01 운영체제의 개요

운영체제(Operating System: OS): 컴퓨터, 노트북, 스마트폰 전원을 키면 가장 먼저 만나는 소프트웨어

- PC운체: 윈도우, Mac OS, 유닉스, 리눅스 등
- 모바일 운체: iOS, 안드로이드 등
- 임베디드 운체: CPU 성능 낮고 메모리 크기 작은 시스템에 내장하도록 만듦
  임베디드 운체가 있는 기계는 기능 계속 향상 가능(스마트 시계, TV)

정의 <운체란?>

컴퓨터 전체를 관리/운영하는 소프트웨어로, 모든 sw위에 존재하는 최고 sw

컴퓨터 관리를 위한 기본 규칙/절차 규정하여 컴퓨터 내 모든 hw와 응용 프로그램 관리

사용자가 자원에 직접 접근 막음으로써 컴퓨터 자원 보호. 이를 위해 응용 프로그램과 사용자에게서 모든 컴퓨터 자원 숨김

사용자가 자원을 이용할 수 있는 다양한 interface 제공

운체는 하드웨어 도움 없이 작동 어려워서 sw와 hw특성 모두 갖춘 형태로 운영됨 -> 펌웨어(firmware)

운체는 커널(kernel)과 인터페이스(interface)로 나뉨

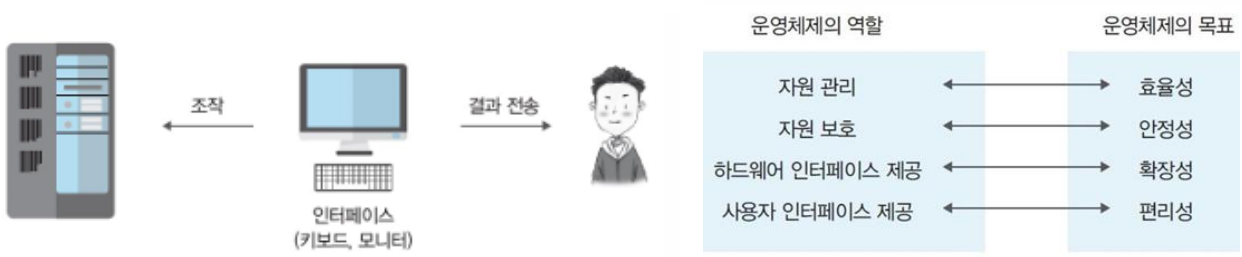
커널은 운체 핵심 기능 모아둔 것으로, 모든 컴퓨터 자원 관리. 사용자/응용 프로그램은 커널 통해서만 컴퓨터 자원 접근 가능. 어떤 사용자나 응.프도 자원에 직접 접근X

인터페이스를 이용해 커널에 명령을 내리면 인터페이스가 결과 전달

컴퓨터 인터페이스 중 일부는 사용자 인터페이스(User Interface; UI)

ex. 디렉터리(폴더) 만들기, 파일 복사 등 작업은 운체가 제공하는 UI 사용

문자 기반 인터페이스 -> 그래픽 사용자 인터페이스(Graphical User Interface; GUI) -> 터치스크린 인터페이스, 음성 인터페이스



역할

1. 자원관리
   : 컴퓨터 시스템 자원을 응용 프로그램에 나눠줘서 사용자가 원활히 작업하게 함
   자원 요청한 프로그램 여러 개면 적당한 순서로 자원 배분하고 적절한 시점에 자원 회수해서 다른 응.프에 나눠줌
2. 자원 보호
   : 비정상저 작업으로부터 컴퓨터 자원 보호
3. 하드웨어 인터페이스 제공
   : 사용자가 복잡한 과정 없이 다양한 장치를 사용할 수 있게 hw인페 제공
   CPU, 메모리, 키보드, 마우스 같은 다양한 hw 일관된 방법으로 사용하도록 지원
4. 사용자 인터페이스 제공
   : 사용자가 운체 편리하게 사용하도록 지원 (ex. 윈도우 그래픽 사용자 인페 GUI)

목표

1. 효율성
   : 자원 효율적으로 관리. 같은 자원으로 더 많은 작업량 처리/같은 작업량 처리에 더 적은 자원 사용
2. 안정성
   : 작업 안정적으로 처리. 사용자와 응.프 안전 문제와 hw적 보안 문제 처리
   시스템에 문제 발생시 이전으로 복구하는 결함 포용 기능 수행
3. 확장성
   : 다양한 시스템 자원 컴퓨터에 추가/제거하기 편리
4. 편리성
   : 사용자가 편리하게 작업할 수 있는 환경 제공

역사

시기	주요 기술	특징
1940년대	없음	• 진공관(0과 1) 사용
1950년대	카드 리더, 라인 프린터	• 일괄 작업 시스템 • 운영체제의 등장
1960년대 초반	키보드, 모니터	• 대화형 시스템
1960년대 후반	C 언어	• 멀티프로그래밍 기술 개발 • 시분할 시스템
1970년대	PC	• 개인용 컴퓨터의 등장 • 분산 시스템
1990년대	웹	• 클라이언트/서버 시스템
2000년대	스마트폰	• P2P 시스템(메신저, 파일 공유) • 클라우드 컴퓨팅 • 사물 인터넷

애니악: 최초 컴퓨터. 진공관 소자가 켜지면 1, 꺼지면 0으로 판단해 컴퓨터가 2진법을 사용하는 계기가 됨

초기 컴퓨터는 키보드, 마우스, 모니터 같은 주변장치X. 전선 연결해서 논리회로 구성하는 하드와이어링(hard wiring) 방식. 운영체제 없음

천공카드 시스템

: 천공카드 리더를 입력장치로, 라인 프린터를 출력장치로 사용

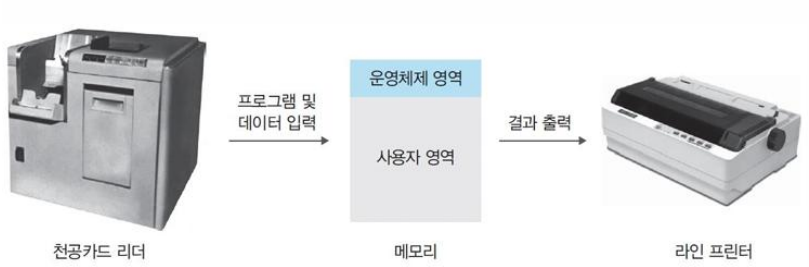
카드에 구멍을 뚫어 프로그램 입력하면 실행 결과 라인 프린터로 출력

hard wiring 시스템은 전선을 일일이 다시 연결해야 다른 작업 가능했지만 천공카드 시스템은 프로그램만 바꾸면 다른 작업 가능해 지금의 프로그래밍과 유사한 방식으로 다양한 sw개발

일괄 작업 시스템(batch job system / batch processing system)

: 천공카드 시스템은 필요한 프로그램과 데이터 동시에 입력해야 작업 가능한데 이를 일괄 작업 시스템 또는 일괄 처리 시스템이라 부름

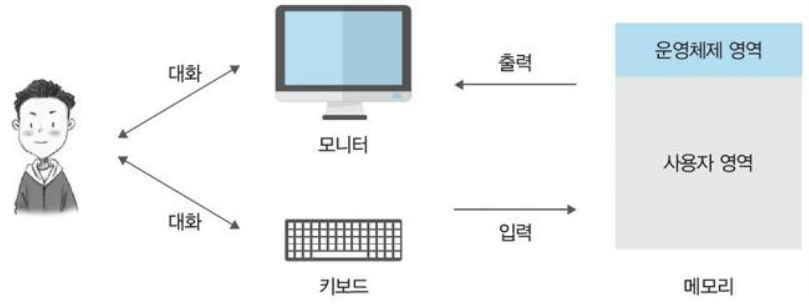
일괄 작업 시스템에서는 작긴 하지만 운체가 사용돼서 메인메모리가 운체 영역과 사용자 영역으로 나뉨



대화형 시스템(interactive system)

: 키보드와 모니터가 등장하며 작업 중간에 입력, 중간 결과값 확인 가능.

중간 입력값에 따라 작업 흐름 바꾸는 것도 가능. 컴퓨터와 사용자의 대화를 통해 작업이 이뤄져서 대화형 시스템



멀티프로그래밍

: 하나의 CPU로 여러 작업 동시에 실행하는 기술. 한 번에 하나의 작업만 가능한 일괄 작업 시스템에 비해 효율성 뛰어남. 시간 분할하는 방법 땜에 여러 작업 동시에 실행되는 것처럼 보임.

시분할 시스템(time sharing system)

: CPU 사용 시간 아주 잘게 쪼개 여러 작업에 나눠줘서 모든 작업 동시에 처리하는 것처럼 보임. 잘게 나뉜 시간 한 조각을 타임 슬라이스(time slice) / 타임 퀀텀(time quantum)

오늘날 컴퓨터 대부분 시분할 시스템 사용.

단점은 여러 작업 동시에 처리하기 위해 추가 작업 필요, 많은 양의 작업이 공존할 경우 중요한 작업이 일정 시간 안에 끝나는 것을 보장하지 못함

특정 시스템에서 일정 시간 안에 작업 처리되도록 보장하는 것이 실시간 시스템(real-time system)

실시간 시스템 종류

- 경성 실시간 시스템(hard real-time system): 지정한 응답 시간을 정확히 지키는 시스템으로 원자력 발전소 원자로 온도 제어나 미사일 요격과 같은 작업에 이용
- 연성 실시간 시스템(soft real-time system): 지정한 응답 시간을 최대한 지키지만 어느정도 융통성이 허용된 시스템. 동영상 재생기의 경우 응답 시간 안에 작업이 처리되지 않으면 끊김 현상 발생, but 치명적 결과를 낳지 않음

분산 시스템(distributed system)

: 개인용 컴퓨터와 인터넷이 보급되며 값이 싸고 크기가 작은 컴퓨터를 하나로 묶어 대형 컴퓨터 능력에 버금가게 만듦

네트워크상에 분산되어 있는 여러 컴퓨터로 작업을 처리하고 그 결과를 상호 교환하도록 구성

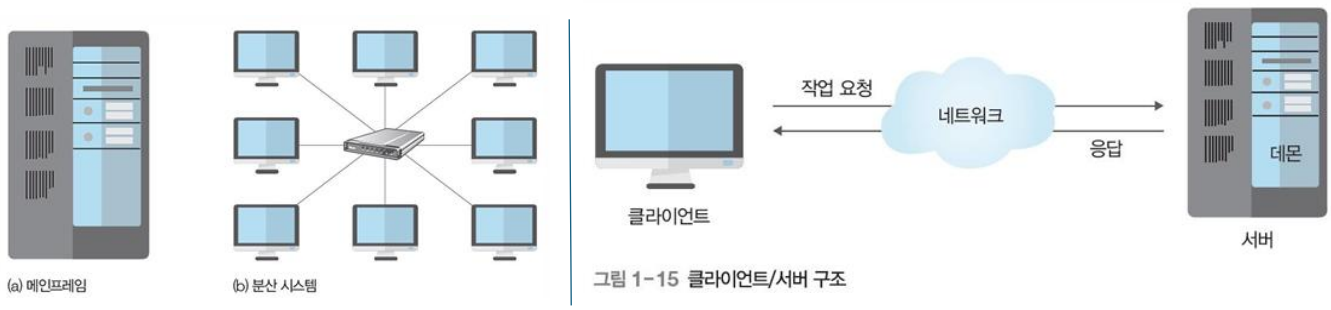


그림 1-15 클라이언트/서버 구조

클라이언트/서버 시스템(client/server system)

: 작업을 요청하는 c와 요청받은 작업 처리하는 s의 이중 구조

웹 시스템이 보급된 이후 일반인들에게 알려짐. 문제점은 모든 요청이 서버로 집중돼서 서버 과부하. 이를 피하려면 많은 서버와 큰 용량 네트워크 필요

P2P시스템

: c/s 구조 단점인 서버 과부하 해결 위해 만든 시스템. 서버를 거치지 않고 사용자와 사용자 직접 연결!

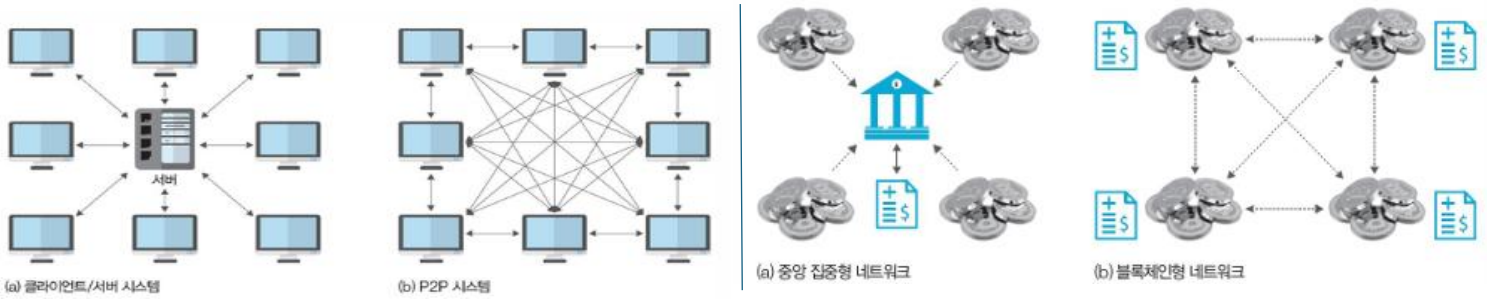
냅스터(mp3 공유 시스템)에서 시작해서 현재는 메신저나 토렌트 시스템에서 사용

- ex. 메신저: 메신저는 사용자가 서버에서 인증 마치면 상대방과 P2P로 직접 연결

서버가 없는 완전한 P2P 시스템 <블록체인(block chain)>

: P2P 시스템 전체에 거래 장부 분산시킨 뒤 전체 시스템 50% 이상이 동의할 때만 거래 장부 변경 가능. 현재 컴퓨팅 기술로 해킹 불가

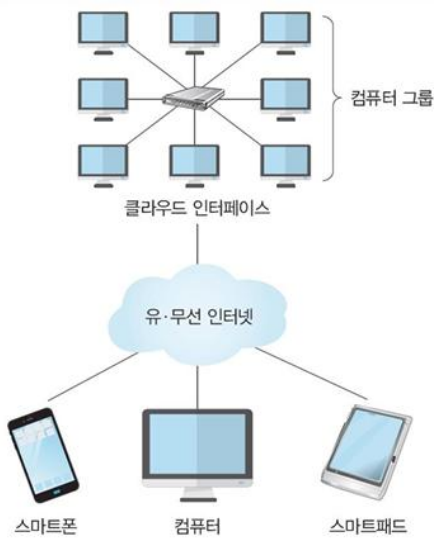
대체 불가 토큰 또는 NFT(Non-Fungible Token)로 불리는 디지털 자산 유일성 증명하는 데 블록체인 사용



클라우딩 컴퓨팅(cloud computing)

