3장 프로세스와 스레드

프로그램

: 저장장치에 저장되어 있는 정적인 상태이다.

프로세스

: 실행을 위해 메모리에 올라온 동적인 상태이다.

일괄 작업 방식의 요리 (ex)

: 일괄 작업 시스템은 한 번에 하나의 작업만 처리하는 것

ex) 레스토랑에 하나의 테이블이 존재, 요리사는 순서대로 주문을 처리

손님의 식사가 끝나야지 다음 손님을 받을 수 있어 작업 효율이 떨어짐.

이때 ,주문목록을 받는데 큐로 처리함.

*큐 : 먼저 들어온 데이터가 먼저 빠져나가는 자료 구조이다.

스택 : 먼저 들어온 데이터가 마지막에 빠져나가는 자료 구조이다.

시분할 방식의 요리 (ex)

: CPU가 시간을 쪼개어 여러 프로세스에 적당히 배분하여 작업을 처리함.

ex) 요리사 1명이 시간을 분배하여 여러가지 요리를 동시에 하는 방식

요리사는 주문 목록에 있는 주문서를 하나 받아 요리함. 모든 요리가 제공되면 주문 목록에서 삭제함

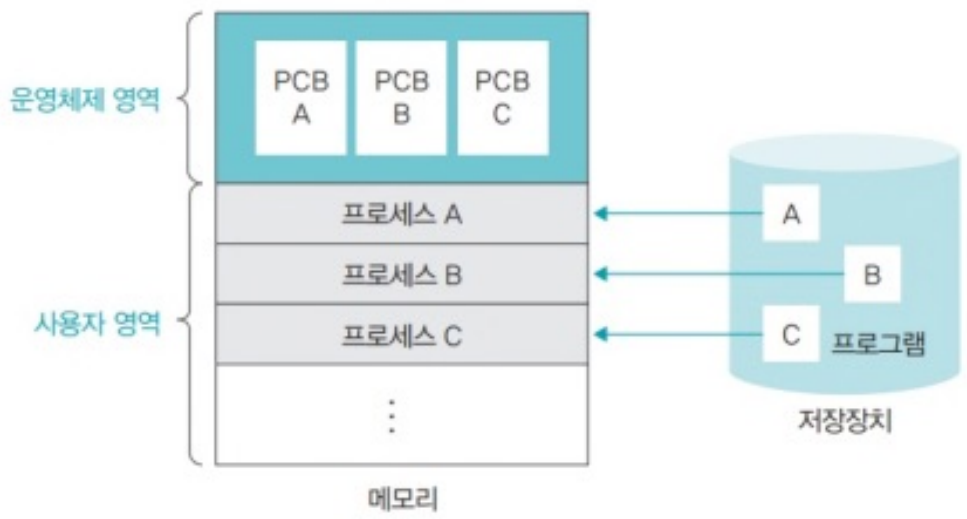
프로세스 제어 블록 (PCB)

: 운영체제가 해당 프로세스를 위해 관리하는 자료 구조

프로세스 구분자 : 각 프로세스를 구분하는 구분자

메모리 관련 정보 : 프로세스의 메모리 위치 정보

각종 중간값 : 프로세스가 사용했던 중간값



-프로그램이 메모리에 올라와 프로세스가 되는 과정-

하나의 프로세스를 실행하려면

프로세스 구분자, 메모리 관련 정보, 프로그램 카운터와 각종 레지스터 같은 중간값을 관리해야 하며, 이러한 정보를 관리하는 데이터구조가

프로세스 제어 블록이다.

프로그램이 프로세스가 되려면 메모리에 올라오는 것과 동시에

프로세스 제어 블록이 생성되어야 한다.

프로세스 제어 블록은 운영체제가 해당 프로세스를 위해 관리하는 데이터구조이기 때문에 운영체제 영역에 만들어진다. 또한 프로세스가 종료되면 프로세스가 메모리에서 삭제되고 프로세스 제어 블록도 폐기된다.

프로세스와 프로그램의 관계

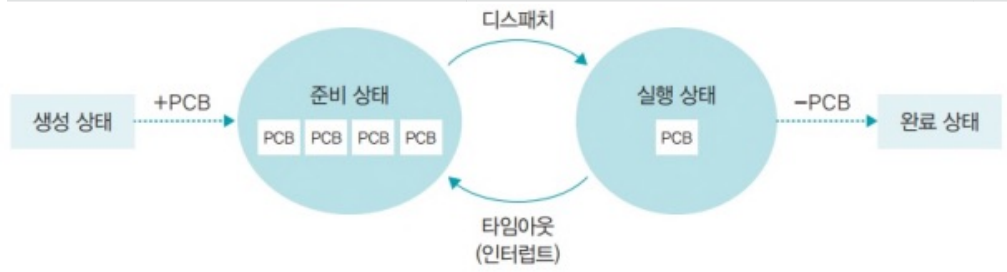
프로그램이 프로세스가 된다는 것은 운영체제로부터 프로세스 제어 블록을 얻는다는 뜻이다.

프로세스가 종료된다는 것은 해당 프로세스 제어 블록이 폐기된다는 뜻이다.

프로세스 = 프로그램 + 프로세스 제어 블록

프로그램 = 프로세스 - 프로세스 제어 블록

프로세스의 상태



생성 상태 : 프로세스가 메모리에 올라와 실행 준비를 완료한 상태이다.
프로세스를 관리하는 데 필요한 프로세스 제어 블록이 생성된다.

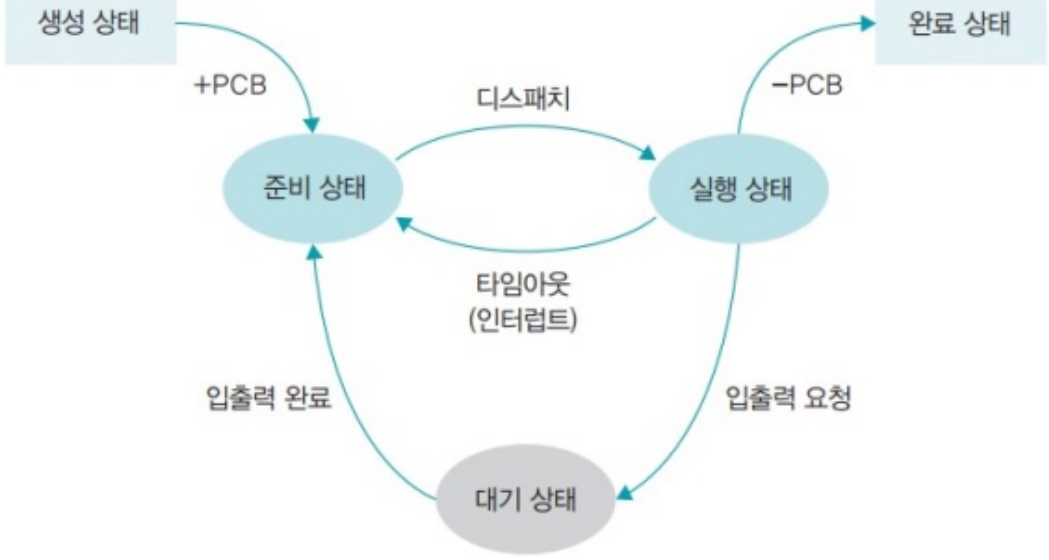
준비 상태 : 생성된 프로세스가 CPU를 얻을 때까지 기다리는 상태이다.

