Raspberry Pi Thread

PMOD 전성현

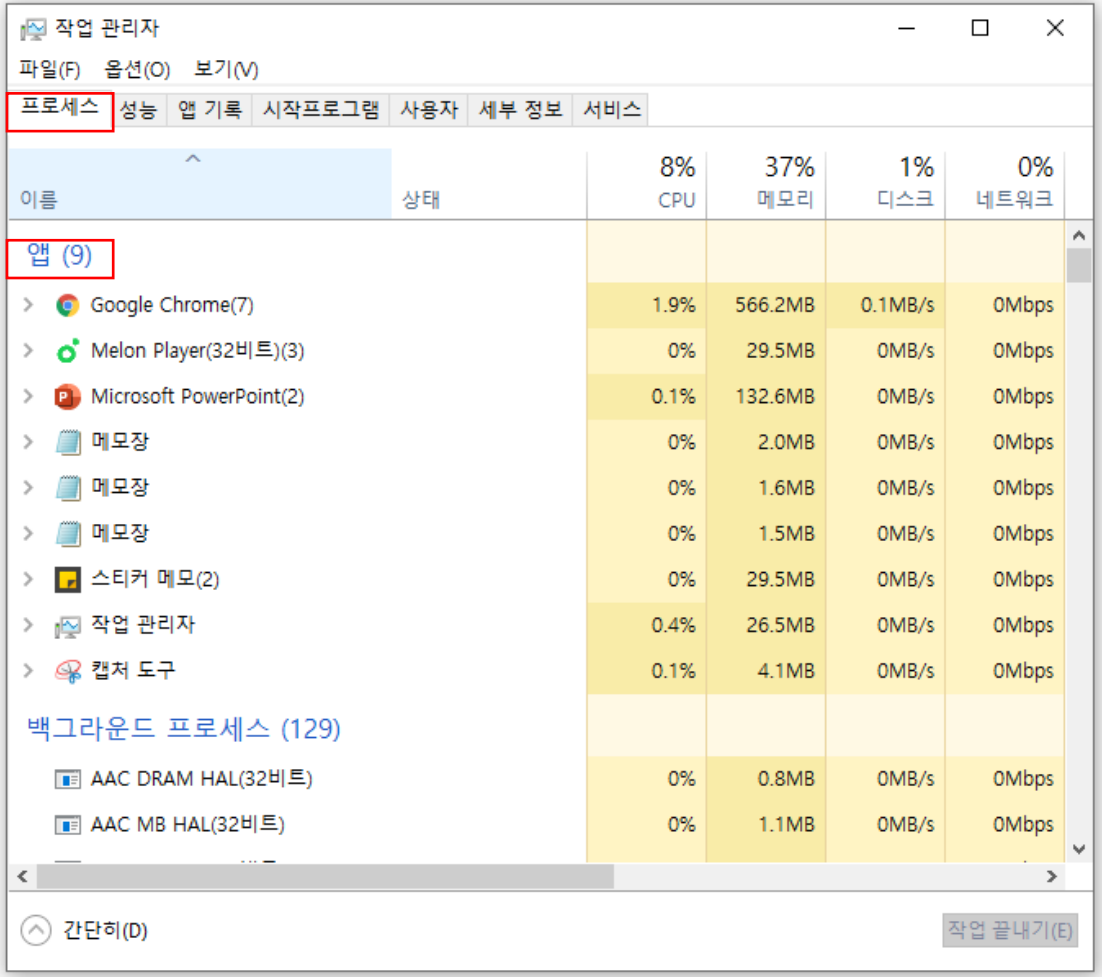
목차

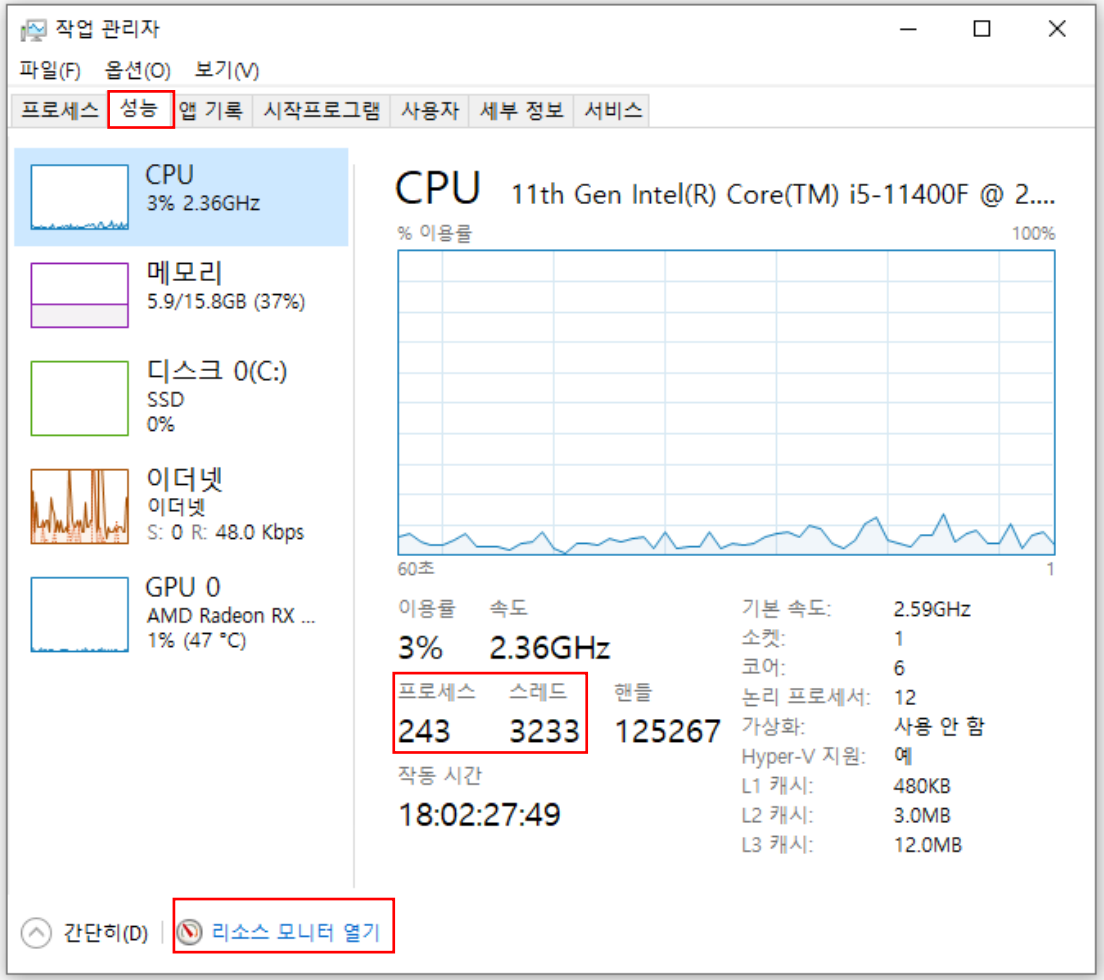
Process

- Program
  실행되기 전 상태의 명령어, 코드 및 정적인 데이터의 묶음

- Process
  실행 중인 Program
  운영체제로부터 시스템 자원을 할당 받는 작업의 단위
  (부연 설명이 없으면)하나의 스레드만 가지고 있는 단일 스레드 프로세스

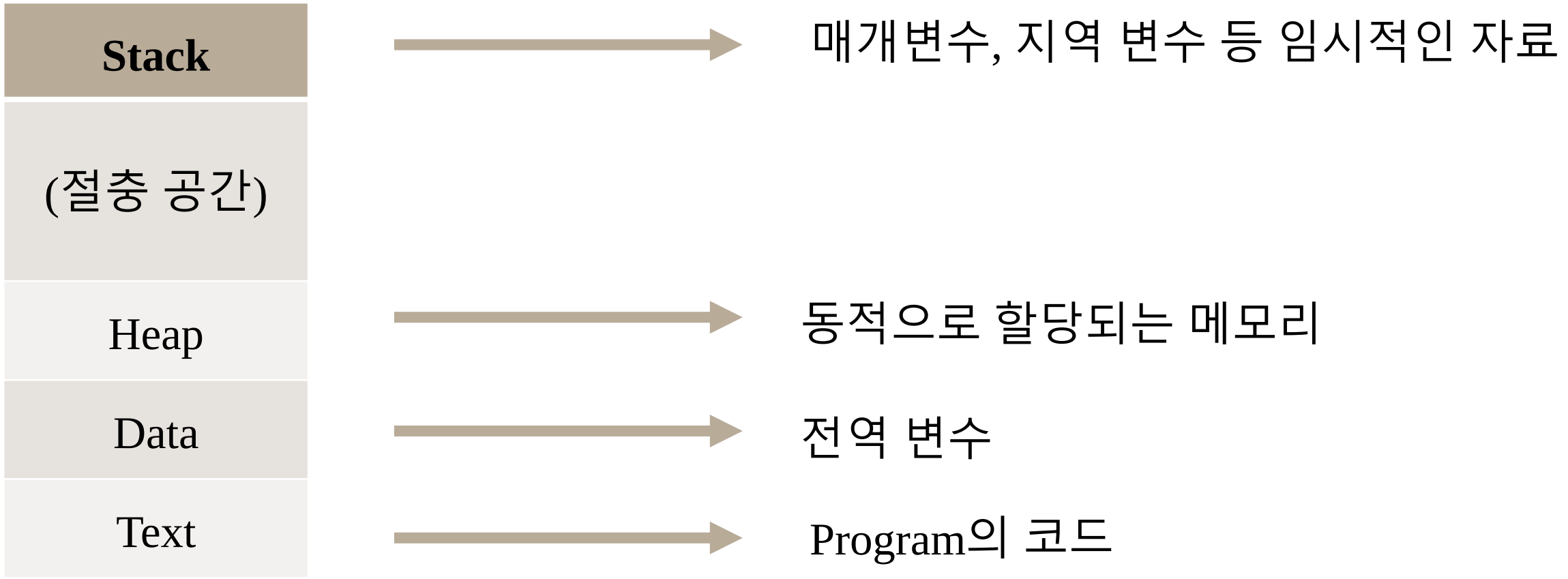
Process


작업 관리자
파일(F) 옵션(O) 보기(V)
프로세스 성능 앱 기록 시작프로그램 사용자 세부 정보 서비스
이름 상태 8% CPU 37% 메모리 1% 디스크 0% 네트워크
앱 (9)
Google Chrome(7) 1.9% 566.2MB 0.1MB/s 0Mbps
Melon Player(32비트)(3) 0% 29.5MB 0MB/s 0Mbps
Microsoft PowerPoint(2) 0.1% 132.6MB 0MB/s 0Mbps
메모장 0% 2.0MB 0MB/s 0Mbps
메모장 0% 1.6MB 0MB/s 0Mbps
메모장 0% 1.5MB 0MB/s 0Mbps
스티커 메모(2) 0% 29.5MB 0MB/s 0Mbps
작업 관리자 0.4% 26.5MB 0MB/s 0Mbps
캡처 도구 0.1% 4.1MB 0MB/s 0Mbps
백그라운드 프로세스 (129)
AAC DRAM HAL(32비트) 0% 0.8MB 0MB/s 0Mbps
AAC MB HAL(32비트) 0% 1.1MB 0MB/s 0Mbps
간단히(D)
작업 끝내기(E)

