

삼성 청년 SW 아카데미

시스템 프로그래밍

<알림>

본 강의는 삼성 청년 SW아카데미의 콘텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사,
촬영, 녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

Day2-1. 시스템 아키텍처 기본

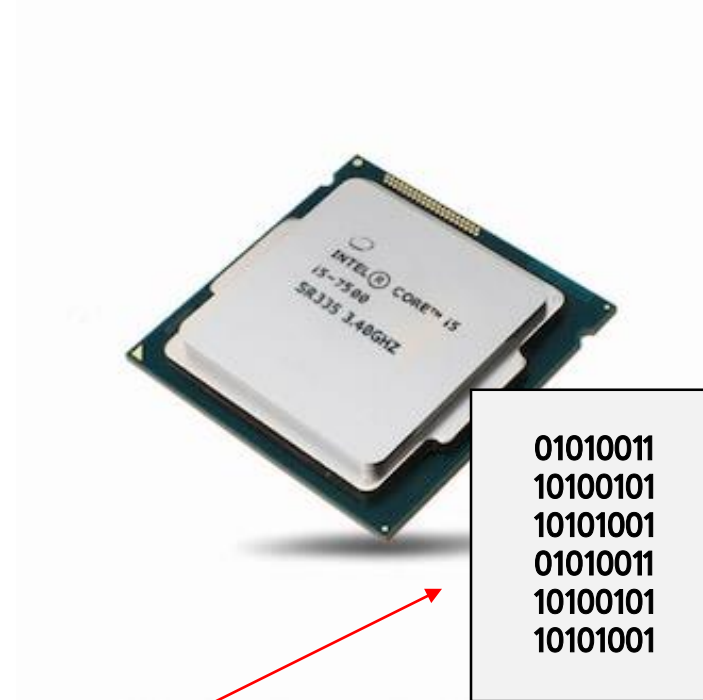
챕터의 포인트

- 폰노이만 아키텍처
- D램과 S램
- 프로그램과 프로세스
- 메모리 구조
- 멀티 프로세스
- 멀티 쓰레드

폰노이만 아키텍처

CPU의 역할

- 0과 1로 구성된 명령어를 하나씩 수행하는 장치



CPU 명령어가
6개 존재한다.

CPU가 동작할 명령어들을
Disk에 저장해둔다.



이곳엔
명령어들을 저장 할
넉넉한 공간이 없다.

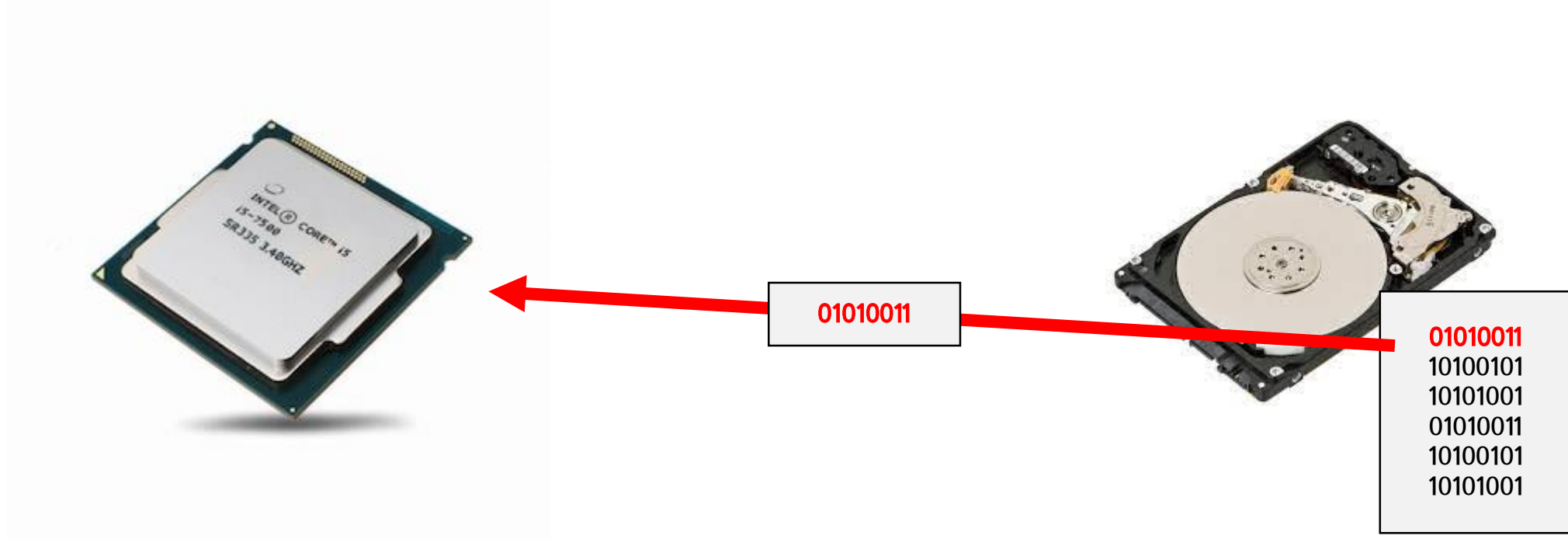


01010011
10100101
10101001
01010011
10100101
10101001

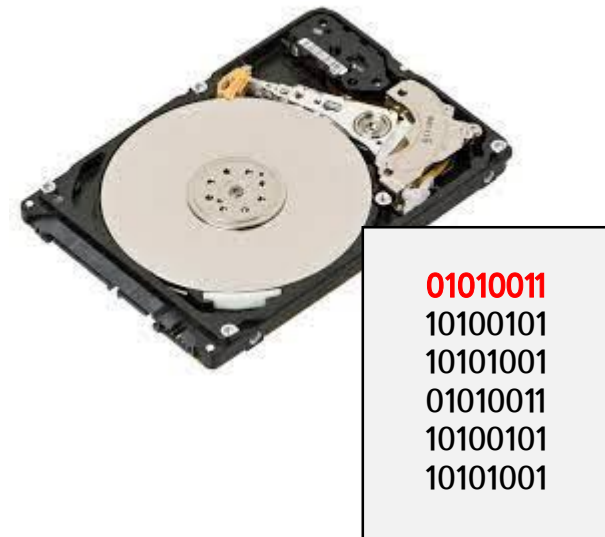
HDD

한줄씩 가져가서 수행한다.

- 명령어가 하나씩 CPU에 전달된다.



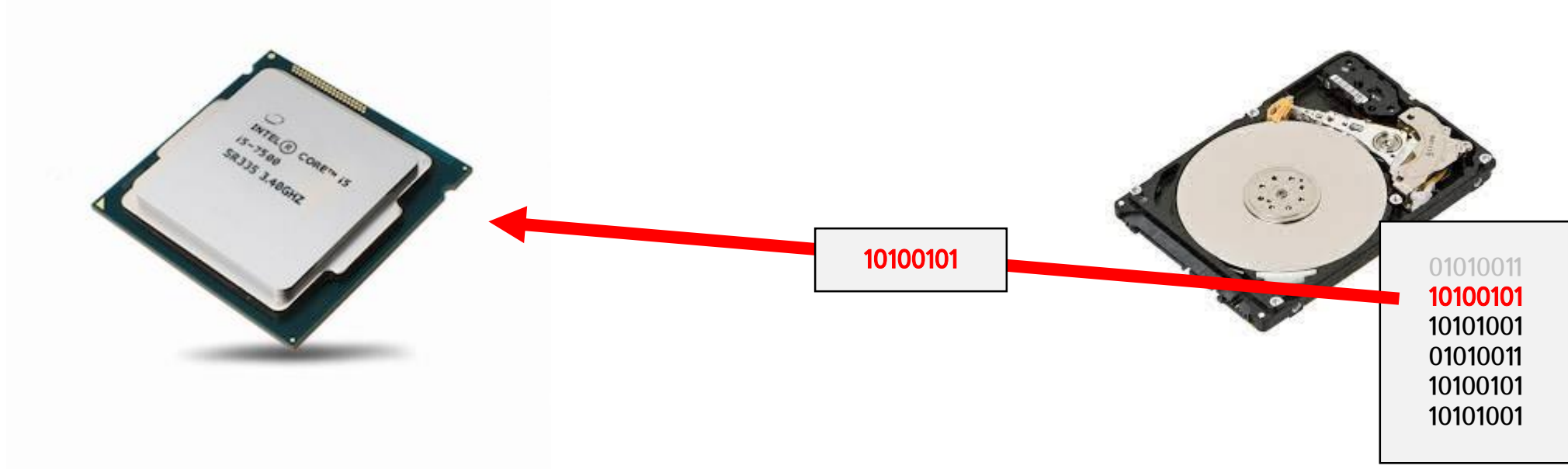
CPU는 받자마자
명령어를 수행한다.



한줄씩 가져가서 수행한다.

- 명령어가 하나씩 CPU에 전달된다.

이런 원리로 차례대로 모두 수행 된다.



이런 방식의 문제점 : CPU의 효율이 너무 떨어진다.

- Disk 가 너~~~~무 느리다.
- CPU는 너~~~~무 빠르다.



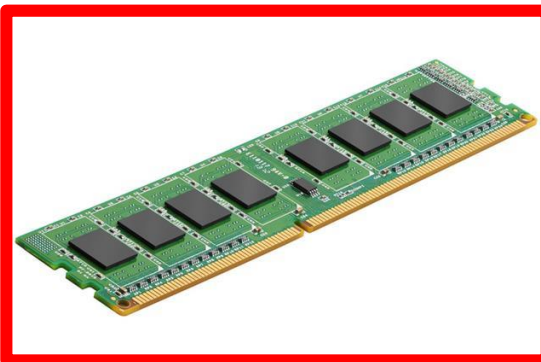
CPU 명령어가
도착될때 까지
맨날 놀고있다.



모터의 성능으로
Disk가 동작된다.
매우 바쁘다.