실행 상태 : 준비 상태에 있는 프로세스 중 하나가 CPU를 얻어 실제 작업을 수행하는 상태

완료 상태 : 실행 상태의 프로세스가 주어진 시간 동안 작업을 마치면 진입하는 상태 (프로세스 제어 블록이 사라진 상태)

디스패치 : 준비 상태의 프로세스 중 하나를 골라 실행 상태로 바꾸는 CPU 스케줄러의 작업

타임아웃 : 프로세스가 자신에게 주어진 하나의 타임 슬라이스 동안 작업을 끝내지 못하면 다시 준비 상태로 돌아가는 것



1. 생성 상태

프로그램 메모리에 올라오고 운영체제로부터 프로세스 제어 블록을 할당받은 상태

생성된 프로세스는 바로 실행되는 것이 아니라 준비 상태에서 자기 순서를 기다리며, 프로세스 제어 블록도 같이 준비 상태로 옮겨짐

2. 준비 상태

실행 대기 중인 모든 프로세스가 자기 순서를 기다리는 상태

프로세스 제어 블록은 준비 큐에 기다리며 CPU 스케줄러에 의해 관리

CPU 스케줄러는 준비 상태에서 큐를 몇 개 운영할지, 큐에 있는 어떤 프로세스의 프로세스 제어 블록을 실행 상태로 보낼지 결정

3. 실행 상태

프로세스가 CPU를 할당받아 실행되는 상태

실행 상태에 있는 프로세스는 자신에게 주어진 시간, 즉 타임 슬라이스 동안만 작업할 수 있음. 그 시간을 다 사용하면 timeout(PID)가 실행되어 프로세스가 정상 종료된다. 실행 상태에 있는 프로세스가 입출력을 요청하면 CPU는 실행 상태에 있는 프로세스가 입출력을 요청하면 CPU는 입출력 관리자에게 입출력을 요청하고 block(PID)를 실행한다. block(PID)는 입출력이 완료될 때까지 작업을 진행할 수 없기 때문에 해당 프로세스를 대기 상태로 옮기고 CPU 스케줄러는 새로운 프로세스를 실행 상태로 가져온다.

4. 대기 상태

실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태

대기 상태의 프로세스는 입출력 장치별로 마련된 큐에서 기다리다가 완료되면 인터럽트가 발생하고, 대기 상태에 있는 여러 프로세스 중 해당 인터럽트로 깨어날 프로세스를 찾는데 이것이 wakeup(PID)

wakeup(PID)로 해당 프로세스의 프로세스 제어 블록이 준비 상태로 이동

5. 완료 상태

프로세스가 종료되는 상태

코드와 사용했던 데이터를 메모리에서 삭제하고 프로세스 제어 블록을 폐기

정상적인 종료는 간단히 exit()로 처리

오류나 다른 프로세스에 의해 비정상적으로 종료되는 강제 종료를 만나면

디버깅하기 위해 종료 직전의 메모리 상태를 저장장치로 옮기는데 이를

코어 덤프(core dump)라고 함.

상태	설명	작업
생성 상태	프로그램을 메모리에 가져와 실행 준비가 완료된 상태이다.	메모리 할당, 프로세스 제어 블록 생성
준비 상태	실행을 기다리는 모든 프로세스가 자기 차례를 기다리는 상태이다. 실행될 프로세스를 CPU 스케줄러가 선택한다.	dispatch(PID): 준비→실행
실행 상태	선택된 프로세스가 타임 슬라이스를 얻어 CPU를 사용하는 상태이다. 프로세스 사이의 문맥 교환이 일어난다.	timeout(PID): 실행→준비 exit(PID): 실행→완료 block(PID): 실행→대기
대기 상태	실행 상태에 있는 프로세스가 입출력을 요청하면 입출력이 완료될 때까지 기다리는 상태이다. 입출력이 완료되면 준비 상태로 간다.	wakeup(PID): 대기→준비
완료 상태	프로세스가 종료된 상태이다. 사용하던 모든 데이터가 정리된다. 정상 종료인 exit와 비정상 종료인 abort를 포함한다.	메모리 삭제, 프로세스 제어 블록 삭제

6. 휴식 상태

프로세스가 작업을 일시적으로 쉬고 있는 상태

유닉스에 프로그램을 실행하는 도중에 Ctrl+z 키를 누르면 볼 수 있음

종료 상태가 아니기 때문에 원할 때 다시 시작할 수 있는 상태

7. 보류 상태

프로세스가 메모리에서 잠시 쫓겨난 상태

프로세스가 보류 상태가 될 경우

-메모리가 꽉 차서 일부 프로세스를 메모리 밖으로 내보낼 때

-프로그램에 오류가 있어서 실행을 미루어야 할 때

-바이러스와 같이 악의적인 공격을 하는 프로세스라고 판단될 때

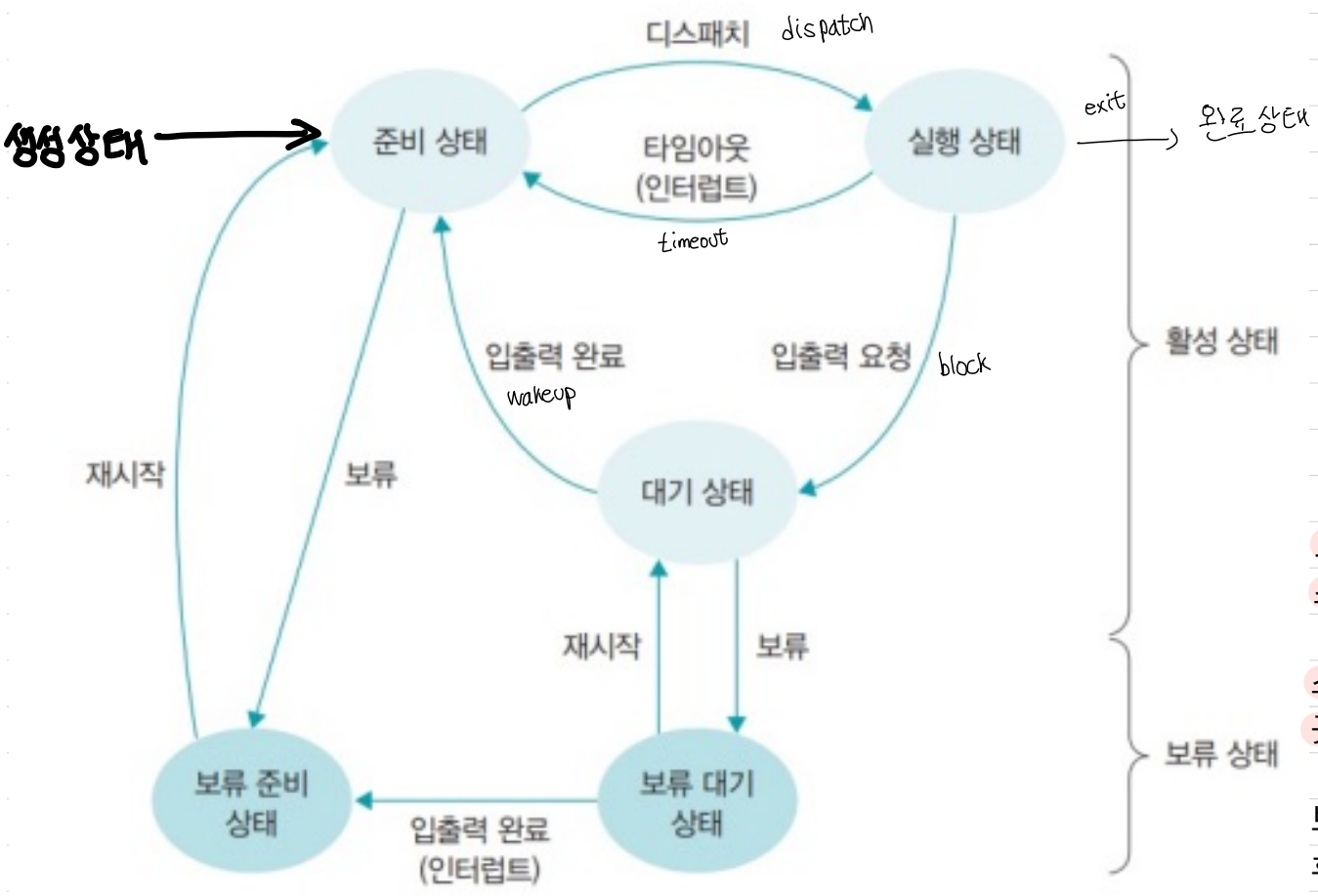
-매우 긴 주기로 반복되는 프로세스가 메모리 밖으로 쫓아내고 큰 문제가 없을 때