: hw와 sw를 클라우드라는 중앙 시스템에 숨기고 사용자는 필요한 서비스만 쉽게 이용하는 컴퓨팅 환경

클라우드 컴퓨팅의 hw구현에 그리드 컴퓨팅 기술 사용 (슈퍼컴퓨터와 맞먹는 높은 수준의 컴퓨팅 파워 제공, 클.컴은 서비스 중심 환경이라는 것이 가장 큰 차이점)



사물 인터넷(Internet of Thing; IoT)

사물(thing)이 인터넷에 연결된 시스템. 다양한 사물이 센서와 통신 기능을 내장하여 스스로 통신하며 지능적인 서비스 제공. ex) 자동차에 무선 인터넷이 연결되면서 sw업그레이드나 설정 변경 등 무선으로 내포(Over-The-Air; OTA)

02 컴퓨터의 구조와 성능 향상

하드웨어 구성

필수장치: 중앙처리장치, 메인메모리(대부분 작업 이뤄짐)

주변장치: 입력장치, 출력장치, 저장장치

메인메모리->메모리, 보조저장장치->저장장치, 중앙처리장치->CPU

CPU: 명령어를 해석해 실행하는 장치로 인간의 두뇌 같은 애

메모리: 작업에 필요한 프로그램과 데이터 저장하는 장소. Byte단위로 분할, 분할 공간마다 주소로 구분

입력장치: 외부 데이터 컴퓨터에 입력하는 장치

출력장치: 컴퓨터에서 처리한 결과를 사용자가 원하는 형태로 출력하는 장치

저장장치: 메모리보다 느리지만 저렴하고 용량 큼. 전원 온오프와 상관없이 데이터 영구적 저장(비휘발성) 느린 저장장치 사용 이유는 저장 용량에 비해 가격 쌈

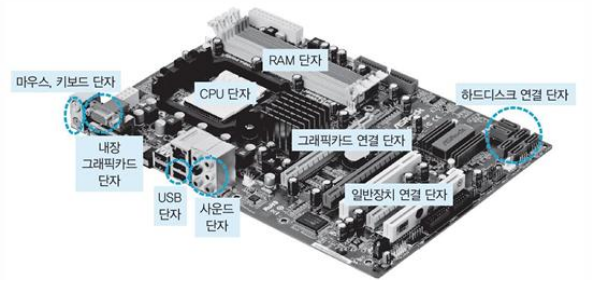
종류

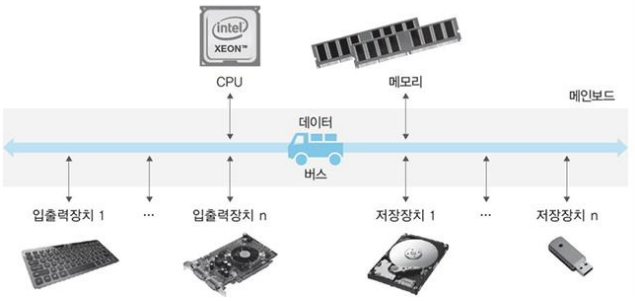
- 자성 이용 장치: 카세트테이프, 플로피디스크, 하드디스크
- 레이저 이용 장치: CD, DVD, 블루레이디스크
- 메모리 이용 장치: USB 드라이버, SD 카드, CF 카드, SSD

메인보드

: CPU와 메모리 등 다양한 부품 연결하는 커다란 판. 다양한 장치들을 bus로 연결. (bus는 데이터 지나다니는 통로)

그래픽카드, 사운드카드, 랜카드 등이 기본으로 장착되어 있기도 하고 성능 향상 위해 따로 장착하기도.





폰노이만 구조

: CPU, 메모리, 입출력장치, 저장장치가 버스로 연결된 구조

프로그램은 하드디스크와 같은 저장 장치에 담겨 있으나 가장 큰 특징은 저장장치에서 바로 실행 못 하고 메모리로 가지고 올라와야 실행 가능 (모든 프로그램은 메모리에 올라와야 실행 가능)

이러한 특징으로 인해 메인 메모리가 유일한 작업 공간, 메모리 관리가 중요한 이슈

요리사 모형

: 운체와 여러 가지 현상에 대한 이해를 돕기 위함 (요리사->CPU, 도마->메모리, 보관 창고->저장장치)

요리사(CPU)가 요리 하려면 보관 창고(저장장치)에 있는 재료 도마(메모리)로 가져와야.

주방에는 도마(메모리)가 핵심적인 작업 공간이고 보관 창고(저장장치)는 보조 공간

메인 메모리 작으면 컴퓨터 느려지는 이유

: 도마가 크면 재료 모두 가져다놓고 요리하면 되는데 도마 작으면 재료 모두 못 가져옴

도마 작으면 하나 하고 보관 창고 갖다 넣고 다른 거 가져와야 해서 재료 손질보다 옮기는 시간이 많이 걸려서 작업 속도 떨어짐. 도마 크기 크면 작업 속도에 영향X

요리사 모형	운영체제 작업
요리 방법 결정	프로세스 관리
도마 정리	메모리 관리
보관 창고 정리	저장장치 관리

크기 단위(데이터 기본 표시 단위 bit, 8bit 묶어서 Byte)

단위	표기	2진 크기	10진 크기	바이트 대비 크기	10진 단위
Byte(바이트)	B	1	1	1B	
Kilo Byte(킬로바이트)	KB	2^{10}	10^{3}	1,000B	천
Mega Byte(메가바이트)	MB	2^{20}	10^{6}	1,000,000B	백만
Giga Byte(기가바이트)	GB	2^{30}	10^{9}	1,000,000,000B	십억
Tera Byte(테라바이트)	TB	2^{40}	10^{12}	1,000,000,000,000B	조
Peta Byte(페타바이트)	PB	2^{50}	10^{15}	1,000,000,000,000,000B	천조

클록(clock)

: 컴퓨터에서 일정 박자 만들어 내는 것. 클록에 의해 일정 간격으로 만들어지는 tick을 펄스(pulse) 혹은 클록 틱(clock tick)

클록이 일정 간격으로 펄스 만들면 거기 맞춰서 컴퓨터 안의 모든 구성 부품들이 작업 진행 (ex. CPU는 클록 한 번에 한 번의 덧셈, 메모리도 클록 발생마다 데이터 저장)

헤르쯔(Hertz; Hz)

: 제품에 따라 제각각인 CPU 성능 나타내는 단위. 컴퓨터에서 헤르츠 단위 사용하는 건 1초 동안 몇 번의 작업이 이루어져서 몇 번의 펄스(클록틱)가 발생했는지를 의미

시스템 버스(system bus)

: 메모리와 주변장치 연결하는 버스. 메인보드 동작 속도 의미함 FSB(Front-Side Bus) 혹은 전면 버스라 부름

CPU버스는 CPU 내부의 다양한 장치 연결하는 것으로 BSB(Back Side Bus) 혹은 후면 버스라 부름

BSB 속도는 CPU 클록과 같고 FSB보다 훨씬 빠름

부품	사양
CPU	인텔 코어 i7(6코어, 12스레드, 기본 3.7GHz, 최대 5.0GHz, 캐시 20MB)
메인보드	FSB 3,200MHz
메모리	DDR4 SDRAM 4GB(3,200MHz)
그래픽카드	RTX 3060 12GB, 베이스클록 1,530MHz
하드디스크(HDD)	1TB, 7200rpm, 256MB

Programing: 컴퓨터 프로그램 만드는 것 (이때 사용하는 언어가 programing language)

기계어(machine language): 컴퓨터가 이해할 수 있는 언어. 0/1 이진수로 이뤄짐

어셈블리어(assembly language): 기계어를 사람들이 이해할 수 있는 문자 형태로 바꿈

저급언어(low level language): 기계어는 아니지만 일반인이 사용하기 어려움

고급언어(high level language): if나 for 같이 일반인이 이해할 수 있는 단어로 만든 언어

기계어, 어셈블리어 제외한 대부분 언어가 고급언어임

compile: 고급언어 기계어로 번역하는 과정 (컴파일 담당 프로그램 compiler)

컴파일러가 소스코드 번역해서 기계어로 이뤄진 실행 파일 만들면 컴퓨터가 실행해서 작업 시작

인터프리터(interpreter): 소스코드를 기계어로 번역하는 다른 방법 (소스코드 한 번에 한 문장씩 번역해서 실행), 실행 파일 따로 안 만들어서 편한데 코드에 잘못된 부분 있는지 파악하기 어렵고 반복 작업 하나로 합치기 어려움

큐(Queue)

: 먼저 들어온 자료가 먼저 처리되는 First In, First Out. 한쪽으로 자료 넣으면 반대쪽으로 나옴

스택(Stack)

: 가장 나중에 들어온 자료가 가장 먼저 처리되는 First In, Last Out. 순서가 뒤집혀 나오고 한쪽으로만 입출력 이뤄짐

배열(Array)

: 형태가 같은 자료 나열해서 메모리에 연속으로 저장. 구현 간단한데 공간 삽입과 삭제 같은 관리가 어려운 게 단점!

연결 리스트는 데이터를 pointer로 연결해 데이터를 중간에 삽입하는 것도 가능

배열 형태 테이블은 크기가 어느 정도 정해진 메모리나 파일 관리에 사용/연결 리스트는 데이터의 삽입과 삭제가 빈번한 테이블에 주로 사용

큐 구현에도 연결 리스트 사용하면 편리

CPU 구성과 동작

산술논리 연산장치(Arithmetic and Logic Unit; ALU)

: 데이터 +-*/ 같은 산술 연산과 AND, OR 같은 논리 연산 수행

제어 장치(control unit)

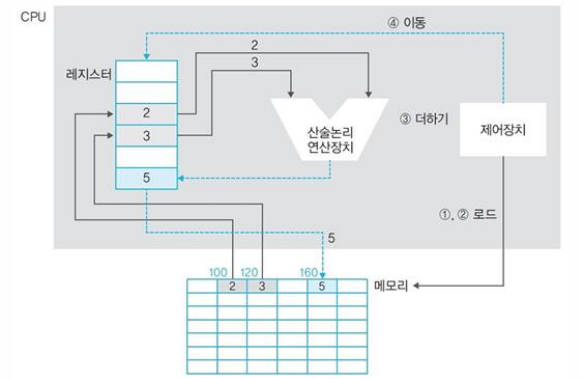
: CPU에서 작업을 지시하는 부분

레지스터(register)

: 작업에 필요한 데이터를 CPU 내부에 보관하는 것

CPU 명령어 처리 과정

: CPU에서는 산술논리 연산장치, 제어장치, 레지스터들이 협업하여 작업 처리

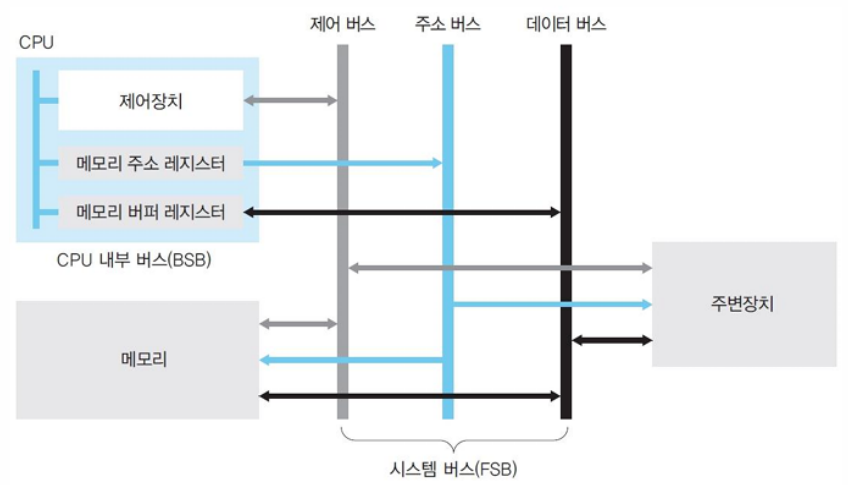


레지스터		특징
일반 레지스터	데이터 레지스터(DR)	CPU가 명령어를 처리하는 데 필요한 데이터를 임시로 저장하는 범용 레지스터다.
	주소 레지스터(AR)	데이터 또는 명령어가 저장된 메모리의 주소를 저장한다.
특수 레지스터	프로그램 카운터(PC)	다음에 실행할 명령어의 위치(코드의 행 번호)를 저장한다.
	명령어 레지스터(IR)	현재 실행 중인 명령어를 저장한다.
	메모리 주소 레지스터(MAR)	메모리 관리자가 접근해야 할 메모리의 주소를 저장한다.
	메모리 버퍼 레지스터(MBR)	메모리 관리자가 메모리에서 가져온 데이터를 임시로 저장한다.
	프로그램 상태 레지스터(PSR)	연산 결과(양수, 음수 등)를 저장한다.

레지스터 종류

- 데이터 레지스터(data register; DR): 메모리에서 가져온 데이터 임시 보관. 일반 레지스터/범용 레지스터라고 부름
- 주소 레지스터(address register; AR): 데이터 또는 명령어가 저장된 메모리의 주소를 저장하는 레지스터
- 프로그램 카운터(program counter; PC): 다음에 실행할 명령어의 주소를 기억 했다가 제어장치에 알려줌. =명령어포인터(instruction pointer)
- 명령어레지스터(instruction register; IR): 현재 실행 중인 명령어 저장
- 메모리 주소 레지스터(memory address register; MAR): 데이터를 메모리에서 가 가져오거 보낼 때 주소 지정하는 데 사용
- 메모리 버퍼 레지스터(memory buffer register; MBR): 메모리에서 가져오거나 옮겨갈 데이터를 임시로 저장. 항상 메모리주소 레지스터와 함께 동작

버스 종류



버스	특징
제어 버스	제어장치와 연결된 버스로, CPU가 메모리와 주변장치에 제어 신호를 보내기 위해 사용한다. 메모리와 주변장치에서도 작업이 완료되거나 오류가 발생하면 제어 신호를 보내기 때문에 양방향이다.
주소 버스	메모리 주소 레지스터와 연결된 버스로, 데이터를 읽거나 쓸 때 메모리나 주변장치에 위치 정보를 보내기 위해 사용하며 단방향이다.
데이터 버스	메모리 버퍼 레지스터와 연결된 버스로, 데이터의 이동이 양방향으로 이루어진다.

CPU 비트의 의미

: 32bit CPU에서 32bit는 CPU가 메모리에서 데이터를 읽거나 쓸 때 한 번에 최대 32bit 처리할 수 있으며 레지스터 크기와 버스 대역폭도 32bit. 32비트용 운체 설치+사용해야

버스 대역폭(bandwidth) : 한 번에 전달할 수 있는 데이터 최대 크기

버스 대역폭 = 레지스터 크기 = 메모리에 한 번에 저장할 수 있는 데이터 크기

워드(word) : CPU가 한 번에 처리할 수 있는 데이터 최대 크기 (1워드 = 32bit / 64bit...)

