프로세스와 스레드

태그	운영체제 - 프로세스와스레드

프로세스란?

프로세스는 일련의 작업 단위로 운영체제로부터 자원을 할당받아 메모리에 적재되어 실행되고 있는 상태의 프로그램을 말한다.

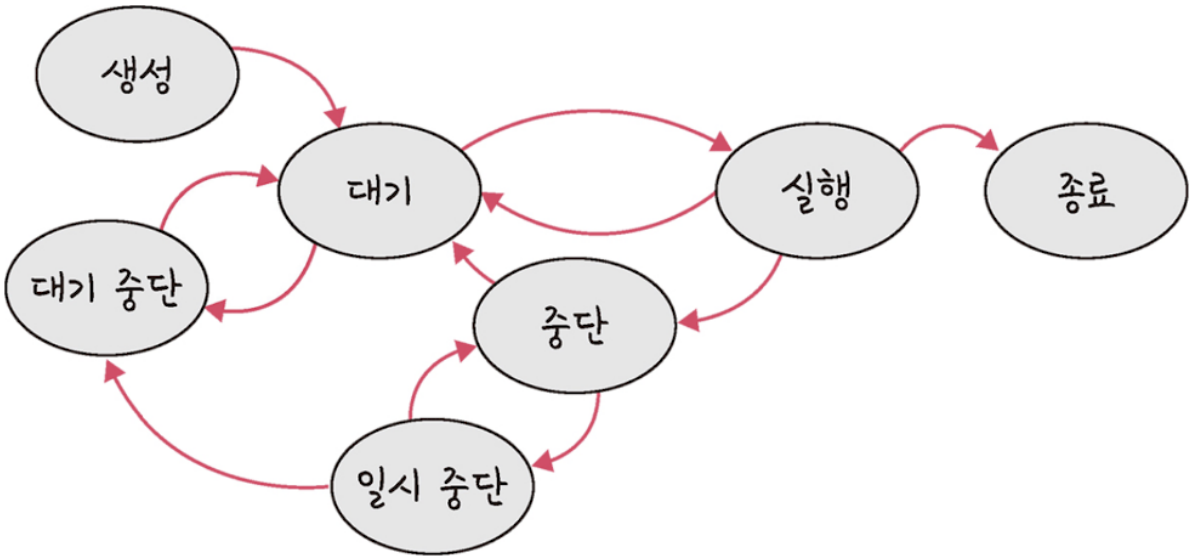
> **프로그램**
> 프로그램이란 하드디크스나 SSD 등 보조기억장치에 존재하는 명령어 및 데이터의 묶음이다. 프로그램은 자체는 스스로 동작을 하지 않는 정적인 개체이다. 이 프로그램을 실행하기 위해서는 운영체제로부터 프로그램이 동작하는데 필요한 자원인 CPU나 메모리 등을 할당받아야 한다.

프로세스의 특징

- 프로세스는 각각 독립된 영역(Code, Data, Stack, Heap의 구조)를 할당받는다.
- 각 프로세스는 별도의 주소 공간에서 실행되며, 일반적으로 한 프로세스는 다른 프로세스의 변수나 자료구조에 접근할 수 없다.
- 한 프로세스가 달느 프로세스의 자원에 접근하려면 프로세스간 통신을 사용해야 한다.

프로세스의 상태

프로세스의 상태는 커널의 자료구조에 의해 관리되며 상태의 종류로는 크게 생성(New), 준비(Ready), 실행(Running), 대기(Waiting) 또는 보류(Block), 종료(Terminated) 등으로 분류 할 수 있다.



생성상태(New)

프로그램이 커널 영역에 PCB를 부여받은 상태를 생성상태라고 한다. 프로세스로 생성되고 대기 또는 준비 상태로 가기 전 상태이다. 운영체제는 프로세스를 생성한 후 주기억장치 공간이 여유가 있는지 확인하고 공간이 충분하면 프로세스 주소 강간을 할당한 후 프로세스를 준비상태로 바꾸어준다. 만약, 주기억장치의 공간이 충분하지 않은 경우 대기중단상태로 넘어간다.

> **프로세스를 생성하는 두가지 함수**
>
> **1. fork()**
> 부모 프로세스의 주소 공간을 복사하여 새로운 자식 프로세스를 생성하는 함수로 부모 프로세스의 주소 공간만 복사할 뿐 비동기 작업 등을 상속하지는 않는다.
>
> **2. exex()**
> **새롭게 프로세스를 생성하는 함수.**

준비상태(Ready)

프로세스가 CPU를 사용하고 있지는 않지만 언제든지 사용할 수 있는 상태로, 메모리 공간이 충분하면 메모리를 할당받고 아니면 아닌 상태로 대기하고 있으며 CPU 스케줄러로부터 CPU 소유권이 넘어오기를 기다리는 상태를 말한다. 일반적으로 준비상태의 프로세스 중 우선순위가 높은 프로세스가 CPU를 할당받는다.

