Process & Thread

1. 프로세스 & 스레드 개념

- **프로세스 (Process)**

  : 운영체제로부터 자원을 할당받은 **작업의 단위**

- **스레드 (Thread)**

  : 프로세스가 할당받은 자원을 이용하는 **실행 흐름의 단위**

2. 프로그램과 프로세스

- **정적 프로그램 (Static Program)**

  : 프로그램은 윈도우의 ***.exe** 파일이나 Mac의 ***.dmg** 파일과 같은 **컴퓨터에서 실행할 수 있는 파일**을 통칭한다. 단, 아직 **파일을 실행하지 않은 상태**이기 때문에 정적 프로그램을 줄여서 프로그램이라고 부른다.

  어떠한 프로그램을 개발하기 위해서는 자바나 C언어와 같은 언어를 이용해 코드를 작성하여 완성된다. 즉, 프로그램은 쉽게 말해서 그냥 코드 덩어리인 것이다.

- **프로세스 (Process)**

  : 프로그램이 그냥 코드 덩어리이면, 프로세스는 프로그램을 실행시켜 정적인 프로그램이 동적으로 변하여 **프로그램이 돌아가고 있는 상태**를 말한다. 즉, 컴퓨터에서 작업 중인 프로그램을 의미하는 것이다.

  모든 프로그램은 운영체제가 실행되기 위한 **메모리 공간**을 할당해줘야 실행될 수 있다. 그래서 프로그램을 실행하는 순간 파일은 컴퓨터 메모리에 올라가게 되고, 운영체제로부터 시스템 자원(CPU)을 할당받아 프로그램 코드를 실행시켜 우리가 서비스를 이용할 수 있게 되는 것이다.

프로그램	프로세스
어떤 작업을 하기 위해 실행할 수 있는 파일	실행되어 작업중인 컴퓨터 프로그램
파일이 저장 장치에 있지만 메모리에는 올라가 있지 않은 정적인 상태	메모리에 적재되고 CPU 자원을 할당받아 프로그램이 실행되고 있는 상태

3. 스레드

- **프로세스의 한계**

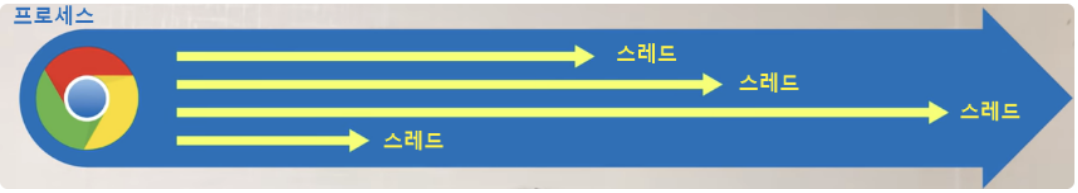
  : 과거에는 프로그램을 실행할 때 프로세스 하나만을 사용해서 이용했었다. 하지만 기술이 발전됨에 따라 프로그램이 복잡해지고 다채로워짐으로써 프로세스 작업 하나만을 사용해서 프로그램을 실행하기에는 한계가 있었다.

  오늘날 컴퓨터는 파일을 다운 받으며 다른 일을 하는 멀티 작업은 너무 당연한 기능이라고 생각할 지 모르지만, 과거에는 파일을 다운받으면 실행 시작부터 실행 끝까지 프로세스 하나만을 사용하기 때문에 다운이 완료될 때까지 하루종일 기다려야 했다. 그렇다고 동일한 프로그램을 여러 개의 프로세스로 만들게 되면, 그만큼 메모리를 차지하고 CPU에서 할당받는 자원이 중복되게 될 것이다. 스레드(Thread)는 이러한 프로세스 특성의 한계를 해결하기 위해 탄생했다.

- **스레드의 개념**

  : 스레드란, **하나의 프로세스 내에서 동시에 진행되는 작업 갈래, 흐름의 단위**를 말한다.

이해하기 쉽게 비유를 들자면, 크롬 브라우저가 실행되면 프로세스가 하나가 생성될 것이다. 그런데 우리는 브라우저에서 파일을 다운 받으며 온라인 쇼핑을 하면서 게임을 하기도 한다.



