스트이(가) 표시된 사진

자동 생성된 설명

프로세스가 위 코드에 접근했을 때, turn이 i가 아니면 대기하고, i가 되면 크리티컬 섹션에 진입할 수 있다.

처리가 끝나면 turn을 j로 바꾸면서 나온다.

이렇게 한 번에 하나의 Thread만 접근하게 되어 Mutual exclusion은 성립하지만, 다음과 같은 상태일 때 문제가 발생한다.

1. turn이 i이고, Thread i가 크리티컬 섹션을 제외한 나머지 코드를 실행중

2. Thread j가 크리티컬 섹션에 들어가려고 한다면 turn이 여전히 i이기 때문에 크리티컬 섹션에 접근 불가

그래서 Progress가 성립되지 않는다.

**알고리즘 2.**

텍스트이(가) 표시된 사진

자동 생성된 설명

이러면 한 상황에 한 스레드만 접근이 되지만,

1. flag[0] = true

2. 컨텍스트 스위칭 0 to 1

3. falg[1] = true

이 순서가 되면 둘다 수행이 되지 않는다.

**피터슨 알고리즘**

텍스트이(가) 표시된 사진

자동 생성된 설명

위에서 살펴 본 1, 2 알고리즘을 결합

- flag : 크리티컬 섹션 진입 의사를 설정. 일단 상대방이 크리티컬 섹션 진입 의사가 없다면 현재 프로세스를 크리티컬 섹션에 진입시킴

- turn : race condition에 의해, turn 설정하는 코드가 늦게 수행된 프로세스가 상대방에게 기회를 양보하도록 함.

많은 시스템에서는 하드웨어에서 크리티컬 섹션에 대한 동기화를 지원한다.

- Uni-Processors : 인터럽드를 비활성화할 수 있음

- 현재 실행되는 코드가 선점 없이 실행될 수 있다.

- 멀티 프로세서 시스템에서 비효율적

- 현재 대부분은 Atomic 하드웨어 명령을 제공한다(Atomin = Non interruptable).

- TestAndSet

- CompareAndSwap

**TestAndSet를 이용한 크리티컬 섹션 구현**

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 구현하면, TestAndSet이 Atomic 연산이므로, 이미 크리티컬 섹션에 진입한 프로세스가 존재하게 되면 해당 프로세스를 제외한 다른 프로세스들은 while에서 대기하게 된다.

**CompareAndSwap**

void CompareAndSwap(int \*value, int expected, int new\_value)

{

int temp = \*value;

if (\*value == expected)

\*value = new\_value;

return temp;

}

do {

while (CompareAndSwap(&lock, 0, 1) != 0); // 이미 다른 프로세스가 lock을 가지고 있다면 대기

/\* Critical Section \*/

lock = 0; // lock 해제

} while(true)

이렇게 하면 CompareAndSwap이 Atomic 연산이므로 lock 가지는 프로세스는 언제나 하나뿐이게 된다.

Semaphores : busy waiting을 피하는 동기화 방법. 커널 오브젝트를 생성해서 커널에서 관리되도록 구현.

- wait(), signal()로 진입, 진입해제를 함

**세마포어 및 wait(), signal()의 대략적인 구현**

- 실제 구현은 하드웨어 동기화 혹은 Interrupt Disable/Enable을 통해 함수 내에서는 컨텍스트 스위칭을 방지하는 방법으로 진행할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

block()을 하게되면 현재 프로세스를 wait\_queue에 넣는다.

wakeup()을 하게 되면 현재 프로세스를 ready\_queue에 넣는다.

**세마포어의 사용 방식은 2가지 존재.**

- Counting : 정수값 제한 없음. 동시에 진행 가능한 프로세스(혹은 스레드)의 수를 설정할 수 있음.

- Binary(Mutex) : 0, 1 두개의 값만 존재. 바이너리 세마포어를 뮤텍스라고 함.

- 뮤텍스를 사용하게 되면 이는 곧 임계구역에 한 번에 한 프로세스(혹은 스레드)만 접근할 수 있게 되는 것을 의미한다.

busy waiting : 프로세스나 스레드가 lock을 획득하기 위한 동기화 상황에서 일반적으로 반복문을 통해 구현되는데, lock을 얻기 위해 이런 식으로 CPU를 낭비하는 것은 비효율적이다.

Spin lock : busy waiting 개념을 이용하여 구현되며, 컨텍스트 스위칭 비용보다 busy waiting으로 대기하는 시간이 더 부하가 적다고 생각될 때 사용됨.

테이블이(가) 표시된 사진

자동 생성된 설명

**DeadLock**

- 2개 이상의 프로세스가 대기 상태의 프로세스 중 하나에서만 발생할 수 있는 Event를 영원히 대기하는 상황

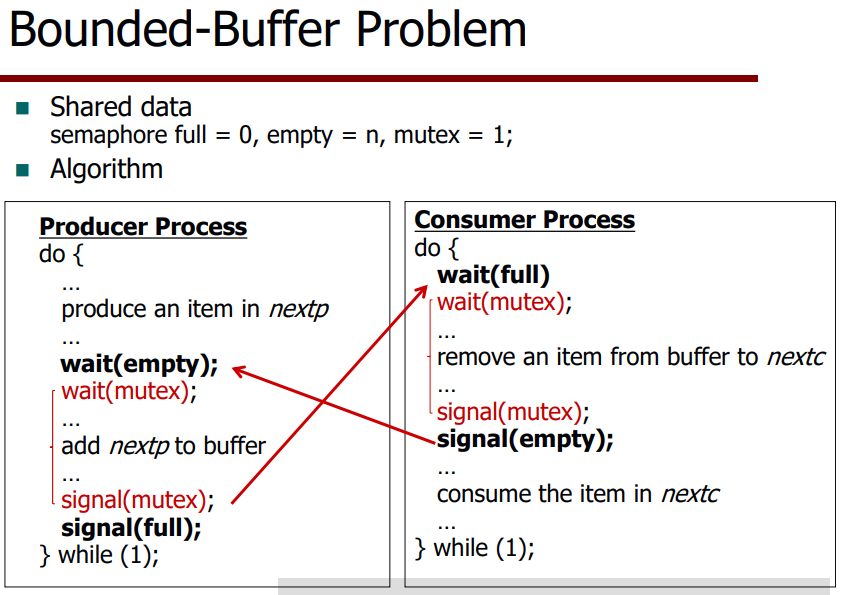
**Starvation**

- 무한하게 Block 상태로 남는 것. 세마포어의 Block상태 프로세스 리스트에서 프로세스를 영원히 제거하지 못하는 상황

**동기화 문제 해결 (당연한 사실이다)**

- Bounded-Buffer Problem : 생산자-소비자 개념이 등장. 동기화를 사용하지 않을 경우, 생산자는 버퍼에 데이터를 넣고, 소비자는 버퍼의 데이터를 가져온다. 이 때, 저장할 공간이 없는 문제, 가져올 데이터가 없는 문제가 발생.

- 데이터가 버퍼에 꽉 채워져 있는지, 데이터가 하나도 없는지 등을 세마포어로 설정



**Readers-Writers Problem**

- 여러 명의 독자 및 저자들이 있고, 하나의 저장 공간을 공유하며 이를 접근할 때 발생하는 문제.

1. 여러 명의 독자가 동시에 데이터를 읽는 것이 가능

2. 한 저자가 공유 공간에 데이터를 쓰고 있는 동안에는 그 저자만 접근이 가능.

\* 독자가 한 명이라도 읽고 있는 중에는 저자가 데이터를 쓸 수 없음

세마포어로 다음과 같이 구현 가능하다(출처 : [위키백과](https://ko.wikipedia.org/wiki/%EB%8F%85%EC%9E%90-%EC%A0%80%EC%9E%90_%EB%AC%B8%EC%A0%9C))

* 변수
  + readcount : 현재 버퍼에 접근 중인 독자의 수를 나타낸다. (초기값=0)
  + wrt : 저자들 사이의 관계를 통제한다. 즉, 동기화한다. (초기값=1)
  + mutex : readcount와 wrt에 접근하는 것이 [원자적](https://ko.wikipedia.org/wiki/%EC%9B%90%EC%9E%90%EC%A0%81)으로 수행될 수 있도록 하는 [세마포어](https://ko.wikipedia.org/wiki/%EC%84%B8%EB%A7%88%ED%8F%AC%EC%96%B4) (초기값=1)
* 저자 프로세스

wait(wrt); // 임계구역에 들어가기 위해 허가가 나기를 기다린다.