이 문제의 해결 방안

- 메모리가 이런 문제를 해결한다.
- 폰노이만의 아이디어

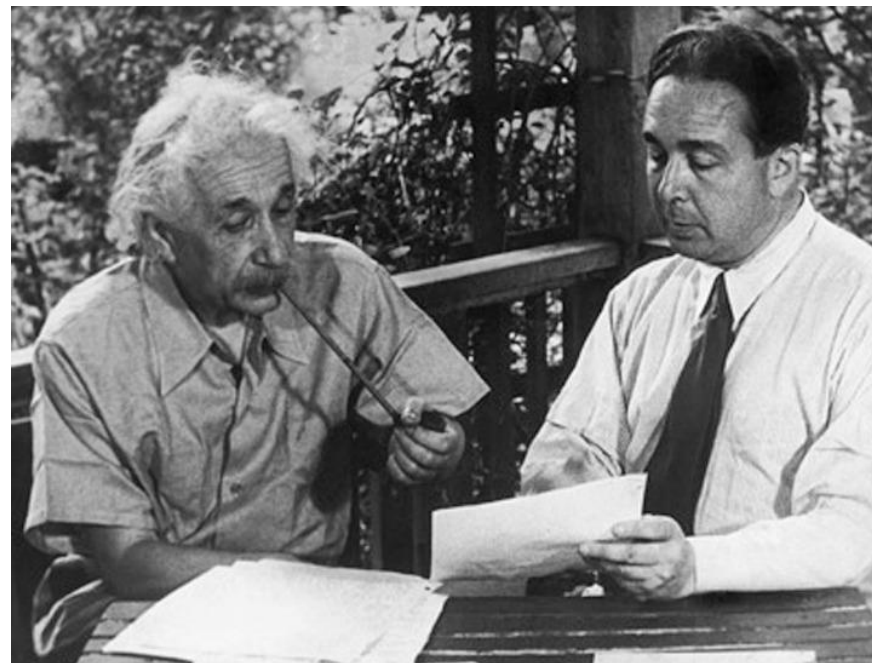


01010011
10100101
10101001
01010011
10100101
10101001

폰노이만

1941년 **프로그램 내장방식**을 통해
메모리가 필요하다고 주장

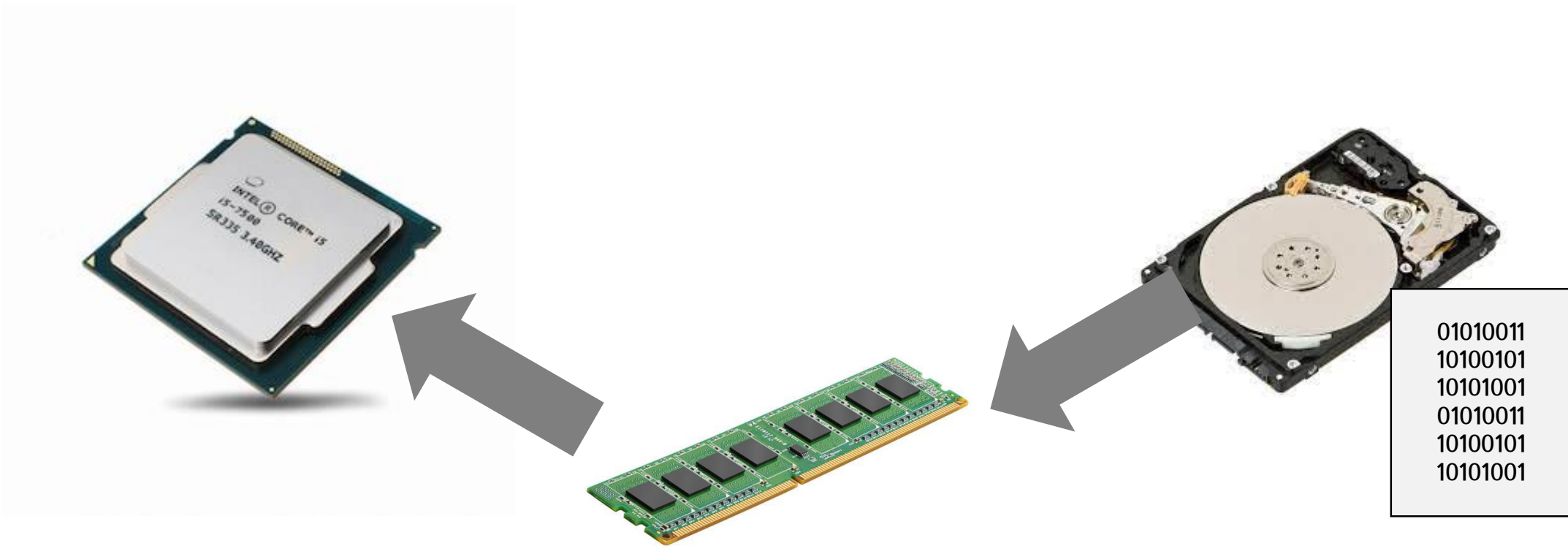
- 악마의 두뇌
- 7개국어 가능
- 미적분은 암산



인류 역사상 천재 중 한 명 : 폰노이만

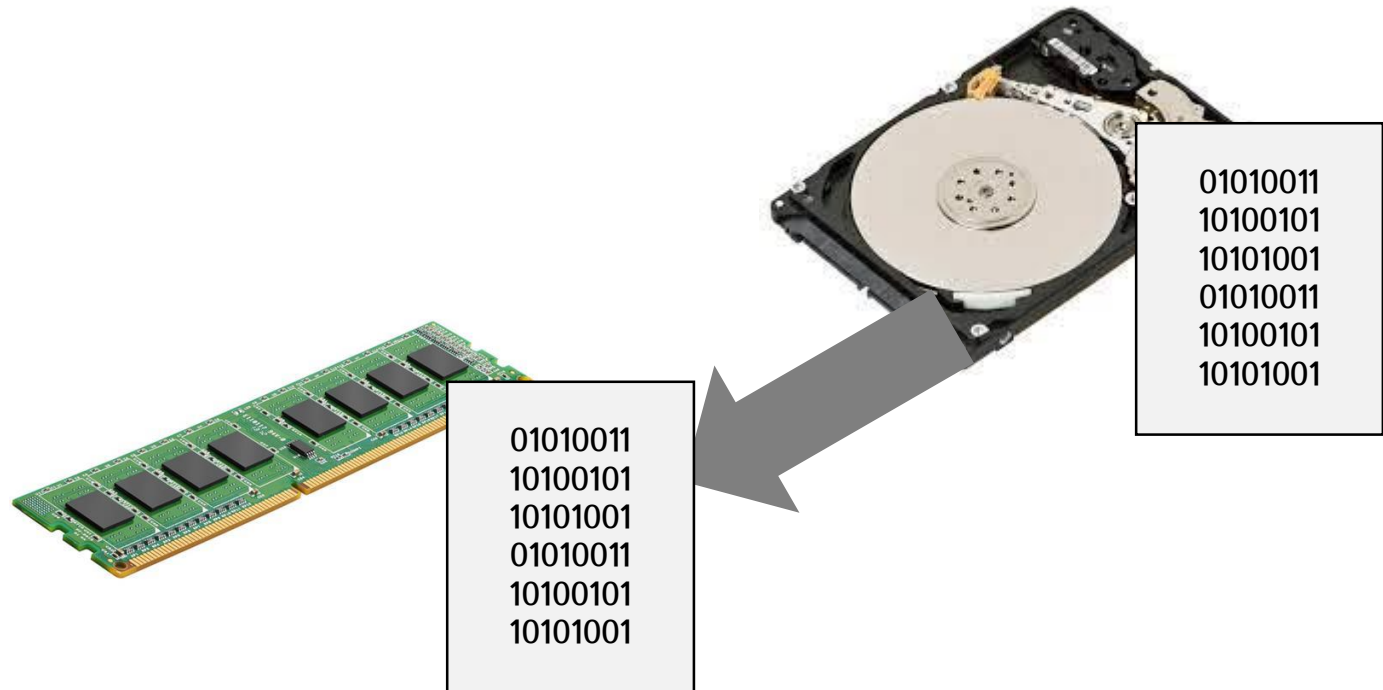
메모리

- 빠른 성능, 저장장치 처럼 저장하는 기능을 갖지만, 전원을 끄면 모두 소멸되는 장치
- 명령어들을 대신 전달해주는 역할을 한다.



Load 작업

- 저장장치에 저장된 0과 1 장치가 메모리에 한꺼번에 복사된다.
- 이때는 시간이 제법 걸린다. (프로그램 전체를 모두 불러와야 하기 때문)
- 이 동작을 적재(Load) 라고 하며, 흔히 Loading 이라고 부르는 말이, 이러한 뜻이다.



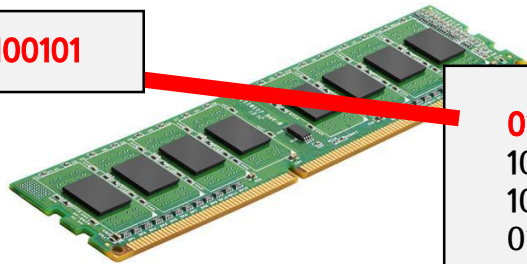
이제 Disk 대신, 메모리가 CPU에게 전달한다.

- 엄청 빠르다! (이전에 비하면)
- CPU와 메모리 사이에 데이터를 주고 받는다.

메모리가 빠르다고는 하나 여전히 CPU 보다는 느리다.
하지만 HDD에 비하면 엄청 빠르다.