즉, 하나의 프로세스 안에서 여러가지 작업들 흐름이 동시에 진행되기 때문에 가능한 것인데, 이러한 일련의 작업 흐름들을 스레드라고 하며 여러개가 있다면 이를 멀티(다중) 스레드라고 부른다.

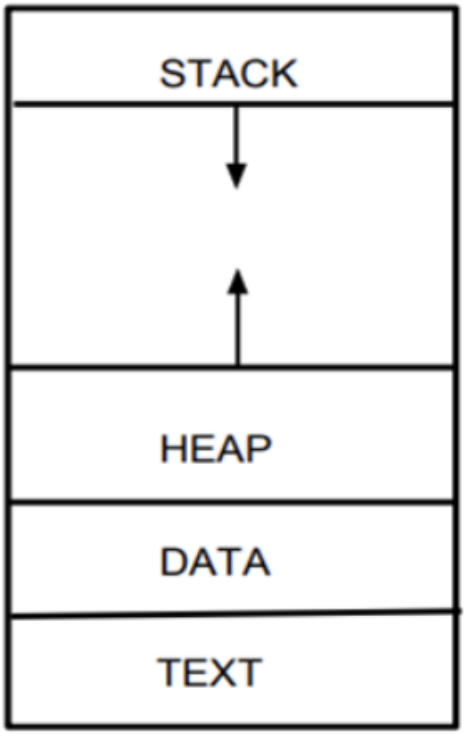
아래 그림에서 보듯이 하나의 프로세스 안에 여러개의 스레드들이 들어있다고 보면 된다. **스레드 수가 많을수록** 당연히 프로그램 속도도 동시에 하는 작업이 많아져 **성능이 올라간다.**

일반적으로 하나의 프로그램은 하나 이상의 프로세스를 가지고 있고, 하나의 프로세스는 반드시 하나 이상의 스레드를 갖는다. **즉, 프로세스를 생성하면 기본적으로 하나의 main 스레드가 생성되게 된다**. 스레드 2개, 3개 ... 는 프로그램을 개발한 개발자가 직접 프로그래밍하여 위치 시켜주어야 한다.

4. 프로세스와 스레드의 내부 작동 방식

- **프로세스& 스레드의 메모리**

  : 프로그램이 실행되어 프로세스가 만들어지면 다음 4가지의 메모리 영역으로 구성되어 할당받게 된다.



1. **코드 영역 (Code / Text) :** 프로그래머가 작성한 프로그램 함수들의 코드가 CPU가 해석 가능한 기계어 형태로 저장되어 있다.
2. **데이터 영역 (Data) :** 코드가 실행되면서 사용하는 전역 변수나 각종 데이터들이 모여있다. 데이터 영역은 .data, .rodata, .bss 영역으로 세분화된다.
    - .data : 전역 변수 또는 static 변수 등 프로그래밍 사용하는 데이터를 저장
    - .BSS : 초기값 없는 전역 변수, static 변수가 저장
    - .rodata : const같은 상수 키워드 선언 된 변수나 문자열 상수가 저장
3. **스택영역 (Stack) :** 지역 변수와 같은 호출한 함수가 종료되면 되돌아올 임시적인 자료를 저장하는 독립적인 공간이다. **Stack 은 함수의 호출과 함께 할당되며, 함수의 호출이 완료되면 소멸**한다. 만일 stack 영역을 초과하면 stack overflow 에러가 발생한다.
4. **힙 영역 (Heap)** : 생성자, 인스턴스와 같은 동적으로 할당되는 데이터들을 위해 존재하는 공간이다. 사용자에 의해 메모리 공간이 동적으로 할당되고 해제된다.

코드 영역과 데이터 영역은 선언할 때 그 크기가 결정되는 정적 영역이지만, **스택 영역과 힙 영역**은 프로세스가 실행되는 동안 **크기가 늘어났다 줄어들기도 하는 동적 영역**이기 때문에 이를 표현한 것이다.