-입출력을 기다리는 프로세스의 입출력이 계속 지연될 때

-보류 상태를 포함한 프로세스의 상태

```
zoch@ubuntu:~$ sleep 100
^Z
[1]+  Stopped                 sleep 100
zoch@ubuntu:~$ jobs
[1]+  Stopped                 sleep 100
zoch@ubuntu:~$ bg
[1]+ sleep 100  &
zoch@ubuntu:~$ jobs
[1]+  Running                 sleep 100 &
```

프로세스 정지 상태 화면



보류 상태에 들어간 프로세스는 메모리 밖으로 쫓겨나 스왑 영역에 보관된다.

스왑 영역은 메모리에서 쫓겨난 데이터를 임시로 보관되는 곳이다.

보류 상태는 스왑 영역에 있는 상태이고

휴식 상태는 프로세스가 메모리에 있으나 멈춘 상태이다.

보류 상태는 대기 상태에서 옮겨진 보류 대기 상태와 준비 상태에서 옮겨진 보류 준비 상태로 구분되며, 각 상태에서 재시작하면 원래의 활성 상태로 들어간다. 또한 보류 대기 상태에서 입출력이 완료되면 활성 상태가 아닌 보류 준비 상태로 옮겨진다.

프로세스 제어 블록

: 프로세스를 실행하는 데 필요한 중요한 정보를 보관하는 자료 구조

프로세스는 고유의 프로세스 제어 블록을 가짐

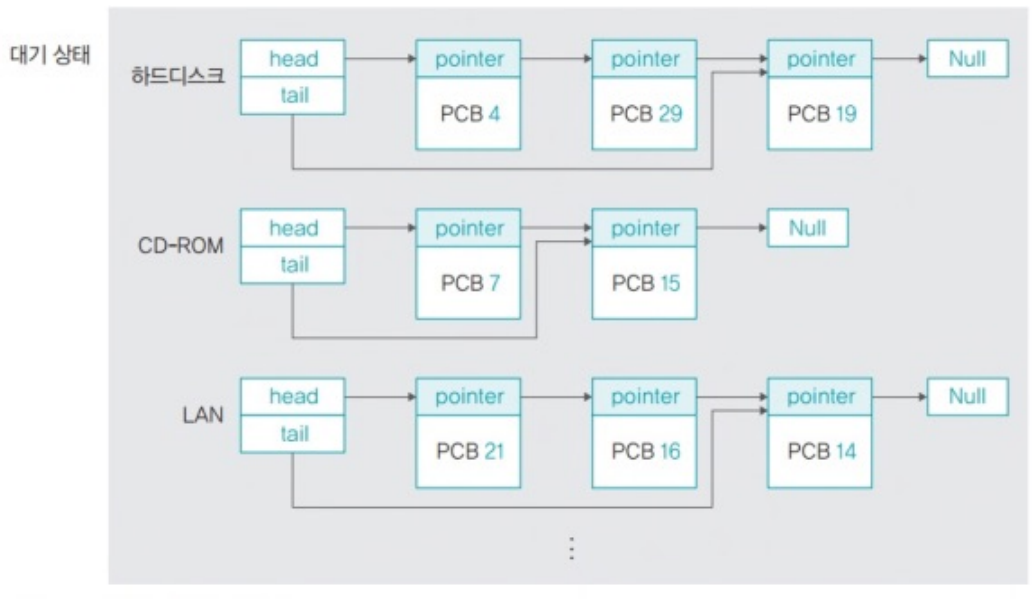
프로세스 생성 시 만들어져야 프로세스가 실행을 완료하면 폐기

프로세스 제어 블록의 구성

1. 포인터 : 준비 상태나 대기 상태의 큐를 구현할 때 사용
2. 프로세스 상태 : 프로세스가 현재 어떤 상태에 있는지를 나타내는 정보
3. 프로세스 구분자 : 운영체제 내에 여러 프로세스를 구현하기 위한 구분자
4. 프로그램 카운터 : 다음에 실행될 명령어의 위치를 가리키는 프로그램 카운터의 값
5. 프로세스 우선순위 : 프로세스의 실행 순서를 결정하는 우선순위
6. 각종 레지스터 정보 : 프로세스가 실행되는 중에 사용하던 레지스터의 값
7. 메모리 관리 정보 : 프로세스가 메모리의 어디에 있는지 나타내는 메모리 위치 정보, 메모리 보호를 위해 사용하는 경계 레지스터 값과 한계 레지스터 값 등
8. 할당된 자원 정보 : 프로세스를 실행하기 위해 사용하는 입출력 자원이나 오픈 파일 등에 대한 정보
9. 계정 정보 : 계정 번호, CPU 할당 시간, CPU 사용 시간 등
10. 부모 프로세스 구분자와 자식 프로세스 구분자 : 부모 프로세스를 가리키는 PPID와 자식 프로세스를 가리키는 CPID 정보