10100101



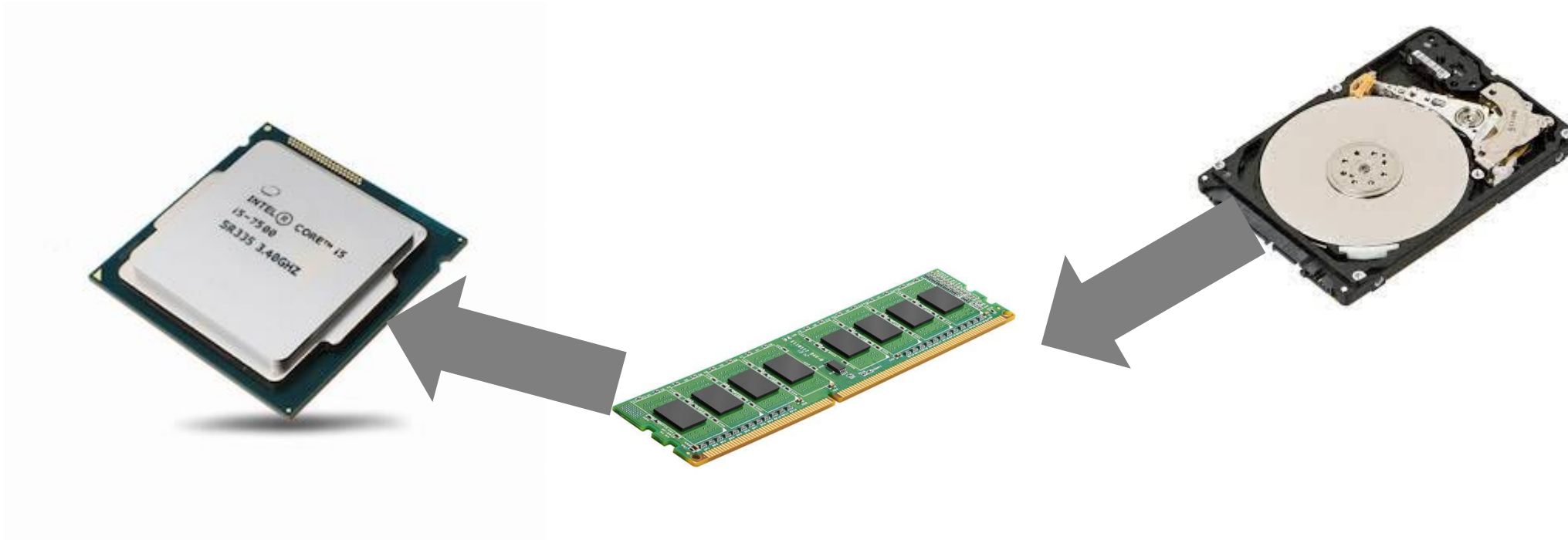
01010011
10100101
10101001
01010011
10100101
10101001



01010011
10100101
10101001
01010011
10100101
10101001

이 원리를 **폰노이만 아키텍처** 라고 한다.

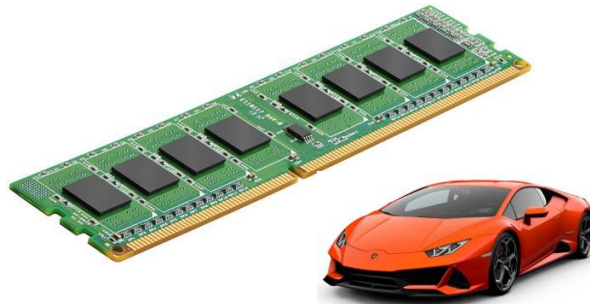
- 컴퓨터는 1941 년 부터 70년 이상 이 구조를 채택하고 있다.
- 이 구조에서 약간 변형한 “하버드아키텍처” 가 있지만, 기본 골격은 폰노이만 아키텍처이다.
- 현대식 컴퓨터 구조, 임베디드도 마찬가지



D램과 S램

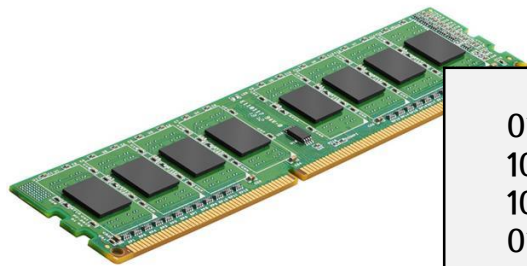
성능에 차이가 있다.

- 저장장치 : 걸어다니는 속도
- 메모리 : 람보르기니 (시속 300 km/h)
- CPU : 전투기 (시속 2,000 km/h)

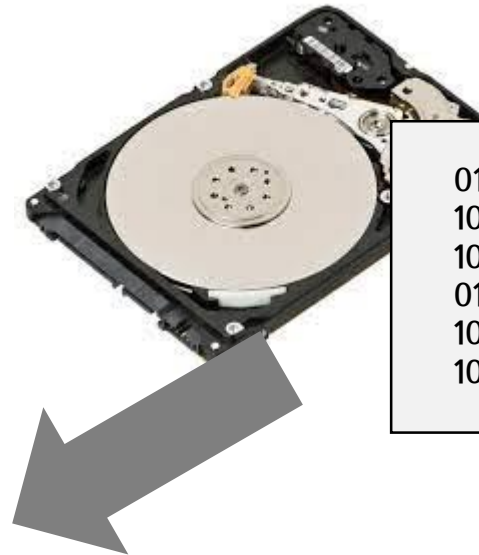


Load 작업은 느리다.

- 저장장치는 여전히 느리다.



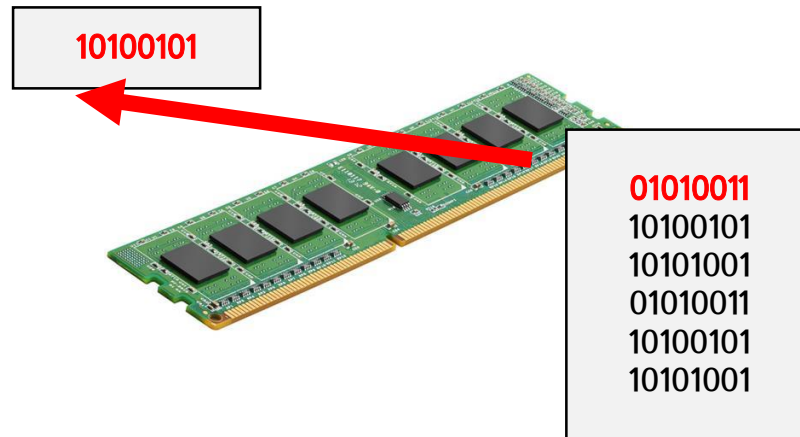
01010011
10100101
10101001
01010011
10100101
10101001



01010011
10100101
10101001
01010011
10100101
10101001

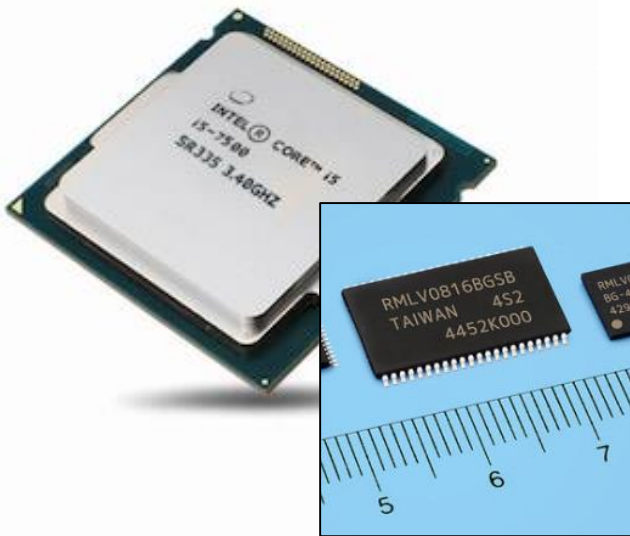
메모리가 명령어를 전달하는 동안
속도가 더 빠른 CPU는 기다리는 시간이 발생한다.

- 대기 시간동안 아무것도 하지 않는다.