프로그램이 여러개 실행된다면 메모리에 프로세스들이 담길 주소 공간이 생성되게 되고 그 안에 Code, Data, Stack, Heap 공간이 만들어지게 된다.

5. 스레드의 자원 공유

스레드는 프로세스가 할당 받은 자원을 이용하는 실행의 단위로서, 스레드가 여러개가 있으면 우리가 파일을 다운받으며 동시에 웹 서핑을 할 수 있게 해준다. 스레드끼리 **프로세스의 자원을 공유하면서** 프로세스 실행 흐름의 일부가 되기 때문에 동시 작업이 가능한 것이다.

이때, 프로세스의 4가지 메모리 영역(Code, Data, Heap, Stack) 중 **스레드**는 **Stack**만 할당 받아 복사하고, **Code, Data, Heap**은 프로세스 내의 **다른 스레드들과 공유**된다. 따라서 각각의 스레드는 별도의 stack을 가지고 있지만, heap 메모리는 고유하기 때문에 서로 다른 스레드에서 가져와 읽고 쓸 수 있게 된다.

stack은 함수 호출 시 전달되는 인자, 되돌아갈 주소값, 함수 내에서 선언하는 변수 등을 저장하는 메모리 공간이기 때문에, 독립적인 스택을 가졌다는 것은 독립적인 함수 호출이 가능하다는 것은 독립적인 실행 흐름이 추가된다는 말이다.
즉, stack을 가짐으로써 스레드는 독립적인 실행 흐름을 가질 수 있게 되는 것이다.

> 💡 반면에 프로세스는 기본적으로 프로세스끼리 다른 프로세스의 메모리에 직접 접근할 수는 없다.

이렇게 구성한 이유는 하나의 프로세스를 다수의 실행 단위인 스레드로 구분하여 자원을 공유하고, 자원의 생성과 관리의 중복성을 최소화하여 수행 능력을 올리기 위해서다.