포인터의 역할

: 대기 상태에는 같은 입출력을 요구한 프로세스끼리 연결할 때 포인터 사용

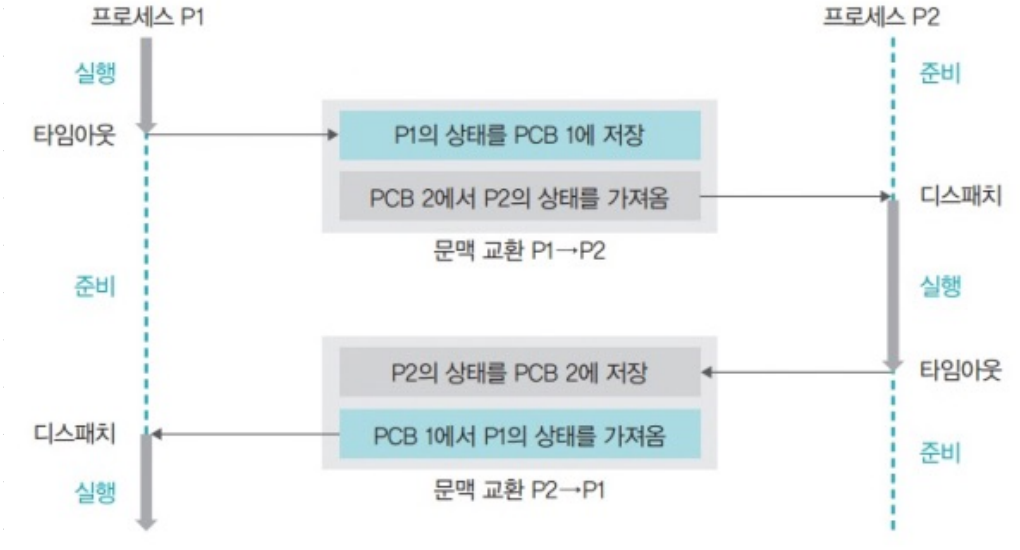


문맥 교환

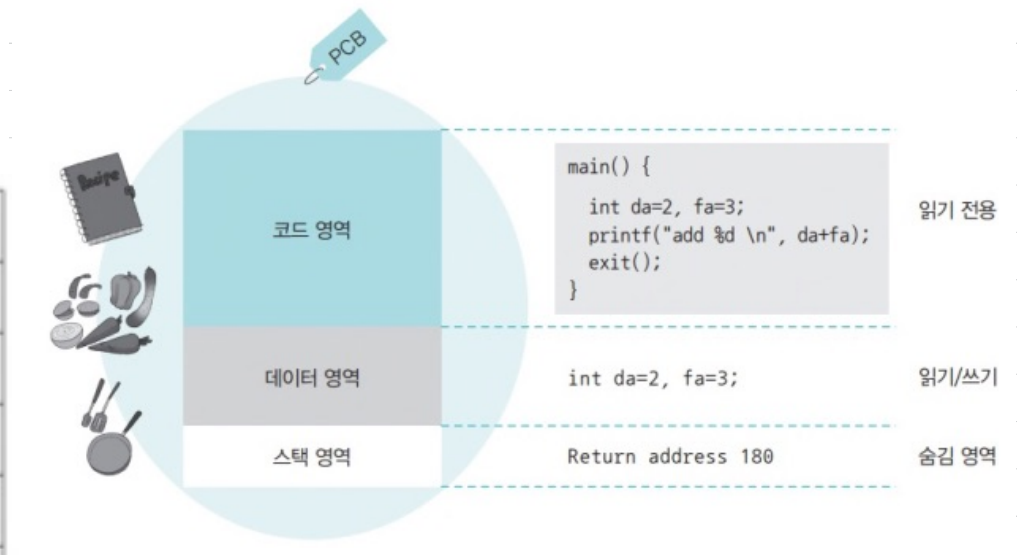
: CPU를 차지하던 프로세스가 나가고 새로운 프로세스를 받아들이는 작업

한 프로세스가 자신에게 주어진 시간을 다 사용하면 발생하고, 인터럽트가 걸렸을 때도 발생한다.

실행 상태에서 나가는 프로세스 제어 블로그에는 지금까지의 내용을 저장하고, 반대로 실행 상태로 들어오는 프로세스 제어 블록의 내용으로 CPU가 세팅됨.



프로세스의 구조



코드 영역

프로그램의 본문이 기술된 곳

프로그래머가 작성한 코드가 탑재되면 탑재된 코드는 읽기 전용으로 처리됨

데이터 영역

코드가 실행되면 사용하는 변수나 파일 등의 각종 데이터를 모아놓은 곳

데이터는 변화는 값이기 때문에 이곳의 내용은 기본적으로 읽기와 쓰기가 가능

스택 영역

운영체제가 프로세스를 실행하기 위해 부수적으로 필요한 데이터를 모아둔 곳

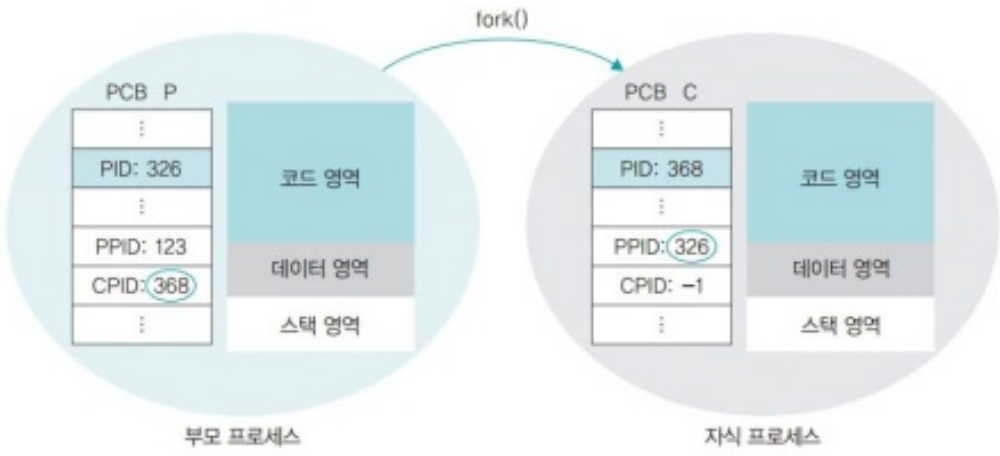
프로세스 내에서 함수를 호출하면 함수를 수행하고 원래 프로그램으로 되돌아올 위치를 이 영역에 저장

운영체제가 사용자의 프로세스를 작동하기 위해 유지하는 영역이므로 사용자에게는 보이지 않음

fork() 시스템 호출

: 실행 중인 프로세스로부터 새로운 프로세스를 복사하는 함수이다.

이때 실행하던 프로세스는 부모 프로세스, 새로 생긴 프로세스는 자식 프로세스로서 부모-자식 관계가 된다.



fork() 시스템 호출을 하면 프로세스 제어 블록을 포함한 부모 프로세스 영역의 대부분이 자식 프로세스에 복사되어 똑같은 프로세스가 만들어짐

-변경 되는 부분

프로세스 구분자

메모리 관련 정보

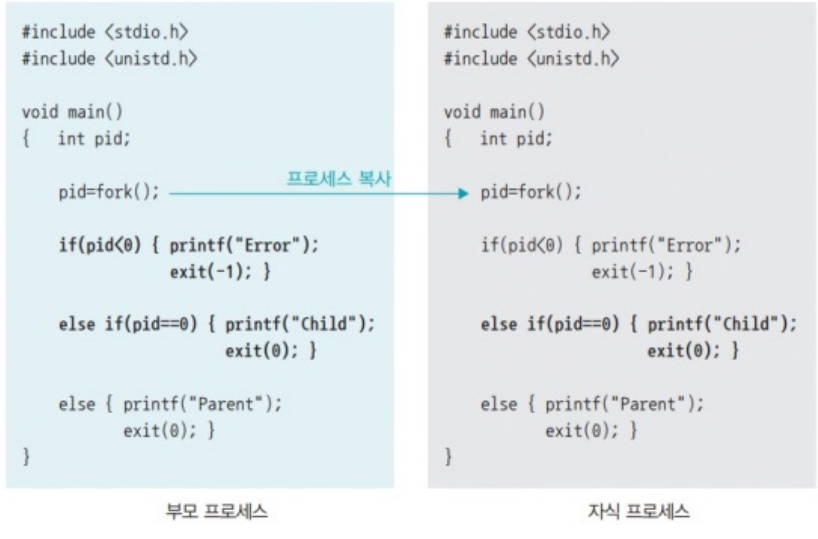
부모 프로세스 구분자와 자식 프로세스 구분자

fork() 시스템 호출의 장점

프로세스의 생성 속도가 빠름

추가 작업 없이 자원을 상속할 수 있음

시스템 관리를 효율적으로 할 수 있음



- 부모 프로세스의 코드가 실행되어 fork( ) 문을 만나면 똑같은 내용의 자식 프로세스를 하나 생성

- 이때 fork( ) 문은 부모 프로세스에 0보다 큰 값을 반환하고 자식 프로세스에 0을 반환
- 만약 0보다 작은 값을 반환하면 자식 프로세스가 생성되지 않은 것으로 여겨 'Error'를 출력

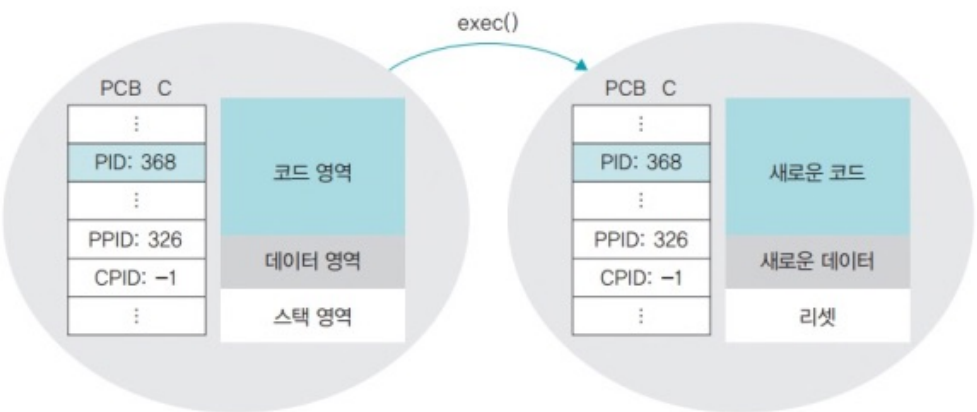
exec() 시스템 호출

: 기존의 프로세스를 새로운 프로세스로 전환하는 함수

이미 만들어진 프로세스의 구조를 재활용하는 것

-fork() : 새로운 프로세스를 복사하는 시스템 호출

-exec() : 프로세스는 그대로 둔 채 내용만 바꾸는 시스템 호출



exec() 시스템 호출을 하면 코드 영역에 있는 기존의 내용을 지우고 새로운 코드로 바꿔버림.