메모리 종류

휘발성 메모리(volatility memory)

- DRAM(Dynamic RAM): 저장된 0/1 데이터 일정 시간 지나면 사라져서 일정 시간마다 다시 재생시켜야 함
- SRAM(Static RAM): 전력 공급되는 동안 데이터 보관할 수 있어서 재생 필요X
- SDRAM(Synchronous Dynamic Random Access Memory): 클록틱(펄스) 발생마다 데이터 저장하는 동기 DRAM

비휘발성 메모리(non-volatility memory)

- 플래시 메모리(flash memory): 디카, MP3, USB같이 전력 없어도 데이터 보관
- SSD: 가격 싼데 빠른 데이터 접근 속도, 저전력, 높은 내구성으로 많이 사용

롬

- 마스크 롬(mask ROM): 데이터 지우거나 쓸 수 없음
- PROM(Programmable ROM): 전용 기계를 이용해서 데이터 한 번만 저장 가능
- EPROM(Erasable Programmable ROM): 데이터를 여러 번 쓰고 지울 수 있음

메모리 보호

: 시분할 기법 사용하는 현대 운체는 여러 프로그램 동시에 실행해서 메모리 보호 중요

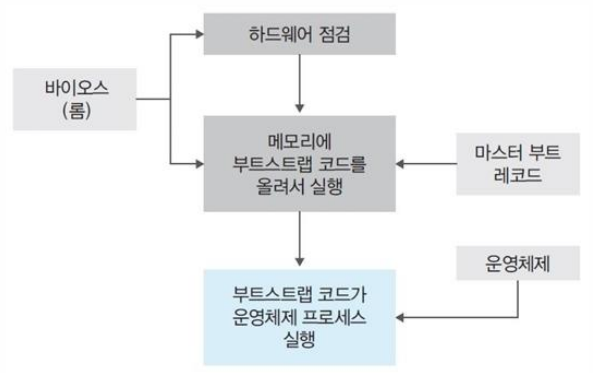
사용자 프로세스가 CPU 차지해 작업 진행하면 운체 작업 중단되는데 사용자 작업으로부터 메모리 보호하려면 hw 도움 필요

메모리 보호 위해 CPU는 현재 진행중인 작업 메모리 시작 주소를 경계 레지스터(bound register)에 저장하고 현재 진행 중인 작업이 차지하고 있는 메모리 크기(마지막 주소까지 차이)를 한계 레지스터(limit register)에 저장

사용자 작업이 진행되는 동안 두 레지스터 주소 범위를 벗어나는지 hw측면에서 점검함으로써 메모리 보호

부팅

: 컴퓨터 켰을 때 운체 메모리에 올리는 과정



버퍼

: 속도에 차이가 있는 두 장치 사이에서 그 차이를 완화하는 역할을 하는 장치.

일정량의 데이터를 모아 옮김으로써 속도 차이 완화

스풀

: CPU와 입출력장치가 독립적으로 동작하도록 고안된 sw적인 버퍼

ex) 스풀러: 인쇄할 내용 순차적으로 출력하는 sw로 출력 명령을 내린 프로그램과 독립적으로 동작. 인쇄물 완료까지 다른 인쇄물 끼어들 수 없으므로 프로그램 간 배타적

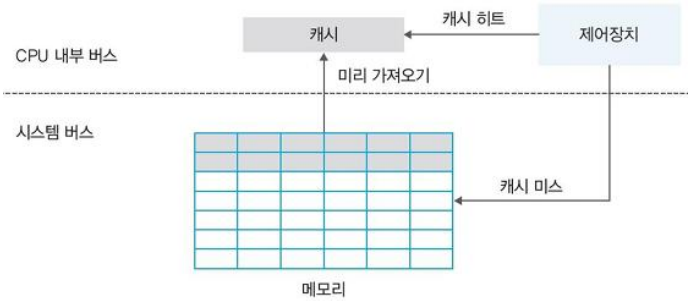
캐시

: 메모리와 CPU간 속도 차(BSB&FSB) 완화 위해 메모리 데이터 미리 가져와 저장해두는 임시 장소. 필요한 데이터 모아 한꺼번에 전달하는 버퍼 일종으로 CPU가 사용할 걸로 예상되는 데이터 미리 갖다둠. CPU는 메모리 접근할 때 캐시 먼저 방문해서 원하는 데이터 찾아봄 ㅇㅇ

캐시 히트(cache hit): 캐시에서 원하는 데이터 찾음. 그 데이터 바로 사용!

캐시 미스(cache miss): 캐시에 원하는 데이터 없어서 메모리 가서 찾음

캐시 적중률(cache hit ratio): 캐시 히트 되는 비율. 일반 컴퓨터는 약 90%



즉시 쓰기(write through)

: 캐시에 있는 데이터가 변경되면 즉시 메모리에 반영! 메모리와 빈번한 데이터 전송으로 성능 느려짐...메모리 최신 값이 항상 유지돼서 급작! 정전에도 데이터 안 잃음

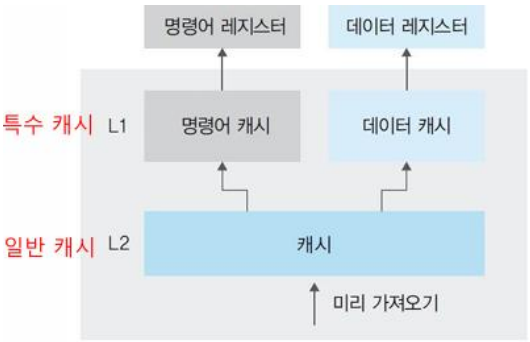
지연 쓰기(write back)

: 캐시에 있는 데이터 변경되면 변경된 내용 모아서 주기적으로 반영. =카피백(copy back), 메모리와 데이터 전송 횟수 줄어서 시스템 성능 향상 가능. 메모리와 캐시된 데이터 사이 불일치 발생 가능...ㅠ

L1캐시와 L2캐시

: 캐시는 명령어와 데이터 구분 없이 모든 자료 가져오는 일반 캐시, 명령어와 데이터 구분해서 가져오는 특수 캐시로 구분함.

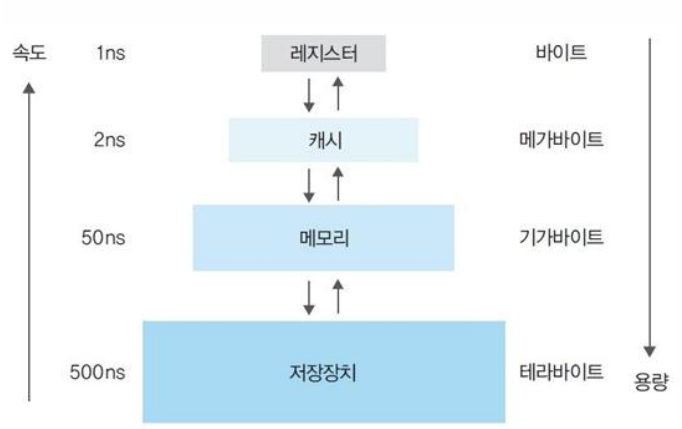
일캐는 메모리와 연결돼서 L2, 특캐는 CPU 레지스터에 직접 연결돼서 L1캐시임



저장장치 계층 구조

: 속도 빠르고 비싼 저장장치를 CPU 가깝게 두고, 싸고 용량 큰 저장장치 반대쪽에 배치해서 적당한 가격으로 빠른 속도와 큰 용량 동시에!

이점-> CPU와 가까운 쪽에 레지스터/캐시 배치해서 CPU가 작업 빨리 진행 가능. 메모리에서 작업한 내용 하드디스크와 같은 저렴하고 용량 큰 저장장치에 영구적으로 저장!



폴링(polling) 방식

: CPU가 직접 입출력장치에서 데이터 가져오거나 내보내는 방식. CPU가 입출력장치 상태 주기적으로 검사해서 일정한 조건 만족할 때 데이터 처리. CPU가 명령어 해석/실행 본래 역할 외에 모든 입출력까지 관여해서 작업 효율 떨어짐

인터럽트(interrupt) 방식

: 입출력 관리자가 대신 입출력 해주는 방식. CPU의 작업과 저장장치 데이터 이동을 독립적으로 운영해서 시스템 효율 높임. 데이터 입출력동안 CPU 다른 작업 할 수 있음!

인터럽트: 입출력 관리자가 CPU에 보내는 완료 신호

인터럽트 번호: 많은 주변장치 중 어떤 것의 작업이 끝났는지 CPU에 알려주기 위해 사용. 윈도우 운체는 IRQ라 부름

인터럽트 벡터: 여러 개 입출력 작업 한꺼번에 처리 위해 여러 개 인터럽트를 하나의 배열로 만듦

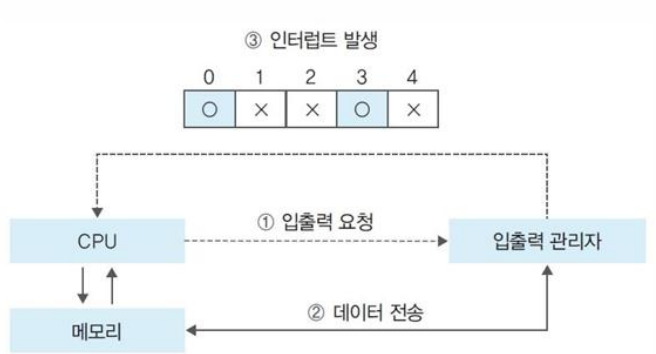


그림 2-25 인터럽트 처리 과정

직접 메모리 접근(Direct Memory Access; DMA)

: 입출력 관리자가 CPU 허락 없이 메모리에 접근할 수 있는 권한. 메모리는 CPU 작업 공간이지만 데이터 전송을 지시받은 입출력 관리자는 직접 메모리 접근 권한이 있어야만 작업 처리 가능

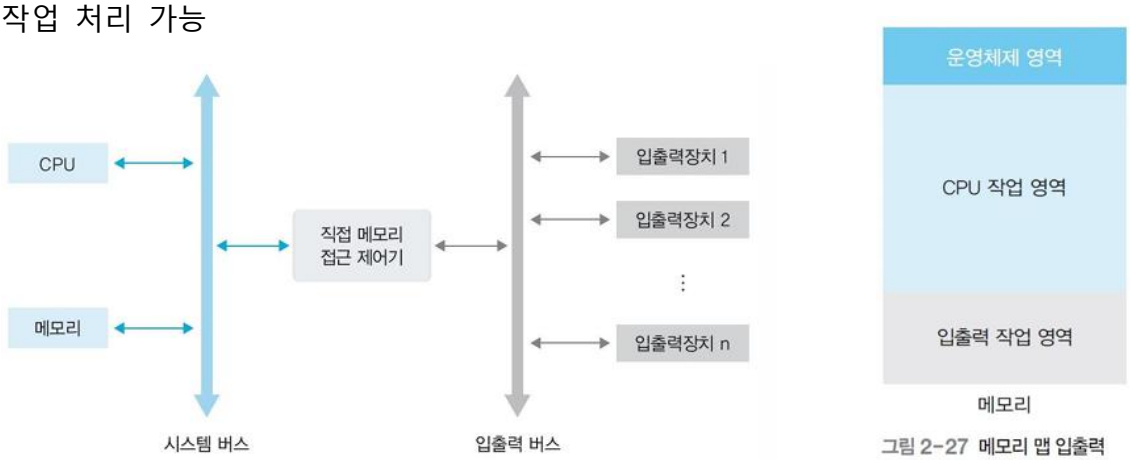


그림 2-27 메모리 맵 입출력

메모리 맵 입출력(Memory Mapped I/O; MMIO)

: 메모리의 일정 공간을 입출력에 할당하는 기법

사이클 훔치기

: CPU와 직접 메모리 접근이 동시에 메모리에 접근하면 보통 CPU가 메모리 사용 권한을 양보. CPU의 작업 속도보다 입출력장치의 속도가 느려서 직접 메모리 접근에 양보하는 것 (격차를 줄이기 위해)

멀티 프로세서 시스템(multi processor system)

: 컴퓨터 성능을 높이기 위해 프로세서를 여러 개 설치해서 사용하는 시스템. 프로세서마다 레지스터와 캐시를 가지며 모든 프로세서가 시스템 버스를 통해 메인메모리 공유.

장점-> 많은 작업 동시 실행 가능!

멀티코어(multi-core) 시스템

: 기존 시스템을 유지한 채 멀티 프로세싱 할 수 있게 하는 시스템. 하나의 chip에 CPU 핵심이 되는 core 여러 개 만들어 여러 작업 동시 처리! dual core는 CPU 주요 기능 담당하는 코어 2개, quad core는 무려 4개~

명령어 병렬 처리(instruction parallel processing)

: 하나의 코어에서 여러 개의 명령어를 동시에 처리

CPU 멀티스레드(multithread)

: 여러 개 스레드 동시에 처리하는 방법. thread란 CPU가 처리할 수 있는 작업 단위

현대 CPU

: 하나의 chip에 멀티코어와 명령어 병렬 처리 기능 한꺼번에 구현.

코어가 4개인데 명령어 병렬 처리 땜에 논리적인 코어 수 8개처럼 보임~

CPU 관련 통용 법칙

1. 무어 법칙(Moore's law)
   : CPU 속도가 24개월마다 2배 빨라진다는 법칙
   요즘은 CPU 처리 속도 올리는 대신 멀티코어 장착해서 CPU 여러 개 사용하는 것 같은 효과 얻음. 하나의 코어에서 여러 개 명령어 동시에 실행하는 멀티스레드 기술도 많이 사용함.