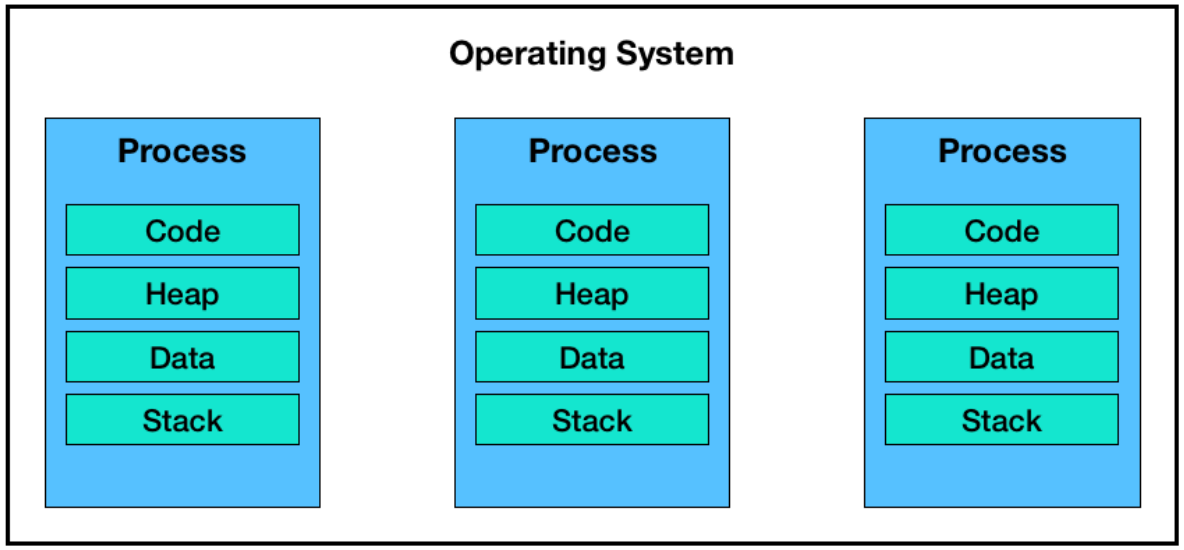

작업 관리자
파일(F) 옵션(O) 보기(V)
프로세스 성능 앱 기록 시작프로그램 사용자 세부 정보 서비스
CPU
3% 2.36GHz
메모리
5.9/15.8GB (37%)
디스크 0(C:)
SSD
0%
이더넷
이더넷
S: 0 R: 48.0 Kbps
GPU 0
AMD Radeon RX ...
1% (47 °C)
CPU 11th Gen Intel(R) Core(TM) i5-11400F @ 2....
% 이용률 100%
60초 1
이용률 속도
3% 2.36GHz
프로세스 스레드 핸들
243 3233 125267
작동 시간
18:02:27:49
기본 속도: 2.59GHz
소켓: 1
코어: 6
논리 프로세서: 12
가상화: 사용 안 함
Hyper-V 지원: 예
L1 캐시: 480KB
L2 캐시: 3.0MB
L3 캐시: 12.0MB
간단히(D)
리소스 모니터 열기

Process

구성


Stack
(절충 공간)
Heap
Data
Text
매개변수, 지역 변수 등 임시적인 자료
동적으로 할당되는 메모리
전역 변수
Program의 코드