실행상태(Running)

프로세스가 CPU 소유권과 메모리를 할당받고 명령어들이 실행중인 상태를 말한다. 준비상태의 프로세스가 CPU를 할당받게 되면(Dispatch) 실행 상태가 되고, CPU는 해당 프로세스를 실행한다.

실행상태의 프로세스는 CPU 스케줄링에 의해 CPU를 빼앗길 수도 있다.(선점) CPU를 빼앗긴 프로세스는 준비 상태로 바뀌게 된다. 또한, 실행 상태인 프로세스가 입출력이 필요하여 시스템 콜을 하게 되면 입출력이 종료될 때 까지 대기 상태로 바뀌게 된다.

입출력이 완료될 때 까지 CPU가 아무일도 하지 않으면 비효율적이기 때문에 입출력이 완료될때까지 해당 프로세스는 대기 상태로 바뀌게 되고 CPU는 준비 상태에 있는 다른 프로세스를 할당받는다.

대기상태(Waiting) 또는 보류상태(Block)

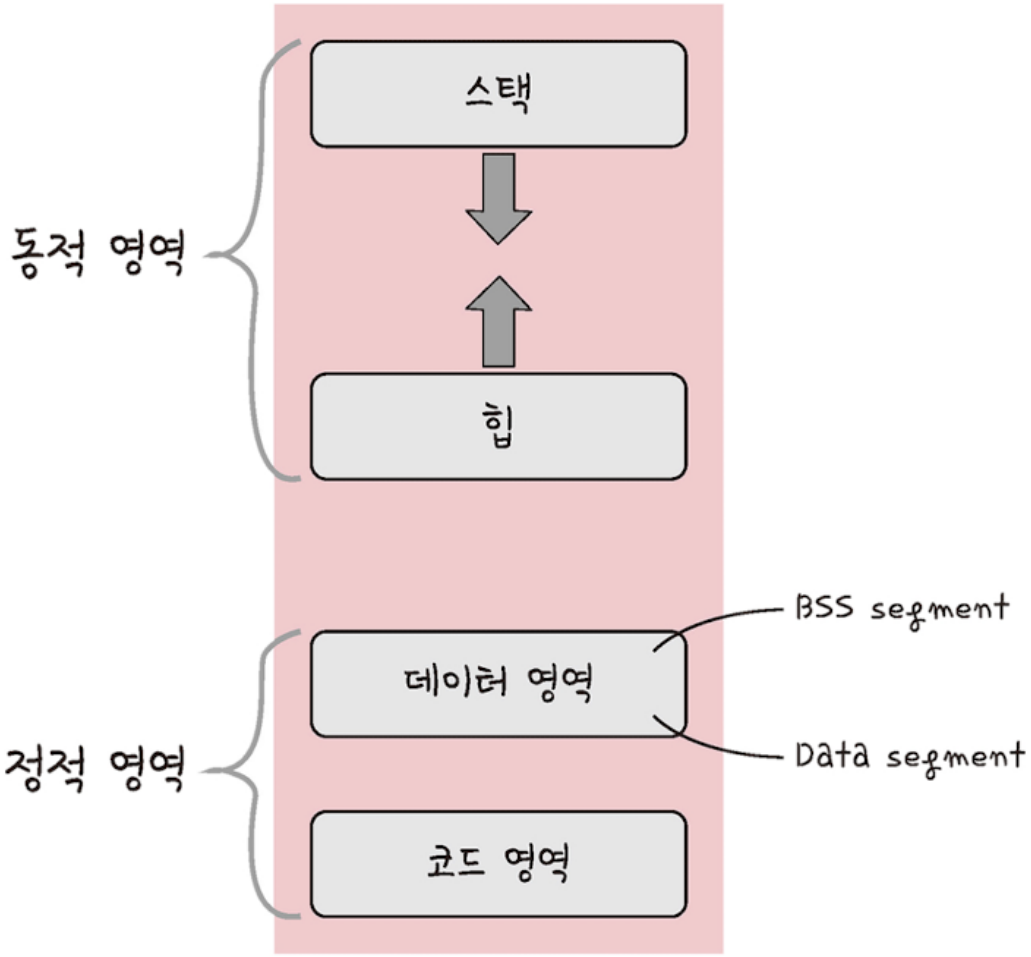
실행상태인 프로세스가 입출력을 요청하거나, 바로 확보될 수 없는 자원을 요청하면 CPU를 반납하고 요청한 일이 완료될 때까지 기다리는 대기 상태가 된다. 대기 상태는 큐 또는 리스트로 관리되며 요청한 일이 완료되면 대기 상태의 프로세스는 다시 준비상태가 된다.