2. 암달 법칙(Amdahl's law)

   : 주변장치의 향상 없이 CPU 속도 2GHz에서 4GHz로 늘리더라도 컴퓨터 성능이 2배 빨라지지 않는다는 법칙

   CPU 속도를 올려도 메모리를 비롯한 주변장치가 CPU 발전 속도를 따라가지 못해서 컴퓨터 전반적인 성능은 저하. 싱글코어 대신 듀얼코어 사용하더라도 CPU 내 다른 부품 병목 현상으로 CPU 성능이 2배가 되지 않음

03 프로세스와 스레드

프로그램: 저장장치에 저장되어 있는 정적인 상태

프로세스: 실행을 위해 메모리에 올라온 동적인 상태

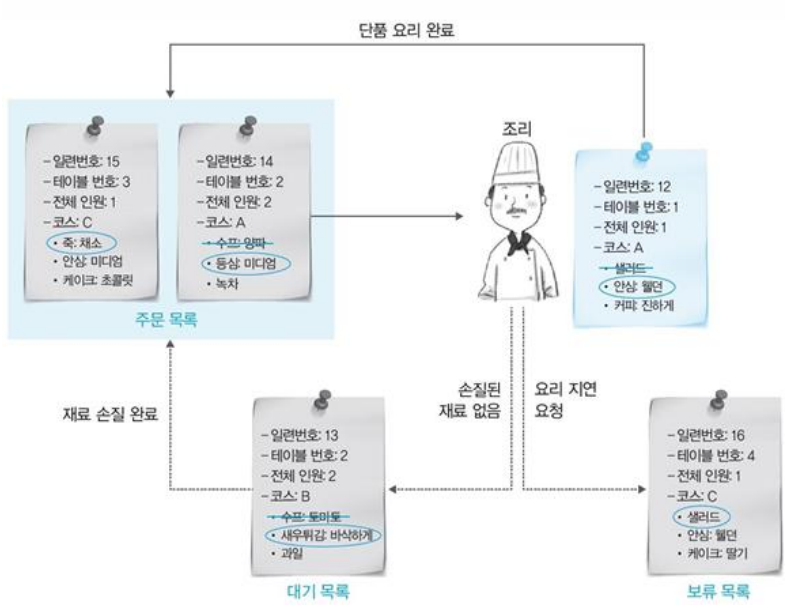
일괄 작업 방식

: 테이블 하나, 요리사 주문서 받은 순대로. (주문 목록 Queue로 처리)

현재 손님 식사 끝나야 다음 손님 받을 수 있어서 작업 비효율적

시분할 방식

: 요리사 1명이 시간 배분해서 여러 요리 동시에! (주문 목록 주문서 중 하나 가져다가. 모든 요리 제공되면 주문 목록에서 삭제)



프로세스 제어 블록(Process Control Bolco; PCB)

: 프로그램이 프로세스로 전환될 때 운체가 만드는 작업 지시서

어떤 프로그램이 프로세스가 된거는 운체로부터 프로세스 제어 블록 받았다는 의미!

PCB에 있는 대표적 3정보

- 프로세스 구분자(Process IDentification; PID): 각 프로세스 구분하는 구분자
- 메모리 관련 정보: 프로세스 메모리 위치 정보, 메모리 보호 위한 경계 레지스터와 한계 레지스터
- 각종 중간값: 프로세스가 사용했던 중간값

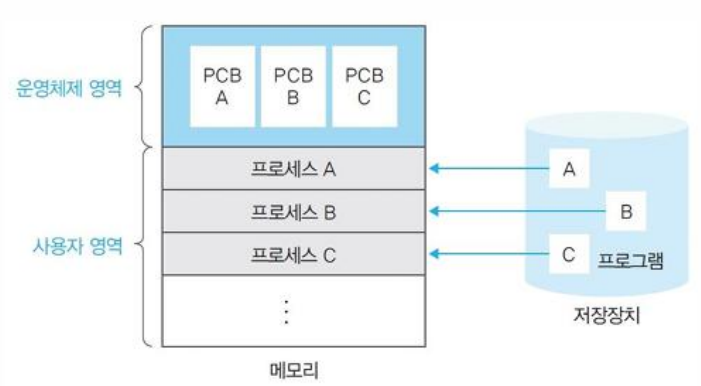


그림 3-7 프로그램이 메모리에 올라와 프로세스가 되는 과정

정의 3-1 프로세스와 프로그램의 관계

프로세스 = 프로그램 + 프로세스 제어 블록

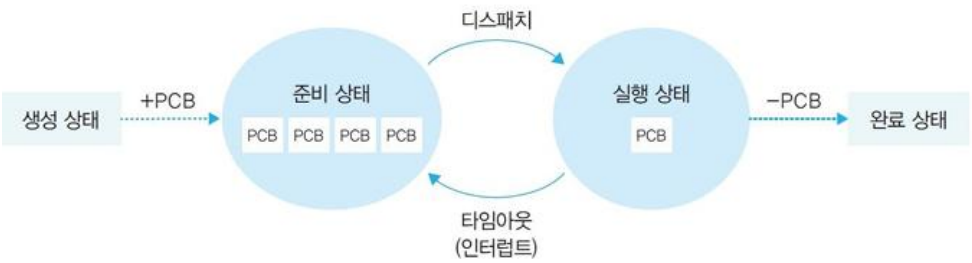
프로그램 = 프로세스 − 프로세스 제어 블록

프로그램이 프로세스가 됐다?->운체로부터 PCB 얻음!

프로세스 중료->해당 프로세스 제어 블록 폐기!

프로세스 4상태

- 생성상태(create status): 프로세스가 메모리에 올라와 실행 준비 완료한 상태. PCB 생성
- 준비상태(ready status): 생성된 프로세스가 CPU를 얻을 때까지 기다리는 상태
- 실행 상태(running status): 준비 상태에 있는 프로세스 중 하나가 CPU를 얻어 실제 작업 수행하는 상태
- 완료 상태(terminate status): 실행 상태의 프로세스가 주어진 시간 동안 작업을 마치면 진입하는 상태. PCB 사라짐



CPU 스케줄러(CPU scheduler): 준비 상태 프로세스 중 다음에 실행할 프로세스 선정

디스패치(dispatch): 준비 상태 프로세스 중 하나를 골라 실행 상태로 바꾸는 CPU 스케줄러의 작업

타임아웃(time out): 프로세스가 자신에게 주어진 하나의 타임 슬라이스 동안 작업을 끝내지 못해 다시 준비 상태로 돌아감

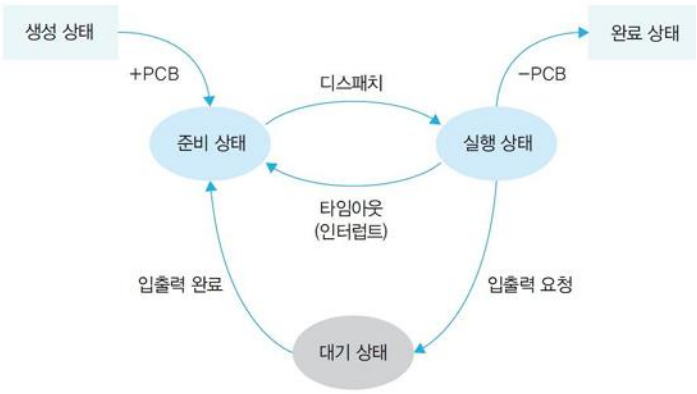


그림 3-9 대기 상태를 추가한 프로세스의 상태

대기 상태(blocking status): 실행 상태 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태로 작업효율을 높이기 위해 만듦

상태	설명
생성 상태	저장장치에 저장된 프로그램이 메모리로 올라와 실행되어 프로세스가 되는 상태로 커널 영역에 프로세스 제어 블록이 만들어진다. 생성된 후에는 준비 상태로 이동한다.
준비 상태	실행을 기다리는 모든 프로세스가 준비 큐에서 자기 차례를 기다리는 상태다. 실행될 프로세스를 CPU 스케줄러가 선택한다.
실행 상태	선택된 프로세스가 타임 슬라이스를 얻어 CPU를 사용하는 상태다. 작업을 마치면 완료 상태로 가고 작업을 끝내지 못하면 준비 상태로 되돌아간다.
대기 상태	실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태다. 입출력이 완료되면 준비 상태로 이동한다.
완료 상태	프로세스가 종료된 상태다. 사용하던 모든 자원은 반납되고, 메모리에서 지워지며, 프로세스 제어 블록은 폐기된다.

휴식 상태(pause status): 프로세스가 작업 일시적으로 쉬고 있는 상태

보류 상태(suspend status): 프로세스가 메모리에서 잠시 쫓겨난 상태 (=일시정지상태)

- 메모리 꽉 차서 일부 프로세스 메모리 밖으로 내보낼 때
- 프로그램에 오류 있어서 실행 미뤄야 할 때
- 바이러스와 같이 악의적인 공격 하는 프로세스라고 판단될 때
- 매우 긴 주기로 반복되는 프로세스라 메모리 밖으로 쫓아내도 큰 문제X
- 입출력을 기다리는 프로세스의 입출력이 계속 지연될 때

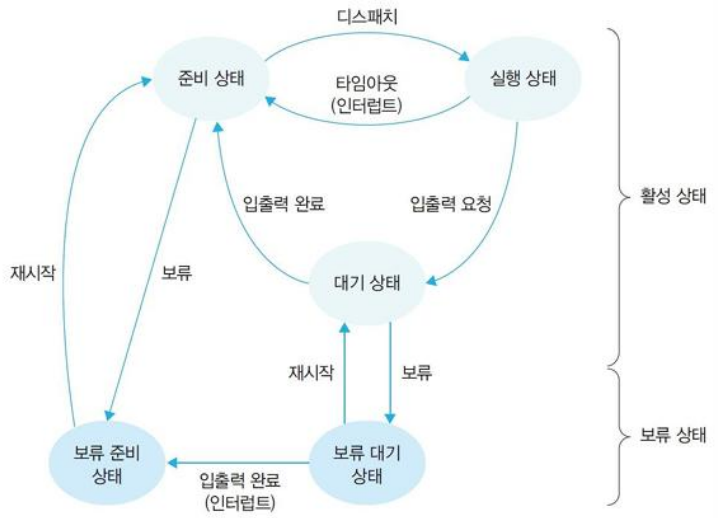


그림 3-11 보류 상태를 포함한 프로세스의 상태

포인터	프로세스 상태
프로세스 구분자	
프로그램 카운터	
프로세스 우선순위	
각종 레지스터 정보	
메모리 관리 정보	
할당된 자원 정보	
계정 정보	
PPID와 CPID	
⋮	

그림 3-12 프로세스 제어 블록의 구성

PCB: 프로세스 실행하는 데 필요한 중요한 정보 보관하는 자료구조. 프로세스는 고유의 프로세스 제어 블록 가짐! 생성 시 만들어지고 실행완료면 폐기

PCB 구성

포인터: 준비 상태나 대기 상태 큐 구현할 때 사용

프로세스 상태: 프로세스가 현재 어떤 상태에 있는지 나타내는 정보

프로세스 구분자: 운체 내에 있는 여러 프로세스 구현하기 위한 구분자

프로그램 카운터: 다음 실행될 명령어 위치를 가리키는 프로그램 카운터 값

프로세스 우선순위: 프로세스 실행 순서 결정하는 우선순위

각종 레지스터 정보: 프로세스가 실행되는 중 사용하던 레지스터 값

메모리 관리 정보: 프로세스가 메모리 어디에 있는지 나타내는 메모리 위치 정보, 메모리 보호를 위해 사용하는 경계 레지스터값+한계 레지스터 값 등

할당된 자원 정보: 프로세스 실행 위해 사용하는 입출력 자원이나 오픈 파일 등에 대한 정보

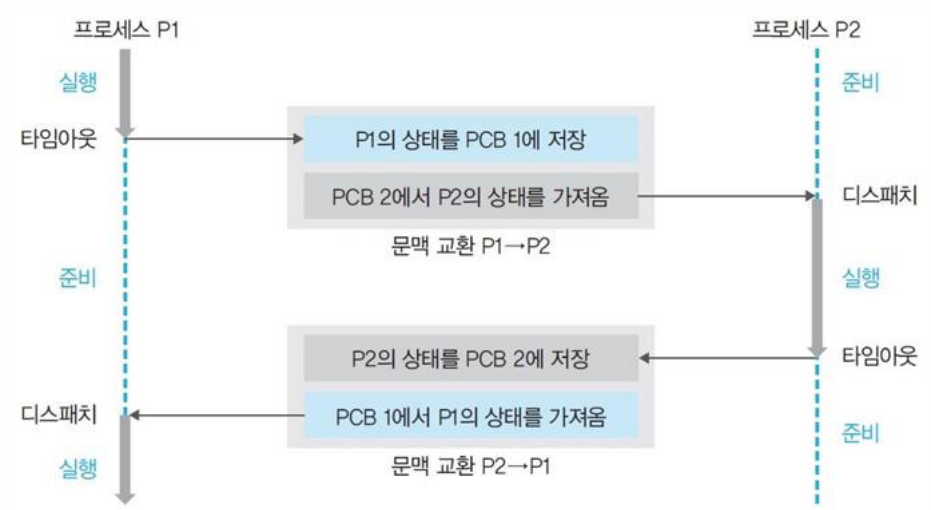
계정 정보: 계정 번호, CPU 할당 시간, CPU 사용 시간 등

부모 프로세스 구분자와 자식 프로세스 구분자: 부모 프로세스를 가리키는 PPID와 자식 프로세스 가리키는 CPID 정보

문맥 교환

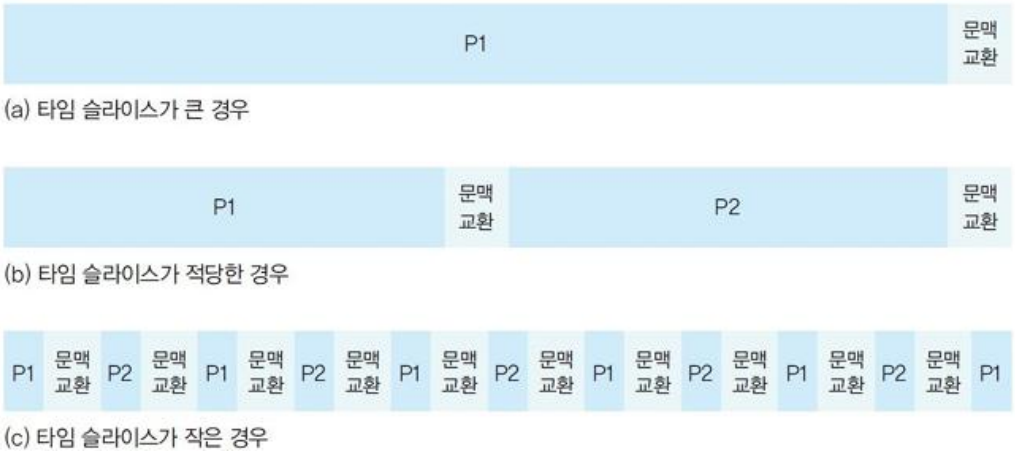
: CPU를 차지하던 프로세스가 나가고 새로운 프로세스를 받아들이는 것. 두 프로세스의 PCB 교환하는 작업

문맥 교환 절차

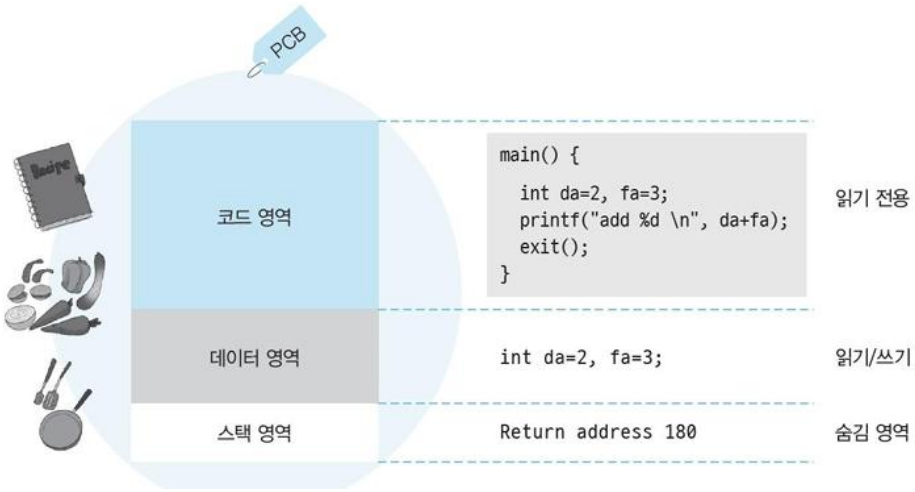


문맥 교환과 타임 슬라이스 크기

: 타임 슬라이스는 되도록 작게 설정하되 문맥 교환에 걸리는 시간 고려해서 적당한 크기로 설정하는 게 중요!