Process

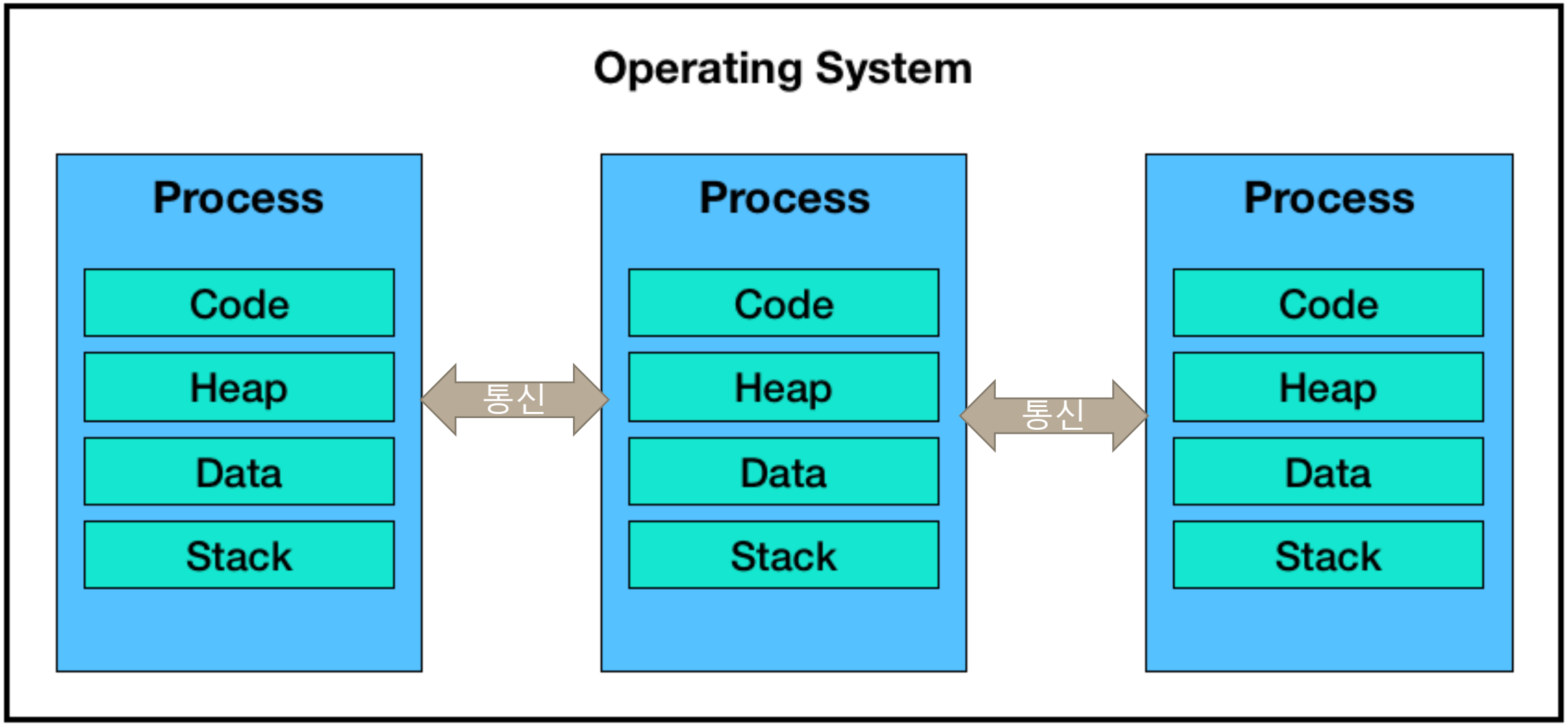


PCB (Process Control Block)

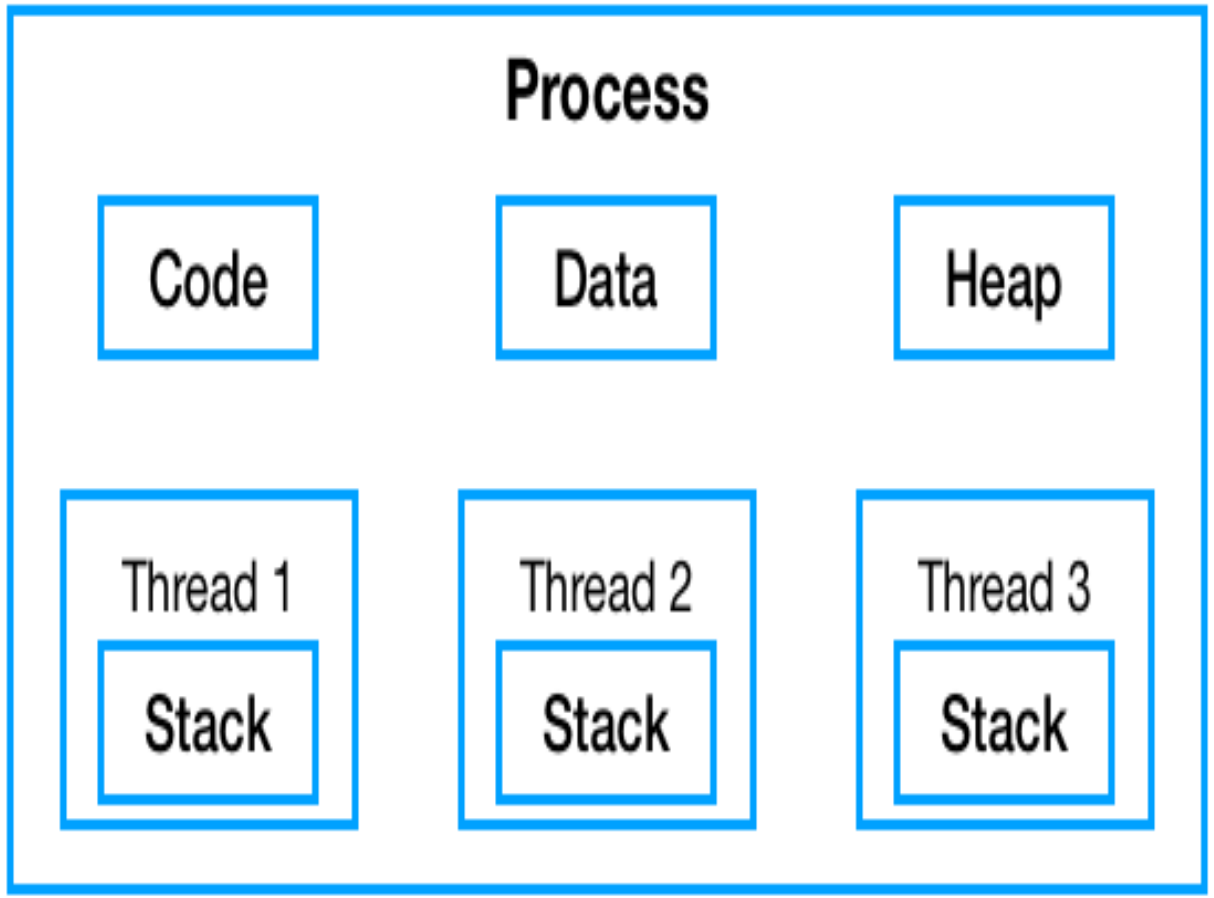
- 각 프로세서는 운영체제에서 PCB로 표현
- PID : 프로세서 식별자
- 프로세서 상태 : new, ready, running,

  waiting, halted등
- 프로세서 카운터: 다음 실행할 명령어의 주소
- 스케줄링 정보 : 우선순위 등

Process


Operating System
Process
Code
Heap
Data
Stack
통신
Process
Code
Heap
Data
Stack
통신
Process
Code
Heap
Data
Stack