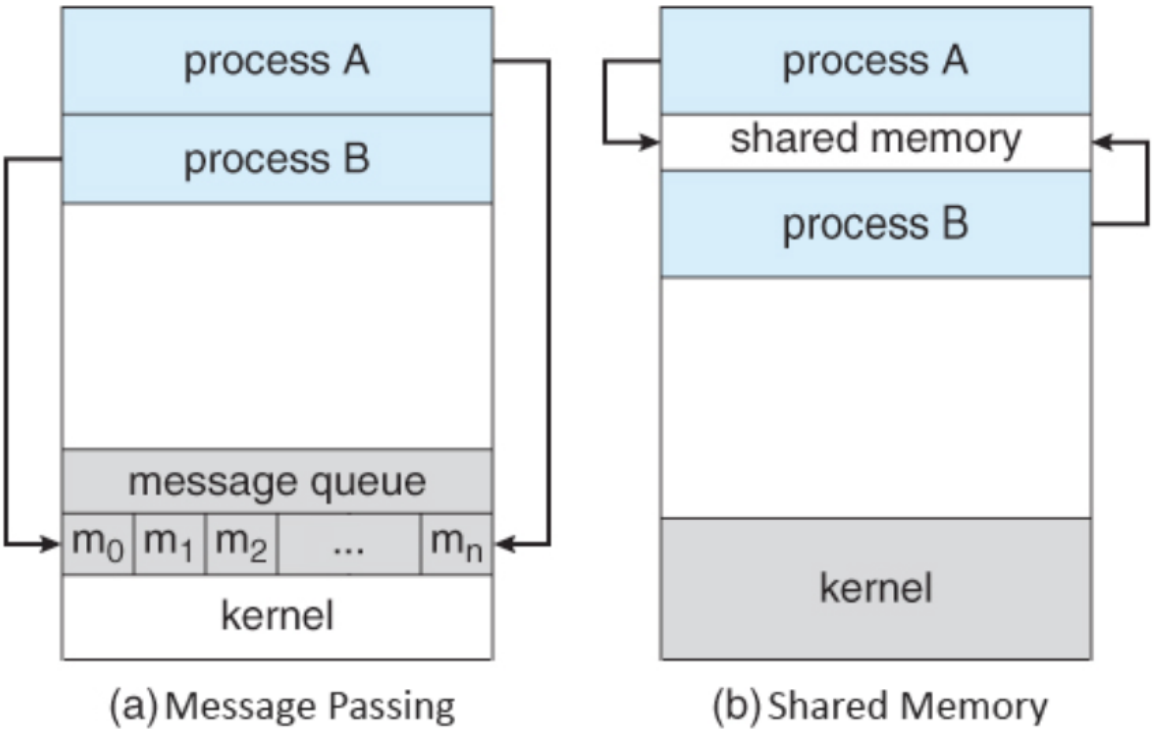
6. 프로세스의 자원 공유

기본적으로 각 프로세스는 메모리에 별도의 주소 공간에서 실행되기 때문에 한 프로세스는 다른 프로세스의 변수나 자료구조에 접근할 수는 없다. 그렇다면 프로세스는 영원히 다른 프로세스 정보에 접근할 수 없을까?

현재 우리가 사용하는 대부분의 컴퓨터 프로그램을 보면 다른 프로그램에 있는 정보를 가져오는 경우는 심심치 않게 볼 수 있을 것이다.

이처럼 특별한 방법을 통해 **프로세스가 다른 프로세스 정보에 접근**하는 것이 가능하다.

1. **IPC (Inter-Process Communication) 사용**
2. **LPC (Local inter-Process Communication) 사용**
3. **별도로 공유 메모리를 만들어서 정보를 주고받도록 설정**



(a) Message Queue : FIFO 방식의 데이터 전송으로 동일한 key값을 이용하여 A프로세스의 데이터를 B프로세스가 받을 수 있다. 반대로, B프로세스에서도 A프로세스로 데이터를 보낼 수 있다. (양방향)

(b) Shared Memory : 커널 공간에 메모리 공간을 만들어 해당 공간을 마치 변수처럼 사용하는 방법이다. 공유 메모리 key를 사용하여 여러 프로세스 접근이 가능하다.

그러나 프로세스 자원 공유는 단순히 CPU 레지스터 교체 뿐만이 아니라 RAM과 CPU 사이의 캐시 메모리까지 초기화되기 때문에 **자원 부담이 크다는 단점**이 있다. 그래서 **다중 작업이 필요한 경우 스레드를 이용하는 것이 훨씬 효율적**이라, 현대 컴퓨터의 운영체제에선 다중 프로세싱을 지원하고 있지만 다중 스레딩을 기본으로 하고 있다.