종료상태(Terminated)

메모리와 CPU 소유권을 모두 반납하는 상태를 말한다. 종료는 자연스럽게 명령어가 모두 실행되어 종료되는것도 있지만 부모 프로세스가 자식 프로세스를 강제시키는 비 자발적으로 종료되기도 한다. 자식 프로세스에 할당된 자원의 한계치를 넘어서거나 부모 프로세스가 종료되거나 사용자가 process.kill 등 여러 명령어로 프로세스를 종료할 때 발생한다.

프로세스의 메모리 구조

프로세스의 메모리 구조는 위에서부터 스택(stack), 힙(heap), 데이터 영역(BSS segment, Data segment), 코드영역(code segment)로 나누어 진다. 스택은 위 주소부터 할당되고 힙은 아래 주소부터 할당된다.



스택영역(Stack)

- 함수 호출 시 생성되는 지역변수와 매개변수가 저장된다.
- 함수 호출 시 할당되며 실행이 끝나면 메모리에서 해제된다.
    - 함수 호출시 스택에 push되고, return을 하면 스택에서 pop

    - 할당과 해제를 반복하기 때문에 데이터의 크기가 불확실하다.
    - 이렇게 스택 영역에 저장된 함수 호출 정보를 Stack Frame 이라고 한다.
- enum과 같은 값 타입의 데이터가 저장된다.
- 메모리의 높은 주소 → 낮은 주소로 순차적으로 할당된다.
- 컴파일 시에 크기가 결정된다.
- 운영체제에 따라 제한된 스택 영역의 크기가 다르다. 이 크기를 초과해 push할 수 없다.

힙영역(Heap)

- 사용자가 직접 관리하는 영역으로 사용자에 의해 메모리가 동적으로 할당, 해제된다.
- 메모리 주소 값에 의해서만 참고되고 사용된다.
- Java의 경우 가비지 콜렉터가 관리한다.
- 런타임 시(프로그램이 실행되는 도중)에 힙 영역의 크기가 결정된다.
- class 나 배열 같은 참조 타입의 데이터가 저장된다.

스택 영역 vs 힙 영역

- 스택 영역과 힙 영역은 같은 공간을 사용한다. 그래서 스택 영역이 클 수록 힙 영역이 작아지고 힙 영역이 클 수록 스택 영역이 작아진다.
- 스택 영역이 높은 주소 → 낮은 주소로 힙 영역이 낮은 주소 → 높은 주소로 할당되기 때문에 자신의 영역이 상대의 영역을 침범하는 사태가 일어날 수 있다. 이를 각각 스택 오버 플로우, 힙 오버 플로우라고 한다.
- 스택의 할당 속도가 힙 할당속도보다 빠르다.
    - 스택은 컴파일 시 이미 할당된 공간을 사용한다.
    - 힙은 사용자가 따로 할당해서 사용하는 공간이다.

데이터영역(Data)

- 컴파일 단계에서 메모리를 할당한다.

- 전역변수와 static 변수등이 저장되어 있는 정적인 데이터 영역이다.
- 프로그램 실행 시 전역변수와 static 변수는 메인 함수가 호출되기 전에 데이터 영역에 할당된다.
- 프로그램이 실행되고 끝날 때까지 메모리에 남아있다.
- 데이터 영역은 BSS 영역과 Code영역으로 구분한다.
  - BSS영역은 초기화 되지 않은 전역변수 또는 static 변수등이 할당된다.
  - Data영역은 초기화 된 전역변수 또는 static 변수등이 할당된다.

코드영역(Code)

- 사용자가 작성한 코드가 저장되는 영역이다. 즉, 실행할 프로그램의 코드가 저장된다.
- 읽기전용이다.
- 프로그램이 실행되고 끝날 때까지 메모리에 남아있다.

PCB (Process Control Block)

PCB란?

운영체제에서 프로세스에 대한 메타 데이터를 저장한 운영체제 커널의 자료구조를 말한다. 운영체제에 따라 다르지만 일반적으로 PCB는 프로세스 식별자, 프로세스 상태, CPU 스케줄링 정보등이 포함된다. PCB는 프로세스의 중요한 정보를 포함하기 때문에 일반 사용자가 접근하지 못하도록 커널 스택의 메모리 가장 앞부분에 위치한다.