Thread

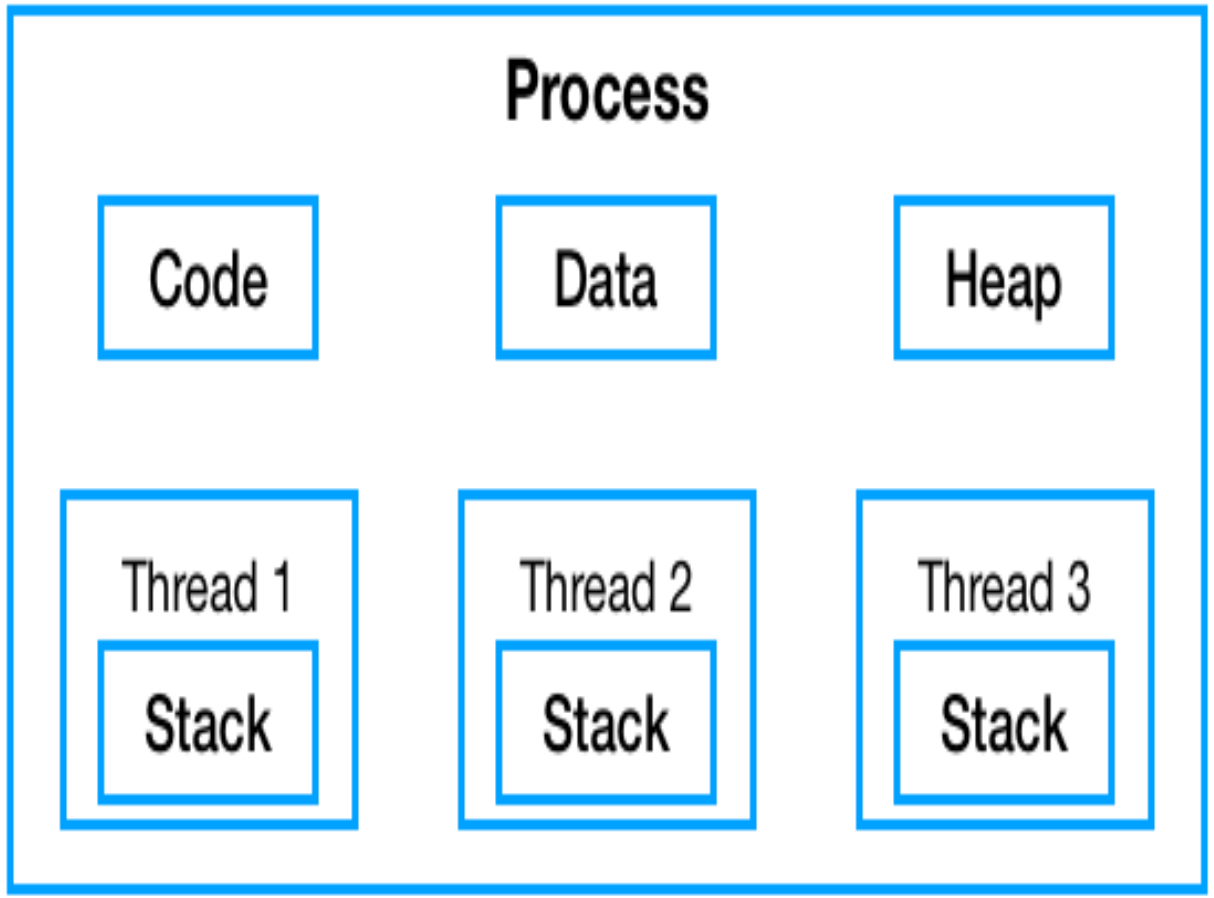


- 프로세스 내에서 실행되는 흐름의 단위
- CPU 이용의 기본 단위

- Code , Data , Heap 영역을 공유
- 각 thread는 별도의 Stack 영역을 가짐

Thread

*Multi Thread



- 프로세스의 자원을 공유
- 향상된 응답성
- Context switching 비용이 적음
- 자원을 공유하는 만큼 충돌을 주의

Thread

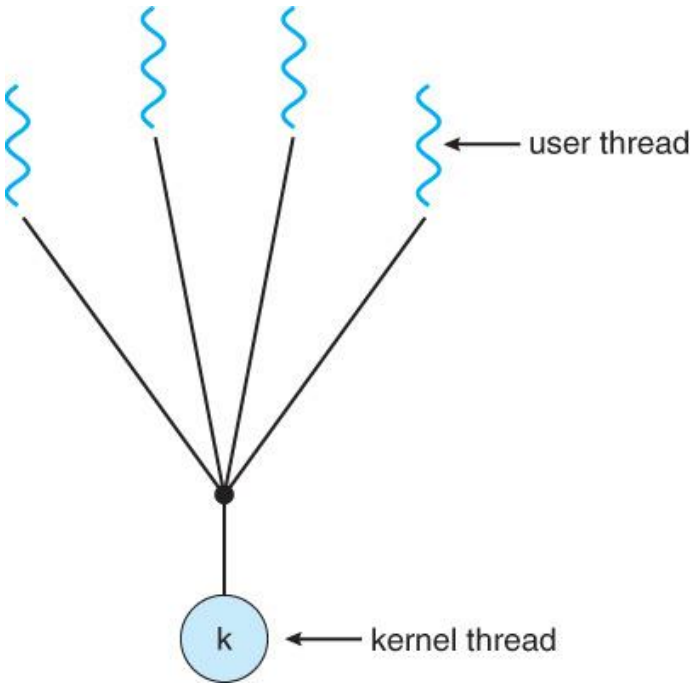
- 사용자 쓰레드(User thread)
  - 커널의 지원/인식 없이 사용자 공간에서 생성, 스케줄링, 관리할 수 있다.
  - 사용자 쓰레드가 멈추면 프로세스 자체가 멈춘다.
  - 쓰레드 라이브러리를 통하여 제공된다.(Pthread, Java 쓰레드 등)