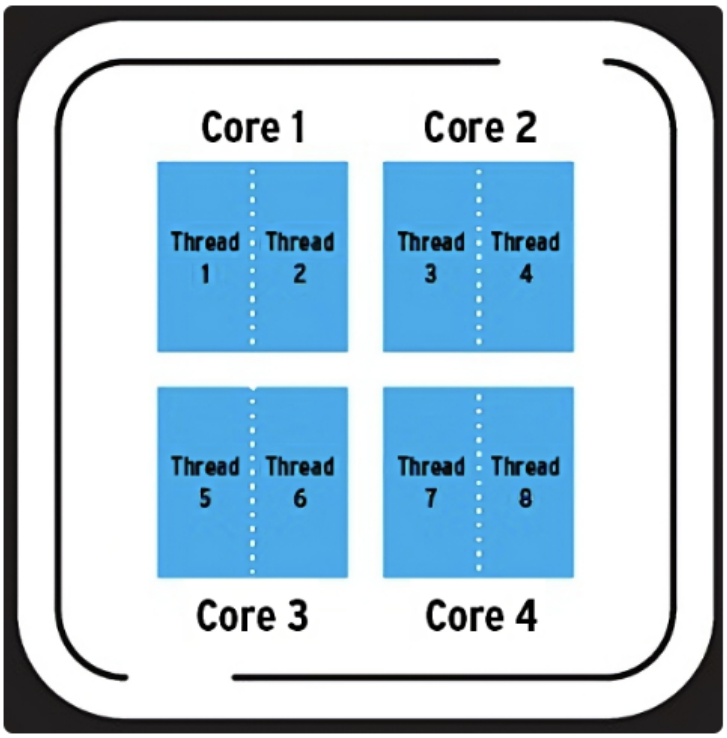
7. 프로세스 & 스레드의 동시 실행 원리

: 우리가 음악을 들으면서, 웹 서핑을 하고, 메신저의 메시지를 확인할 수 있는 이유는 컴퓨터 내부적으로 **프로세스와 스레드를 동시에 처리하는 멀티태스킹 기술** 때문이다. 하지만 여기서 동시에 처리한다는 것이 심플하게 CPU 프로세서가 프로그램들을 한꺼번에 동시에 돌리는 것으로 생각하겠지만, 내부적으로 복잡한 원리에 의해 처리가 된다.

- **멀티 코어와 스레드**

  : 한번 컴퓨터 견적을 맞춰본 경험이 있는 독자분들은 **4코어 8쓰레드 CPU**에 대한 단어를 본 적이 있을 것이다.

  CPU 한 개는 여러개의 코어를 가질 수 있다. 코어는 말 그대로 CPU 코어 유닛이다. 즉, 명령어를 메모리에서 뽑아 해석하고 실행하는 반도체 유닛이 4개가 있는 것이다. **4코어가 물리적 코어 개수**이면, **8쓰레드는 논리적 코어 개수**이다. 이 경우 **물리적 코어 하나가 스레드 두개 이상을 동시에 실행가능하다는 의미**가 된다. 즉, 운영체제가 8개의 작업을 동시에 처리할 수 있다는 뜻이다. 이를 **하이퍼스레딩 기술**이라 말한다.



> 💡 단, 여기서 CPU의 쓰레드는 우리가 배운 프로세스의 쓰레드와는 조금 다른 개념이다. 엄밀히 말하자면 **CPU의 스레드**는 **하드웨어적 스레드**이고, **프로그램의 스레드**는 **소프트웨어적 스레드**로 구분한다.

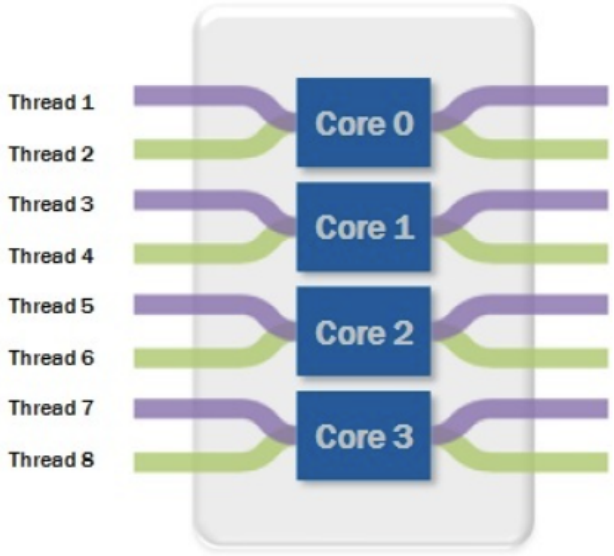
8. CPU의 작업 처리 방식

그런데 우리는 컴퓨터를 이용할 때 프로그램을 수십, 수백개를 켜놓고 이용한다. 그럼 그 수십수백개의 프로세스들을 고작 8개의 논리적인 스레드로 어떻게 처리하는 것일까?

이 원리를 알기 위해선 **병렬성(Parallelism)**과 **동시성(Concurrency)**이라는 개념을 알고 있어야 한다. 이 개념은 운영체제의 프로세스, 스레드를 이해하는데 있어 가장 핵심 골자이다.

- **병렬성(Parallelism)**

  : 병렬성은 직관적으로 명령어를 메모리에서 뽑아 해석하고 실행하는 반도체 유닛인 **여러개의 코어에 맞춰** 여러개의 프로세스, 스레드를 돌려 병렬로 작업들을 동시 수행하는 것을 말한다.



  듀얼 코어, 쿼드 코어, 옥타 코어 등 이런 명칭이 붙는 **멀티코어 프로세스**가 달린 컴퓨터에서 할 수 있는 방식이다.