...

쓰기 작업 수행

...

signal(wrt); // 임계구역에서 빠져나왔음을 알린다.

* 독자 프로세스

wait(mutex);

readcount++; // 독자 수 1 증가

if readcount = 1

wait(wrt); // 쓰고 있는 저자가 없을 때까지 기다린다.

signal(mutex);

...

읽기 작업 수행

...

wait(mutex);

readcount--; // 독자 수 1 감소

if readcount = 0

signal(wrt); // 독자가 없다면 이를 알린다.

signal(mutex);

**Dining-Philosophers(식사하는 철학자) Problem**

- Dead Lock(교착 상태)을 설명하기 위한 내용이다.

- N명의 철학자가 원탁에 앉아서 식사를 한다. 철학자들 사이에는 포크가 하나씩 놓여 있고, 다음의 규칙을 순서대로 수행하며 식사를 한다.

1. 왼쪽 포크가 사용 가능해질 때까지 대기한다. 사용 가능해지면 집는다.

2. 오른쪽 포크가 사용 가능해질 때까지 대기한다. 사용 가능해지면 집는다.

3. 양쪽의 포크를 집으면 정해진 시간(CPU Time Slice)만큼 식사를 한다.

4. 오른쪽 포크를 내려놓는다.

5. 왼쪽 포크를 내려놓는다.

6. 1번을 수행한다.

- 만일 모든 철학자가 동시에 자신의 왼쪽 포크를 잡는다면 모든 철학자들이 자신의 오른쪽 포크를 영원이 기다려야 한다.

이를 Deadlock이라고 하는데, Deadlock이 일어나는 필요 조건 중 하나를 깨버리면 된다.

필요조건은 다음과 같다.

1. Mutual Exclusion(상호 배제) : 프로세스들이 공유 자원을 동시에 쓸 수 없는 상황

2. Hold and Wait(보유 및 대기) : 공유 자원을 점유한 상태에서 다른 자원을 기다리는 상황. 점유한 자원을 또다른 프로세스가 대기하게 되고, 최악의 경우 Circular Wait(순환성 대기) 상황에 빠진다.

3. No Preemption(선점 불가) : 프로세스들이 서로 공유 자원을 뺏을 수 없다.

**Monitor**

- 동기화 기법 중 하나. 어떤 공유 데이터에 모니터를 지정해 놓으면, 프로세스는 그 데이터에 접근하기 위해 모니터에 진입해야 한다. 그리고 모니터 내부에 들어간 프로세스에게만 공유 데이터를 접근할 수 있는 기능을 제공한다.

- 이미 모니터에 프로세스가 존재하면 대기 큐가 존재해서 해당 대기 큐에서 대기한다.

뮤텍스와의 차이 : 뮤텍스는 프로세스 간의 동기화가 가능하고, 모니터는 한 번에 한 프로세스만 진입 가능하므로 스레드 간의 동기화라고 할 수 있다.

Java에서는 synchronized라는 명령어로 이를 제공한다.

## [OS](https://teraphonia.tistory.com/category/OS)

[**[OS] System call, Interrupt, Trap, 등**](https://teraphonia.tistory.com/790)

2018. 9. 24. 23:38

Interrupt

- 하드웨어(I/O 포트, 그래픽 카드, 하드 디스크, 등)에서 발생.  처리가 필요할 경우에 CPU에게 그 상황을 처리할 수 있도록 알려주는 것

- 인터럽트가 발생하면 실행 중인 코드를 중단하고, 인터럽트 서비스 루틴을 실행하여 인터럽트를 처리함.

- 인터럽트가 발생하면 해당 프로세스는 ready queue에 들어가고 다시 스케줄링됨.

- I/O 혹은 Event가 발생하면 해당 작업이 완료될 때까지 wait queue에 들어가게 된다(block 상태). 완료되면 다시 ready\_queue로 들어감.

System Call

- 커널 기능을 커널에 요청하는 방식. Trap을 통해서 실제로 커널에 정보가 전달된다. System Call은 그 Interface라고 할 수 있음.

Exception

- Trap

- Register와 스택 포인터가 저장되고, Context Switching이 일어나고, 실행이 재개될 수 있다는 면에서 인터럽트와 굉장히 흡사하다. 그래서 이를 CPU 내부 인터럽트라고도 한다.

- User Mode -> Kernel Mode 전환해 준다.

- System Call은 주변 장치, 메모리 관리, 등의 커널 모드에서만 조작 가능한 코드를 실행하기 위해 Trap을 통해 구현됨.

- Fault

- 0으로 나누기, 잘못된 메모리 엑세스, 등으로 발생

- abort

- 하드웨어 고장, 등

프로그램이 실행되려면 메모리로 가져와야 한다.

Input queue(job queue) : 프로그램을 실행하기 위해 메모리로 가져오는 것을 기다리는 프로세스 집합 - Long-term 스케줄러의 역할.

유저 프로그램은 실행되기 전에 다음 과정을 거친다. 자세한 내용은 나중에 적어두어야겠다.

- Instruction Fetch : 명령어를 메모리로부터 가져온다. 메모리 -> CPU Cache

- Instruction Decode : 가져온 명령어에서 PC가 가리키는 주소값을 읽고 다음 명령어가 있는 곳으로 PC값(가상 메모리 주소)을 갱신한다.

- Operand Fetch : 명령어에서 사용되는 피연산자를 읽는다.

- Instruction 실행 : 명령어를 실행한다.

- 명령 실행 결과를 메모리에 저장

**메모리 접근 속도**

- 메인 메모리와 Register는 CPU에서 직접적으로 엑세스할 수 있다.

- Register에 엑세스하는 것은 CPU Clock의 1 cycle 또는 그 이하의 속도다.

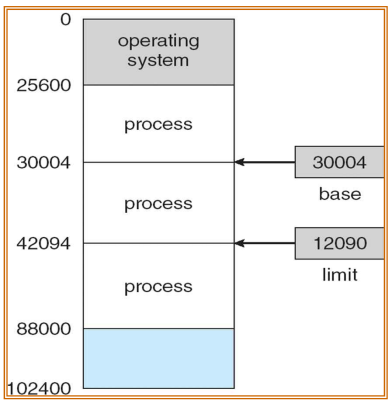
- 메인 메모리 엑세스는 그보단 많은 사이클을 요한다. 그렇기 때문에 이전에 언급한 Memory Stall 현상(Memory에서 Bus를 통해 CPU로 오기까지의 대기시간 발생)이 발생한다.

- Cache는 메인 메모리와 CPU Register 사이에 존재한다. 일반적으로 CPU 칩에 있다.

**메모리 보호 HW**

- 올바른 작동을 보장하기 위한 메모리 보호

- base / limit 레지스터는 논리 주소 공간을 정의한다.



Logical Address : 프로세스마다 독립적으로 가지는 주소 공간(Virtual Address이다). 0번지부터 시작한다. CPU가 인식하는 주소 체계가 이것이다.

Physical Address : 물리적인 메모리 주소로, 실제 하드웨어의 메모리 주소.

Symbolic Address : 프로그래머가 변수 이름을 선언하고 이 변수 이름을 통해 주소에 접근하는 것.

**주소 바인딩**

- Instruction 및 Data를 메모리 주소에 바인딩하는 것

- Ex) Symbolic 주소를 실제 주소로 바인딩해 주는 것.

**주소 바인딩은 다음 세 단계로 이루어진다.**

- Compile Time

- 주소 변환이 컴파일 시에 이루어진다.

- 물리 메모리가 충분히 비었어도 이미 주소가 결정되어 변경할 수 없다.(비효율적)

- 주소 변환을 하려면 다시 컴파일해야 한다.

- 이를 Absolute Code라고 한다.

- Load Time

- 컴파일 시 메모리 위치를 알 수 없는 경우, 일단 Relocatable code(재배치 가능한 코드)로 생성해야 한다.

- Symbol과 실제 주소의 바인딩은 프로그램이 메인 메모리로 적재될 때 이루어진다.