- 커널 쓰레드(Kernel thread)
  - 커널이 쓰레드를 인식하고 프로세스와 유사하게 스케줄링한다.
  - 커널의 오버헤드로 쓰레드 생성과 스케줄링이 느리다.
  - Windows, Linux, Mac OS X 등 많은 운영체제에서 지원한다.
  - 각 커널 스레드는 CPU 스케줄링을 통해 병렬적으로 실행된다.
  - 멀티프로세서 작업 수행 능력이 높다.

Thread

*Multi Thread

- 다대 일 모델(Many - to - One Model) / 유저 레벨 쓰레드(User Level Thread)

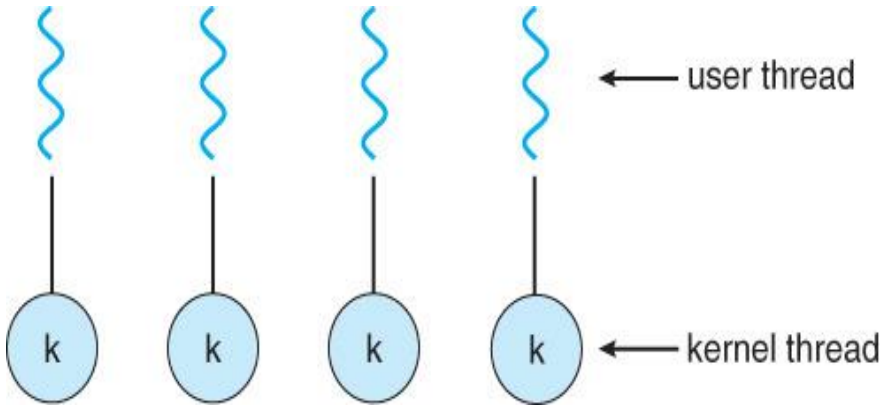


- 사용자 영역의 쓰레드 라이브러리로 구현(pthread, java thread 등)
  - 쓰레드 라이브러리는 쓰레드의 생성과, 종료, 통신, 스케줄링과 컨텍스트 등 정보를 보관한다.
- 장점
  - 커널의 지원/인식 없이 실행되기 때문에 쓰레드 연산이 빠르다..
  - API에 의존하지 않아 이식성이 높다.
  - 개발자가 쓰레딩 라이브러리의 스케줄링 알고리즘을 수정할 수 있다.
  - 시스템 호출과 같은 인터럽트가 발생할 때 오버헤드가 발생하는데 커널에 의존해야 하는 쓰레드보다 오버헤드가 적다.
- 단점
  - 커널이 멀티 쓰레드 프로세스를 한 개의 스레드로 간주하기 때문에 프로세스들을 여러 프로세서에 동시에 할당할 수 없다.
  - 다대일 쓰레드 맵핑이기에 하나의 스레드가 입출력 작업이 수행되어 블록 될 경우 프로세스 전체가 블록 된다.
  - 스케줄링 우선순위를 지원하지 않는다.

Thread

*Multi Thread

- 일대 일 모델(One - to - One Model) / 커널 레벨 쓰레드(Kernel Level Thread)