> **메타 데이터**
> 데이터에 관한 구조화된 데이터이자 데이터를 설명하는 작은 데이터, 대량의 정보 가운데서 찾고 있는 정보를 효율적으로 찾아내서 이용하기 위해 일정한 규칙에 따라 콘텐츠에 대해 부여되는 데이터이다.

PCB의 구조

- 프로세스 ID(PID)

  프로세스를 구분할 수 있는 고유의 번호

- **프로세스 스케줄링 상태**
  생성, 준비 등 프로세스가 CPU에 대한 소유권을 얻은 이후 프로세스의 상태를 나타낸다.

- 프로세스 권한

  컴퓨터 자원 또는 I/O 디바이스에 대한 권한

- CPU 레지스터

  프로세스를 실행하기 위해 저장해야 할 레지스터에 대한 정보

- CPU 스케줄링 정보

  CPU 스케줄러에 의한 우선 순위, 최종 실행 시각, CPU 점유 시간 등에 대한 정보

- 프로세스 계정 정보

  프로세스 실행에 사용된 CPU 사용량, 실행한 유저의 정보

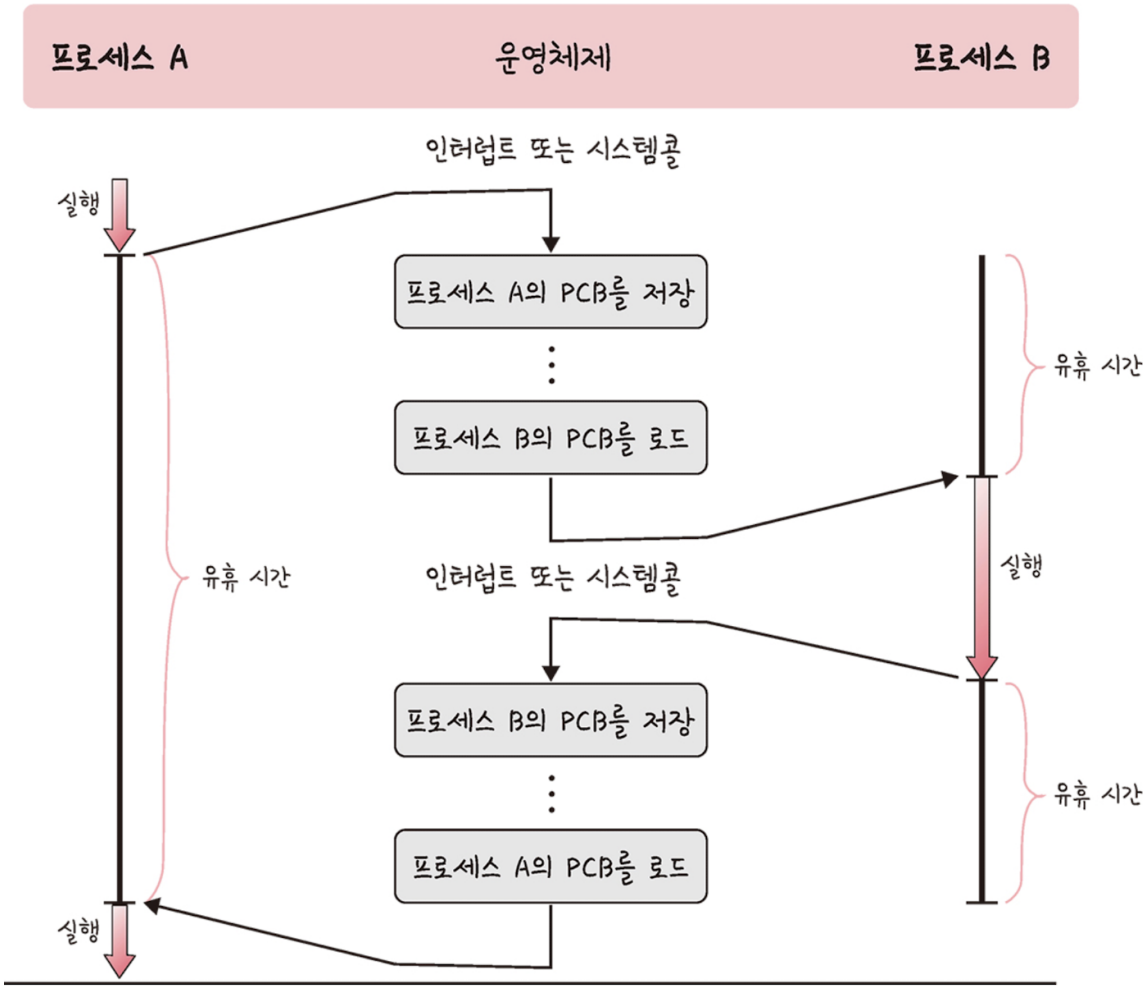
- 프로그램 카운터(PC)

  프로세스에서 실행해야 할 다음 명령어의 주소에 대한 포인터

컨텍스트 스위칭(Context Switching)

PCB를 교환하는 과정을 말한다. 하나의 프로세스가 CPU를 사용한중인 상태에서 다른 프로세스가 CPU를 사용하도록 하기 위해 이전의 프로세스의 상태를 보관하고 새로운 프로세스의 상태를 메모리에 적재하는 작업을 말한다. 프로세스의 Context는 PCB에 저장된다. 그래서 switching이 발생해도 이전에 하던일을 이어서 하는 것이 가능하다.