데이터 영역이 새로운 변수로 채워지고 스택 영역이 리셋된다.

프로세스 제어 블록의 내용 중 프로세수 구분자, 부모 프로세스 구분자, 자식 프로세스 구분자, 메모리 관련 사항 등은 변하지 않지만 프로그램 카운터 레지스터 값을 비롯한 각종 레지스터와 사용한 파일 정보가 모두 리셋된다.

```c
#include <stdio.h>
#include <unistd.h>

void main()
{    int pid;

     pid=fork();

     if(pid<0) { printf("Error");
                 exit(-1); }

    else if(pid==0) {  /* child process */
              execlp("mplayer", "mplayer", NULL);
              exit(0); }

    else {  wait(NULL);
            printf("mplayer  Terminated");
            exit(0); }
}
```

부모 프로세스

```c
/* mplayer - music player */
        ⋮
void main()
{
        ⋮
      mplayer
      실행 코드
        ⋮
}
```

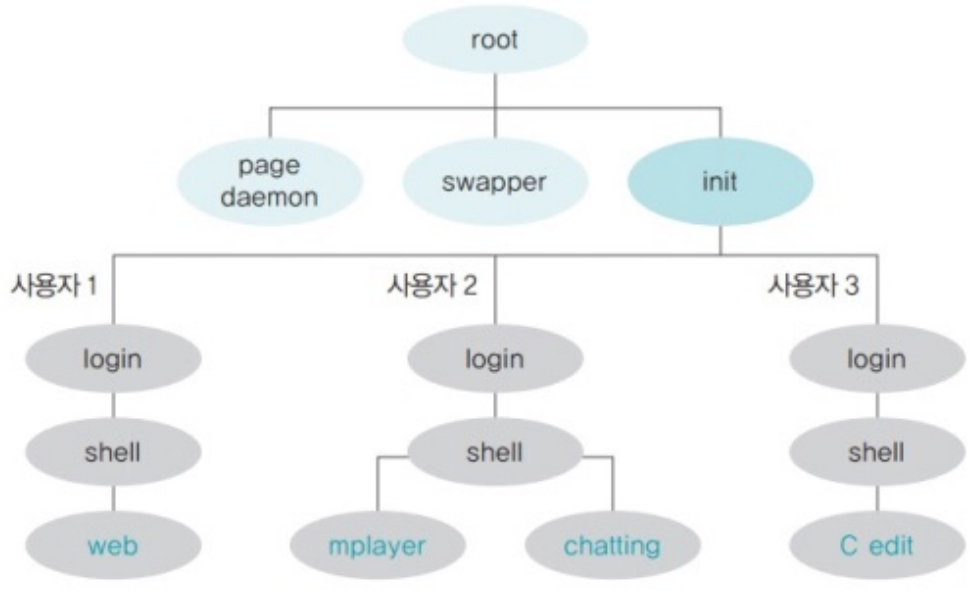
자식 프로세스(exec( ) 실행)

- 부모 프로세스의 fork() 문을 실행하여 자식 프로세스를 생성하고, wait() 문을 실행하여 자식 프로세스가 끝날 때까지 기다림
- 새로 생성된 자식 프로세스는 부모 프로세스의 코드와 같음
- exec() 시스템 호출을 사용하여 새로운 프로세스로 전환하더라도 프로세스 제어 블록의 각종 프로세스 구분자(PID, PPID, CPID)가 변경되지 않기 때문에, 프로세스가 종료된 후 부모 프로세스로 돌아올 수 있음

작업이 끝난 프로세스의 자원을 회수하는 행위 (garbage collection)

유닉스 프로세스 구조

: 유닉스의 모든 프로세스들은 init프로세스의 자식이 되어 트리구조를 이룸

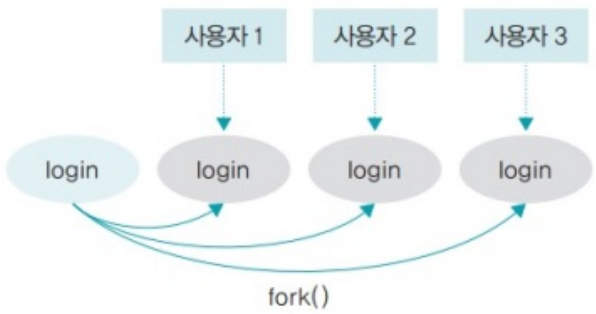


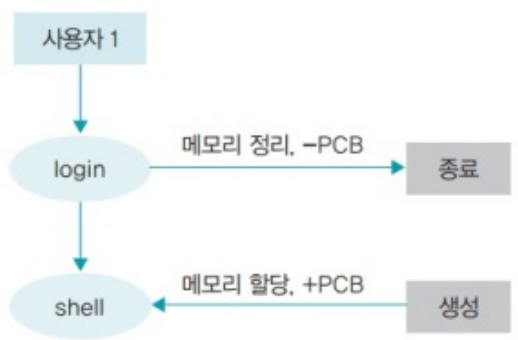
fork()와 exec() 시스템 호출을 이용하여 자식 프로세스를 만든다.

init 프로세스의 자식으로는 login 프로그램, shell 프로그램 등이 있다.

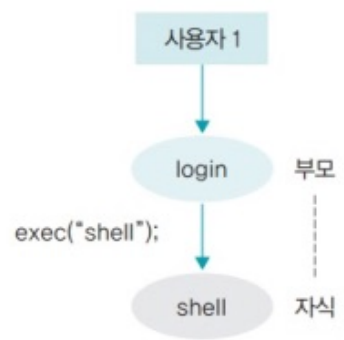
프로세스 계층 구조의 장점

1. 동시에 여러 작업을 처리하고 종료된 프로세스의 자원을 회수하는데 유용함





2. 프로세스의 재사용 용이



3. 자원 회수가 쉬움

-프로세스를 계층 구조로 만들면 프로세스 간의 책임 관계가 분명해져서 시스템 관리하기가 수월함

미아 프로세스 (고아 프로세스)

:프로세스가 종료된 후에도 비정상적으로 남아 있는 프로세스

C언어의 exit() 또는 return() 문은 자식 프로세스가 작업이 끝났음을 부모 프로세스에 알리는 것으로 미아 프로세스 발생을 미연에 방지함

스레드

: 프로세스의 코드에 정의된 절차에 따라 CPU에 작업을 요청하는 실행 단위

CPU 스케줄러가 CPU에 전달하는 일 하나가 스레드이다.

운영체제 입장에서의 작업 단위는 프로세스이고,

CPU 입장에서의 작업 단위는 스레드이다.

