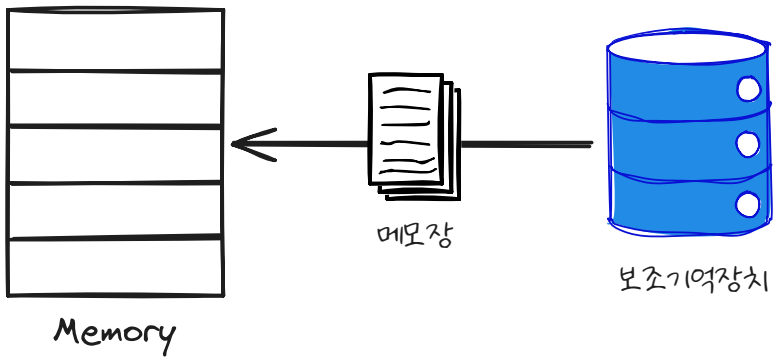
프로세스와 스레드

프로세스

실행 중인 프로그램을 프로세스라고 칭한다.

프로그램은 실행되기 전에는 보조기억장치에 존재하는 단순한 데이터이지만, 보조기억장치에 저장된 프로그램을 메모리에 적재하고 실행하는 순간 프로그램은 프로세스가 된다.

위와 같은 일련의 과정을 프로세스를 생성한다고 칭한다.



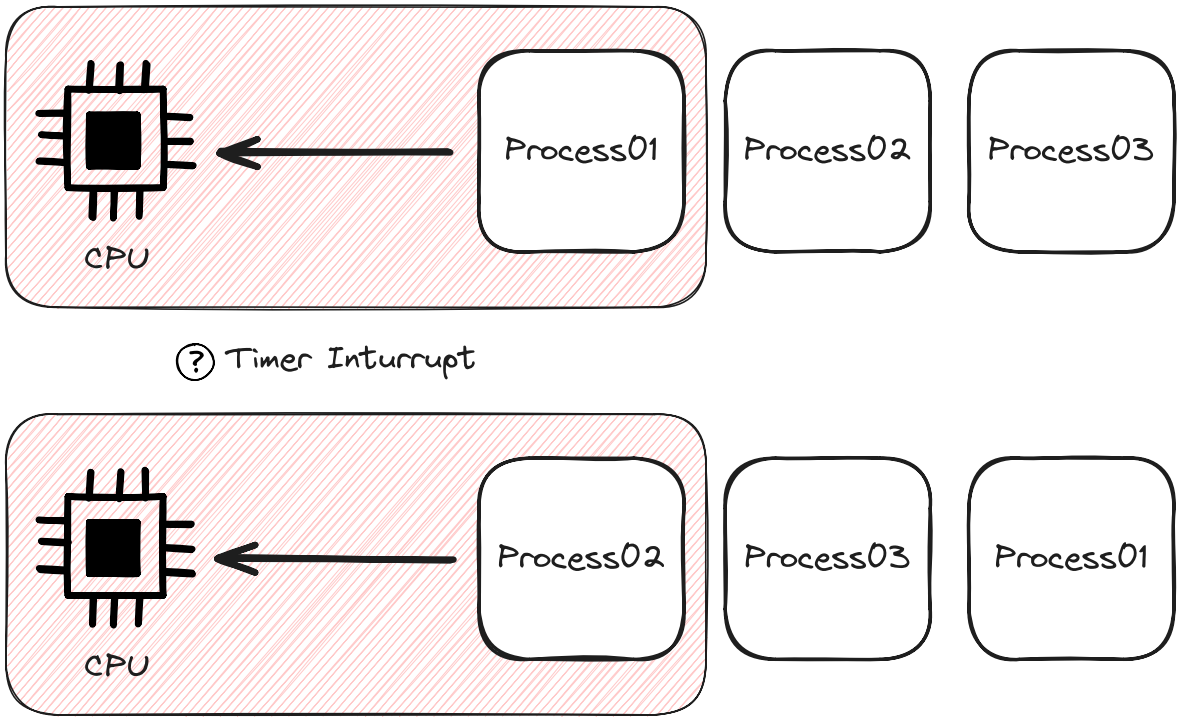
실행되는 여러 프로세스 중 사용자가 볼 수 있는 공간에서 실행되는 프로세스를 **포그라운드 프로세스**, 사용자가 볼 수 없는 곳에서 실행되는 프로세서를 **백그라운드 프로세스**라고 칭한다.