이런 컨텍스트 스위칭은 한 프로세스에 할당된 시간이 끝나거나 인터럽트에 의해 발생한다. 컴퓨터는 많은 프로그램을 동시에 실행하는 것처럼 보이지만 어떠한 시점에서 실행되고 있는 프로세스는 한개이며, 많은 프로세스가 동시에 구동되는 것처럼 보이는 것은 다른 프로세스와의 컨텍스트 스위칭이 빠른 속도로 실행되기 때문이다.(싱글코어 기준)



컨텍스트 스위칭 과정은 위 그림처럼 컨텍스트 스위칭은 프로세스 A가 실행하다 멈추고, 프로세스 A의 PCB를 저장하고 다시 프로세스 B를 로드하여 실행한다. 그리고 다시 프로세스 B의 PCB를 저장하고 프로세스 A의 PCB를 로드한다.

컨텍스트 스위칭 과정에서 유휴시간이 발생하는 것을 볼 수 있다. 이런 유휴시간은 컨텍스트 스위칭에 드는 비용을 증가시킨다. 컨텍스트 스위칭에 비용을 증가시키는 요인으로는 캐시 미스도 있다.

> **유휴시간**
> 유휴시간은 CPU가 일을 하지 않는 시간을 말한다.

> **캐시미스**
> 컨텍스트 스위칭 발생 시 프로세스가 가지고 있는 메모리 주소가 그대로 있으면 잘못된 주소 변환이 생겨 캐시 클리어 과정을 겪게 되고 이 때문에 캐시미스가 발생한다.

> **스레드에서 컨텍스트 스위칭**
> 스레드에서도 컨텍스트 스위칭은 일어난다. 스레든느 스택 영역을 제외한 모든 메모리를 공유하기 때문에 비용이 더 적고 시간도 적게 걸린다.

멀티 프로세싱

여러 개의 프로세스 즉, 멀티 프로세스를 통해 동시에 두 가지 이상의 일을 수행할 수 있는 것을 말한다. 이를 통해 하나 이상의 일을 병렬로 처리할 수 있으며 특정 프로세스의 메모리, 프로세스 중 일부에 문제가 발생해도 다른 프로세스를 이용해서 처리할 수 있어 신뢰성이 높다. 대표적으로 웹 브라우저가 멀티프로세스 구조를 가지고 있다.

IPC(Inter Process Communication)