프로세스 구조



코드 영역

: 프로그램 본문이 기술된 곳. 프로그래머가 작성한 코드가 탑재되며 탑재된 코드는 읽기 전용으로 처리

데이터 영역

: 코드가 실행되면서 사용하는 변수나 파일 등 각종 데이터 모아둠. 데이터는 변하는 값이라서 기본적으로 읽/쓰 가능

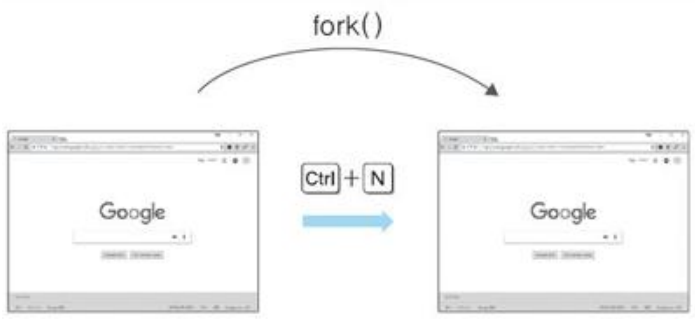
스택 영역

: 운체가 프로세스 실행하기 위해 필요한 데이터 모아둔 곳. 프로세스 내에서 함수 호출하면 함수 수행하고 원래 프로그램으로 돌아올 위치 저장하는 곳. 운체가 사용자 프로세스 작동하기 위해 유지하는 영역으로 사용자에게 안 보임

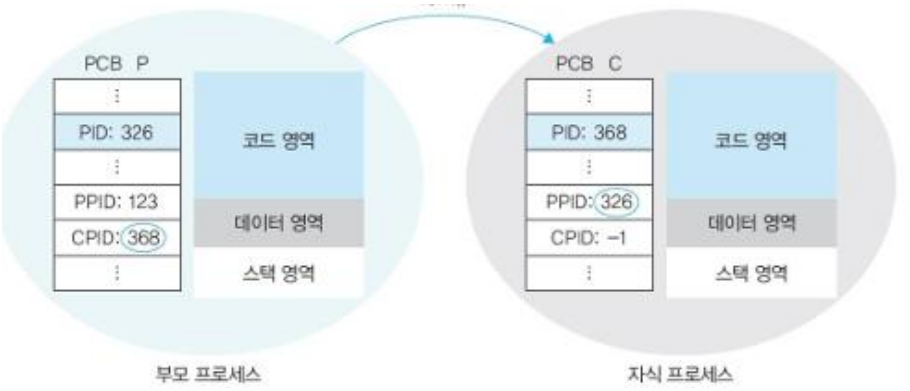
프로세스 생성과 복사

fork() 시스템 호출

: 실행 중인 프로세스로부터 새로운 프로세스 복사하는 함수. 실행 중인 프로세스랑 똑같은 프로세스 하나 더 만들어짐! (실행하던 거 부모프로세스, 생긴 거는 자식 프로세스)



fork() 시스템 호출 동작 과정



fork() 시스템 호출 해서 PCB 변경 내용!

- 프로세스 구분자(PID)
- 메모리 관련 정보
- 부모 프로세스 구분자(PPID), 자식 프로세스 구분자(CPID)

포크 시스템 호출 장점은?

- 프로세스 생성 속도 빠름
- 추가 작업 없이 자원 상속 가능
- 시스템 관리 효율적으로 가능!

<예시>

부모 프로세스 코드 실행돼서 fork()문 만나서 똑 같은 내용 자식 프로셋 생성!

이때 fork()문은 부.프에 0보다 큰 값 반환, 자.프에 0 반환! (만약 0보다 작은 값 반환하면 자.프가 생성 안된 것으로 여기고 Error 출력)

프로세스 전환

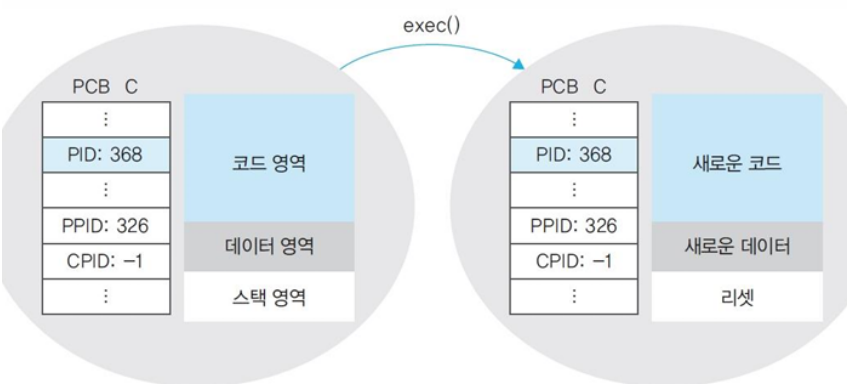
exex() 시스템 호출

: 기존 프로세스를 새로운 프로세스로 전환(재사용)하는 함수 (그대로 두고 구조 재활용)

exec() 시스템 호출 동작 과정

exec() 시스템 호출하면 코드 영역에 있는 기존 내용 지우고 새로운 코드로 바꿈!

데이터 영역이 새로운 변수로 채워지고 스택 영역은 리셋~ PCB 내용 중 프로세스 구분자, 부.프.구분자랑 자.프.구분자랑 메모리 관련 사항 등 안 변하지만! 프로그램 카운터 레지스터 값 비롯한 각종 레지스터와 사용한 파일 정보 모두 리셋

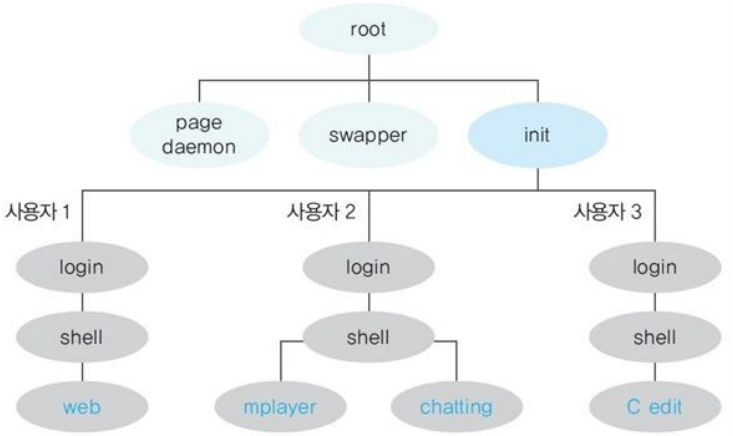


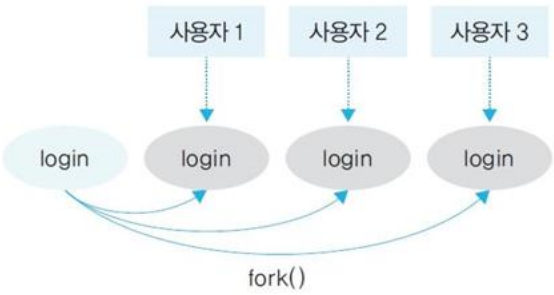
<예시>

부모 프로세스 fork()문 실행해서 자.프 생성하고 wait()문 실행해서 자.프 끝날 때까지 기다림. 새로 생성된 자.프는 부.프 코드랑 같음 ㅇㅇ exec() 시스템 호출 사용해서 새로운 프로세스로 전환해도 프로세스 구분자(PID, PPID, CPID) 안 변해서 프로세스 종류 후에도 부.프로 돌아올 수 있음

유닉스의 프로세스 계층 구조

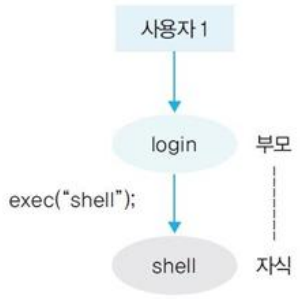
: 유닉스 모든 프로세스는 init 프로세스 자식 돼서 트리 구조!





프로세스 계층 구조 장점:

- 여러 작업 동시 처리 가능!
- 프로세스 재사용 용이!



- 자원 회수 쉬움! (프로세스 간 책임 관계 분명해져서 시스템 관리 수월)

고아 프로세스(orphan process): 부모 프로세스가 먼저 종료되어 돌아갈 곳X

좀비 프로세스(zombie process): 자식 프로세스가 종료되었는데도 부모 프로세스가 뒤처리 안 해서 발생

C에서는 exit() / return() 사용해서 자식 프로세스 작업 끝났다고 부모 프로세스한테 알림

스레드

: CPU 스케줄러가 CPU에 전달하는 일 하나(CPU가 처리하는 작업단위는 프로세스로부터 전달받은 스레드. 프로세스 코드에 정의된 절차 따라 CPU에 작업 요청하는 실행 단위))

- 운체 작업 단위는 프로세스 / CPU 작업 단위는 스레드

프로세스끼리는 약하게 연결, 스레드끼리는 강하게 연결!

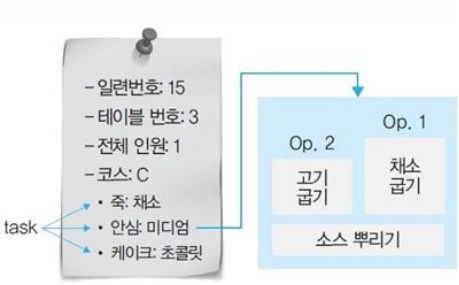
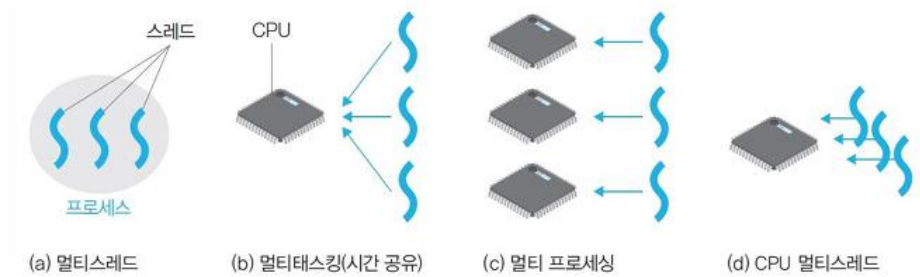


그림 3-27 작업(task)과 일(Op)의 차이

멀티태스크와 멀티스레드 차이

(a) 멀티태스크 (b) 멀티스레드

스레드 관련 용어



(a) 멀티스레드 (b) 멀티태스킹(시간 공유) (c) 멀티 프로세싱 (d) CPU 멀티스레드

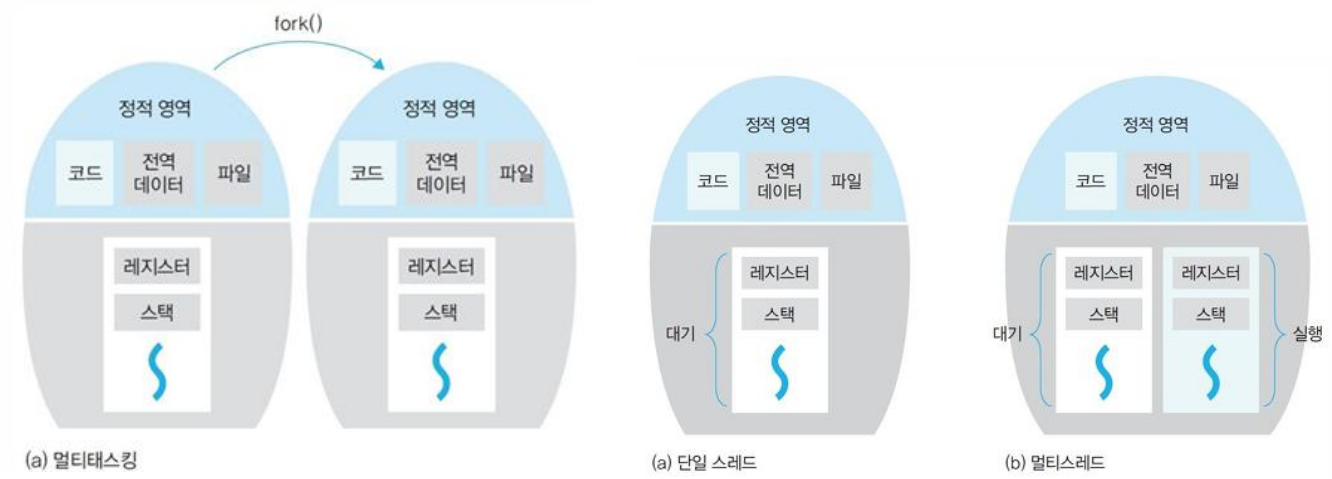
멀티스레드: 프로세스 내 작업을 여러 개 스레드로 분할해서 작업 부담 줄이는 프로세스 운영 기법

멀티태스킹: 운체가 CPU에 작업 줄 때 시간 잘게 나눠 배분하는 기법

멀티 프로세싱: 여러 개 CPU로 여러 개 스레드 동시 처리하는 작업 환경

CPU 멀티스레드: 한 번에 하나씩 처리해야 하는 스레드 잘게 쪼개서 동시 처리하는 명령어 병렬 처리 기법

멀티태스킹과 멀티스레드 구조



멀티스레드 장점

- 응답성 향상
- 자원 공유
- 효율성 향상
- 다중 CPU 지원

(a) 멀티스레드를 이용하는 인터넷 익스플로러

(b) 멀티태스킹을 이용하는 크롬

멀티스레드 단점

: 모든 스레드가 자원 공유하기 때문에 한 스레드 문제 생기면 전체 프로세스에 영향.

인터넷 익스플로러에서 여러 개 화면 동시에 띄웠는데 그중 하나에 문제 생기면 전체 종료됨

>멀티스레드 모델

커널 스레드: 커널이 직접 생성하고 관리하는 스레드

사용자 스레드: 라이브러리에 의해 구현된 일반적인 스레드

사용자 스레드(1 to N 모델)

: 사용자 프로세스 내 여러 개의 스레드가 커널의 스레드 하나와 연결

+라이브러리가 직접 스케줄링 하고 작업에 필요한 정보 처리해서 문맥 교환 필요X

-커널 스레드가 입출력 작업 위해 대기 상태 들가면 모든 사용자 스레드가 같이 대기!