-Execution Time

- 바인딩이 런타임 이후에도 진행된다.

- CPU가 주소를 참조할 때마다 바인딩 상태를 점검해야 한다.

- MMU이라는 하드웨어 지원이 필요하다.

- MMU(Memory Management Unit) : 논리 주소를 물리 주소로 매핑해 주는 하드웨어이며, relocation register와 limit register를 이용하여 매핑을 수행한다.

- 유저 프로그램은 논리 주소를 다루며, 실제 물리 주소를 고려하지 않는다.

- relocation register : 물리 메모리 주소의 최소값

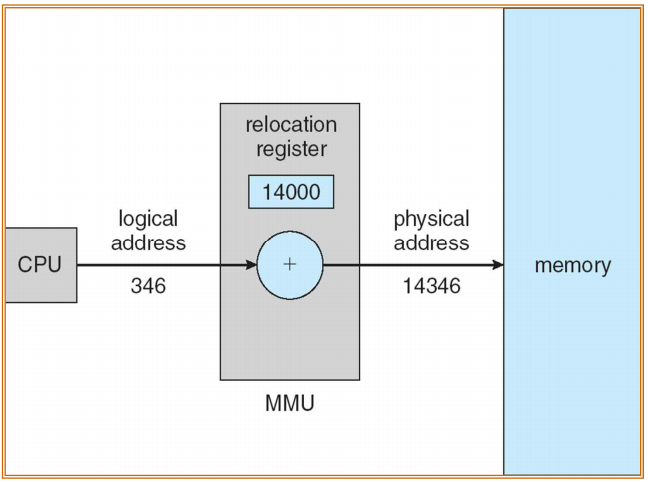
- limit register : 논리 주소의 범위

- CPU가 사용한 값에 relocation register의 값을 주소에 더해줌으로써 실제 물리 메모리 주소값으로 변환한다.

- 범위를 넘어간다면 Trap(Software interrupt)을 통해 오류를 알린다.

- 대부분의 운영체제가 사용하는 방법.

Dynamic relocation



**컴파일부터 프로그램 실행까지의 과정**

1. Compile : 각각의 소스코드를 컴파일하면 Object(\*.o) 파일이 생성된다.

2. Link : Object 파일을 하나의 Object 파일로 묶어주는 것. 이것을 Executable File이라고 한다(ex \*.exe파일).

3. Loader

- Executable File을 읽음

- 메인 메모리에 적재

- 프로세스 생성

- OS의 기능

**Dynamic Loading VS Dynamic Linking**

- Dynamic Loading

- 루틴이 호출되는 시점에 로드하는 방식

- 메모리 공간 사용 효율 증대 : 사용되지 않는 루틴은 로드되지 않는다.

- 사용 빈도가 낮은데 코드량이 많은 루틴이 필요할 때 유용.

- 운영체제가 이런 기능을 지원하지는 않고, 프로그래머에 의해 구현된다. (Program API: dlopen(), dlsym(), dlclose() etc.)

- Dynamic Linking(Ex DLL 파일 사용)

- 실제로 실행될 때까지 Linking이 지연된다(Dynamic loading + Auto linking).

- 컴파일을 통해 생성된 파일과 라이브러리 사이의 Linking을 실제로 수행되기 전까지 지연시킨다.

- 프로그램 실행 중에 라이브러리를 링크할 필요가 있을 때 사용된다.

- Stub이 포함되는데, 메모리에 존재하는 라이브러리 루틴을 찾는 데 사용되는 적은 양의 코드다. Stub은 자신을 루틴의 주소로 변경하고 실행한다.

- OS에서 지원하며, OS는 해당 루틴이 프로세스의 메모리 공간에 있는지 확인해야 한다.

- Shared Library에 유용 : 라이브러리 루틴을 여러 프로그램에 공유된다.

**Swapping**

- 프로세스는 일시적으로 메모리에서 디스크로 스왑되었다가 실행이 재게될 때 다시 메모리로 가져오게 될 수 있다.

- 반드시 보조메모리가 필요. 보통 디스크가 사용된다.

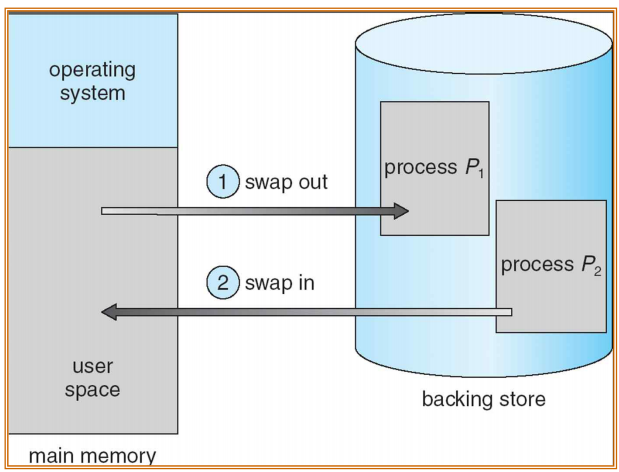
- CPU는 ready\_queue에서 프로세스를 가져와 CPU를 할당하고, CPU 스케줄러는 다음 프로세스를 선택할 때 Dispatcher를 호출하며, Dispatcher는 다음에 실행할 프로세스가 ready\_queue에 존재하는지 확인하고 없다면 디스크에서 가져온다.

- Swap 수행시간의 대부분은 메모리 전송 시간이다. 스왑에서 전송되는 메모리의 양에 비례한다.

- Roll out, Roll in

- 우선 순위 스케줄링에서 사용하는 방식

- 낮은 우선순위를 가진 프로세스가 스왑아웃되고, 높은 우선순위를 가진 프로세스가 로드되고 실행될 수 있도록 한다.



**Memory Allocation**

- Contiguous Allocation

- Multiple Partition Allocation

- 고정 크기 분할

- 가변 크기 분할

- Non-Contiguous Allocation

- Paging

- Segmentation

**Contiguous Allocation**

- 메인 메모리는 2개의 파티션으로 나뉘어진다.

- OS는 인터럽트 벡터와 함께 낮은 메모리 주소에 올라간다.

- 유저 프로세스들은 높은 메모리 주소에 올라간다.

- relocation register에 들어 있는 시작 주소 및 limit register에 들어 있는 메모리 범위를 이용해 프로세스들이 서로의 영역을 침범하지 못하도록 막는다.

- Multiple-Partition Allocation (고정 크기)

- 메모리를 고정된 크기의 파티션으로 분할한다.

- 각 파티션은 하나의 프로세스를 가진다.

- 파티션의 수에 따라 멀티프로그래밍의 정도가 결정된다.

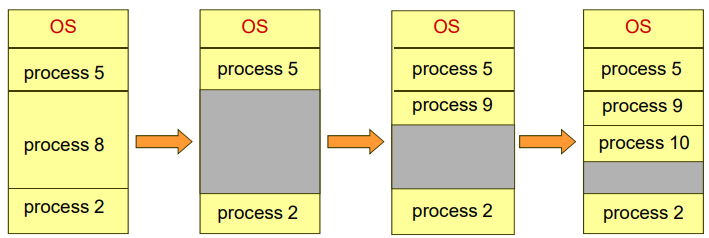
- IBM OS/360에서 사용되었지만 이제 사용되지 않는다.

- Multiple-Partition Allocation (가변 크기)

- Hole : Hole이라는 개념이 있는데, 사용 가능한 메모리 블럭을 의미한다. 다양한 크기의 Hole이 메모리 전체에 흩어져 있다.

- 프로세스가 도착하면 프로세스가 할당될 수 있는 충분한 크기의 Hole을 찾아 할당한다.

- OS는 Allocated-Partition과 Free-Partition(Hole)에 대한 정보를 유지한다.



**Dynamic Storage Allocation Problem**

- Hole list로부터 N 바이트 블록 요청을 어떻게 응답해줄 수 있는지를 결정하는 문제. 해결 방안 3가지 존재한다.

1. First-Fit : 요청을 만족시키는 충분히 큰 첫 번째 Hole을 할당해 준다.

2. Best-Fit : 요청을 만족시키는 충분히 큰 공간들 중에서 제일 작은 Hole을 할당해 준다.

3. Worst-Fit : 가장 큰 Hole을 할당해 준다.

당연히, 1, 2번은 3번보다 속도 및 메모리 사용 효율이 훨씬 낫다.