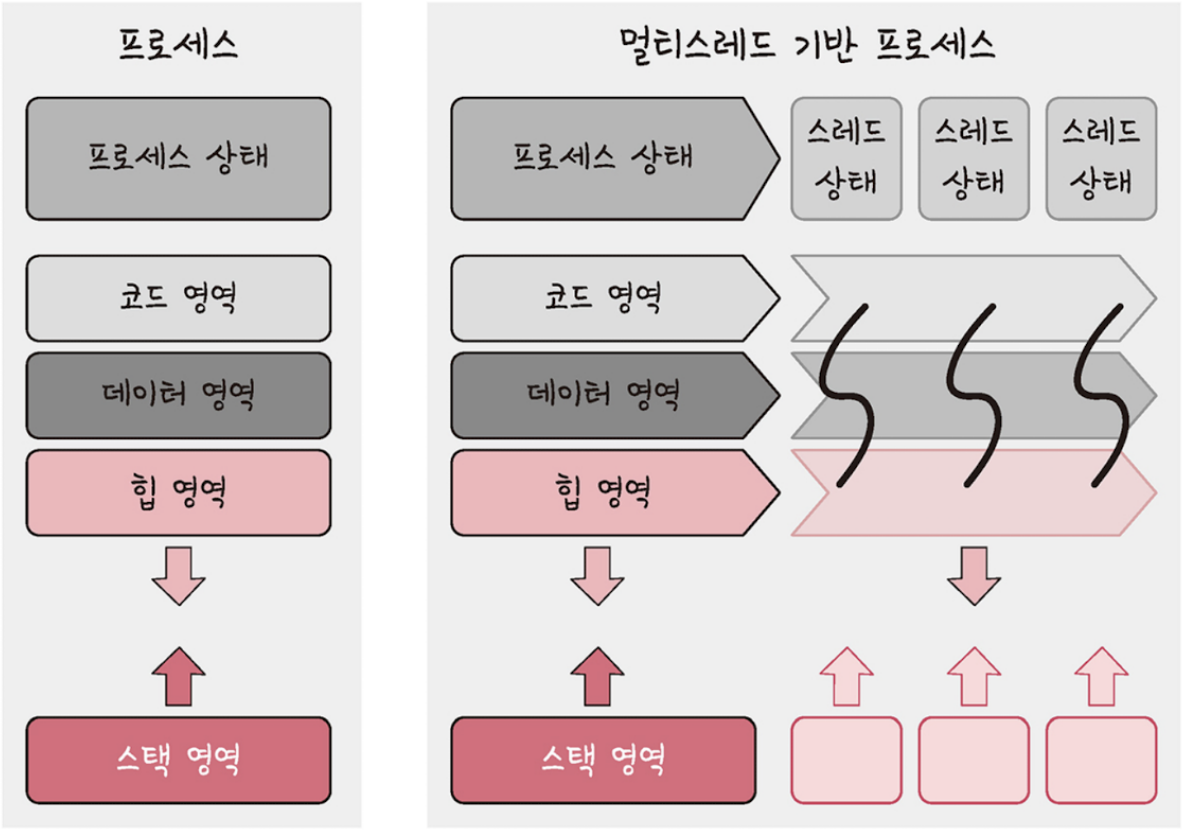
프로세스끼리 데이터를 주고받고 공유 데이터를 관리하는 메커니즘을 말한다. 클라이언트와 서버를 예로 들 수 있는데 클라이언트는 데이터를 요청하고 서버는 클라이언트 요청에 응답하는 것도 IPC의 예이다.

IPC의 종류로는 공유 메모리, 파일, 소켓, 메시지 큐 등이 있다. 이들은 모두 메모리가 완전 공유되는 스레드보다는 속도가 떨어진다.

스레드와 멀티스레딩

스레드(Thread)란?

스레드는 프로세스에서 작업을 실행하는 가장 작은 단위를 말한다. 프로세스는 하나 이상의 스레드가 존재하며 작업을 수행한다. 또한, 두 개 이상의 스레드를 갖는 프로세스를 멀티스레드 프로세스라고 한다.



코드, 데이터, 스택 힙을 각각 생성하는 프로세스와 달리 스레드는 코드, 데이터, 힙은 스레드끼리 서로 공유한다. 그 외의 영역은 각각 생성된다.

멀티스레딩(multi threading)

멀티 스레딩은 프로세스 내 작업을 여러 개의 스레드, 멀티 스레드로 처리하는 기법이며 스레드끼리 서로 자원을 공유하기 때문에 효율성이 높다.

[멀티스레딩의 장점]

- 한 프로세스내의 스레드는 자원들과 메모리를 공유하기 때문에 메모리공간과 시스템 자원의 효율성이 높다.

- 서로 독립적인 작업들을 작은 단위로 나누고 동시에 실행하는 것 처럼 보여주는 동시성에 장점을 갖는다.
- Data, Heap 영역을 이용해 데이터를 주고 받아 스레드 간 통신이 간단하다.

[멀티스레딩의 단점]

- 설계가 복잡하다.
- 하나의 스레드에 문제가 생기면 전체 스레드가 영향을 받을 수 있다.
- 스레드간의 자원 공유는 전역변수(데이터 세그먼트)를 이용하기 때문에 다른 스레드가 동시에 사용할 때 충돌이 발생할 수 있다.
- 서로 다른 스레드가 Stack을 제외한 메모리 공간을 공유하기 때문에 동기화 문제가 발생할 수 있다.

공유자원과 임계영역

공유자원(Shared Resource)

공유 자원은 시스템 안에서 각 프로세스, 스레드가 함께 접근할 수 있는 메모리, 데이터 등의 자원이나 변수를 의미한다. 이 공유 자원을 두 개 이상의 프로세스가 동시에 읽거나 쓰는 상황을 **경쟁 상태**라고 한다. 동시에 접근을 시도할 때 접근의 타이밍이나 순서 등이 결과값에 영향을 줄 수 있다.

임계영역(Critical Section)

둘 이상의 프로세스, 스레드가 공유 자원에 접근할 때 순서등의 이유로 결과가 달라지는 프로그램의 코드 영역을 말한다. 임계구역에서는 프로세스들이 동시에 작업할 수 없다.