-한 프로세스 타임 슬라이스를 여러 스레드가 공유해서 여러 개 CPU 동시에 사용X

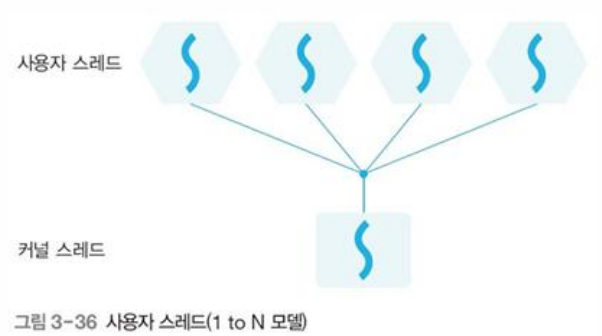


그림 3-36 사용자 스레드(1 to N 모델)

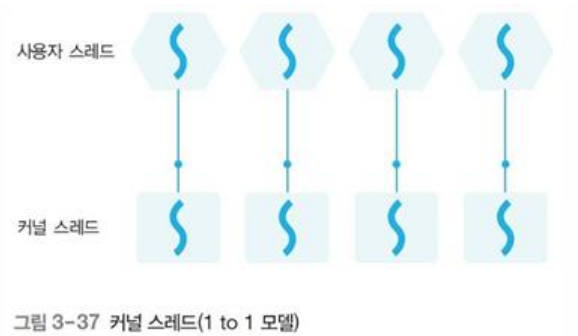


그림 3-37 커널 스레드(1 to 1 모델)

커널 스레드(1 to 1 모델)

: 하나의 사용자 스레드가 하나의 커널 스레드와 연결

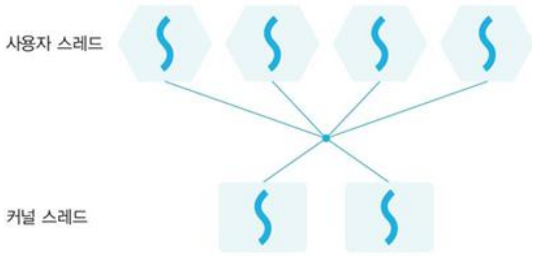
+독립적으로 스케줄링 돼서 특정 스레드가 대기 상태 들가도 다른 스레드는 계속 작업 할 수 있음.

+커널 레벨에서 모든 작업을 지원해서 멀티 CPU 사용 가능!

+하나의 스레드가 대기 상태에 있어도 다른 스레드는 작업 계속 가능!

+커널 기능 사용해서 보안에 강하고 안정적으로 작동

-문맥 교환 때 오버헤드 땜에 느리게 작동



멀티레벨 스레드(M to N 모델)

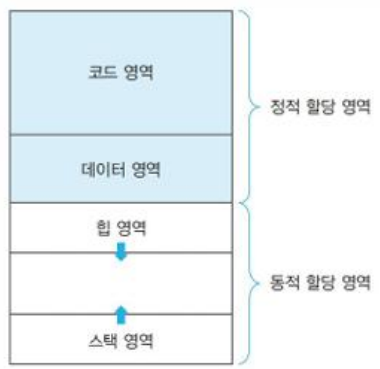
: 사용자 스레드와 커널 스레드 혼합한 방식

+커널 스레드가 대기 상태 들가면 다른 커널 스레드가 대신 작업 해서 사용자 스레드보다 유연한 작업 처리

-커널 스레드 같이 사용해서 문맥 교환 시 오버헤드 있어서 사용자 스레드만큼 빠르진X

+빠르게 움직여야 되는 스레드는 사용자 스레드로, 안정적으로 움직여야 하는 스레드는 커널 스레드로 작동!

프로세스 구조



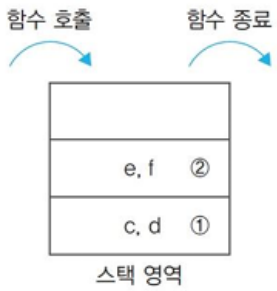
코드 영역: 프로그램 본체 있는 곳

데이터 영역: 프로그램 사용하려고 정의한 변수/데이터 있는 곳

스택 영역과 힙 영역: 프로세스 실행되는 동안 만들어지는 영역! 크기 늘었다 줄었다 하는 동적 할당 영역

스택 영역

: 스레드 작동하는 동안 추가되거나 삭제되는 동적 할당 영역. 스레드가 진행됨에 따라 커지거나 작아지기도



힙 영역

: 프로그램이 실행되는 동안 할당되는 변수 영역. 포인터, malloc() 함수, calloc() 함수 등은 메모리를 효율적으로 사용하기 위해 만들어진 것으로, 어쩌다 한 번 쓰는 큰 배열을 처음부터 선언하고 끝까지 놔두는 일이 없어야!

exit() 시스템 호출

: 작업 종료 알려주는 시스템 호출. exit() 함수 선언함으로써 부모 프로세스는 자식 프로세스가 사용하던 자원을 빨리 거둬 갈 수 있음. exit() 함수는 전달하는 인자를 확인해서 자식 프로세스가 어떤 상태로 종료되었는지를 알려줌! 인자가 0이면 정상 종료고 -1이면 비정상 종료

wait() 시스템 호출

: 자식 프로세스가 끝나기를 기다렸다가 자식 프로세스가 종료되면 다음 문장 실행하는 시스템 호출. 부모 프로세스와 자식 프로세스 간 동기화에도 사용

wait() 함수 응용

: 전면 프로세스에서는 셸이 wait() 함수 사용하기 때문에 자식 프로세스가 끝날 때까지 다음 명령어 입력받을 수X. 후면 프로세스에선 셸이 wait() 함수 사용 안 해서 다음 명령어 받아들일 수 있음! ㅎㅎ

```
01  sleep 100    // 전면 프로세스
02  sleep 100&   // 후면 프로세스
```

04 CPU 스케줄링

CPU 스케줄러

: 운체에서 관리자 역할 담당. 여러 프로세스 상황 고려해 CPU와 시스템 자원 배정 결정

고수준 스케줄링(high level scheduling)

: 시스템 내 전체 작업 수 조절. 어떤 작업을 시스템이 받아들일지/거부할지 결정

시스템 내에서 동시 실행 가능한 프로세스 총 개수 정해짐. (=장기 스케줄링(long-term scheduling), 작업 스케줄링(job scheduling), 승인 스케줄링(admission scheduling))

저수준 스케줄링(low level scheduling)

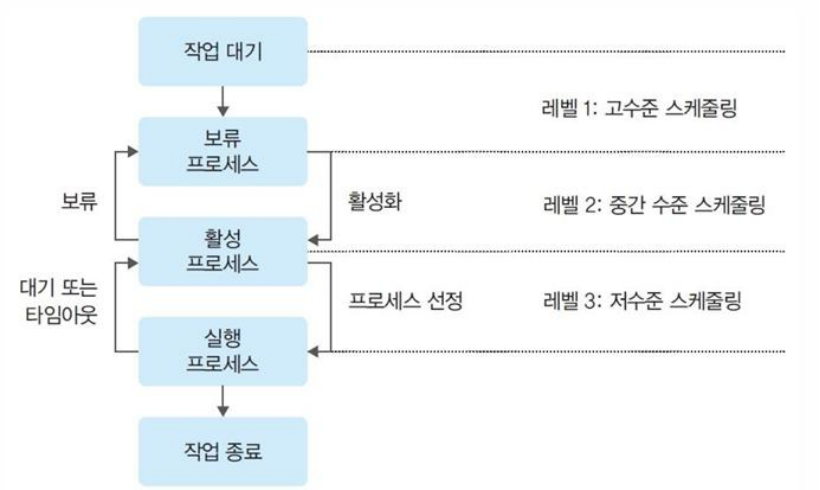
: 어떤 프로세스에 CPU 할당할지 어떤 프로세스를 대기 상태로 보낼지 결정. 아주 짧은 시간에 일어남 (=단기 스케줄링(short-term scheduling))

중간 수준 스케줄링(middlelevel scheduling)

: 중지(suspend)와 활성화(active)로 전체 시스템의 활성화된 프로세스 수 조절해서 과부하 막음

일부 프로세스를 중지상태로 옮겨 나머지 프로세스가 원만히 작동하도록 지원. 저수준 스케줄링이 원만하게 이루어지도록 완충하는 역할

고수준 스케줄링	전체 시스템의 부하를 고려하여 작업을 시작할지 말지 결정
중간 수준 스케일링	시스템에 과부하가 걸려서 전체 프로세스 수를 조절해야 한다면 이미 활성화된 프로세스 중 일부를 보류 상태로 보냄
저수준 스케줄링	실제 작업 수행



CPU 스케줄링 목적

공평성	모든 프로세스가 자원을 공평하게 배정받아야 하며, 자원 배정 과정에서 특정 프로세스가 배제되어서는 안 됨
효율성	시스템 자원이 유휴 시간 없이 사용되도록 스케줄링을 하고, 유휴 자원을 사용하려는 프로세스에는 우선권을 주어야 함
안정성	우선순위를 사용하여 중요 프로세스가 먼저 작동하도록 배정함으로써 시스템 자원을 점유하거나 파괴하려는 프로세스로부터 자원을 보호해야 함
확장성	프로세스가 증가해도 시스템이 안정적으로 작동하도록 조치해야 하며 시스템 자원이 늘어나는 경우 이 혜택이 시스템에 반영되게 해야 함
반응 시간 보장	응답이 없는 경우 사용자는 시스템이 멈춘 것으로 가정하기 때문에 시스템은 적절한 시간 안에 프로세스의 요구에 반응해야 함
무한 연기 방지	특정 프로세스의 작업이 무한히 연기되어서는 안 됨

선점형스케줄링(preemptive scheduling)

: 운체가 필요하다 판단하면 실행 상태에 있는 프로세스 작업 중단시키고 새로운 작업 시작할 수 있는 방식

하나의 프로세스가 CPU 독점할 수 없기 땜에 빠른 응답 시간 요구하는 대화형 시스템/시분할 시스템에 적합. 대부분 저수준 스케줄러가 선점형 스케줄링 방식 사용함

비선점형스케줄링(non-preemptive scheduling)

: 어떤 프로세스가 실행 상태에 들가서 CPU 사용하면 그 프로세스 종료되거나 자발적으로 대기 상태 들어가기 전까지 계속 실행되는 방식

+선점형보다 스케줄러 작업량 적고 문맥 교환에 의한 낭비도 적음

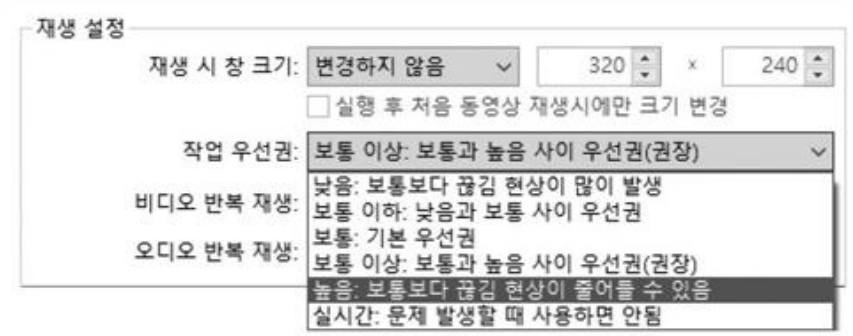
-CPU 사용 시간이 긴 프로세스 땜에 CPU 사용 시간 짧은 여러 프로세스가 오랫동안 기다리게 돼서 전체 시스템 처리율 떨어짐. 과거 일괄 작업 시스템에서 사용하던 방식

구분	선점형 스케줄링	비선점형 스케줄링
작업 방식	실행 상태에 있는 작업을 중단시키고 새로운 작업을 실행할 수 있다.	실행 상태에 있는 작업이 완료될 때까지 다른 작업이 불가능하다.
장점	프로세스가 CPU를 독점할 수 없어 대화형이나 시분할 시스템에 적합하다.	CPU 스케줄러의 작업량이 적고 문맥 교환의 오버헤드가 적다.
단점	문맥 교환의 오버헤드가 많다.	기다리는 프로세스가 많아 처리율이 떨어진다.
사용	시분할 방식 스케줄링에 사용된다.	일괄 작업 방식 스케줄링에 사용된다.
중요도	높다.	낮다.

프로세스 우선순위

: 커널 프로세스 > 일반 프로세스

시스템엔 다양한 우선순위 프로세스 공존하고, 우선순위 높은 프로세스가 CPU 먼저 더 오래 차지. 시스템에 따라 높은 숫자가 높은 우선순위 나타낼 수도 있고 반대도 ㅇㅇ

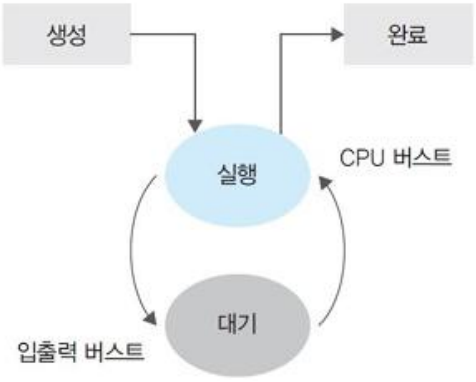


CPU 집중 프로세스

: 수학 연산과 같이 CPU 많이 사용하는 프로세스로, CPU burst가 많은 프로세스

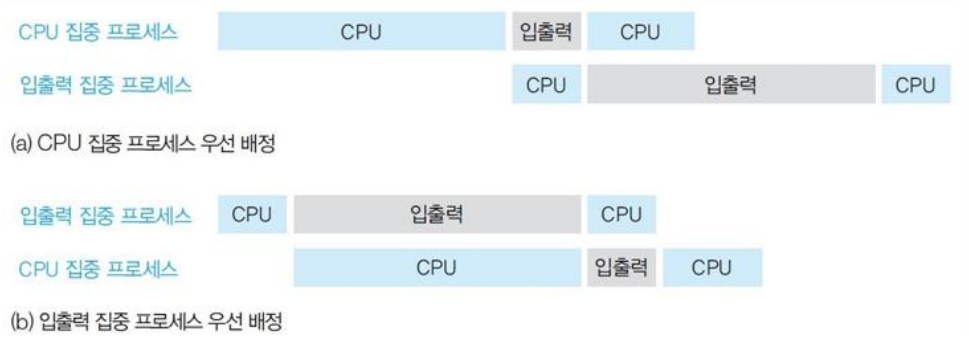
입출력 집중 프로세스

: 저장장치에서 데이터 복사하는 일과 같이 입출력 많이 사용하는 프로세세스로, I/O burst가 많은 프로세스



우선 배정

: 스케줄링 할 때 입출력 집중 프로세스(I/O bound process) 우선순위를 CPU 집중 프로세스(CPU bound process)보다 높이면 시스템 효율이 향상!