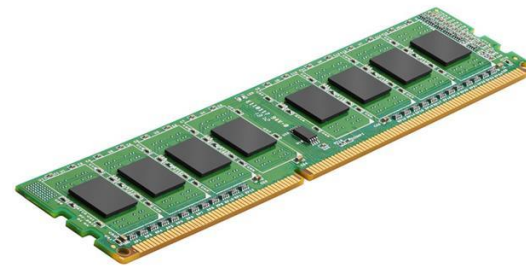


CPU 안에, 더 빠른 메모리를 내장하면 된다. → S램

- 저장공간은 작지만, 기존에 메모리보다 더 빠른 성능을 내는 메모리



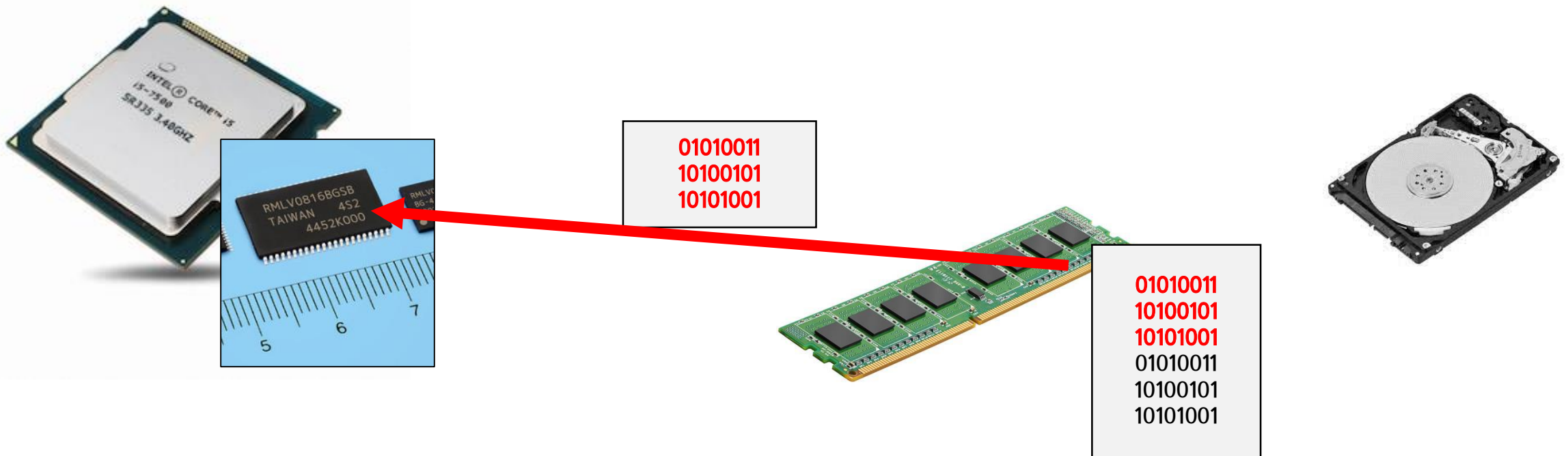
메모리 : S램



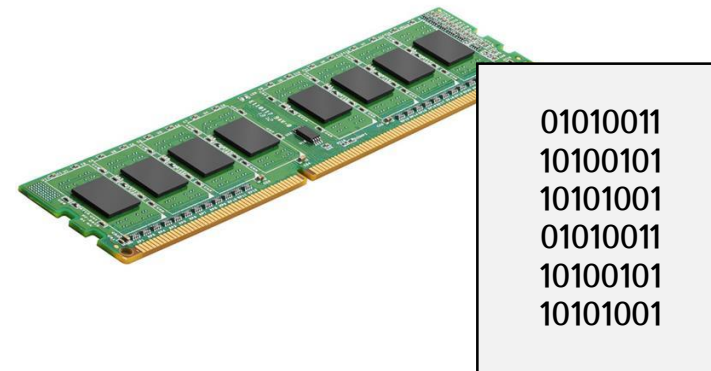
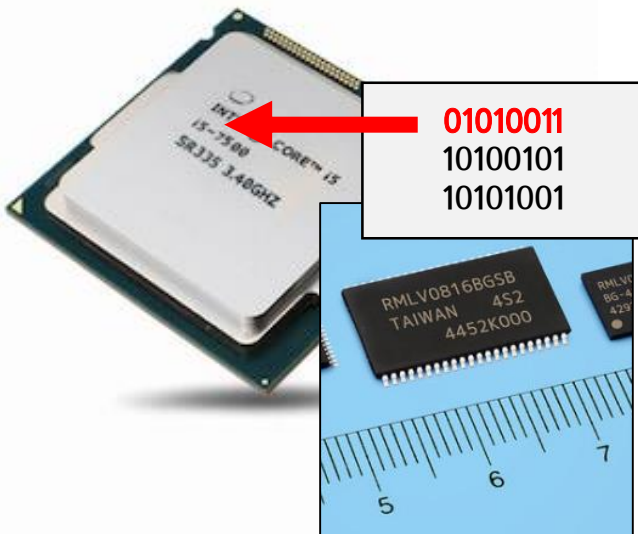
메모리 : D램



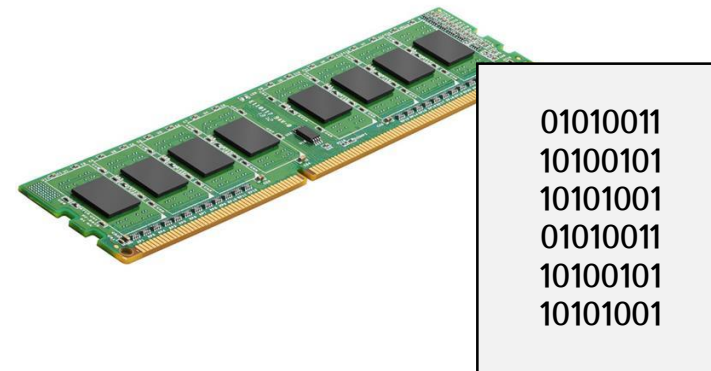
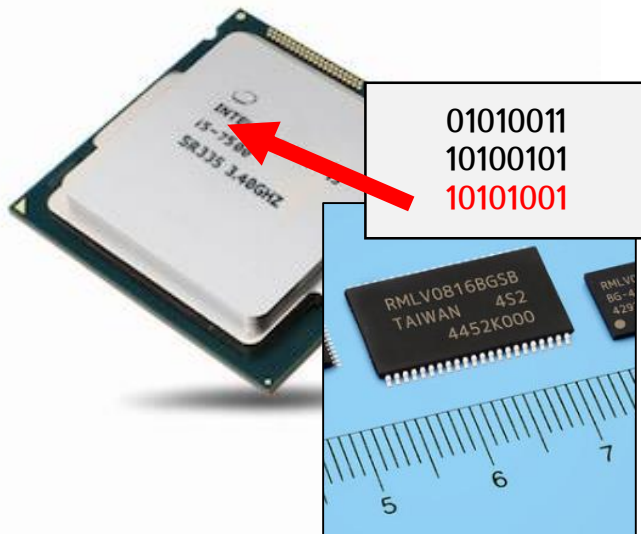
D램에서 S 램으로
여러줄의 명령어를 한꺼번에 전달한다.



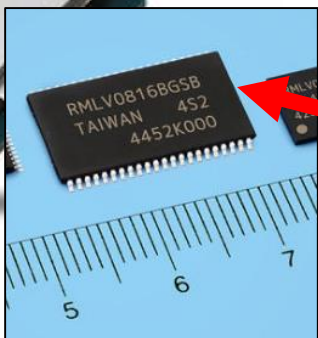
전달 받으면,
기존보다 더 빠르게 한줄씩, 명령어를 수행 한다.



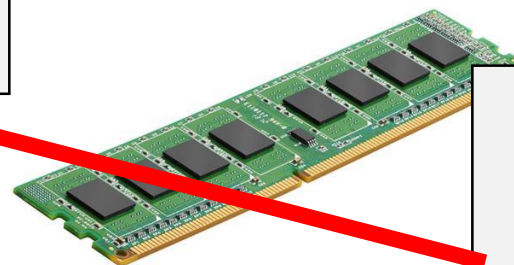
한줄씩 모든 명령어 수행을 빠르게 한다.



모든 명령어를 다 수행했을 경우,
이제 다음 명령어 덩어리들을, D램에게 전달 받는다.



01010011
10100101
10101001

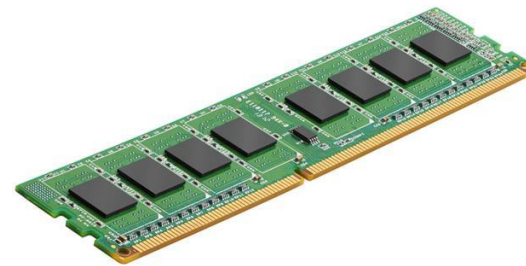
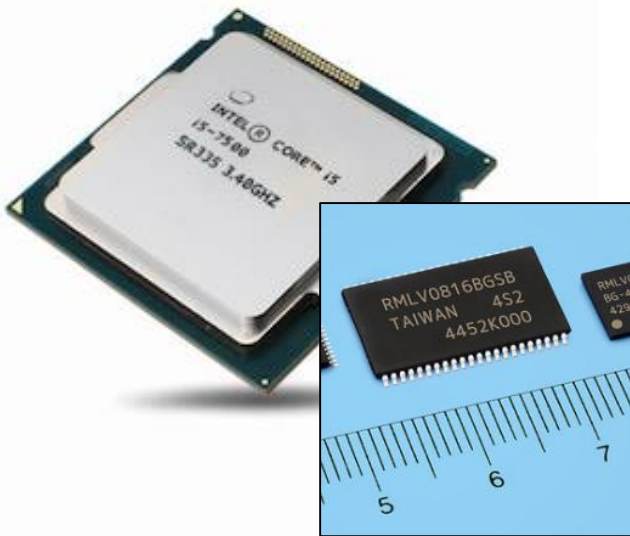


01010011
10100101
10101001
01010011
10100101
10101001

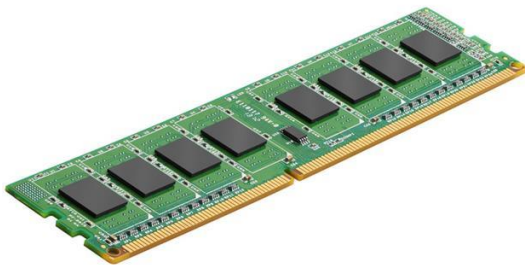
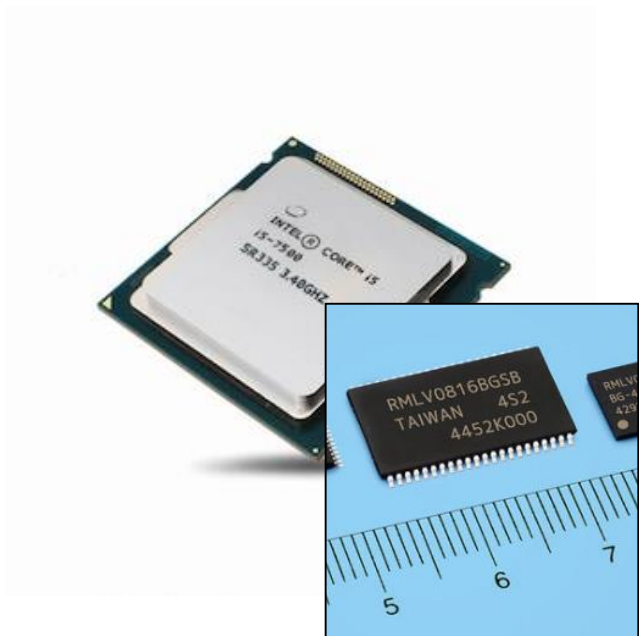


이 S램을 Cache 메모리라고 한다.

- Cache 메모리에 Hit / Miss 전략을 통해, 효율성을 더 높이는 방법도 존재한다.
- S램 만을 이용하면 더 빠르지 않을까? → S램의 가격이 너무 비싸다..