- 50% rule : N개의 메모리 블록이 존재할 때, 0.5N개의 메모리 블록은 Fragmentation으로 인해 손실된다.

**Fragmentation(단편화)**

- External Fragmentation(외부 단편화)

- 메모리의 전체적인 공간을 계산해 보면 분명히 프로세스가 할당될 크기의 빈 메모리 공간이 존재하지만, 연속적이지 않기 때문에 프로세스를 할당할 수 없는 경우.

- Compaction

- Compaction(비어 있는 Hole들을 하나로 합쳐 하나의 큰 Hole로 만들어주면서 단편화 영역을 한 쪽으로 몰아 옮겨 주는 작업.)을 진행해서 외부 단편화를 줄일 수 있다.

- 시스템은 Compaction 진행 중에 다른 작업을 할 수 없다.

- Compaction을 Garbage Collection이라고도 한다.

- Compaction은 각 프로세스의 주소들을 동적으로 relocation해야 하기 때문에 실행 부하가 있다.

- Interal Fragmentation(내부 단편화)

- 할당된 메모리 영역이 프로세스 크기보다 약간 큰 경우, 프로세스에서 사용할 수 없지만 할당된 메모리 공간이 존재하게 된다. 이를 내부 단편화라고 한다.

**Paging**

- 프로세스의 가상 주소 및 물리 주소 공간은 Non-Contiguous일 수 있다.

- Page라는 블록을 사용하여 메모리 할당을 하는 방법.

- 물리 메모리를 고정된 크기의 블록으로 나누는데, 이를 Frame이라고 한다. 크기는 2의 승수이며, 512(2^9) bytes와 8192(2^13) bytes

- 가상 주소 메모리를 같은 크기의 블록으로 나누는데, 이를 Page라고 부른다.

- 크기가 N 페이지인 프로그램을 실행한다면, N Free Frame을 찾고, 프로그램을 로드한다.

- 가상 주소를 물리 주소로 변경하기 위해 Page Table을 사용한다.

- 정해진 크기의 가상 주소와 물리 주소를 사용하며, 페이지 테이블을 사용한다. 이는 연속된 물리 메모리를 다루는 것이 아니기 때문에 때문에 외부 단편화 문제는 해결되지만 내부 단편화 문제는 여전히 존재하게 된다.

- 페이지 테이블은 32비트 기준 4GB에 해당하는 페이지 수를 가진다.

**주소 변환**

- 가상 주소(CPU가 인식하는 주소)는 다음으로 나눠진다.

- Page 번호 (p)

- 페이지 테이블에서 페이지의 index로 사용됨.

- 각 페이지 별로 프레임(물리 메모리)의 시작 주소를 가지고 있다.

- Page Offset (d) : 시작 주소와 이 값을 합치면 물리 메모리 주소가 된다.

페이지 크기가 2^n이고 가상 주소 공간이 2^m일 때, 가상 주소의 내용은 다음과 같다.

테이블이(가) 표시된 사진

자동 생성된 설명

1. 가상메모리르 4GB 사용. [2^32] => m = 32

2. 페이지 크기가 4KB. [4KB = 2^12] => n = 12

따라서 p = m - n = 32 - 12 = 20, 상위 20bit는 페이지 번호를, 하위 12bit는 offset을 나타낸다.

- 페이지 수는 4GB / 4KB로, 2^20개이다. 그러니 2^20개의 번호가 존재하게 된다.

- 페이지 내의 변위를 offset에 담는다. 한 페이지가 2^12이므로 0 ~ 2^12-1의 수가 들어갈 것.

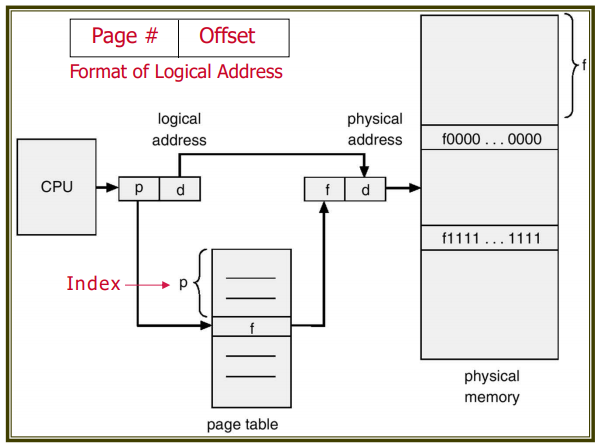
다음 그림을 보면 이해가 될 것이다.

**가상 메모리 -> 물리 메모리 변환 과정**

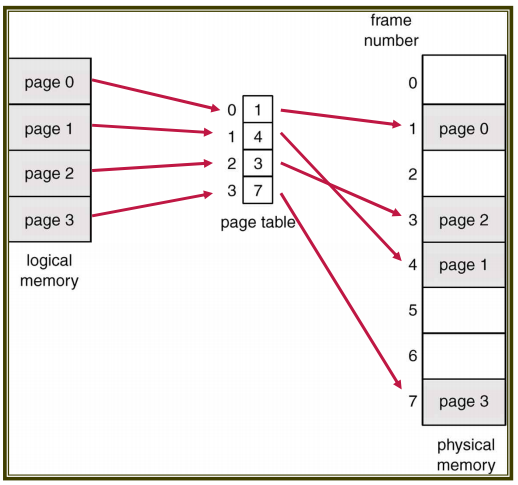
1. 가상 메모리 주소에서 Page 번호 및 Offset 추출.

2. Page 번호로 Page Table에서 해당 페이지를 찾아 프레임의 주소로 변환.

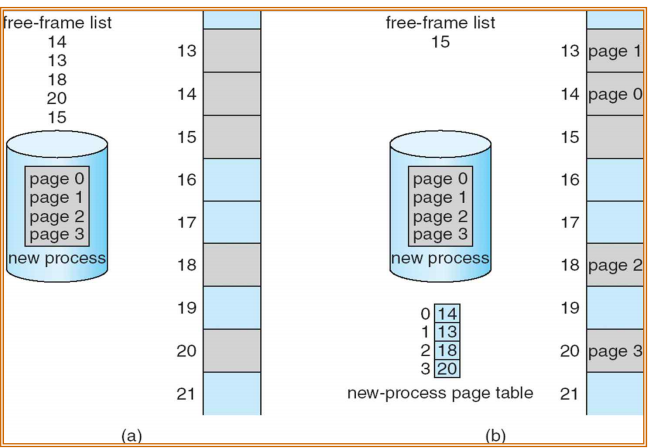
3. 프레임 주소 + Offset을 해서 실제 해당 물리 주소 조회



페이징



페이지 - 프레임 관계



**Page Table 구현**

- 페이지 테이블은 크기가 크기 때문에 레지스터에 저장되지 못하고 메인 메모리에 유지되며, 다음 레지스터들로 페이지 테이블을 다루도록 한다.

- PTBR(Page Table Base Register)로 페이지 테이블을 가리킴.

- PTLR(Page Table Length Register)로 페이지 테이블의 크기를 나타냄.

- 이 방식은, 물리 메모리 엑세스를 2번 거쳐야 한다.

1. 메인 메모리의 PCB에 존재하는 페이지 테이블 자체에 엑세스.

2. 1번을 통해 물리 메모리 주소를 추출하고 해당 물리 메모리 엑세스

이 과정이 느리기 때문에 이것을 해결하기 위해 TLB(Translation Look-Aside Buffers)라고 하는 소형 하드웨어 캐시가 사용된다.

**TLB**

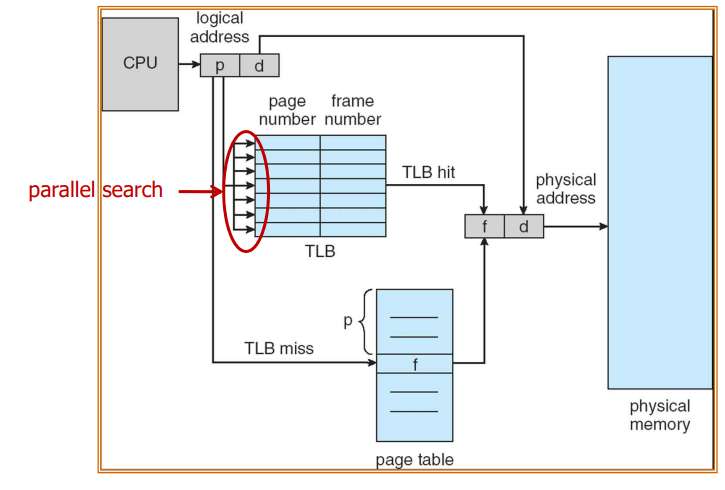
- CPU와 CPU 캐시 사이, CPU 캐시와 메인 메모리 사이 등에 존재.