- **동시성(Concurrency)**

  : 동시성은 **둘 이상의 작업이 동시에 실행**되는 것을 의미한다. 이 '동시'라는 의미에서 병렬성과 동시성의 한글 의미가 헷갈릴 수 있다. **Parallelism가 물리적으로 정말로 동시에 실행하는 것**이라고 하면, **Concurrency는 동시에 실행하는 것처럼 보이게 하는 것**으로 이해하면 된다.

단, 이때 작업들을 번갈아가면서 실행할 때 작업들을 **아주 작게 나누어 아주 조금씩만 작업을 수행**하고 다음 작업으로 넘어가는 식으로 동작된다. 이렇게 하는 이유는 여러 작업을 동시에 처리하는 것 처럼 보이게 만들어, 사용자에게 **더 빠른 반응성**을 제공하기 위해 **Context Switching**이라고 부른다.

- **동시성이 필요한 이유**

  : 그런데 상식적으로 생각해보면 동시성(Concurrency)은 '동시에 돌아가는 것처럼' 보이는 거지, 정말 실제로 동시에 돌아가는 것이 아니기 때문에 최종 작업이 걸리는 시간은 거의 차이가 없을 것이다. 병렬성은 정말로 각 코어에 프로세스를 나눠 실행하는거니까 듀얼 코어면 반 이상 줄어들텐데 말이다. 그렇다면 왜 이렇게 번거롭게 작업들을 스위칭하며 처리하는 것일까?

  1. **하드웨어적 한계**

     : CPU 발열 때문에 클럭으로 성능을 올리기에는 한계에 봉착했기 때문에 코어의 성능을 올리는 대신 코어를 여러개 탑재하여 쿼드 코어, 옥타 코어 CPU들을 출시하고 있다. 하지만 아무리 코어를 많이 넣어도 수십개의 코어를 넣을 순 없으니 결국 하드웨어적 제한이 걸리게 되고, 수십 수백개의 프로세스를 돌리기 위해선 결국 동시성이 필요한 것이다.

  2. **논리적 효율**

     : 4코어 8스레드의 CPU 환경에선 현재 총 16개의 작업이 있다고 가정을 해보자. 그중 8개는 오래 걸리는 작업이고, 나머지 8개는 짧은 시간을 필요로 하는 작업이라고 하자.

     논리적인 8개의 코어이니 최대 8개까지 동시에 실행할 수 있을텐데, 만일 최악의 경우 8개의 오래 걸리는 작업이 먼저 동시에 처리되기 시작했다고 하자. 이 겨웅 나머지 가벼운 8개의 작업은 처리하는데 짧은 시간이 걸리는데도 불구하고 현재 처리중인 8개의 작업이 다 끝날때까지 기다려야 할 것이다 따라서 이러한 비효율적인 면을 극복하기 위해 **작업을 아주 잘게 나눠 번갈아 가면서 처리**하는 동시성 개념을 채택한 것이다.

     따라서 최대 8개의 작업에 대해서 8개의 논리적인 스레드가 병렬적으로 아주 빠르게 동시적으로 작업을 하면서, 그보다 많은 수십개의 소프트웨어적 스레드가 있다면 **적절히 병렬성과 동시성**을 섞어 동시에 돌리게 된다.

9. 프로세스& 스레드의 생명 주기

: **프로세스와 스레드는 각각의 생명 주기**를 가지고 있으며, **운영체제는 이러한 생명 주기를 관리**하고, 프로세스와 스레드를 조정하여 시스템 자원을 효율적으로 사용할 수 있게 된다.

- **프로세스 스케줄링**

  : 프로세스 스케줄링(Process Scheduling)은 운영체제에서 CPU를 사용할 수 있는 프로세스를 선택하고, CPU를 할당하는 작업을 말한다. **프로세스 스케줄링은 프로세스의 우선순위, 작업량 등을 고려하여 효율적으로 배치**하여, 이를 통해 운영체제는 CPU를 효율적으로 사용하며 시스템 전반적인 성능을 향상시킨다. 그래서 스케줄링은 멀티 태스킹 작업을 만들어내는데 있어 핵심적인 부분이다.