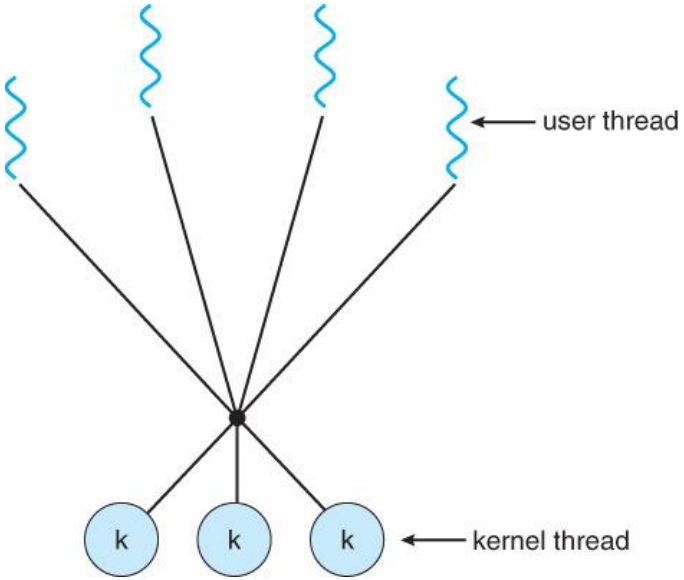


- 프로세스가 주기억 장치에 여러 개가 적재되어 CPU 할당을 기다리며 동작한다.
- 장점
  - 일대일 쓰레드 맵핑이기에 프로세스는 쓰레드들의 병렬 실행 작업 성능이 높다.
  - 커널이 각 쓰레드를 개별적으로 관리할 수 있다.
  - 입출력 작업이 끝날 때까지 다른 쓰레드를 사용해 다른 작업을 진행할 수 있다.
  - 사용자 입력에 빠르게 반응할 수 있어서 프로그램 사이의 상호작용성이 높다.
  - 우선순위 지정을 통한 서비스의 수준을 조절할 수 있다.
- 단점
  - 스케줄링과 동기화를 위해 커널을 호출해야 하는데 이는 오버헤드를 증가시킨다.
  - 사용자 쓰레드를 사용할 때보다 이식성이 떨어진다.
  - 프로그래머는 실행될 시스템이 달라지면 운영체제에서 제공하는 쓰레드 API를 사용해 프로그램을 수정해야한다.
  - 자원을 많이 소비하며, 쓰레드 생성 개수가 제한된다.

Thread

*Multi Thread

- 다대 다 모델(Many to Many Model) / 멀티 레벨 쓰레드(Multi Level Thread)



- 다대 일 모델과 일대 일 모델의 장점을 활용하여 동작하는 방식이다.
- 장점
  - 동시에 수행되는 쓰레드 개수가 많을수록 스케줄링하는 데 소비되는 시간이 많아지는데 쓰레드 풀링을 통해 오버헤드 문제를 해결한다.
    1. 미리 생성되어 있는 쓰레드를 재사용한다.
    2. 쓰레드의 개수를 제한한다.
- 단점
  - 한 개의 사용자 레벨 쓰레드가 블록되면 운영체제가 멀티스레드 프로세스 전체를 블록한다.
    - 스케줄러 활성화로 문제를 해결 가능
    1. 사용자 쓰레드가 블록 되면 쓰레드의 상태를 스케줄러 활성화에 저장
    2. 새로운 스케줄러 활성화를 위해 사용자 레벨 라이브러리에 자신의 스레드 중 하나가 블록되었음을 알림
    3. 사용자 레벨 라이브러리는 스케줄러 활성화로부터 블록된 쓰레드의 상태를 저장하고 다른 쓰레드를 할당
  - 멀티프로세서에서도 프로세서의 쓰레드들을 동시에 실행할 수 없다.
  - 설계가 복잡하고, 구현에 대한 표준적인 방법이 없다.

Thread 함수

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

이 함수는 thread를 생성하는 함수이다.
첫번째 인자는 생성할 thread의 id
두번째 인자는 thread의 특성인데, 보통 NULL을 집어넣는다.
세번째 인자는 thread를 실행할 함수가 온다. thread를 실행할 함수는 포인터 함수이므로 인자값으로 올 수 있다.
네번째 인자는 thread를 실행할 함수에 넣어줄 인자값이 온다.

리턴 값 : 성공적으로 pthread가 생성될 경우 0 반환

int pthread_join(pthread_t thread, void **rval_ptr);

이 함수는 main이나 부모 thread에서 자식 thread가 종료할 때까지 대기하는 함수이다.
main이나 부모thread가 끝나버리면 자식 thread 또한 종료되기 때문이다.
첫번째 인자는 생성한 thread의 id
두번째 인자는 해당 thread가 종료되면 return받을 변수. 리턴받을 값이 없으면 NULL을 넣으면 된다.

Thread 함수

void pthread_exit(void *rval_ptr);

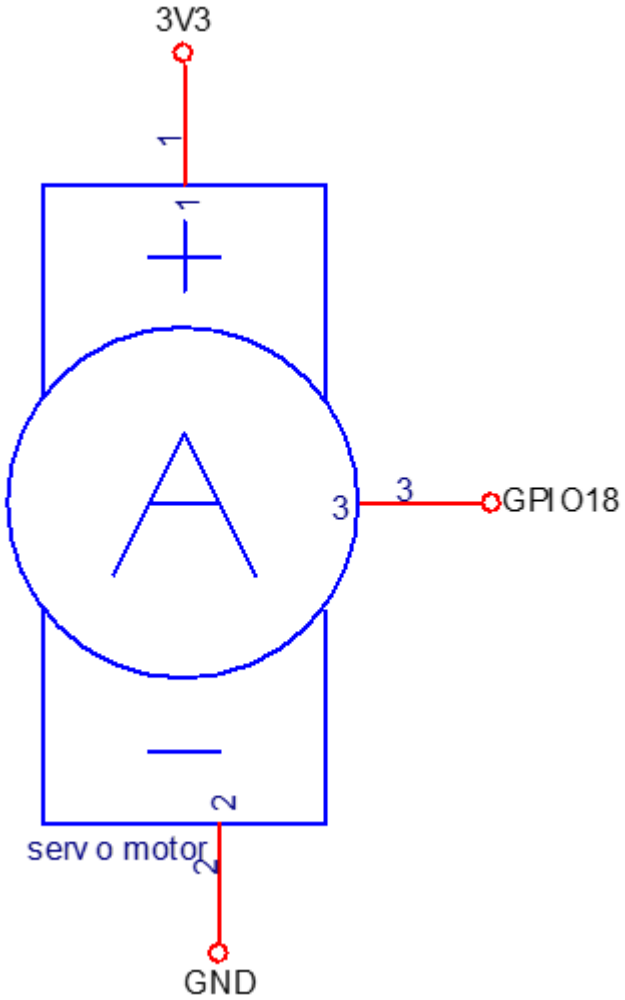
이 함수는 thread를 종료할 때 사용한다.
첫번째 인자는 thread를 종료하고 pthread_join() 함수 두번째 인자에 리턴할 변수.
리턴할 값이 없으면 NULL을 넣으면 된다.