- Key / Value 형태의 Entry로 테이블이 구성됨

- Key에는 Page 번호, Value에는 Frame 번호가 저장됨.

- 하드웨어 성능상 페이지 테이블의 일부만 저장해둘 수 있어서, 페이지 교체 알고리즘이 적용된다.

- TLB를 거치면 TLB 1번 메인 메모리에 1번 접근하는 것으로 끝난다.



**Memory Protection**

- protection bit

- 페이지에 대해 접근 권한 제어

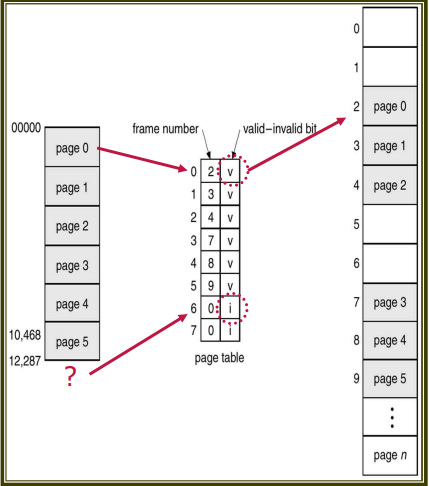
- code 영역 : 수정을 막기 위해 read only 부여

- data / stack 영역 : 수정 가능하도록 read / write 권한 부여

- 페이지 테이블 엔트리에는 각각 Valid-Invalid bit를 붙인다.

- Valid :  페이지가 프로세스의 가상 주소 공간에 있음을 나타낸다.

- Invalid : 페이지가 프로세스의 가상 주소 공간에 없음을 나타낸다.



**페이지 테이블의 구조**

- Hierarchical Paging

- 가상 주소 공간을 N개의 페이지 테이블로 나눈다.

- Example : 2-Level 페이지 테이블

- 가상 주소(32bit machine / 4KB page size)를 다음과 같이 분류한다.

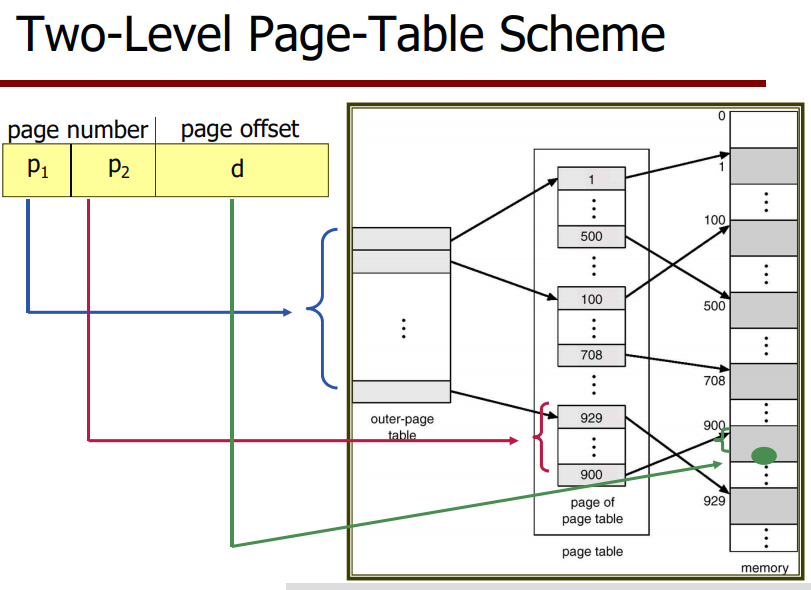
- 페이지 번호 : 20bits

- 페이지 offset : 12bits

- 페이지 테이블의 크기가 크기 때문에, 테이블을 하나 더 둔다.

- 2^20을 둘로 나눠서 2^10 / 2^10으로, 하나는 outter page table, 하나는 page table이다.

- 이렇게 나눠서, 실제로 해당 페이지가 사용될 때마다 나눠진 구역을 사용하기 때문에 사용되지 않은 부분의 페이지 테이블은 생성되지 않는다.



- Hashed Page Table

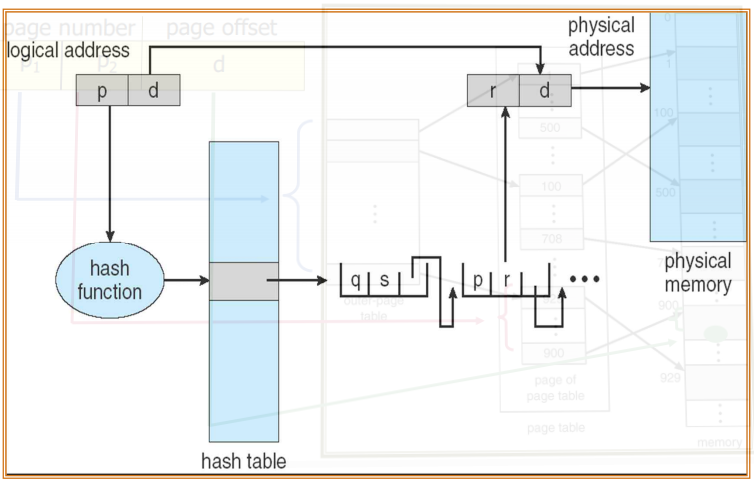
- 페이지 테이블을 Chaining 방식의 Hash table로 구현한 것.

- 32bit보다 큰 주소 체계에서 사용됨.

- 페이지 번호가 hash 함수를 거쳐 페이지 테이블에 매핑됨.

- Chaining 방식이기 때문에, 데이터가 중복되어 들어오면 linked list에 추가된다.

- 페이지 번호는 이 list에서 비교하며 검색한다. 해당 페이지 번호에 해당하는 엔트리를 찾으면 물리 메모리를 추출한다.



**Segmentation**

- 페이징에서는 같은 크기의 블록으로 나누었지만, 세그멘테이션은 가상 메모리를 서로 다른 크기로 나누는 기법이다.

- 크기가 서로 다르기 때문에, 페이징을 할 때처럼 메모리를 미리 분할해 둘 수 없다.

- 하나의 프로그램은 세그먼트의 집합이다. 세그먼트는 다음과 같은 논리적 요소이다.

- main program

- procedure

- function

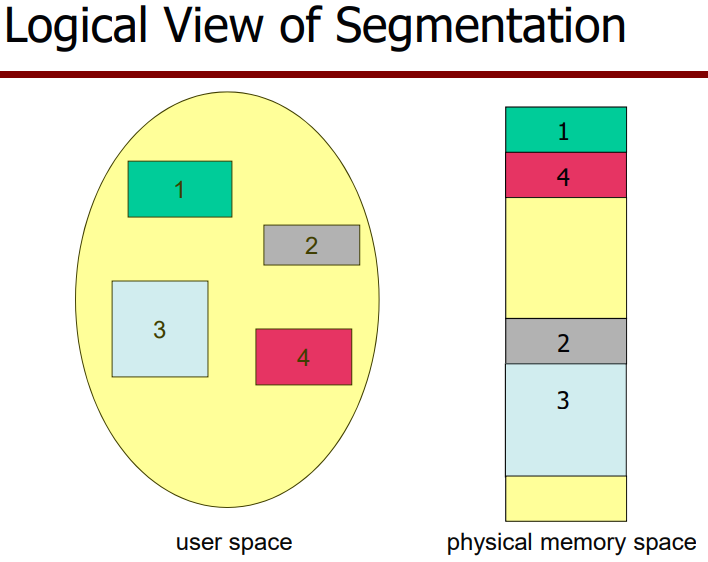
- object

- local variables, global variables

- common block

- stack

- arrays



**세그먼테이션 구조**

- 가상 주소는 <세그먼트 번호, offset> 이렇게 두 개 데이터의 튜플로 구성됨.