> 백그라운드 프로세스
>
> 백그라운드 프로세스 중에는 사용자와 직접 상호작용할 수 있는 백그라운드 프로세스도 존재하지만, 사용자와 상호작용하지 않는 백그라운드 프로세스도 존재한다.
>
> 이를 유닉스 체계의 운영체제에서는 **데몬(Daemon)** 윈도우에서는 **서비스(Service)** 라고 부른다

프로세스 제어 블록

모든 프로세스는 실행을 위해 CPU를 필요로 하지만, CPU 자원은 한정되어 있기 때문에 모든 프로세스가 CPU를 동시에 사용할 수 없다.

그렇기에 모든 프로세스는 차례대로 돌아가며 한정된 시간만큼만 CPU를 이용한다. 이용 시간이 지나면 타이머 인터럽트가 발생하고, CPU를 점유하고 있던 프로세스는 CPU 사용을 중단하고 다음 차례가 올 때까지 대기 상태로 전환된다.



운영체제는 빠르게 번갈아 수행되는 프로세스의 실행 순서를 관리하고, 프로세스에 CPU를 비롯한 자원을 배분한다. 이를 위해 운영체제는 프로세스 제어 블록 이하 PCB(Process Controller Block) 을 이용한다.

PCB는 프로세스와 관련된 정보를 저장하는 자료 구조이며, 해당 프로세스를 식별하기 위해 꼭 필요한 정보들이 저장된다. 운영체제는 PCB로 특정 프로세스를 식별하고 해당 프로세스를 처리하는데 필요한 정보를 판단한다.

PCB의 라이프 사이클은 프로세스와 거의 동일한데, 프로세스가 생성될 때 함께 만들어지고 프로세스의 실행이 끝나면 폐기된다.

프로세스 제어 블록에 담기는 정보들

> 프로세스 ID (PID)
>
> 특정 프로세스를 식별하기 위해 부여하는 고유한 번호이며, 동일한 작업을 수행하는 프로그램이라 하더라도 두 번 실행한다면, PID가 다른 두 개의 프로세스가 생성된다.

> 레지스터 값
>
> 프로세스는 자신의 실행 차례가 돌아오면 이전 진행 작업들을 이어하기 위해, 이전까지 사용했던 레지스터의 중간값들을 모두 복원한다.
>
> PCB 내부에는 해당 프로세스가 실행하며 사용했던 프로그램 카운터를 비롯한 레지스터 값들이 담긴다.

> 프로세스 상태
>
> 현재 프로세스가 입출력장치를 사용하기 위해 기다리는 건지 CPU를 사용하기 위해 기다리고 있는 상태인지, 혹은 이용하고 있는 중인지를 나타내는 프로세스 상태 정보가 저장된다.

> CPU 스케줄링 정보
>
> 프로세스가 언제, 어떤 순서로 CPU를 할당받을지에 대한 정보도 PCB에 기록된다.

> 메모리 관리 정보
>
> 프로세스마다 메모리에 저장된 위치가 다르기 때문에 어느 주소에 저장되어 있는지에 대한 정보가 저장된다. PCB에는 베이스 레지스터, 한계 레지스터 값, 페이지 테이블 정보가 저장된다.