S램, D램 : HW 구조적으로 Static / Dynamic 용어로 만들어진 이름
S램은 **Cache 메모리** 라고도 하며,
D램은 **Main 메모리** 라고 불리운다.



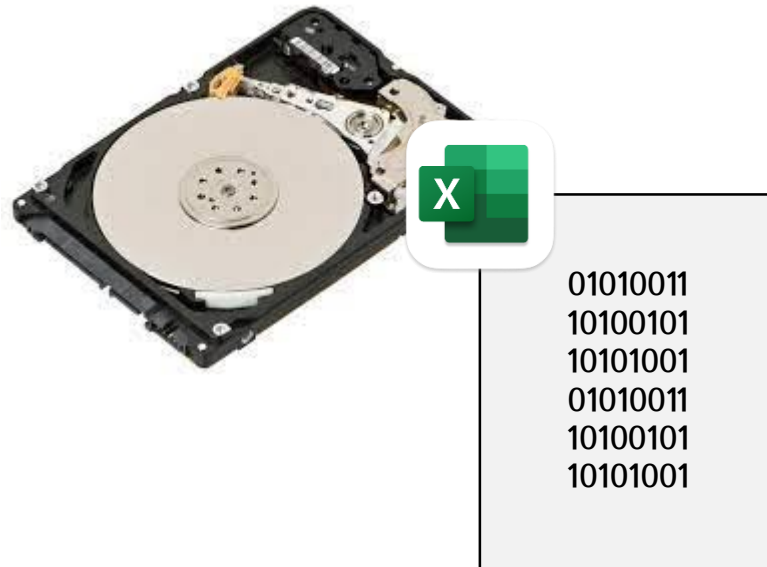
프로그램과 프로세스

프로그램의 정의는 무엇일까?



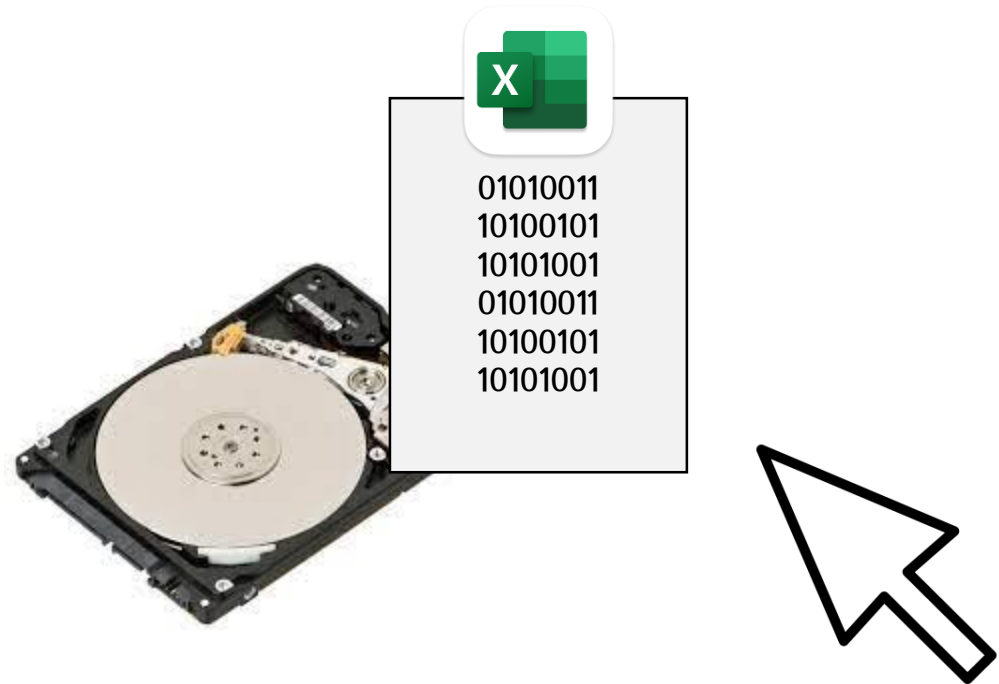
프로그램이란

- CPU의 2진수 명령어들의 집합

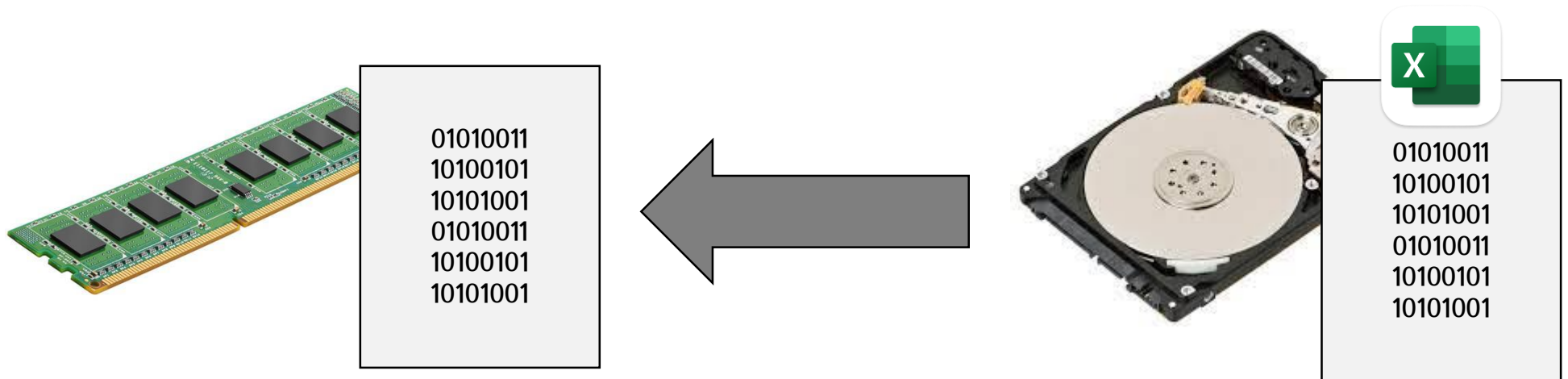


프로그램을 실행하면...

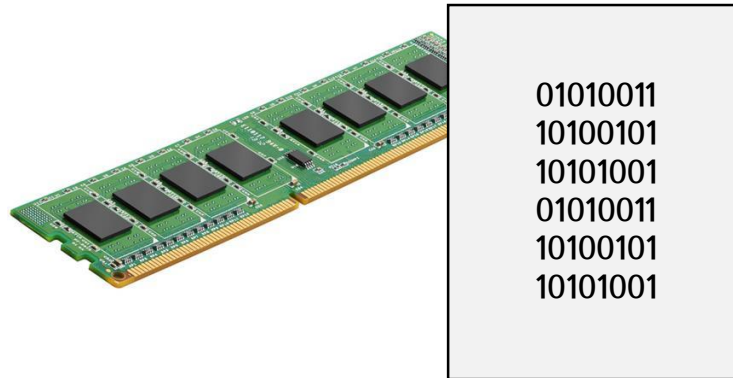
- 어떤 일이 첫 번째로 발생할까?
- 힌트 : L 로 시작함



로딩>Loading)된다. 라고 표현한다.

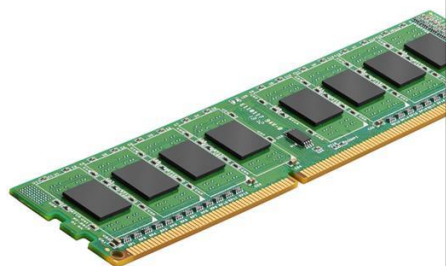


실행된 프로그램을 뜻하는 용어는 무엇인가?



프로세스

- 실행된 프로그램을 뜻하는 용어



01010011
10100101
10101001
01010011
10100101
10101001

작업 관리자						
파일(F) 옵션(O) 보기(V)						
프로세스 성능 앱 기록 시작프로그램 사용자 세부 정보 서비스						
이름	상태	62% CPU	60% 메모리	2% 디스크	0% 네트워크	5% GPU
> Google Chrome(10)		2.5%	874.4MB	0MB/s	0Mbps	0%
> Mattermost(7)		0%	514.0MB	0MB/s	0Mbps	0%
> Microsoft PowerPoint		0%	245.9MB	0MB/s	0Mbps	0%
> Antimalware Service Executable		5.9%	229.4MB	0.1MB/s	0Mbps	0%
Google Chrome		0%	202.3MB	0MB/s	0Mbps	0%
> Microsoft Visual Studio 2019(32비트)(6)		1.2%	193.1MB	0MB/s	0Mbps	0%
Google Chrome		0%	159.6MB	0MB/s	0Mbps	0%
데스크톱 창 관리자		2.8%	131.8MB	0MB/s	0Mbps	7.9%
> Windows 탐색기		2.1%	94.2MB	0.1MB/s	0Mbps	0%
Google Chrome		0%	51.6MB	0MB/s	0Mbps	0%
Google Chrome		0%	50.9MB	0MB/s	0Mbps	0%
> KakaoTalk(32비트)		0%	42.3MB	0MB/s	0Mbps	0%
Google Chrome		0%	41.9MB	0MB/s	0Mbps	0%
> 화면 캡처		2.9%	40.2MB	0MB/s	0Mbps	0%
Google Drive		1.3%	37.4MB	0MB/s	0Mbps	0%
Webex		0.3%	36.7MB	0MB/s	0Mbps	0%
> 서비스 호스트: Diagnostic Policy Service		0%	34.7MB	0MB/s	0Mbps	0%
> 작업 관리자		10.5%	31.8MB	0.3MB/s	0Mbps	0%

프로세스 리스트

프로그램 : CPU의 2진수 명령어의 집합

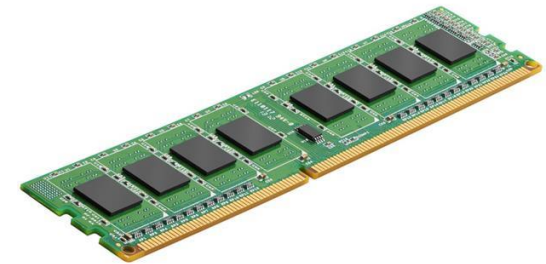
프로세서 : 프로그램을 실행하는 칩셋 (HW)