임계 영역을 해결하기 위한 방법으로 뮤텍스, 세마포어, 모니터 세가지 방법이 있다. 이 방법 모두 상호 배제, 한정 대기, 융통성이란 조건을 만족한다.

이 방법에 토대가 되는 매커니즘은 락(Lock)이다.

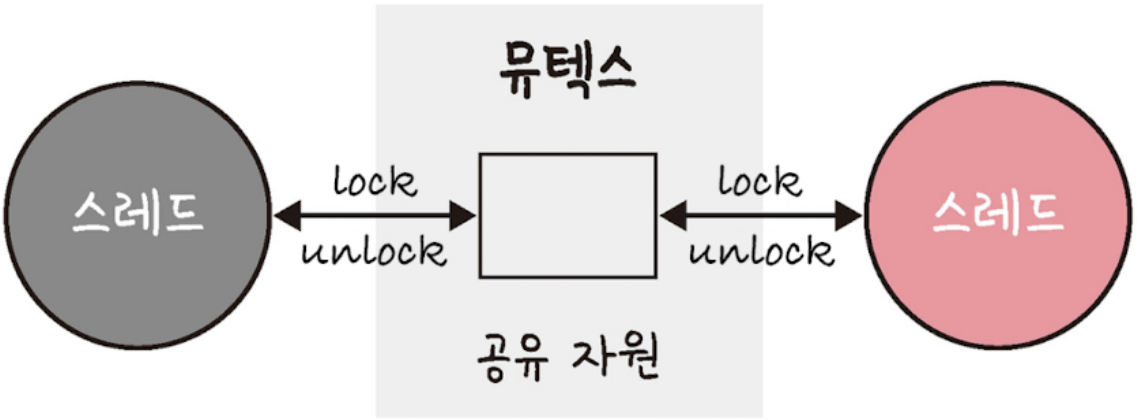
임계영역 문제를 해결하기 위한 기본조건

- **상호배제**
  한 프로세스가 임계 영역에 들어갔을 때 다른 프로세스는 들어올 수 없다.
- **한정 대기**
  특정 프로세스가 영원히 임계 영역에 들어가지 못하면 안된다.
- **융통성**
  한 프로세스가 다른 프로세스의 일을 방해해서는 안된다.

임계영역 문제를 위한 해결책

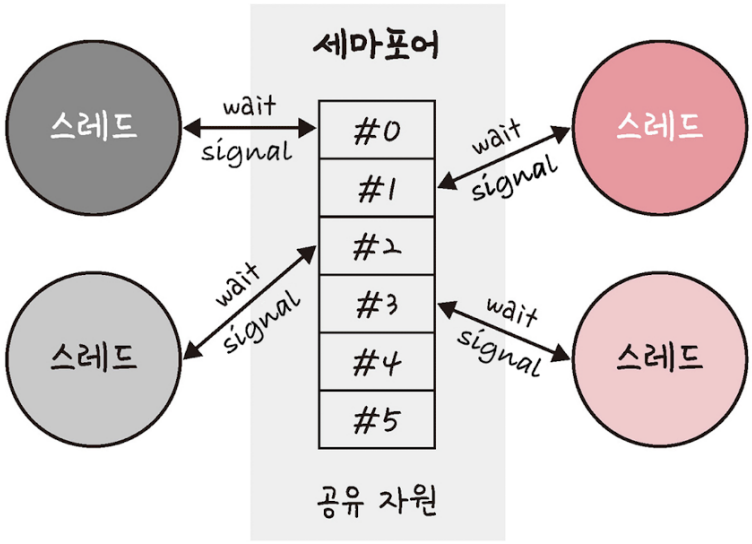
뮤텍스

뮤텍스는 프로세스나 스레드가 공유자원을 lock()을 통해 잠금 설정을하고 사용한 후에는 unlock()을 통해 잠금 해제하는 객체이다. 잠금이 설정되면 다른 프로세스나 스레드는 잠긴 코드 영역에 접근할 수 없고 해제는 그 와반대이다. 뮤텍스는 잠금 또는 잠금 해제라는 상태만을 갖는다.