전면 프로세스

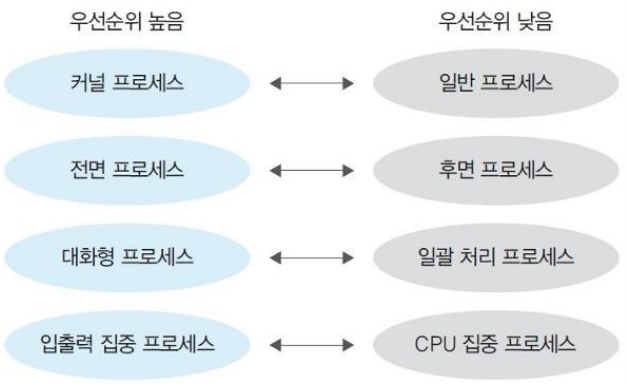
: GUI를 사용하는 운체에서 화면 맨앞에 놓인 프로세스. 현재 입력과 출력을 사용하는프로세스. 사용자와 상호작용이 가능(=상호작용 프로세스)

후면 프로세스

: 사용자와 상호작용X. 사용자 입력 없이 작동(=일괄 작업 프로세스)

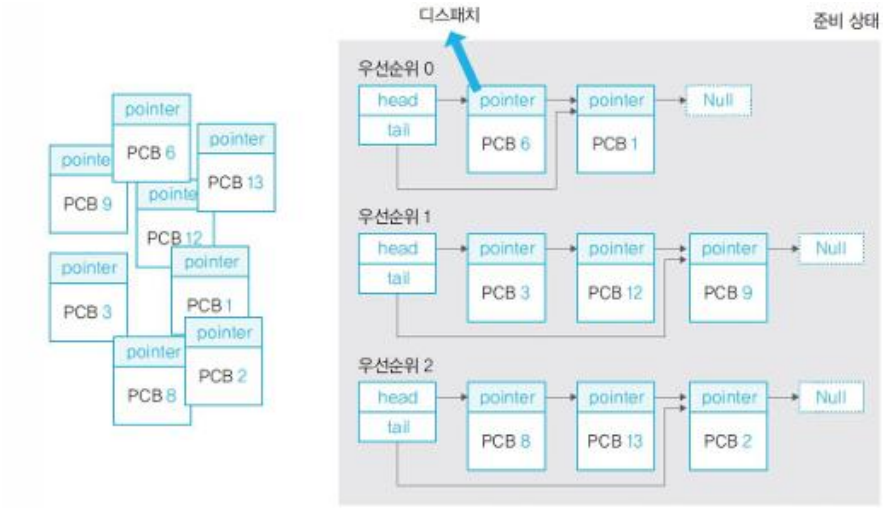
우선순위는 전면 프로세스 > 후면 프로세스

CPU 스케줄링 시 고려사항



준비 상태 다중 큐(multiple queue)

: 프로세스는 준비 상태에 들어올 때마다 자신의 우선순위에 해당하는 큐의 마지막(tail)에 삽입. CPU 스케줄러는 우선순위가 가장 높은큐(0번 큐)의 맨 앞에 있는프로세스 6에 CPU 할당



프로세스 우선순위 배정 방식

고정 우선순위 방식(static priority)

: 운체가 프로세스에서 우선순위 부여하면 프로세스 끝날 때까지 바뀌지 않는 방식.

프로세스가 작업하는 동안 우선순위 안 바꿔서 구현 쉬움! 근데 시스템 상황이 시시각각 변하는데 우선순위 고정하면 시스템 변화에 대응 어려워서 작업 효율 떨어짐

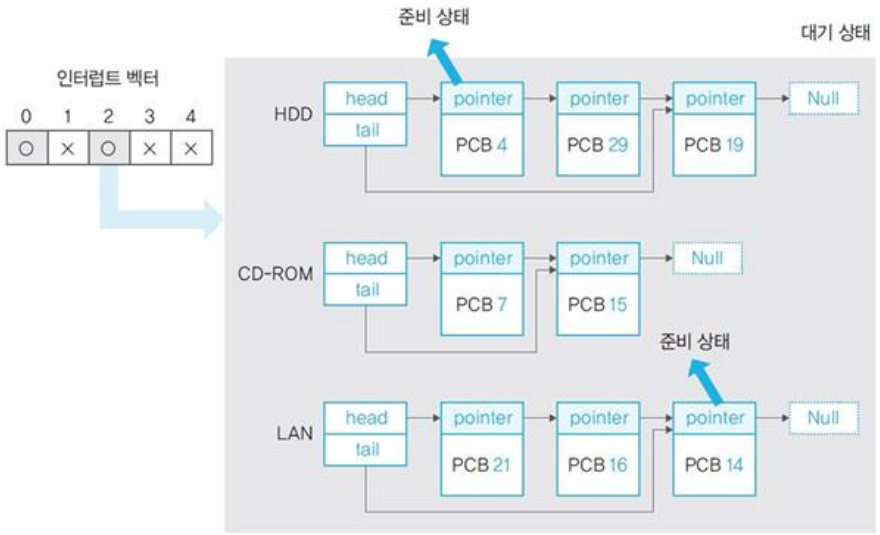
변동 우선순위 방식(dynamic priority)

: 프로세스 생성시 부여받은 우선순위가 프로세스 작업 중간에 변하는 방식.

구현 어려워도 시스템 효율성 높임!

대기 상태 다중 큐

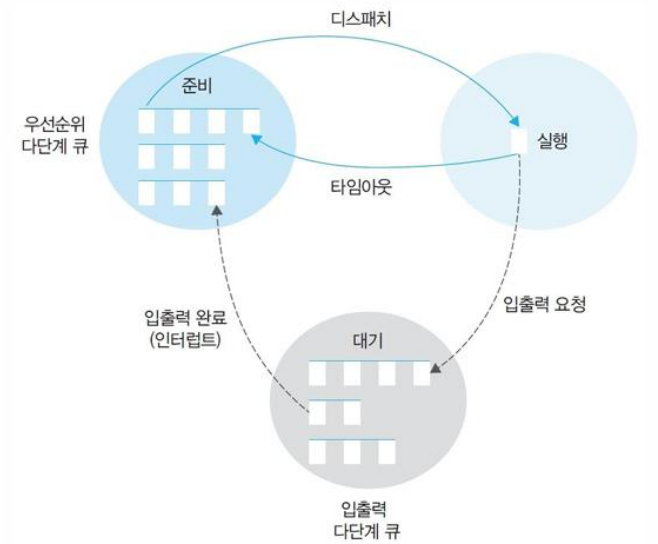
: 시스템 효율 높이기 위해 같은 입출력 요구한 프로세스끼리 모아둠



다중 큐 비교

준비 큐: 한 번에 하나의 프로세스 꺼내서 CPU를 할당

대기 큐: 여러 개 프로세스 제어 블록 동시에 꺼내서 준비 상태로 옮김. 대기 큐에서 동시에 끝나는 인터럽트 처리 위해 interrupt vector 자료구조 사용

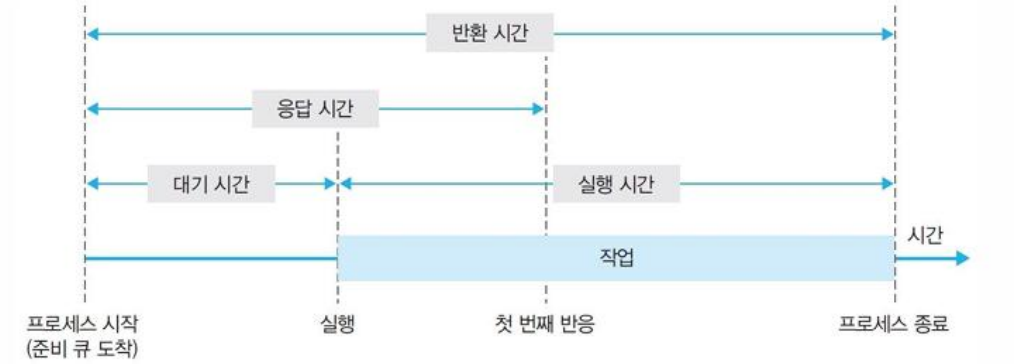


스케줄링 알고리즘 종류

구분	종류
비선점형 알고리즘	FCFS 스케줄링, SJF 스케줄링, HRN 스케줄링
선점형 알고리즘	라운드 로빈 스케줄링, SRT 스케줄링, 다단계 큐 스케줄링, 다단계 피드백 큐 스케줄링
둘 다 가능	우선순위 스케줄링

스케줄링 알고리즘의 평가 기준

- CPU 사용률: 전체 시스템 동작 시간 중 CPU가 사용된 시간을 측정하는 방법. 가장 이상적인 수치는 100%지만 실제 여러 가지 이유로 90%도 못 미침
- 처리량: 단위 시간당 작업 마친 프로세스 수. 수치가 클수록 좋은 알고리즘!
- 대기 시간: 작업 요청한 프로세스가 작업 시간 전까지 대기하는 시간. 짧을수록 좋음!
- 응답 시간: 프로세스 시작 후 첫 번째 출력or반응 나올 때까지 걸리는 시간. 짧을수록 좋음!
- 반환 시간: 프로세스 종료돼서 사용하던 자원 모두 반환하는 데까지 걸리는 시간. (대기 시간 + 실행 시간)



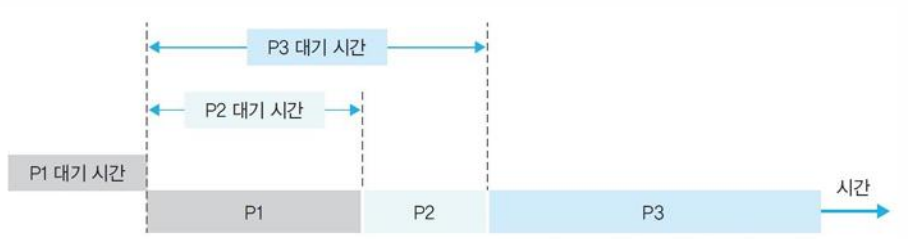
대기 시간: 프로세스가 생성된 후 실행되기 전까지 대기하는 시간

응답시간: 첫 작업을 시작한 후 첫번째 출력(반응)이 나오기까지 시간

실행시간: 프로세스 작업이 시작된 후 종료되기까지 시간

반환시간: 대기시간 포함해서 실행이 종료될 때까지 시간

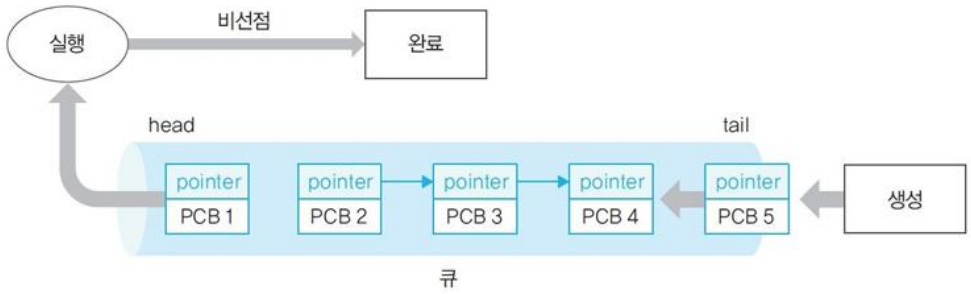
평균 대기 시간 (모든 프로세스 대기 시간 합 / 프로세스 수)



FCFS(First Come First Served) 스케줄링의 동작 방식

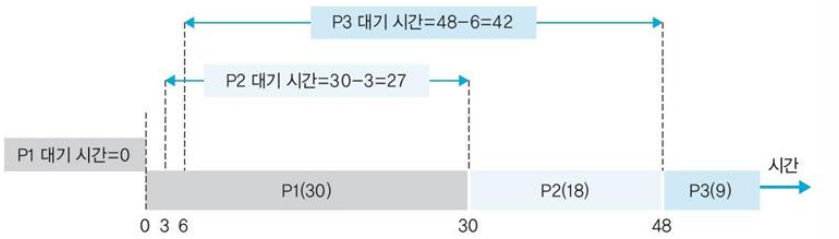
: 준비 큐에 도착한 순서대로 CPU 할당하는 비선점형 방식 (배열로 구현)

한 번 실행되면 그 프로세스 끝나야 다음 프로세스 실행 가능. 큐가 하나여서 모든 프로세스는 우선순위 동일



FCFS 스케줄링 성능

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9



※ 평균 대기 시간
(0+27+42)÷3=23밀리초

FCFS 스케줄링 평가

: 처리 시간 긴 프로세스가 CPU 차지하면 다른 프로세스는 하염없이...기다려서 효율성X

특히 현재 작업 중인 프로세스가 입출력 작업 요청하면 CPU 쉬는 시간 많아져서 효율X

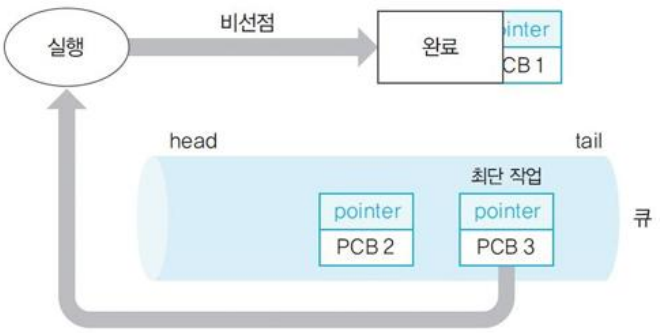
(콘보이 효과: CPU를 많이 필요로 하지 않는 프로세스들이 CPU를 오랫동안 사용하는 프로세스가 끝나기를 기다리는 현상)

SJF(Shortest Job First) 스케줄링 동작 방식

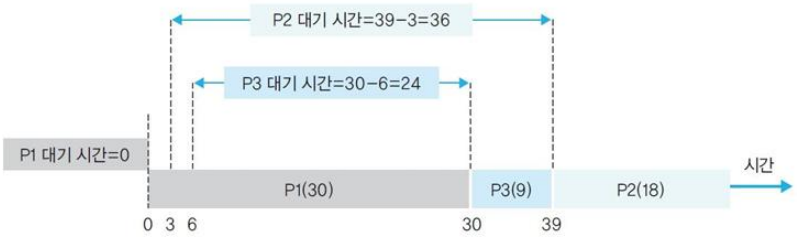
: 준비 큐에 있는 프로세스 중에서 실행시간이 가장 짧은 작업부터 CPU 할당하는 비선점형 방식 (=최단작업우선스케줄링)

=>이론상 존재하며 구현 불가함. (길이를 전부 측정할 수 없기 때문)

콘보이효과를 완화해서 시스템 효율성 높임



도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9



※ 평균 대기 시간
(0+24+36)÷3=20밀리초

SJF 스케줄링 평가

: 운체가 프로세스 종료 시간을 정확히 예측 어려움. 작업 시간이 길단 이유만으로 자꾸만 뒤로 밀려서 공평성 떨어짐. 아사(starvation)현상