프로세스 : 실행 된 프로그램

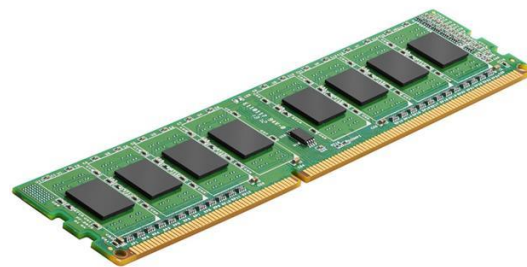
메모리 구조

메모리의 역할

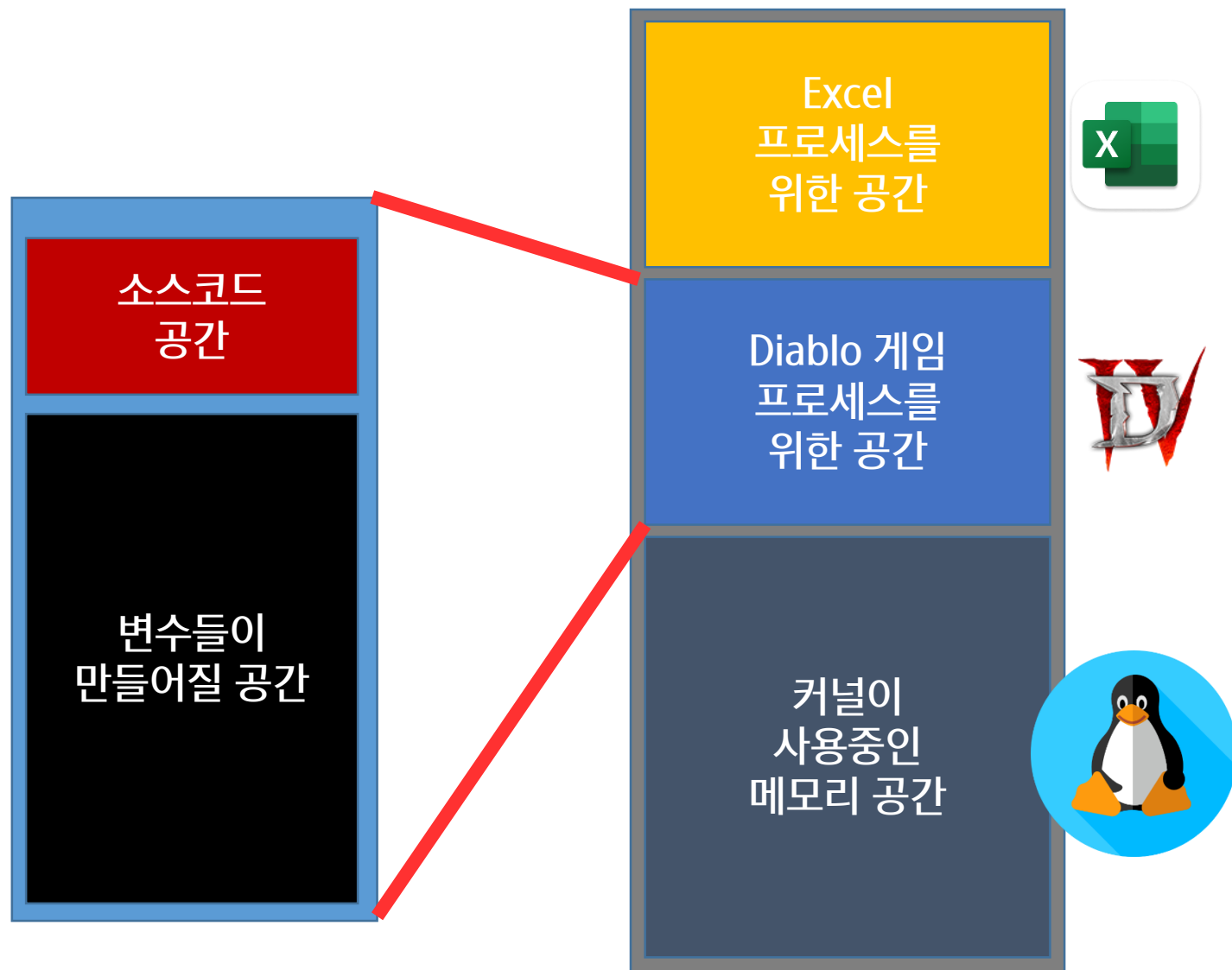
1. 0과 1로 된 CPU 명령어를 저장하는 역할
2. 변수가 만들어지는 공간



프로세스는
메모리 공간을 오른쪽 이미지처럼
나누어서 활용한다.



각 프로세스는
세부적인 공간을 가진다.



하나의 Process 는 text / data / bss / stack / heap 으로 세그먼트로 구성

- **.text** (닷 텍스트 or text segment 라고 읽는다.)

- 코드 영역

- **.data** : 초기화된 전역변수

- 하드코딩 데이터

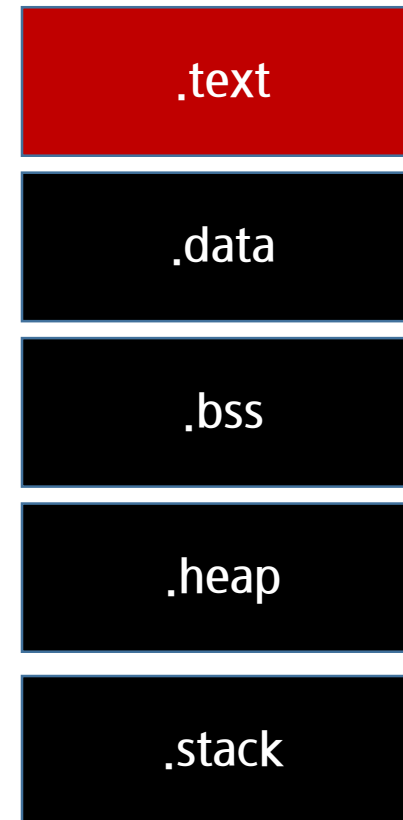
- **.bss** : 초기화 안된 전역변수

- 초기값이 **없**bss

- **.heap / .stack**

- .stack : 지역변수

- .heap : malloc으로 만든 변수들



코드를 보고, 각 변수가 어떤 세그먼트에 속한 지 맞춰보자.

- t, g, q, p

```
5    int t;  
6    int g = 32;  
7  
8    int main(){  
9        int q = 31;  
10       int *p = (int*)malloc(4);
```

$t \rightarrow \text{.bss}$

$g \rightarrow \text{.data}$

$q \rightarrow \text{.stack}$

$p \rightarrow \text{.stack}$

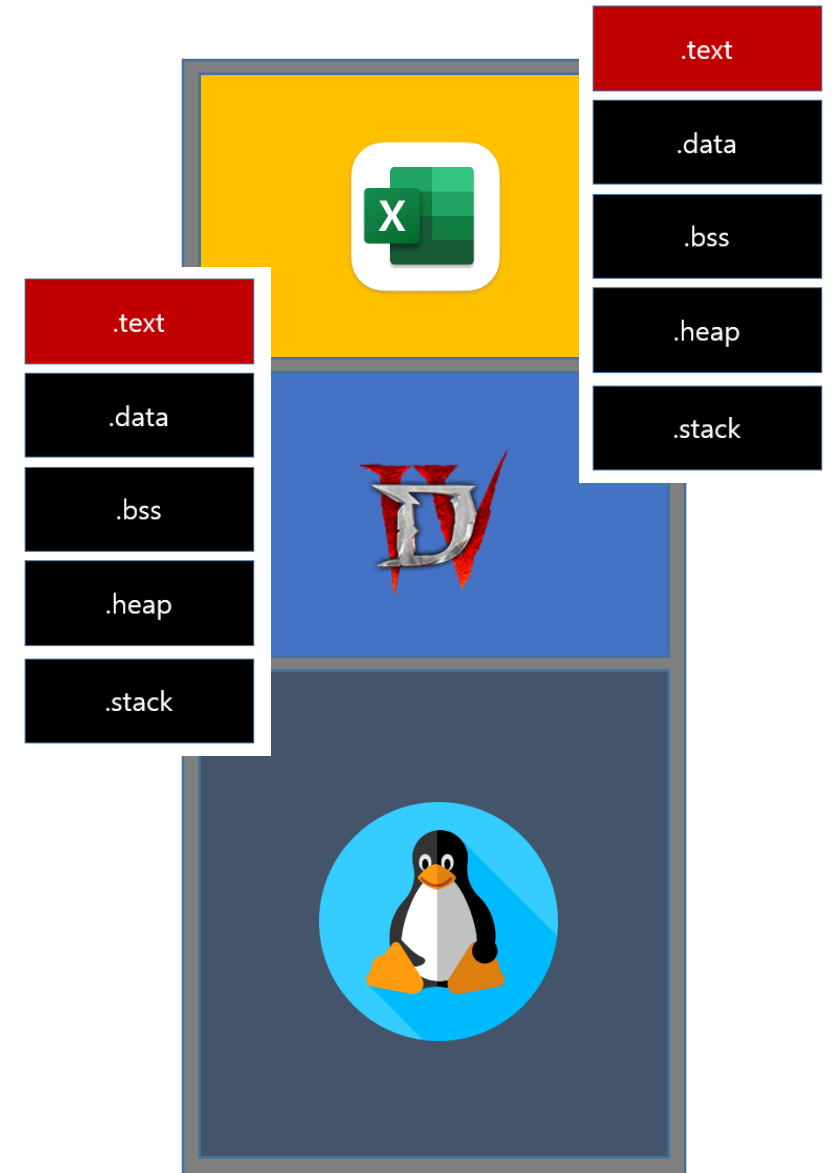
$\text{malloc} \rightarrow \text{.heap}$

```
5    int t;  
6    int g = 32;  
7  
8    int main(){  
9        int q = 31;  
10       int *p = (int*)malloc(4);
```

각각의 프로세스는
독립적인 메모리 공간을 가지고 있다.