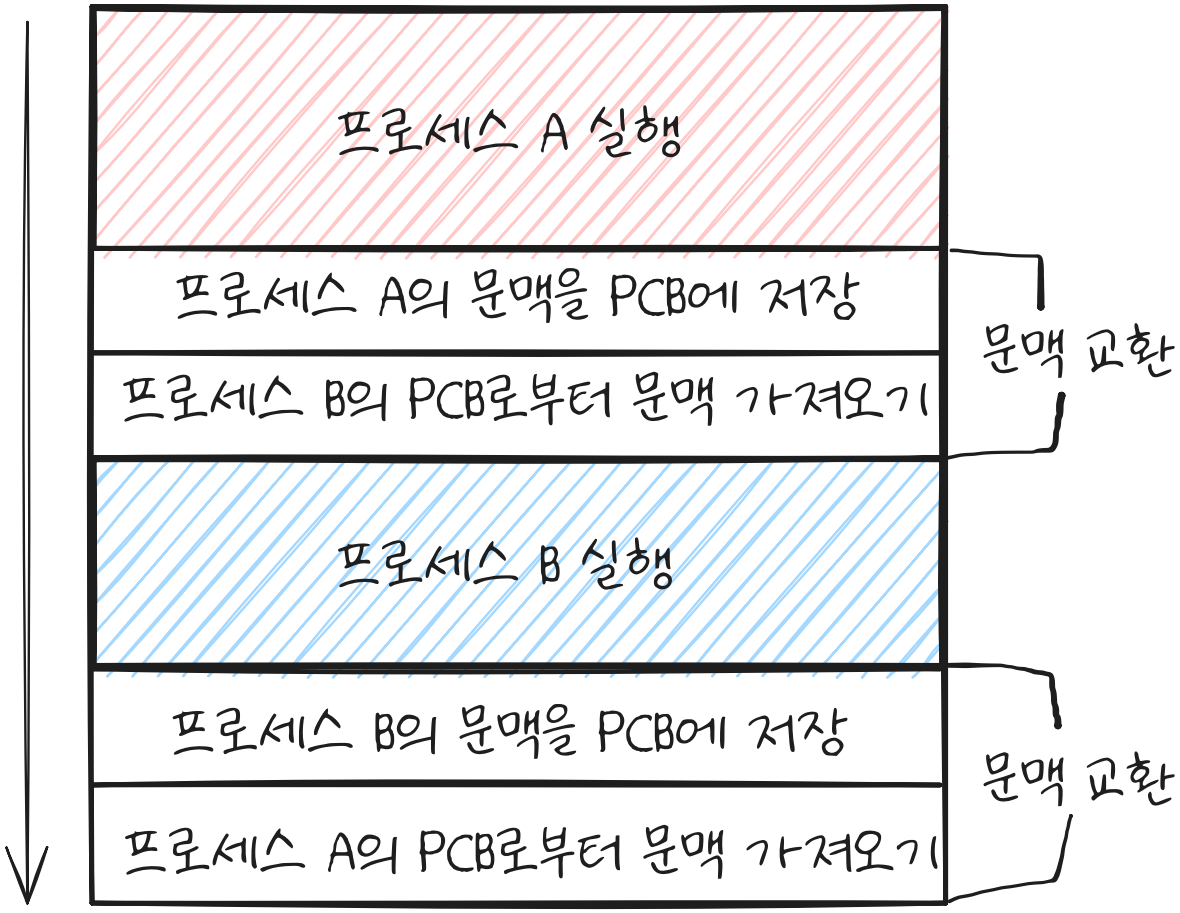
> 사용한 파일과 입출력장치 목록
>
> 프로세스가 실행 중인 과정에서 특정 입출력장치나 파일을 사용하면 PCB에 해당 내용이 명시된다.

문맥 교환

하나의 프로세스에서 다른 프로세스로 실행 순서가 넘어갈 때, 직전까지 실행되던 프로세스는 중간 정보를 백업해야 한다. 이때, 하나의 프로세스 수행을 재개하기 위해 기억해야 할 정보를 문맥(Context) 라고 한다.

하나의 프로세스 문맥은 해당 프로세스의 PCB에 기록되어 있다. PCB에 기록되는 정보들을 문맥이라고 봐도 무방하다. CPU 점유 시간이 종료되거나 인터럽트가 발생하면 운영체제는 해당 프로세스의 문맥을 PCB에 백업한다. 그리고 뒤이어 실행될 프로세스의 문맥을 복구한다.

이처럼 기존 프로세스의 문맥을 백업하고 새로운 프로세스를 실행하기 위해 문맥을 PCB로부터 복구하여 새로운 프로세스를 실행하는 것을 문맥 교환(Context Switching) 이라고 한다.



⚠ 문맥 교환을 너무 자주하면 오버헤드가 발

프로세스의 메모리 영역

프로세스는 메모리에 할당될 때, 크게 코드 영역, 데이터 영역, 힙 영역, 스택 영역으로 분리되어 저장된다.

코드 영역

코드 영역은 텍스트 영역이라고도 불리며, 기계어로 이루어진 명령어가 저장된다. 코드 영역에는 데이터가 아닌 CPU에서 실행될 명령어가 담겨 있기 때문에 write 가 금지되어 있다. 즉, 코드 영역은 읽기 전용 공간이다.