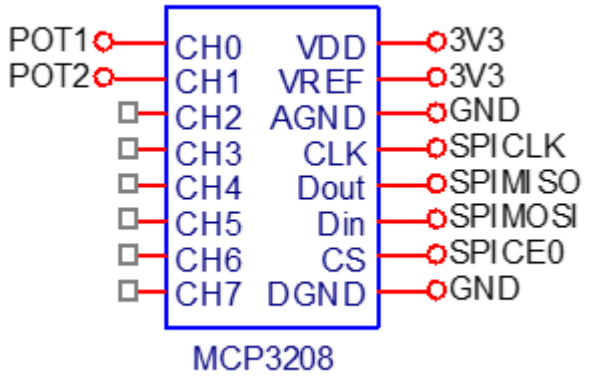
pthread_t pthread_self(void)

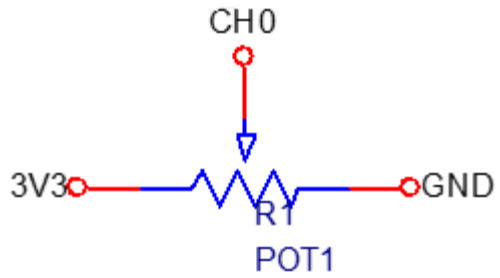
이 함수는 실행하고 있는 thread의 id를 리턴한다.

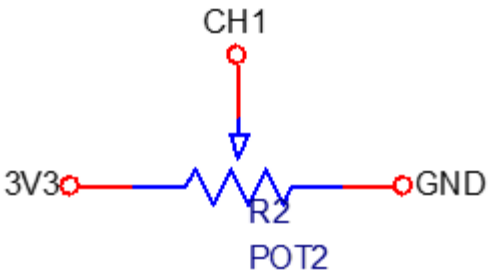
또한 gcc에서 PThread를 사용할 소스를 컴파일 할때 맨 뒤에 -lpthread 명령어를 붙여줘야 한다.

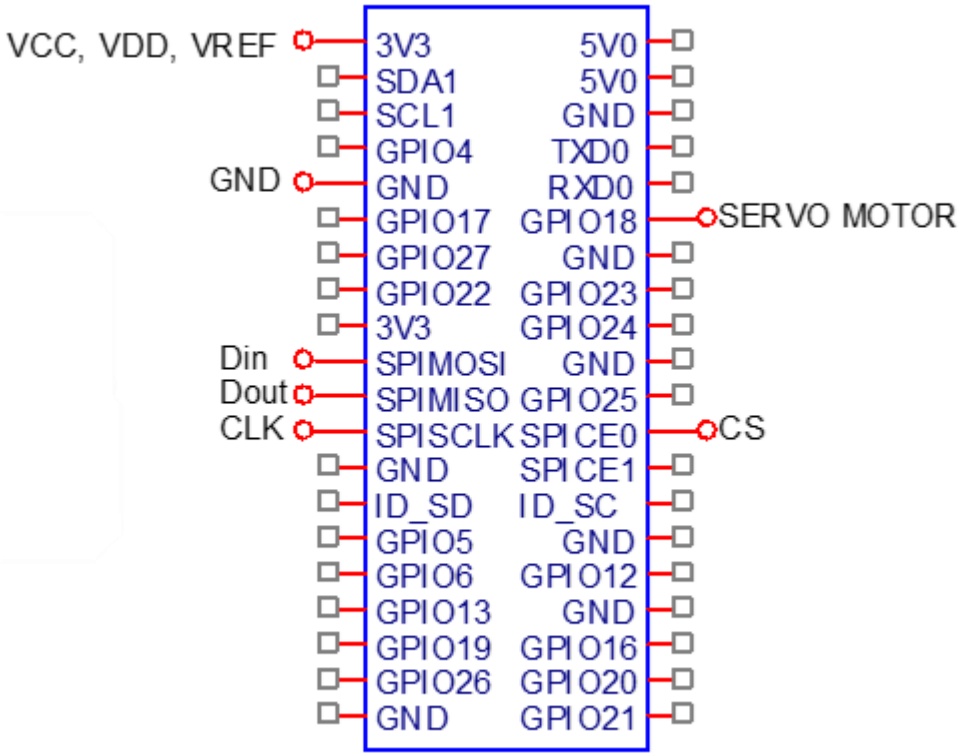
회로도


3V3
1
1
3
3
GPIO18
servo motor
2
2
GND


POT1
POT2
CH0
CH1
CH2
CH3
CH4
CH5
CH6
CH7
VDD
VREF
AGND
CLK
Dout
Din
CS
DGND
3V3
3V3
GND
SPICLK
SPIMISO
SPIMOSI
SPICE0
GND
MCP3208


CH0
3V3
R1
GND
POT1


CH1
3V3
R2
GND
POT2


VCC, VDD, VREF
3V3
SDA1
SCL1
GPIO4
GND
GND
GPIO17
GPIO27
GPIO22
3V3
Din
SPIMOSI
Dout
SPIMISO
CLK
SPISCLK
GND
ID_SD
GPIO5
GPIO6
GPIO13
GPIO19
GPIO26
GND
5V0
5V0
GND
TXD0
RXD0
GPIO18
GND
GPIO23
GPIO24
GND
GPIO25
SPICE0
SPICE1
ID_SC
GND
GPIO12
GND
GPIO16
GPIO20
GPIO21
SERVO MOTOR
CS