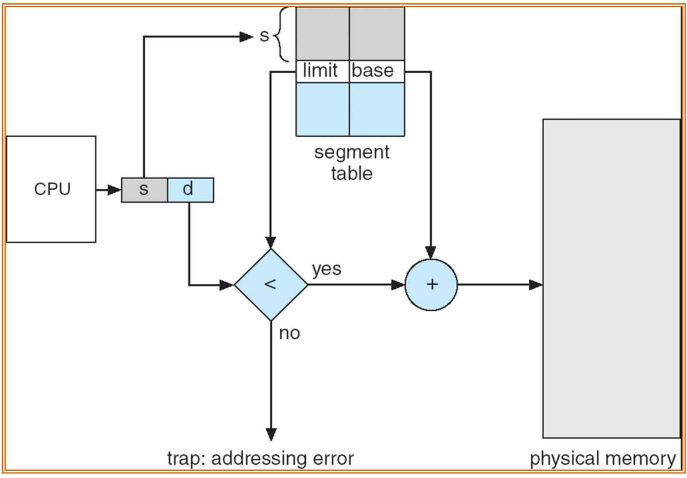
- Segment table : 세그먼트의 base address와 limit address를 담고 있다.

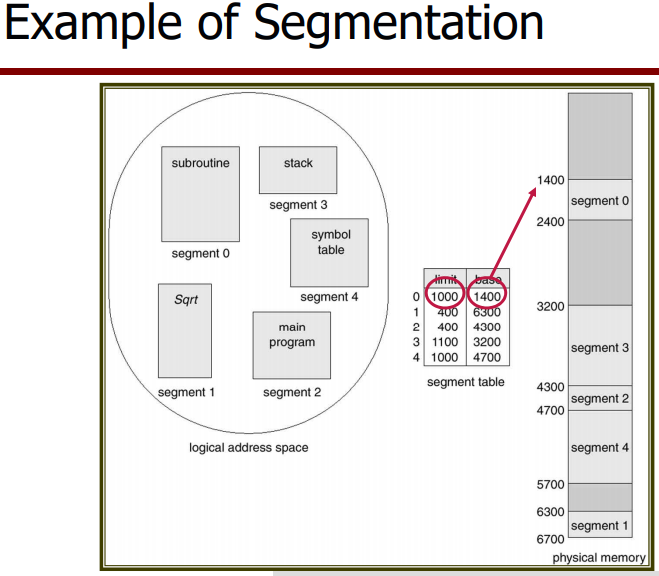
- Offset 단위로 관리된다. 시작 주소와 Offset으로 실제 물리 메모리 주소를 추출할 수 있다.

- Segment-Table Base Register (STBR) : 세그먼트 테이블의 메모리 주소를 가리킴

- Segment-Table Length Register (STLR) : 프로그램에서 사용하는 세그먼트들의 수를 나타냄

세그먼트 번호 s는 s < STLR일 때 유효하다.





**세그먼테이션 설계**

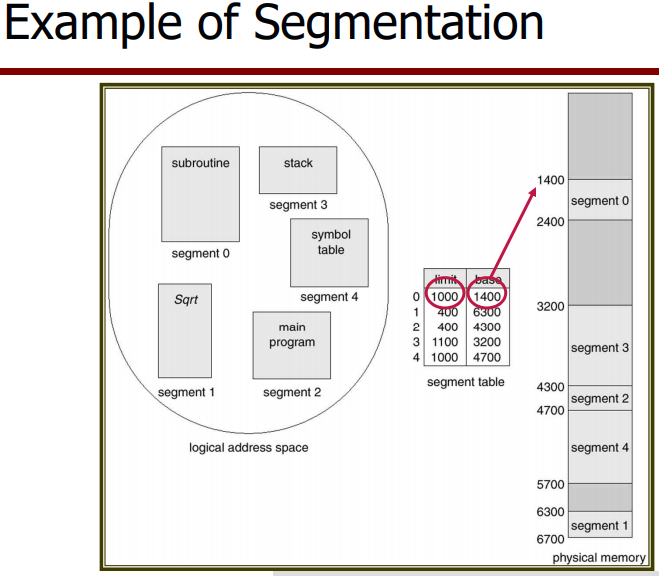
- Relocation

- dynamic

- by segment table

- 공유

- 엔트리에서 같은 세그먼트 주소값을 가지도록 하는 것으로 간단하게 공유가 가능하다.



- Allocation

- First fit / Best fit

- External Fragmentation(외부 단편화)

- Protection

- 세그먼트 테이블의 각 엔트리가 연결된다.

- validation bit = 0이면 유효하지 않은 세그먼트다.

- read / write / execute 권한 부여 가능

**하지만 Hole 설명에서처럼 Dynamic-Storage-Allocation 문제가 발생한다.**

**장점 : 세그멘테이션을 이용하면 프로세스의 Code / Data / Stack 등을 종류별로 분류하여 저장하므로, 공유 및 보호가 간단하다.**

**단점 : 메모리 크기가 고르지 않으므로 외부 단편화가 발생할 수 있다.**

결론적으로 페이징을 쓰면 내부 단편화 발생,

세그먼테이션을 쓰면 외부 단편화가 발생한다.

이를 해결하기 위해 Intel386에서는 Segmentation과 Paging을 결합한 방식의 CPU가 개발되었다.

- 세그먼트 각각이 페이지 테이블을 가지고 있음.

- 세그먼트 테이블의 엔트리의 base가 페이지 테이블을, limit이 페이지 수를 나타냄

**이렇게 하면 메모리 사용 효율은 올라가지만 결국 물리 메모리를 3번 참조해야 함**

1. 세그먼트 테이블 참조

2. 페이지 테이블 참조

3. 추출된 물리메모리 참조

**그래서 속도가 느려진다.**

**exit**

- 종료 처리 진행 후 프로세스 종료

- atexit() 함수 호출하여 exit handler에 등록된 함수 호출

- 전역 or 정적 객체 소멸자 호출

**abort**

- 프로세스 즉시 종료

- atexit() 호출하지 않음

- 소멸자 호출되지 않음

- 잘못된 기능을 수행하고 있어 더 진행되면 위험할 경우 abort 사용

**프로세스가 사용하는 실제 메모리 공간**

- 일반적으로 0 ~ max의 연속된 메모리. 물리 메모리는 연속적이지 않을 수 있다.

- MMU가 Page를 Frame으로 매핑한다.

**Copy On Write**

- 부모 프로세스와 자식 프로세스가 같은 페이지를 공유하는 것이다.

- 두 프로세스 중 하나가 공유 페이지를 수정하게 되면, 복사된 페이지를 쓰게 한다.

- 이 때 어떤 free page를 할당할지 결정하는 게 중요한데, OS는 이 free page들의 pool을 제공하는데 이 때 사용되는 기법을 Zero-Fill-On-Demand라 하며, 다음과 같은 경우에 갱신된다.

- 프로세스들이 Stack / Heap을 확장할 때

- Copy On Write가 수행될 때

**Demand Paging(요구 페이징)**

- 물리 메모리와 Swap Device 사이의 데이터 전송이 최소한으로 발생하도록 하기 위함.

- 스와핑을 사용하는 페이징 시스템. 전체 프로세스를 메모리에 스와핑하지 않고, 필요한 페이지만 스와핑해서 메모리에 올린다. 이를 Lazy Swapper 또는 Pager라고 한다.

- 극단적인 경우, 메모리에 페이지를 올리지 않고 프로세스를 실행시킬 수도 있다. 이 방식을 Pure Demand Paging이라 한다. 요구되기 전까지는 페이지를 메모리에 올리지 않는다.

- Demand Paging의 장점

- I/O가 적게 발생

- 적은 메모리

- 빠른 응답

- 더 많은 유저수용

- Demand Paging 단계

1. Page가 필요하다면 그것을 참조한다.

2. invalid reference라면 abort, 메모리에 없으면 메모리에 올린다.

**valid-invalid bit**

- Page table의 각 Entry에는 valid-invalid bit가 존재.