에이징(aging) =>구현 불가 ㅇㅇ

: starvation 현상 완화 방법. 프로세스가 양보할 수 있는 상한선 정하는 방식

프로세스가 자신 순서 양보할 때마다 나이 한 살씩 먹어 최대 몇 살까지 양보하게 규정

HRN(Highest Response Ratio Next) 스케줄링 동작 방식

: SJF 스케줄링에서 발생하는 아사 현상 해결하기 위해 만들어진 비선점형 알고리즘

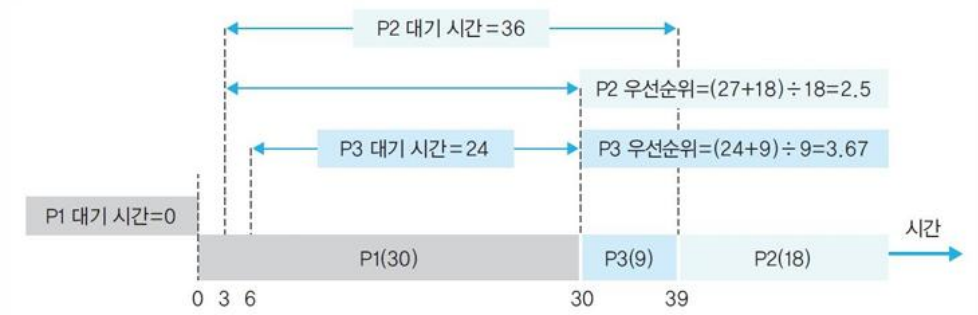
(=최고 응답률 우선 스케줄링). 서비스 받기 위해 기다린 시간과 CPU 사용 시간 고려해서 스케줄링 하는 방식. 프로세스 우선순위 결정 기준은~

$$\text{우선순위} = \frac{\text{대기 시간} + \text{CPU 사용 시간}}{\text{CPU 사용 시간}}$$

HRN 스케줄링 성능

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

※ 평균 대기 시간
(0+24+36)÷3=20밀리초



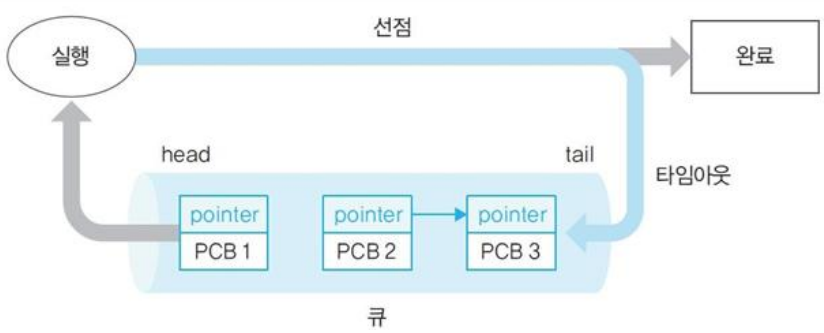
HRN 스케줄링 평가

: 실행 시간 짧은 프로레스 우선순위를 높게 설정하면서 대기 시간 고려해서 아사 현상 완화. 대기 시간 긴 프로세스 우선순위 높여서 CPU 할당받을 확률 높임 (공평성 위배돼서 많이 사용X)

라운드로빈(Round Robin) 스케줄링 동작 방식

: 한 프로세스가 할당받은 시간(타임 슬라이스) 동안 작업 하다 작업 완료 못하면 준비 큐 맨 뒤로 가서 자기 차례 기다리는 방식

선점형 알고리즘 중 가장 단순하고 대표적! 프로세스들이 작업 완료까지 계속 순환하며 실행

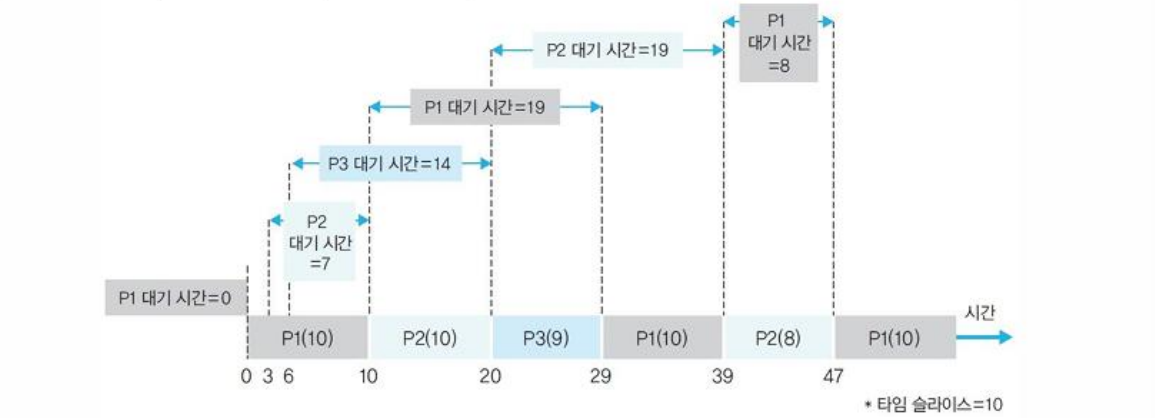


라운드 로빈 스케줄링 성능

도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

※ 총 대기 시간
0(P1)+7(P2)+14(P3)+19(P1)+19(P2)+8(P1)=67밀리초

※ 평균 대기 시간
67÷3=22.33밀리초

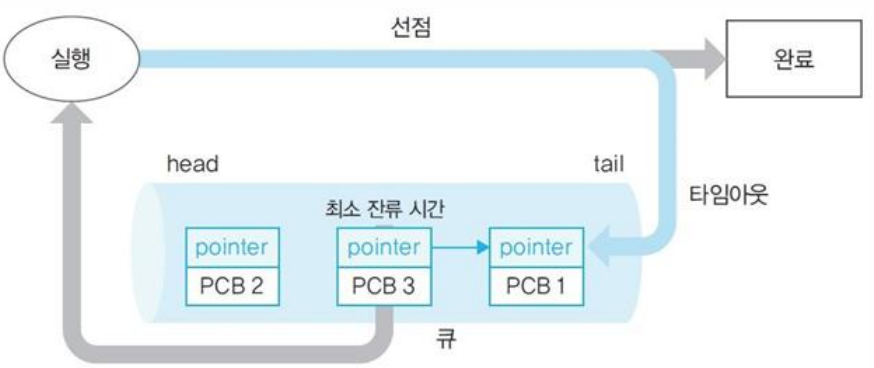


라운드 로빈 스케줄링 평가

: 라운드 로빈스케줄링과 FCFS 스케줄링 평균 대기시간이 같다면 라운드로빈 스케줄링이 더 비효율적임 (라운드로빈 스케줄링 같은 선점형방식에는 문맥교환 시간이 추가)

SRT(Shortest Remaining Time) 스케줄링의 동작 방식

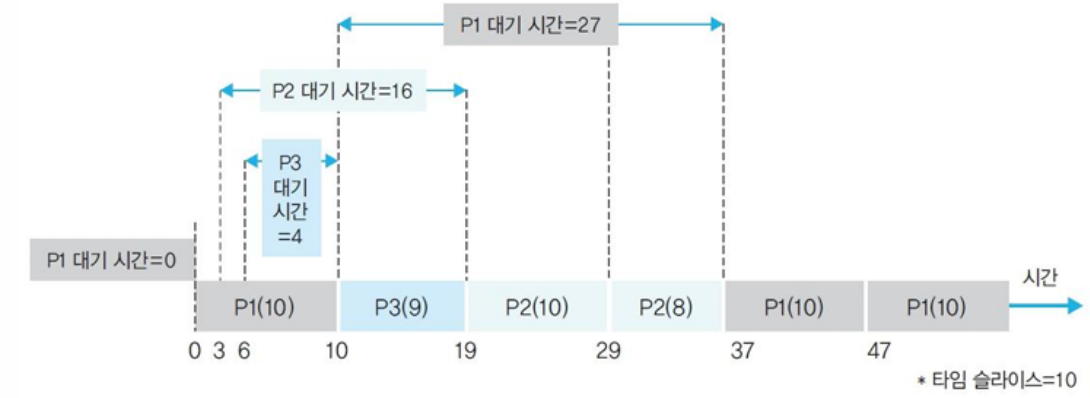
: 기본적으로 라-로 스케줄링 사용하지만 CPU 할당받을 프로세스 선택할 때 남아 있는 작업 시간이 가장 적은 프로세스를 선택



도착 순서	도착 시간	작업 시간
P1	0	30
P2	3	18
P3	6	9

※ 총 대기 시간
0(P1)+4(P3)+16(P2)+27(P1)=47밀리초

※ 평균 대기 시간
47÷3=15.66밀리초



SRT 스케줄링 평가

: 현재 실행중인 프로세스와 큐에 있는 프로세스의 남은 시간을 주기적으로 계산하고, 남은 시간이 더 적은 프로세스와 문맥교환을 해야해서 SJF스케줄링에 없는 작업이 추가됨

운체가 프로세스 종료 시간 예측하기 어렵고, 아사 현상 일어나서 잘 사용X

우선순위(priority) 스케줄링 동작 방식

: 프로세스 중요도에 따른 우선순위 반영한 스케줄링 알고리즘

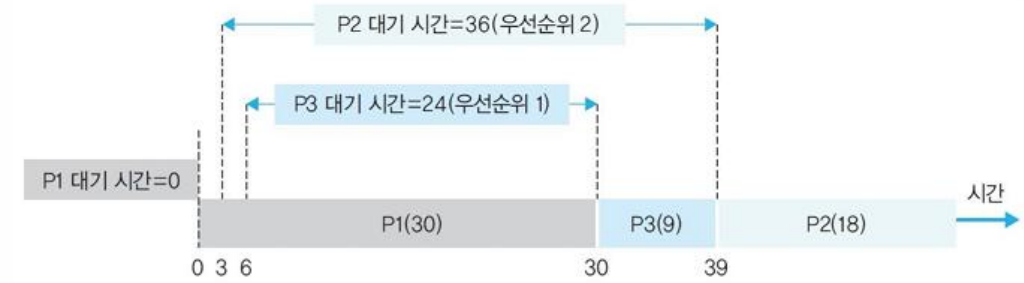
우선순위 적용

: 우선순위는 비선점형 방식+선점형 방식에 모두 적용 가능!

- (비) SJF 스케줄링: 작업 시간 짧은 프로레스에 높은 우선순위 부여
- (비) HRN 스케줄링: 작업 시간 짧거나 대기 시간 긴 프로세스에 높은 우선순위 부여
- (선) SRT 스케줄링: 남은 시간이 짧은 프로세스에 높은 우선순위 부여

표 4-5 프로세스의 우선순위

도착 순서	도착 시간	작업 시간	우선순위
P1	0	30	3
P2	3	18	2
P3	6	9	1



고정 우선순위 알고리즘

: 한 번 우선순위 부여받으면 종료까지 우선순위 고정~ (단순 구현. 시시각각 변하는 시스템 상황 반영 못해서 효율성X)

변동 우선순위 알고리즘

: 일정 시간마다 우선순위 변해서 우선순위 새로 계산하고 이를 반영. (복잡, 시스템 상황 반영해서 효율적인 운영 가능)

우선순위 스케줄링 평가

: 준비 큐에 있는 프로세스 순서 무시하고 우선순위 높은 프로세스에 먼저 CPU 할당해서 공평성 위배하고 아사 현상 일으킴. 준비 큐에 있는 프로세스 순서 무시하고 프로세스 우선순위 매번 바꿔야 해서 오버헤드 발생! (시스템 효율성 떨어짐)

다단계큐(MLQ; Multi-Level Queue) 스케줄링 동작 방식

: 우선순위에 따라 준비 큐 여러 개 사용. 프로세스는 운체로부터 부여받은 우선순위에 따라 해당 우선순위 큐에 삽입. 우선순위는 고정형 우선순위 사용함.

상단 큐에 있는 모든 프로세스 작업 끝나야 다음 우선순위 큐 작업 시작됨

다단계 피드백 큐(MLFQ;Multi-Level Feedback Queue) 스케줄링 동작 방식

: 프로세스가 CPU 한 번씩 할당받아 실행마다 프로세스 우선순위 낮춰서 다단계 큐에서 우선순위 낮은 프로세스 실행이 연기되는 문제 완화.

우선순위 낮아지더라도 커널 프로세스가 일반 프로세스 큐에 삽입X.

우선순위에 따라 time slice크기 다름. 우선순위 낮아질수록 CPU 얻을 확률 적어짐. 그래서 한번 CPU 잡을 때 많이 작업하라고 낮은 우선순위 time slice 크게 함.

마지막 큐(우선순위 가장 낮은) 프로세스는 무한대 time slice 얻음. 마지막 큐는 들어온 순대로 작업 마치는 FCFS 스케줄링 방식으로 동작.

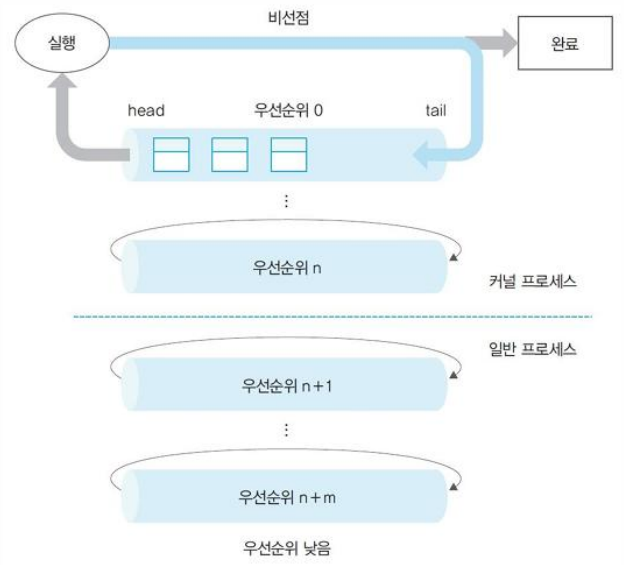


그림 4-25 다단계 큐 스케줄링의 동작

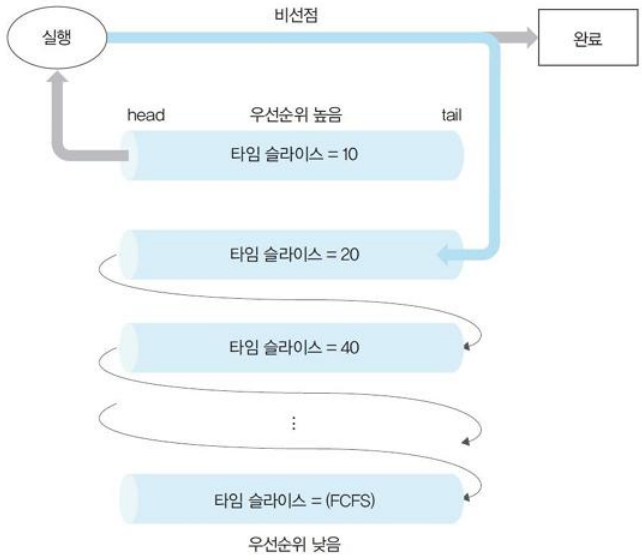


그림 4-26 다단계 피드백 큐 스케줄링의 동작