프로세스는 다섯 개의 segment 로 구성된다.

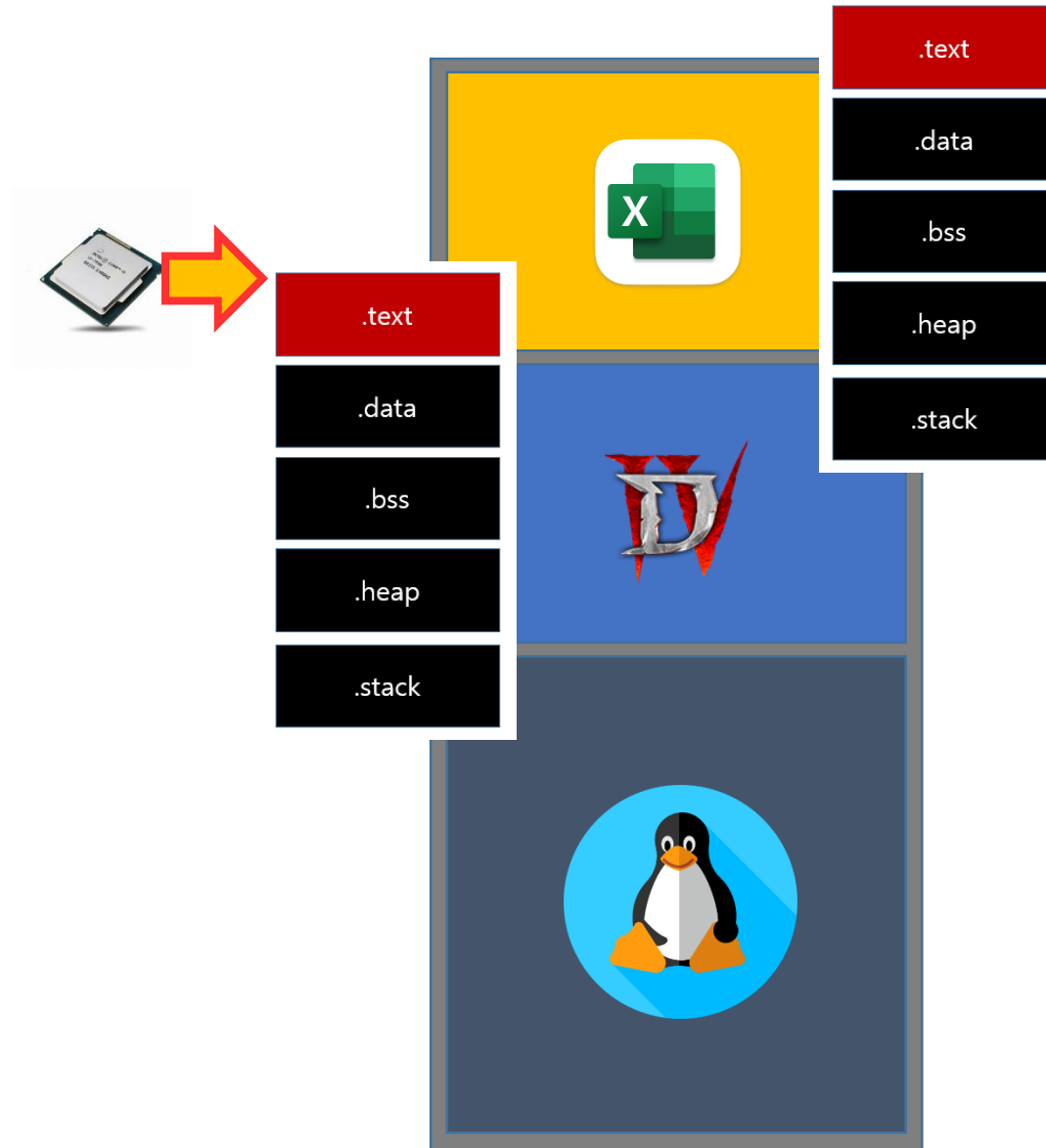
- .text , .bss , .data, .stack , .heap



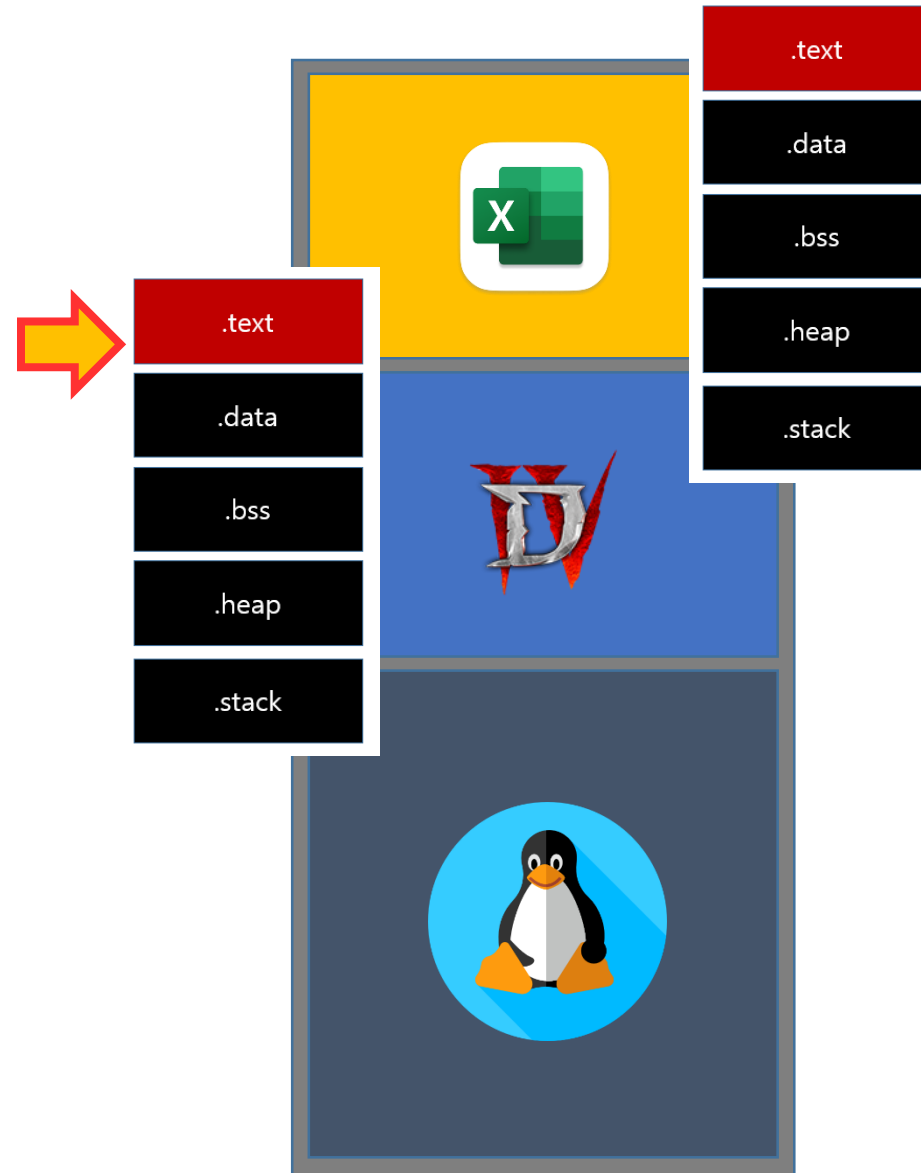
멀티 프로세스

CPU는 한번에
한가지 동작만 수행한다.

- 현재 Diablo Process의 **.text**를 수행 중.