0 : not in memory

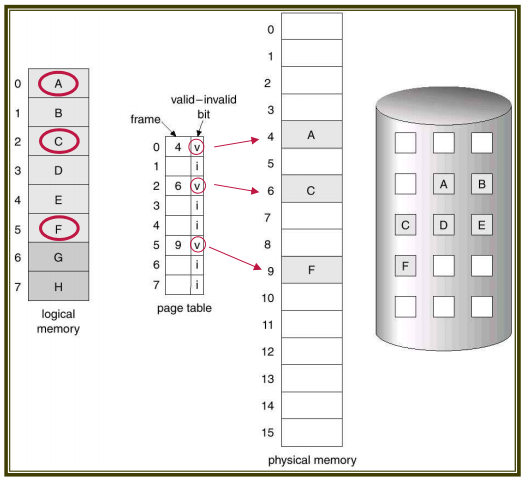
1 : in memory

- 처음엔 모두 0으로 설정됨

- 페이지를 찾을 때 valid bit가 0이면 Page-Fault 발생. 이 경우 메모리에 가져온다.

테이블이(가) 표시된 사진

자동 생성된 설명



**Page Fault**

- 가상 메모리 주소가 가리키는 Page를 찾았을 때, Frame 정보가 없어서 인터럽트를 발생시켜 운영체제에 알려주는 것.

1. Trap을 발생시켜 OS에 알린다.

2. OS는 해당 프로세스의 PCB에 있는 Page Table을 확인하여 다음을 결정한다.

- 아예 잘못된 참조일 경우 => abort

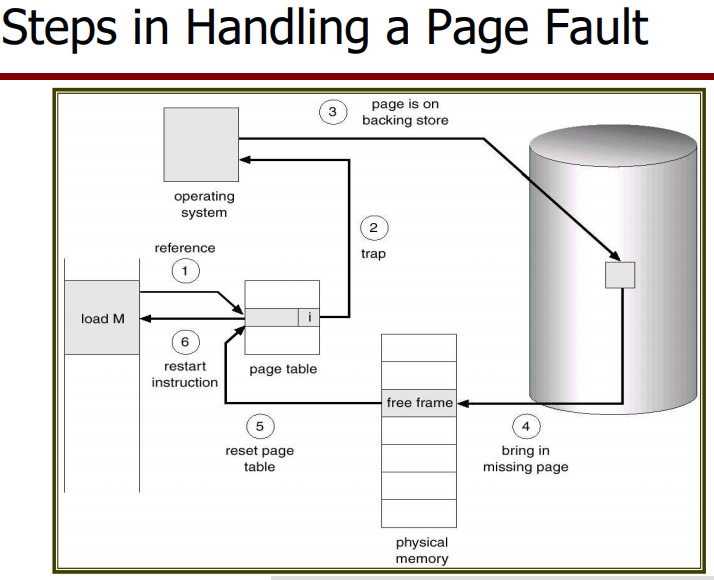
- 메모리에만 존재하지 않는 경우(valid-invalid bit가 0인 경우)

3. Free Frame을 찾는다.

4. 해당 페이지를 Frame에 넣는다.

5. 테이블을 갱신하고 해당 페이지의 validation bit를 1로 설정한다.

6. 다시 해당 Instruction을 수행



위의 Page Fault 과정 6번에서 Instruction을 재시작하는 것에 대한 문제가 있음

- Block Move : Block이 겹치는 경우가 있음. 이에 대해서는 나중에 다시 포스팅

**Demand paging의 성능은 다음과 같이 측정한다.**

- Page Fault Rate : 0 <= p <= 1.0

- p = 0이면 Page Fault가 없다는 것이고, 1이면 항상 fault가 발생한다는 것이라 할 때,

- Effective Access Time(EAT:유효 엑세스 시간)

- EAT = (1 - p) \* 메모리 엑세스 시간 + p \* (Page Fault 처리시간 + 페이지 스왑(in/out) 시간 + Instruction 재시작 시간)

다음 예시를 보면 이해가 잘 될 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**Free Frame이 존재하지 않는 경우엔?**

- 메모리에 존재하지만 실제로 사용되지 않는 페이지를 Swap-out

- Page Fault를 최소하하는 알고리즘이 필요

- modify bit(dirty bit)를 사용하는 방법.

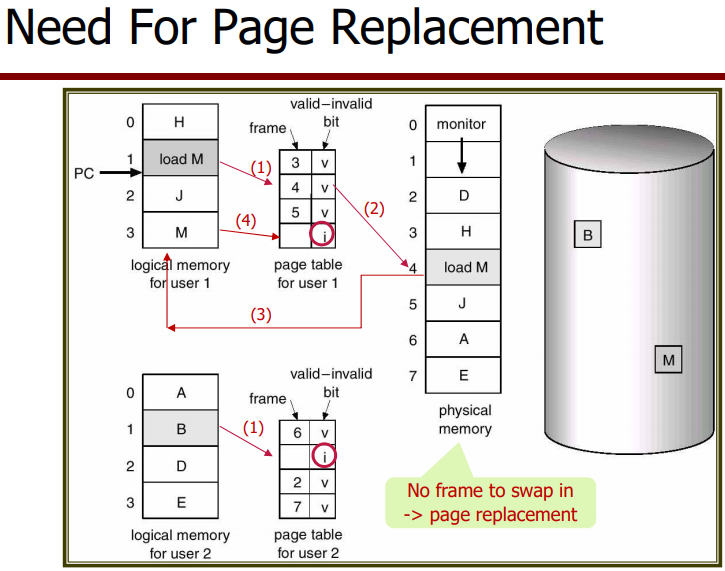
- 페이지는 swap in될 때 다시 dirty bit를 0으로 만든다.

- Swap in-out 발생 시 Swap out되는 페이지의 dirty bit가 0이라면 수정된 적이 없다는 뜻이므로, 디스크에 저장된 페이지와 지금 Swap out될 페이지의 정보가 일치한다는 뜻이다. 그래서 Swap-out을 다시 할 필요가 없고 그대로 해당 프레임을 할당해 주면 된다.

- dirty bit가 1이라면 Swap-out 수행

- 페이지 교체가 되면 가상 메모리와 물리 메모리 간의 분리가 완료된다. 물리 메모리가 더 적어도 이런 식으로 페이지 교체를 통해 마치 가상 메모리 크기만큼의 물리 메모리가 존재하는 것처럼 사용할 수가 있게 된다.

다음 그림을 보자.



**상황**

- user1, user2 프로세스가 각각 페이지 4개를 사용하고 있음

- user1, user2 페이지 테이블에 실제 물리 메모리에 할당된 페이지가 3개씩 존재

- 물리 메모리는 꽉 차 있음

**수행**

1. PC가 1을 가리키고 있으므로, Page Table 참조하여 페이지 번호 1 확인.

2. valid 이므로 물리 메모리에 있는 프레임 주소 4 참조하여 load M 이라는 명령 실행

3. 페이지 M을 추출해 보니 페이지 번호 3, 따라서 페이지 테이블 3번 Entry 확인.

4. invalid이므로, 디스크에 존재하는 페이지 M을 물리 메모리에 적재해야 함

5. 물리 메모리가 모두 사용중이라서 페이지 교체 알고리즘을 통해 어떻게 Swap in-out을 할지 결정

**페이지 교체 알고리즘**

- 이건 알아서 공부해야겟다.

## Process란?

- PC와 Register Set, Process ID, State, 메모리 등의 정보를 가진다.

- 프로세스는 적어도 하나의 Thread를 가진다.

**Thread란?**

- CPU 스케쥴링의 기본 단위.

- Program Counter, Register들과 Stack으로 구성

- Process 내부에서 여러 Thread 생성 가능하며, 코드와 메모리를 공유한다.