*작업의 크기를 크기 순으로 나열하면 job > task> operation이고,

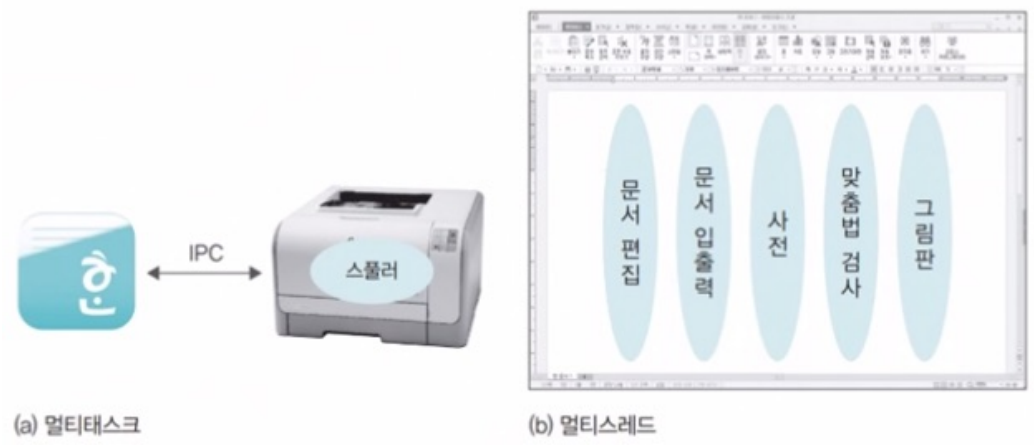
이를 프로세스와 스레드 관계에 대입하면

처리(job) > 프로세스(task) > 스레드(operation)가 된다.

여러 개의 스프레드가 모여 프로세스를 이루고 여러 개의 프로세스가 모여 처리가 되며, 여러 개의 프로세스를 모아서 한꺼번에 처리하는 방법을 일괄 작업(batch job)이라고 한다.

프로세스와 스레드의 차이

프로세스끼리는 약하게 연결되어 있는 반면, 스레드는 강하게 연결되어 있음



멀티태스크

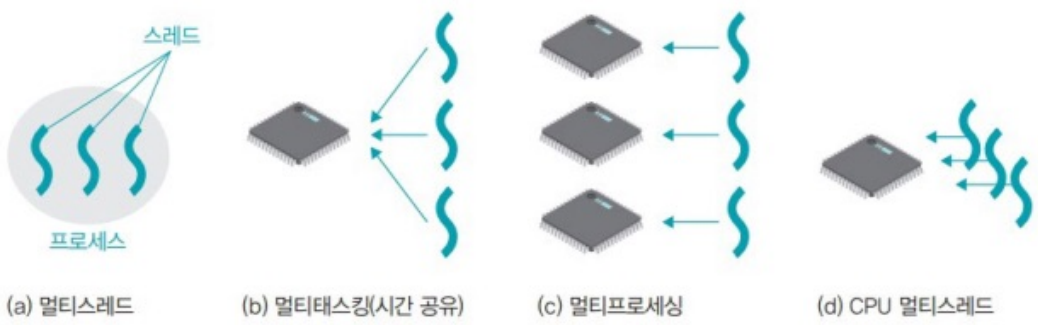
: 한 개의 컴퓨터로 동시에 여러 개의 프로그램을 실행할 수 있게 하는 시스템

ex. 워드프로세서와 프린터 스풀러의 예처럼 서로 독립적인 프로세스는 데이터를 주고받을 때, 프로세스 간 통신(IPC)을 이용한다.

멀티스레드

: 프로세스 내 작업을 여러 개의 스레드를 분할함으로써 작업의 부담을 줄이는 프로세스 운영 기법

ex. 스레드들은 강하게 연결되어 있어 워드프로세서가 종료되면 프로세스 내의 스레드도 강제 종료된다. 멀티 스레드는 변수나 파일 등을 공유하고 전역 변수나 함수 호출 등의 방법으로 스레드 간 통신을 말한다.



멀티스레드

: 프로세스 내 작업을 여러 개의 스레드로 분할함으로써 작업의 부담을 줄이는 프로세스 운영 기법이다.

멀티태스킹

: 운영체제가 CPU에 작업을 줄 때 시간을 잘게 나누어 배분하는 기법이다.

여러 스레드에 시간을 잘게 나눠주는 시스템을 시분할 시스템이라고 한다.

멀티프로세싱

: CPU를 여러 개 사용하여 여러 개의 스레드를 동시에 처리하는 작업 환경

병렬 처리에서의 슈퍼스칼라 기법과 같다.

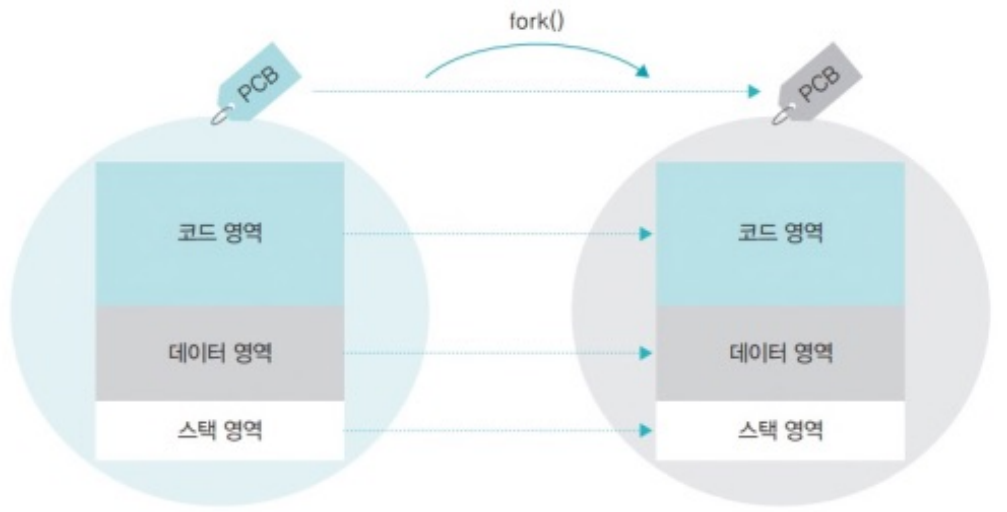
네트워크로 연결된 여러 컴퓨터에 스레드를 나누어 협업하는 분산 시스템도 멀티프로세싱이라고 부른다.

CPU 멀티스레드

: 한 번에 하나씩 처리해야 하는 스레드를 파이프라인 기법을 이용하여 동시에 여러 스레드를 처리하도록 만든 병렬 처리 기법

멀티스레드 : 운영체제가 소프트웨어적으로 프로세스를 작은 단위의 스레드로 분할하여 운영하는 기법

CPU 멀티스레드 : 하드웨어적인 방법으로 하나의 CPU에서 여러 스레드를 동시에 처리하는 병렬 처리 기법

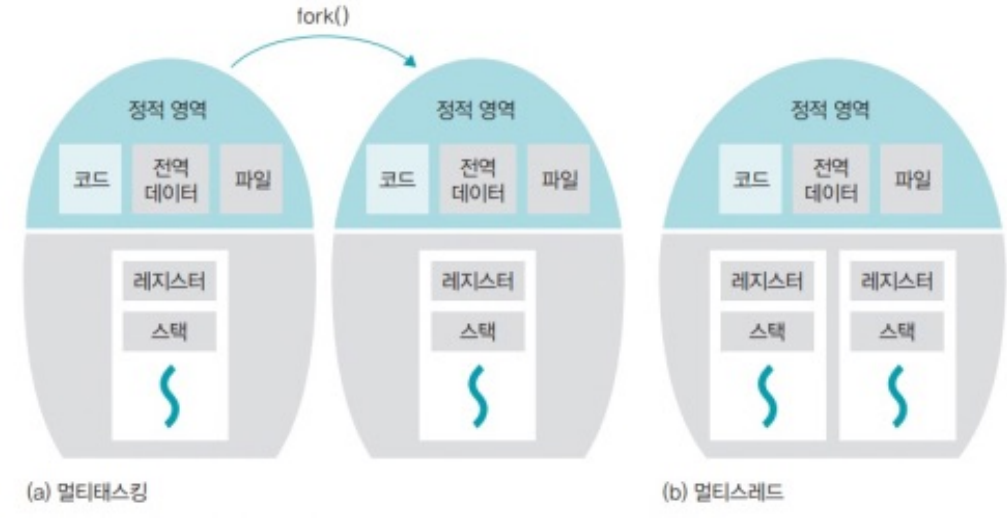


멀티태스킹의 낭비 요소

fork() 시스템 호출로 프로세스를 복사하면 코드 영역과 데이터 영역의 일부가 메모리에 중복되어 존재하며, 부모-자식 관계이지만 서로 독립적인 프로세스이므로 이러한 낭비 요소를 제거할 수 없음