데이터 영역

프로그램이 실행되는 동안 유지할 데이터가 저장되는 공간이다. 이런 데이터로는 전역 변수가 대표적이다. 코드 영역과 데이터 영역은 크기가 고정되어 변하지 않기 때문에 정적 할당 영역이라고 부른다.

힙 영역

힙 영역은 프로그래머가 직접 할당할 수 있는 저장 공간이다. 프로그래밍 과정에서 힙 영역에 메모리 공간을 할당했다면 언젠가는 해당 공간을 반환해야 한다.

메모리 공간을 반환한다는 것은 해당 메모리 공간을 사용하지 않겠다고 운영체제에게 선언하는 것과 같다.

만약 메모리 공간을 반환하지 않는다면 할당한 공간은 메모리 내에 계속 남아 메모리 낭비를 초래한다. 이런 문제를 메모리 누수라고 한다.

스택 영역

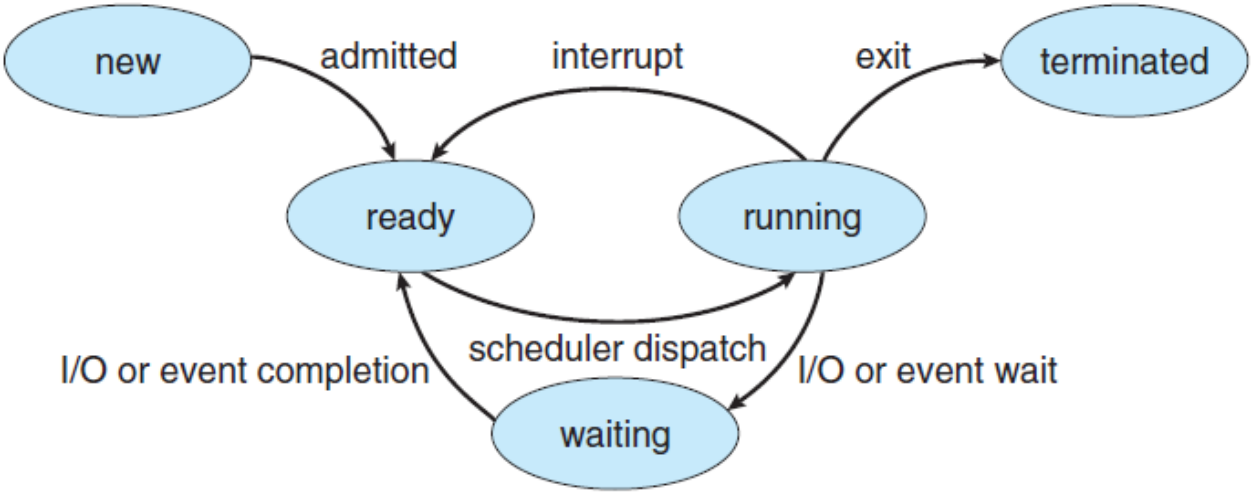
스택 영역은 데이터를 일시적으로 저장하는 공간으로 데이터 영역에 담기는 값과는 달리 일시적으로 사용할 값들이 저장되는 공간이다. 매개 변수, 지역 변수 등이 대표적이다.

일시적으로 저장할 데이터는 스택 영역에 Push 되고, 더 이상 필요하지 않은 데이터는 Pop 되는 것으로 스택 영역에서 사라진다.

힙 영역과 스택 영역은 실시간으로 그 크기가 변할 수 있기 때문에 동적 할당 영역이라고 한다. 그래서 일반적으로 힙 영역은 메모리의 낮은 주소에서 높은 주소로 할당되고, 스택 영역은 높은 주소에서 낮은 주소로 할당된다.

프로세스 상태

여러 프로세스들이 번갈아 실행되는 과정에서 각 프로세스들은 여러 상태를 거치며 실행된다. 운영체제는 프로세스의 상태를 PCB를 통해 인식하고 관리한다.



생성 상태 (New)

프로세스를 생성 중인 상태를 생성 상태라고 한다. 이제 막 메모리에 적재되어 PCB를 할당 받은 상태를 의미한다. 생성 상태를 거쳐 실행할 준비가 완료된 프로세스는 곧바로 실행되지 않고 준비 상태가 되어 CPU 할당을 기다린다.