세마포어

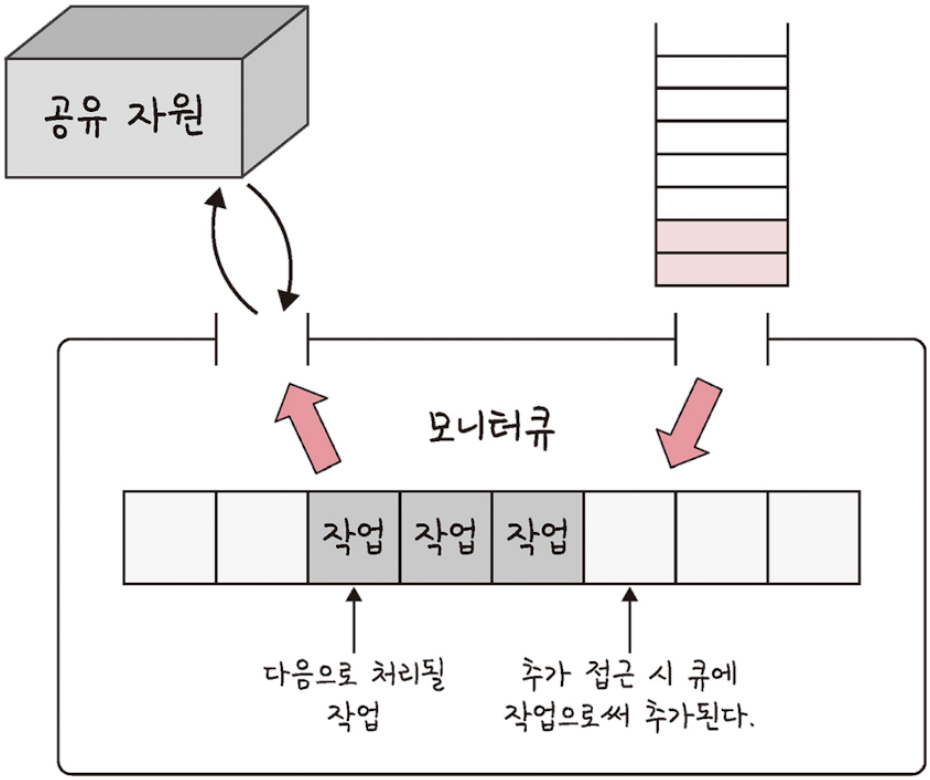
일반화된 뮤텍스이다. 간단한 정수 값과 두 가지 함수 wait(P 함수라고도 한다.) 및 signal(V 함수라고도 한다.)로 공유 자원에 대한 접근을 처리한다. wait()는 자신의 차례가 올 때까지 기다리는 함수이며, signal()은 다음 프로세스로 순서를 넘겨주는 함수이다.



프로세스나 스레드가 공유 자원에 접근하면 세마포어에서 wait() 작업을 수행하고 프로세스나 스레드가 공유 자원을 해제하면 세마포어에서 signal() 작업을 수행한다. 세마포어에는 조건 변수가 없고 프로세스나 스레드가 세마포어 값을 수정할 때 다른 프로세스나 스레드는 동시에 세마포어 값을 수정할 수 없다.

모니터

모니터는 둘 이상의 스레드나 프로세스가 공유 자원에 안전하게 접근할 수 있도록 공유 자원을 숨기고 해당 접근에 대해 인터페이스만 제공한다.



모니터는 모니터 큐를 통해 공유 자원에 대한 작업들을 순차적으로 처리한다. 모니터는 세마포어보다 구현하기 쉬우며 모니터에서 상호 배제는 자동인 반면 세마포어에서 상호 배제는 명시적으로 구현해야 한다.

교착상태(DeadLock)

교착상태란?

두 개 이상의 프로세스들이 서로가 가진 자원을 기다리며 중단된 상태를 말한다.

교착 상태의 원인

- 상호배제

  한 프로세스가 자원을 독점하고 있으며 다른 프로세스들은 접근이 불가능한 상태

- 점유대기

  특정 프로세스가 점유한 자원을 다른 프로세스가 요청하는 상태

- 비선점

  다른 프로세스의 자원을 강제적으로 가져올 수 없는 상태

- 환영대기

  프로세스 A는 프로세스 B의 자원을 요구하고 프로세스 B는 프로세스 A의 자원을 요구하는 등 서로가 서로의 자원을 요구하는 상황

교착상태 해결방법

1. 자원을 할당할 때 교착 상태가 발생하는 조건이 성립하지 않도록 한다.
2. 교착 상태 가능성이 없을 때만 자원을 할당하며, 프로세스당 요청할 자원들의 최대치를 통해 자원 할당 가능 여부를 파악하는 **은행원 알고리즘**을 사용한다.
3. 교착 상태가 발생하면 사이클이 있는지 찾아보고 이에 관련된 프로세스를 한 개씩 지운다.
4. 교착 상태는 매우 드물게 일어나기 때문에 이를 처리하는 비용이 더 커져서 교착상태가 발생하면 사용자가 작업을 종료한다.

> **은행원 알고리즘**
> 총 자원의 양과 현재 할당한 자원의 양을 기준으로 안정 또는 불안정 상태로 나누고 안정 상태로 가도록 자원을 할당하는 알고리즘