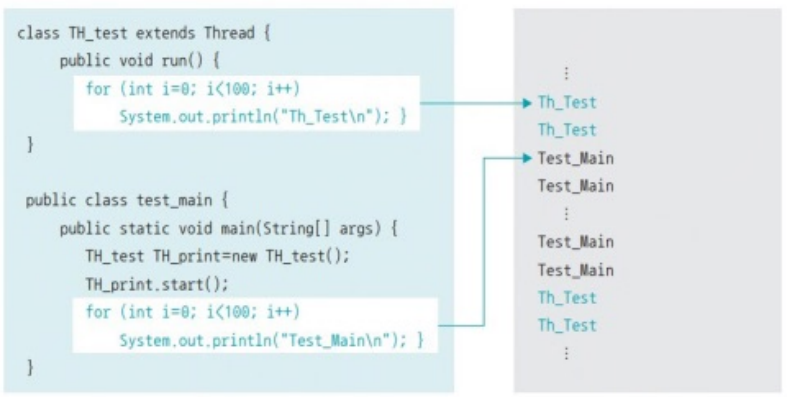
스레드는 이러한 멀티태스킹의 낭비 요소를 제거하기 위해 사용한다.

비슷한 일을 하는 2개의 프로세스를 만드는 대신 코드, 데이터 등을 공유하면서 여러 개의 일을 하나의 프로세스 내에서 사용하는 것이다.



fork() 시스템 호출로 여러 개의 프로세스를 만들면 필요 없는 정적 영역이 여러 개가 된다. 멀티스레드는 코드, 파일 등의 자원을 공유함으로써 자원의 낭비를 막고 효율성 향상한다.

스레드는 가벼운 프로세스(LWP)라고 부르며, 스레드가 1인 일반 프로세스는 무거운 프로세스(HWP)라고 부른다.



main{ } 위쪽에 있는 class TH_test extends Thread{ }는 스레드 객체로, 이는 TH_test 객체를 확장하여 스레드를 만듦

TH_test 객체의 run() 부분을 스레드로 만들어 실행하며 'Th_Test'를 백 번 출력

fork() 시스템 호출 사용 코드

```
#include <stdio.h>
#include <unistd.h>

int main()
{   int i=0, pid;

    pid=fork();

    if(pid<0) { printf("Error");
                exit(-1); }

    else if(pid==0) { for(i=0; i<100; i++)
                        printf("Th_Test\n");
                      exit(0); }

    else { for(i=0; i<100; i++)  /* parent */
                printf("Test_Main\n");
           exit(0); }
}
```

부모 프로세스

```
#include <stdio.h>
#include <unistd.h>

int main()
{   int i=0, pid;

    pid=fork();

    if(pid<0) { printf("Error");
                exit(-1); }

    else if(pid==0) { for(i=0; i<100; i++)
                        printf("Th_Test\n");
                      exit(0); }

    else { for(i=0; i<100; i++)  /* parent */
                printf("Test_Main\n");
           exit(0); }
}
```

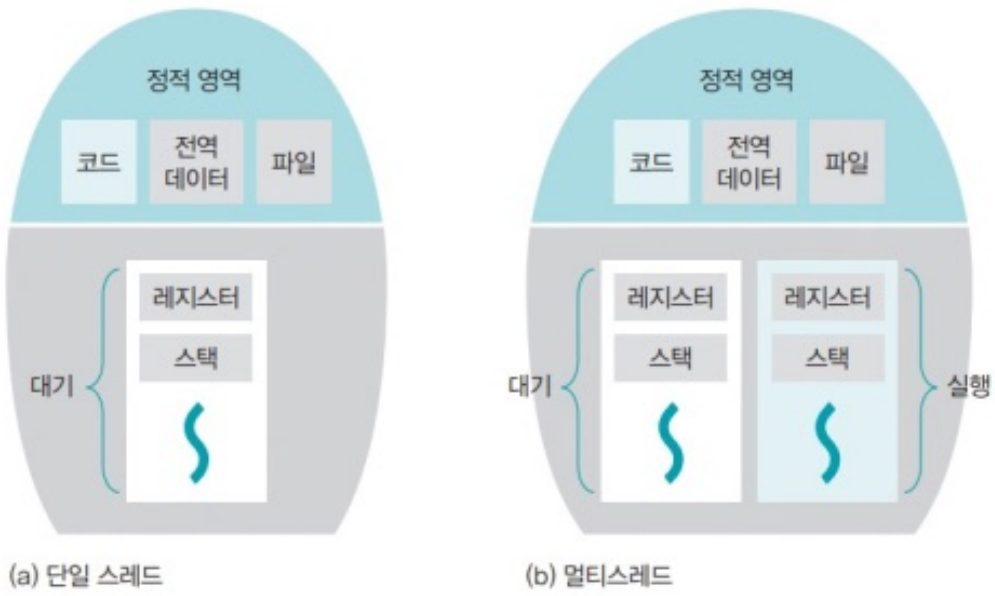
자식 프로세스

fork() 시스템 호출을 사용하여 프로세스 제어 블록, 코드, 데이터 등이 모두 2배가 됨으로써 스레드를 사용하는 것보다 낭비가 심함

*스레드 라이브러리 :

스레드는 운영체제가 지원 라이브러리를 통해 생성된다.

멀티스레드의 장단점



장점	단점
1. 응답성 향상 2. 자원 공유 3. 효율성 향상 4. 다중 CPU 지원	모든 스레드가 자원을 공유하기 때문에 한 스레드에 문제가 생기면 전체 프로세스에 영향을 미침 ex) 인터넷 익스플로러에서 여러 개의 화면을 동시에 띄웠는데 그중 하나에 문제가 생기면 인터넷 익스플로러 전체가 종료

멀티스레드 모델

커널 스레드 : 커널이 직접 생성하고 관리하는 스레드이다

사용자 스레드 : 라이브러리에 의해 구현된 일반적인 스레드이다

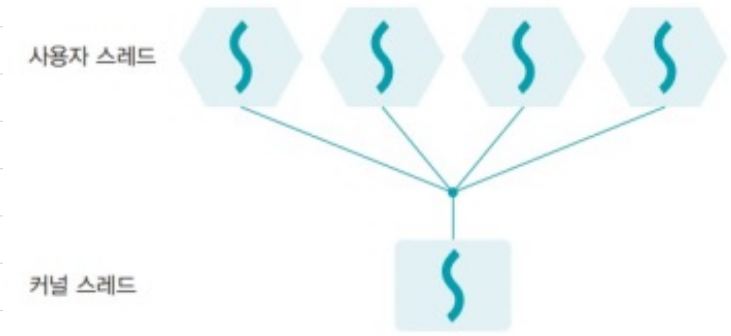
사용자 레벨 스레드

사용자 프로세스 내에 여러 개의 스레드가 커널의 스레드 하나와 연결 (1 to N 모델)