준비 상태 (Ready)

준비 상태는 당장이라도 CPU를 할당받아 실행할 수 있지만 아직 자신의 차례가 아니기에 기다리고 있는 상태를 의미한다. 준비 상태 프로세스는 차례가 되면 CPU를 할당받아 실행 상태가 된다.

실행 상태 (Running)

실행 상태는 CPU를 할당받아 실행 중인 상태를 의미한다.

실행 상태인 프로세스는 할당된 일정 시간 동안만 CPU를 사용할 수 있으며, 프로세스가 할당된 시간을 모두 사용한다면 다시 준비 상태로 전환되고 실행 도중 입출력장치를 사용하여 입출력장치의 작업이 끝날 때까지 기다려야 한다면 대기 상태가 된다.

대기 상태 (Blocked)

입출력 작업을 CPU에 비해 처리 속도가 느리기에, 입출력 작업을 요청한 프로세스는 입출력 장치가 입출력을 끝낼 때까지, 즉 입출력 완료 인터럽트를 받을 때까지 기다려야 한다.

이처럼 입출력장치의 작업을 기다리는 상태를 대기 상태라고 한다. 입출력 작업이 완료되면 해당 프로세스는 다시 준비 상태로 CPU 할당을 기다린다.

대기 상태가 되는 원인에는 입출력 작업만이 있는 것은 아니다. 특정 이벤트가 일어나길 기다릴 때 프로세스는 대기 상태가 된다.

종료 상태

종료 상태는 프로세스가 종료된 상태다. 프로세스가 종료되면 운영체제는 PCB와 프로세스가 사용한 메모리를 정리한다.

스레드

스레드는 프로세스를 구성하는 실행의 흐름 단위다. 하나의 프로세스는 여러 개의 스레드를 가질 수 있다.

스레드를 이용하면 하나의 프로세스에서 여러 부분을 실행할 수 있다.

스레드는 프로세스 내에서 각기 다른 스레드 ID, 프로그램 카운터 값을 비롯한 레지스터 값, 스택으로 구성된다. 각자 프로그램 카운터 값을 비롯한 레지스터 값, 스택을 가지고 있기에 스레드마다 각기 다른 코드를 실행할 수 있다.

프로세스의 스레드들은 실행에 필요한 최소한의 정보만을 유지한 채 프로세스 자원을 공유하며 실행된다.

멀티프로세스와 멀티스레드

여러 프로세스를 동시에 실행하는 것을 멀티프로세스, 여러 스레드로 프로세스를 동시에 실행하는 것을 멀티스레드라고 한다.

프로세스끼리는 기본적으로 자원을 공유하지 않지만, 스레드끼리는 같은 프로세스 내의 자원을 공유한다.

프로세스를 fork 하여 같은 작업을 행하는 동일한 프로세스 두 개를 동시에 실행하면 코드 영역, 데이터 영역, 힙 영역 등을 비롯한 **모든 자원이 복제되어 메모리에 적재된다.** 이는 같은 프로그램을 실행하기 위해 **메모리에 동일한 내용들이 중복해서 존재하는 낭비라고 볼 수 있다.**

반면, 스레드들은 각기 다른 스레드 ID, 프로그램 카운터 값을 포함한 레지스터 값, 스택을 가질 뿐 프로세스가 가지고 있는 자원을 공유한다. 여러 프로세스를 병행 실행하는 것보다 메모리를 더 효율적으로 사용할 수 있는 것이다.

또한 스레드는 프로세스의 자원을 공유하기 때문에 서로 협력과 통신에 유리하다.

싱글스레드

멀티스레드

> ⚠ 프로세스의 자원을 공유한다는 특성은 단점
>
> 멀티프로세스 환경에서는 하나의 프로세스에 문제가 생겨도 다른 프로세스에는 지장이 적거나 없지만, 멀티스레드 환경에서는 하나의 스레드에 문제가 생기면 프로세스 전체에문제가 생길 수 있다.
>
> 모든 스레드가 프로세스 자원을 공유하고 하나의 스레드에 문제가 생기면 다른 스레드도 영향을 받기 때문이다.