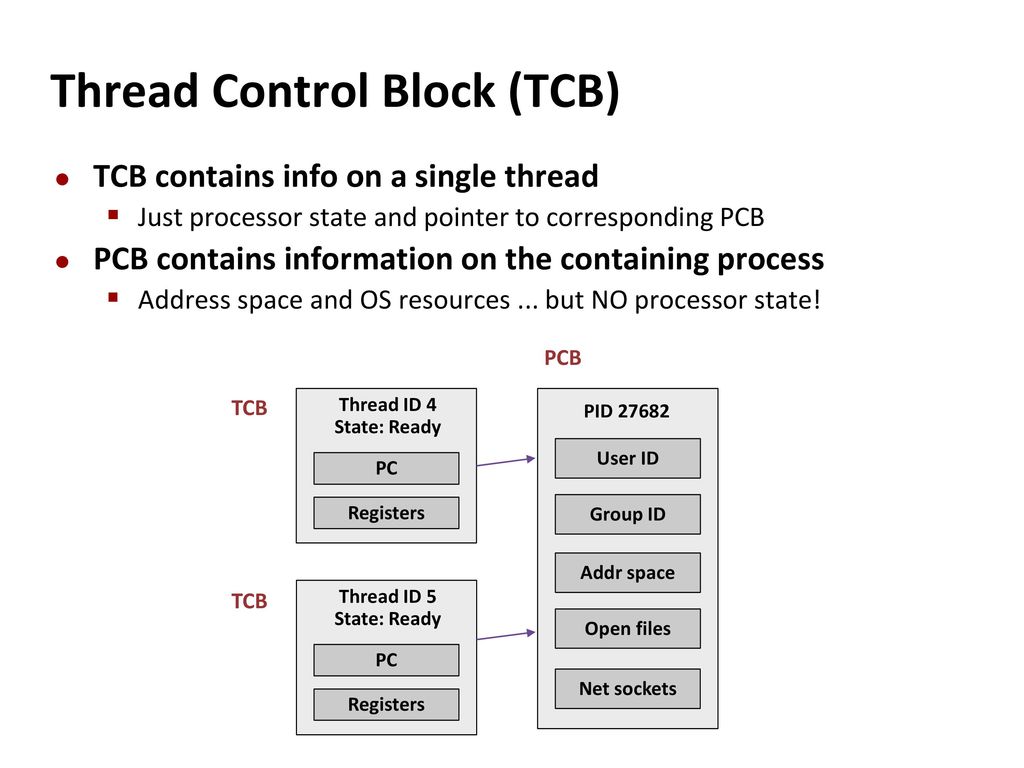
운영체제 책에는 이렇게 설명되어 있지만, 의문이 든다.

"프로세스가 여러개의 스레드를 가지면 PCB 하나로 어떻게 저장하지? 각 Thread의 정보를 저장해야 할 텐데."

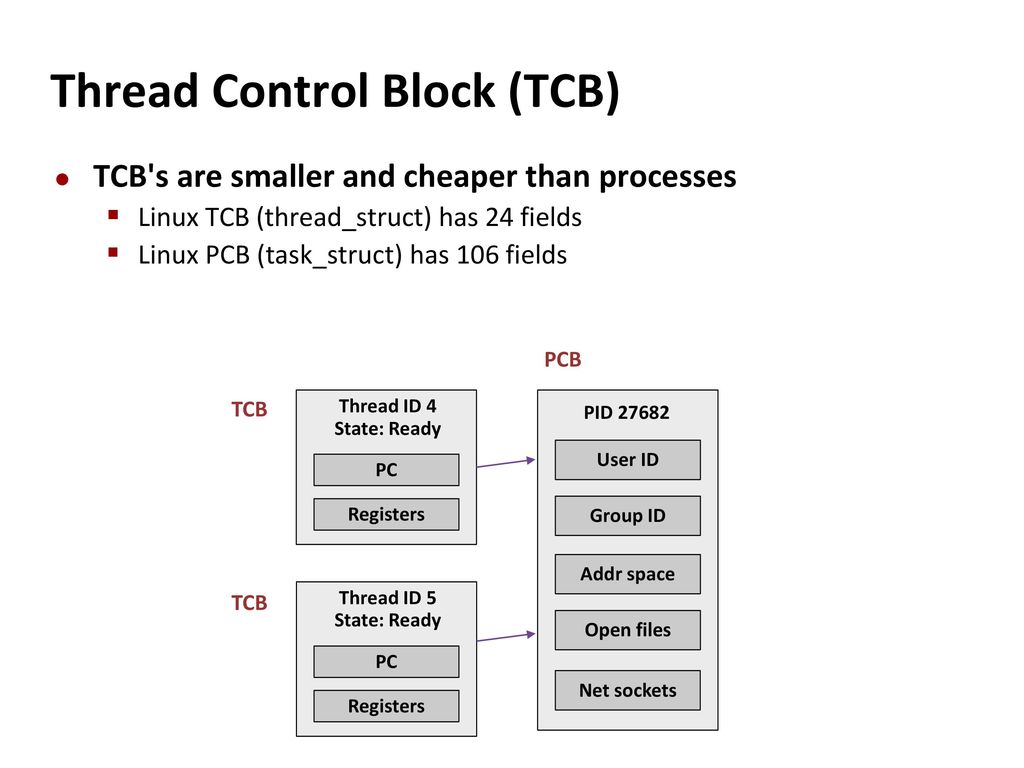
이렇게 생각해서 알아본 것이 TCB이다.

**PCB**는 Process ID와 상태, 우선순위, 메모리 정보 등을 저장한다. 멀티스레드가 아닌 멀티프로세스 환경에서는 PCB가 PC와 Register Set 정보도 포함한다. 여기서는 멀티스레드 환경이라 가정한다.

**TCB**는 Thread별로 존재하는 자료구조이며, PC와 Register Set(CPU 정보), 그리고 PCB를 가리키는 포인터를 가진다.

TCB는 PCB를 가리키는 포인터를 가진다

그리고 TCB는 PCB보다 적은 데이터를 가지는 자료구조이다. 해당 Thread에 대한 정보만 저장하면 되기 때문이다.

TCB가 PCB보다 데이터가 적다

보통 TCB는 커널 레벨에서 Context Switching의 기본 단위가 되며, 같은 프로세스에서의 스위칭에 대해서는 TCB 정보만 저장하면 된다.

하지만 다른 프로세스 간의 스위칭을 할 때에는 PCB / TCB 정보를 모두 저장해야 한다.



**User Level Thread**

- Kernel에서 관리되지 않는 Thread

- 커널에서 보기에 프로세스와 스레드 비율 1:1인 프로그램이므로, Time slice를 받게 되면 User Level에서 분할하여 사용하게 된다.

- 스케줄러도 User Level(Runtime/RAM)에 존재하며, Context Switching을 진행할 때 TCB 정보만 저장/복구하면 되기 때문에 빠르다.

- System Call이 발생하면 프로세스 자체가 Block된다.

- Interrupt 발생(I/O작업 등)이 없는 프로그램이라면 이 방식이 유리하다.

- 운영체제에 따른 영향을 덜 받기 때문에 이식성이 좋다.

**Kernel Level Thread**

- Kernel에서 관리되는 Thread

- System Call이 발생해도 해당 Thread만 Block되어 wait\_queue에 들어간다.

- 운영체제에 따라 정책이 다르기 때문에, 이식성이 낮다.

**\*추가정보 : JVM, V8...**

JVM같은 경우 User Level Thread를 이용하며, 내부적으로는 Kernel Level Thread로 1:1매핑한다.

Node.js의 엔진인 V8 같은 경우, 내부적으로 C++을 사용하는데, 리눅스에서는 C++ 에서 Thread를 이용할 때 User / Kernel Level Thread를 1:1매칭하는 pThread 라이브러리를 이용한다.

게다가 V8은 내부적으로 IOCP를 이용하여 비동기 IO를 구현하므로 IO 관련 System Call에 대해서는 Block되지 않는다.

그렇기 때문에 Spring 서버와 Node.js 서버의 차이는 다음과 같다.

**Node.js**

- IOCP를 통해 비동기 I/O를 이용하므로, Block되지 않는다.

- 동기화 작업을 수행할 필요가 없지만 I/O를 제외한 CPU 작업을 수행하게 되면 그동안 Event Loop가 블락되기 때문에 조심해야 한다.

**Spring**

- 멀티스레드로 I/O를 관리하면 Block되지 않는다. 예를 들면 Network I/O 등이 있다. Spring은 요청 당 하나의 스레드를 할당하여 Block되지 않도록 한다. 하지만 이 때 요청 수가 스레드 수보다 많다면 처리중인 요청이 모두 완료될 때까지 대기하게 된다.

- 동기화 처리를 제대로 하지 못한다면 문제가 발생한다.

- CPU 작업을 다른 스레드에서 진행할 수 있다.