라이브러리가 직접 스케줄링을 하고 작업에 필요한 정보를 처리하기 때문에 문맥 교환이 필요 없음

커널 스레드가 입출력 작업을 위해 대기 상태에 들어가면 모든 사용자가 스레드가 같이 대기하게 됨

한 프로세스의 타임 슬라이스를 여러 스레드가 공유하기 때문에 여러 개의 CPU를 동시에 사용할 수 없음



커널 레벨 스레드

하나의 사용자 스레드가 하나의 커널 스레드와 연결(1 to 1 모델)

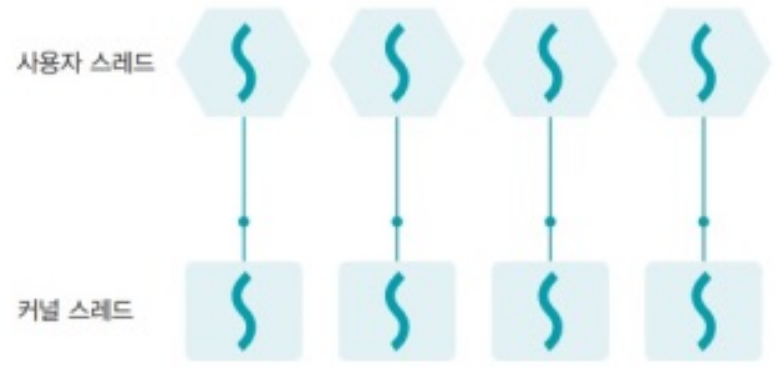
독립적으로 스케줄링이 되므로 특정 스레드가 대기 상태에 들어가도 다른 스레드는 작업을 계속할 수 있음

커널 레벨에서 모든 작업을 지원하기 때문에 멀티 CPU를 사용할 수 있음

하나의 스레드가 대기 상태에 있어도 다른 스레드는 작업을 계속할 수 있음

커널의 기능을 사용하므로 보안에 강하고 안정적으로 작동

문맥 교환할 때 오버헤드 때문에 느리게 작동



멀티레벨 스레드

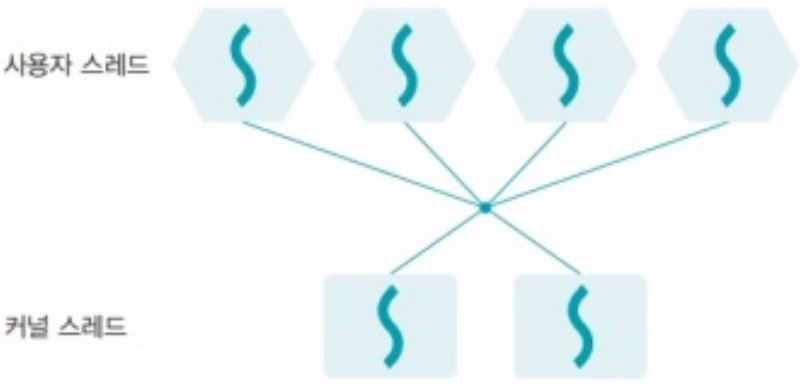
사용자 레벨 스레드와 커널 레벨 스레드를 혼합한 방식(M to N 모델)

커널 스레드가 대기 상태에 들어가면 다른 커널 스레드가 대신 작업을 하여 사용자 레벨 스레드보다 유연하게 작업을 처리할 수 있음

커널 레벨 스레드를 같이 사용하기 때문에 여전히 문맥 교환 시 오버헤드가 있어 사용자 레벨 스레드만큼 빠르지 않음

빠르게 움직여야 하는 스레드는 사용자 레벨 스레드로 작동하고,

안정적으로 움직여야 하는 스레드는 커널 레벨 스레드로 작동



구분	선점형	비선점형
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 스케줄러에 사용된다.	일괄 작업 방식 스케줄러에 사용된다.
중요도	높다.	낮다.

4장

CPU 스케줄링

: CPU 스케줄러는 프로세스가 생성된 후 종료될 때까지 모든 상태 변화를 조정하는 일을 하며, CPU 스케줄링은 CPU 스케줄러가 하는 모든 작업을 가르킴

스케줄링 단계

고수준 스케줄링(high level scheduling)

: 시스템 내의 전체 프로세스 수를 조절하는 것이다.
어떤 작업을 시스템이 받아들일지 또는 거부할지를 결정한다.
시스템 내에서 동시에 실행 가능한 프로세스의 총 개수가 정해진다.

중간 수준 스케줄링(middle level scheduling)

: 전체 시스템의 활성화된 프로세스 수를 조절하여 과부하를 막는 것이다.
시스템 과부하가 걸려서 전체 프로세서 수를 조절해야한다면 이미 활성화된 프로세스 중 일부를 보류 상태로 보낸다. 보류된 프로세스는 처리 능력에 여유가 생기면 다시 활성화가 된다.

저수준 스케줄링(low level scheduling)

: 어떤 프로세스에 CPU를 할당할지, 어떤 프로세스를 대기 상태로 보낼 지 등을 결정하는 것이다. 실제 작업이 이루어진다.

스케줄링 목적

공평성 : 모든 프로세스가 자원을 공평하게 배정받아야 하며, 자원 배정 과정에서 특정 프로세스가 배제되어서는 안된다.

효율성 : 시스템 자원이 유휴 시간 없이 상용되도록 스케줄링을 하고, 유휴 자원을 사용하려는 프로세스에는 우선권을 주어야 한다.

안정성 : 우선순위를 사용하여 중요 프로세스가 먼저 작동하도록 배정함으로써 시스템 자원을 점유하거나 파괴하려는 프로세스로부터 자원을 보호해야 한다.

확장성 : 프로세스가 증가해도 시스템이 안정적으로 작동하도록 조치해야 한다. 시스템 자원이 늘어나는 경우 이 혜택이 시스템에 반영되게 해야 한다.

반응 시간 보장 : 응답이 없는 경우 사용자는 시스템이 멈춘 것으로 가정하기 때문에 시스템은 적절한 시간 안에 프로세스의 요구에 반영되게 해야 한다.

무한 연기 방지 : 특정 프로세스의 작업이 무한히 연기되어서는 안된다.

스케줄링 시 고려 사항

선점형 스케줄링 : 어떤 프로세스가 CPU를 할당받아 실행 중이더라도 운영체제가 CPU를 강제로 빼앗을 수 있는 스케줄링 방식

비선점형 스케줄링 : 어떤 프로세스가 CPU를 점유하면 다른 프로세스가 이를 빼앗을 수 없는 스케줄링 방식

선점형 스케줄링 방식의 대표적인 예로는 인터럽트 처리가 있다.

문맥 교환 같은 부가적인 작업으로 인해 낭비가 생기는 것이 단점이다.

하나의 프로세스가 CPU를 독점할 수 없기 때문에 빠른 응답 시간을 요구하는 대화형 시스템이나 시분할 시스템에 적합하다.

저수준 스케줄러는 선점형 스케줄링 방식을 사용한다.

비선점형 스케줄링에서는 어떤 프로세스가 실행 상태에 들어가 CPU를 사용하면 그 프로세스가 종료되거나 자발적으로 대기 상태에 들어가기 전까지는 계속 실행된다. 스케줄러의 작업 적고 문맥 교환에 의한 낭비도 적다.

CPU 사용 시간이 긴 프로세스 때문에 CPU 사용 시간이 짧은 여러 프로세스가 오랫동안 기다리게 되어 전체 시스템의 처리율이 떨어진다.