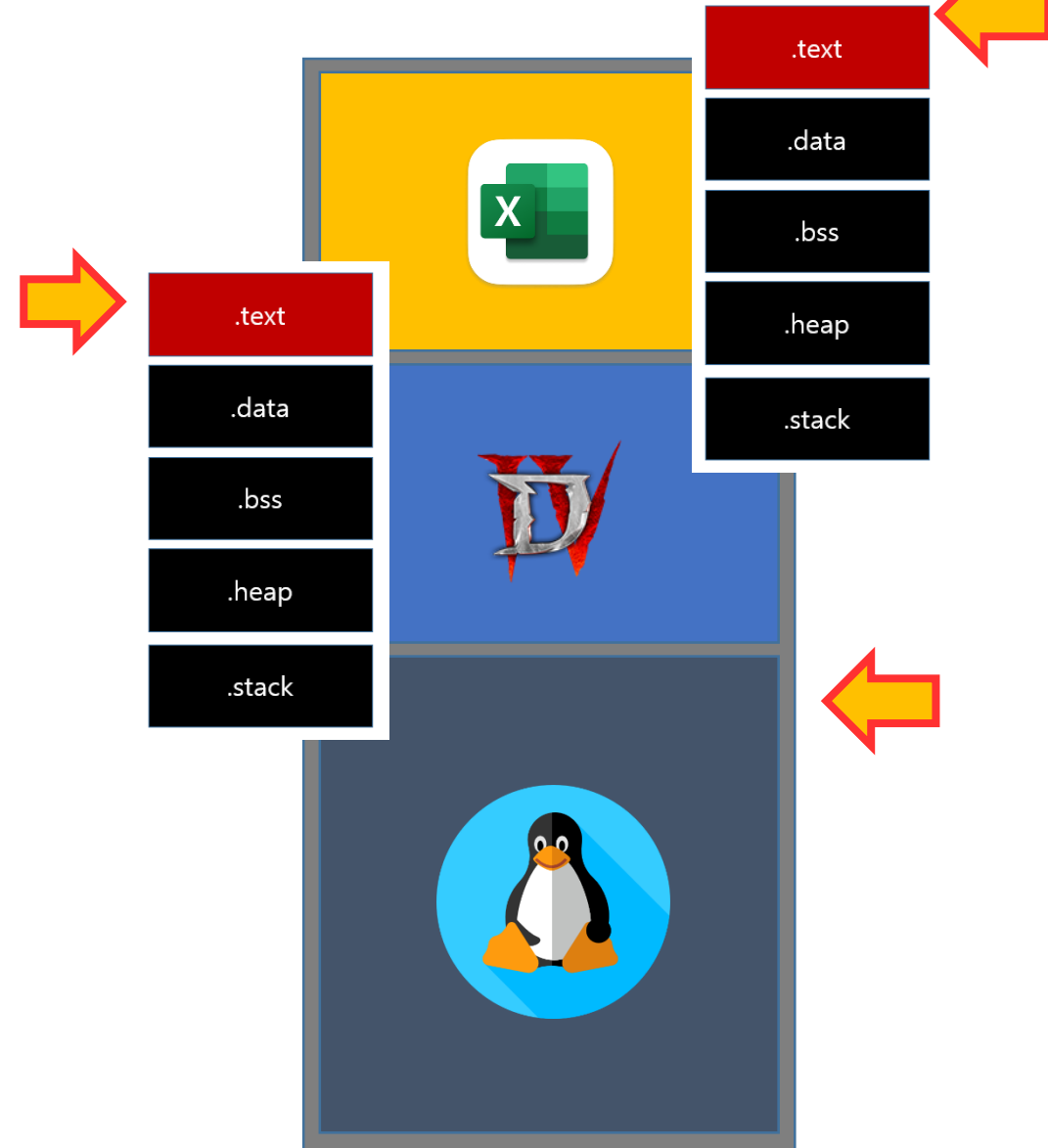


그런데 한 프로세스의
.text만 수행하면
다른 프로세스는 멈춰있는 상태



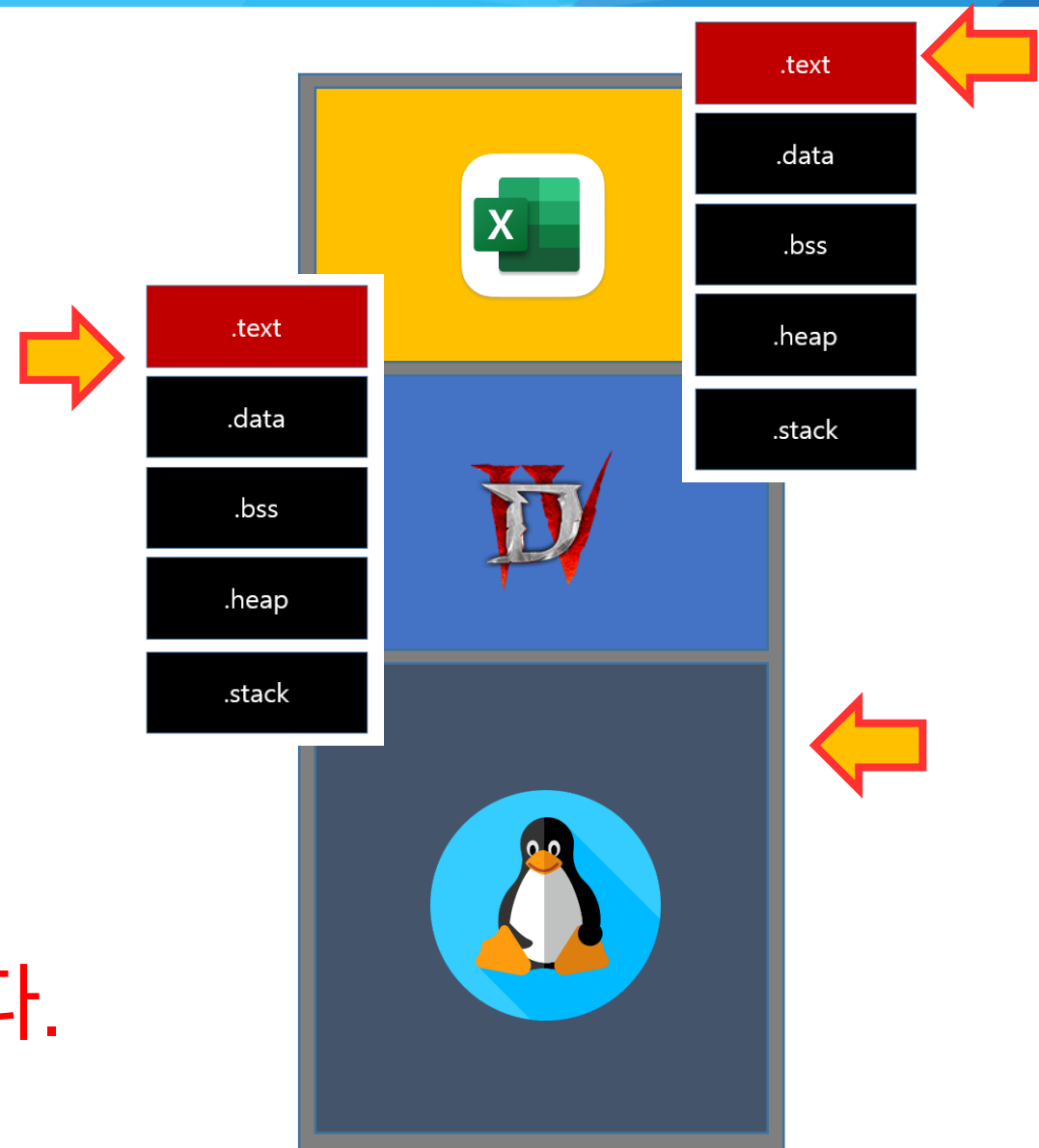
동시에 여러 프로세스가
동시에 수행되어야 한다.

- 어떻게 동시에 수행될까?



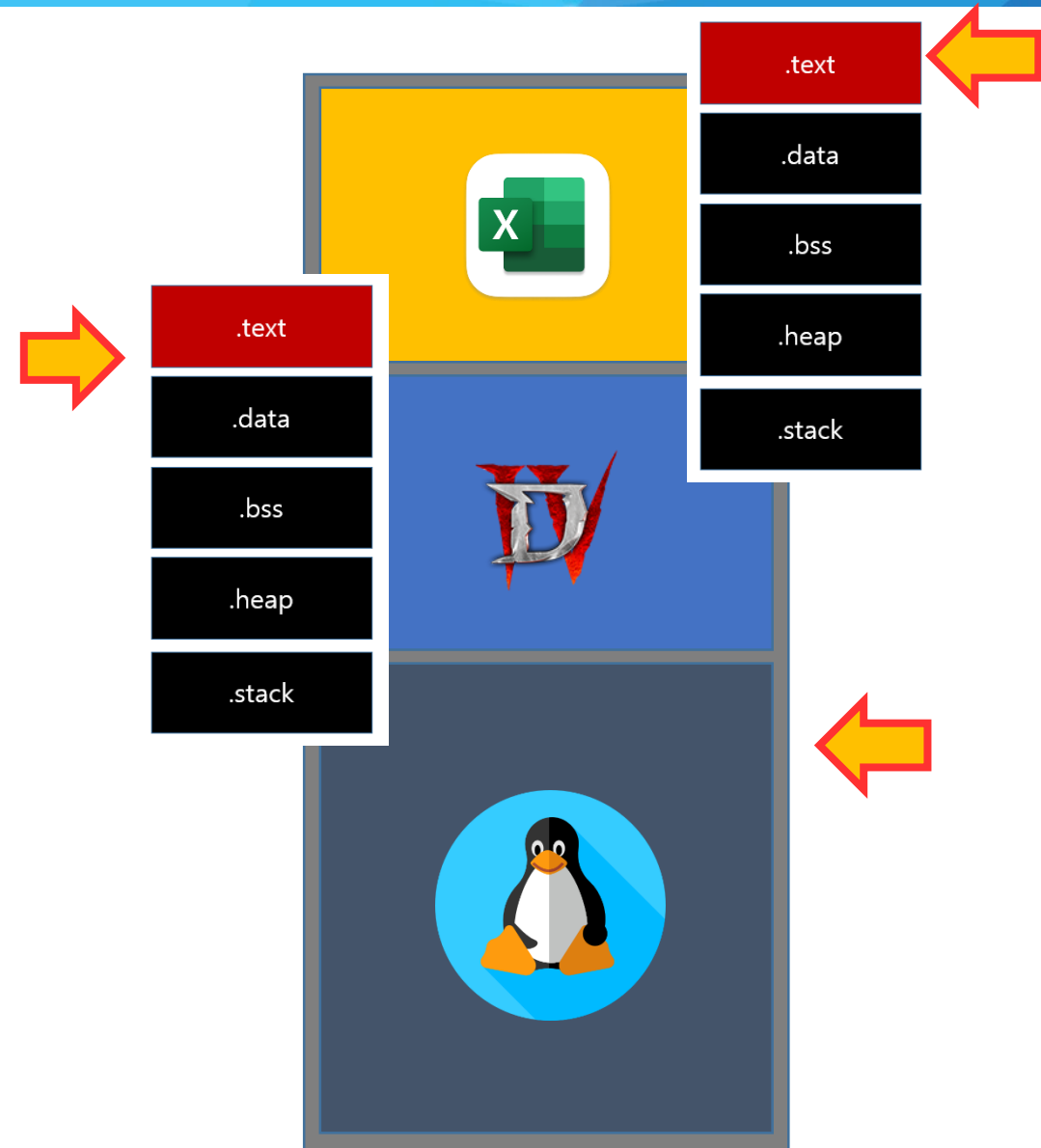
그런데,
CPU는 하나이기에
해결 방법은

매우 빠른 속도로
하나씩 돌아가면서 수행하는 것이다.



이런 작업을
커널이 지원해야 한다.

그런데 임베디드 S/W 개발자가
다음과 같이 동작되도록 개발하기
매우 어려운 일이다.

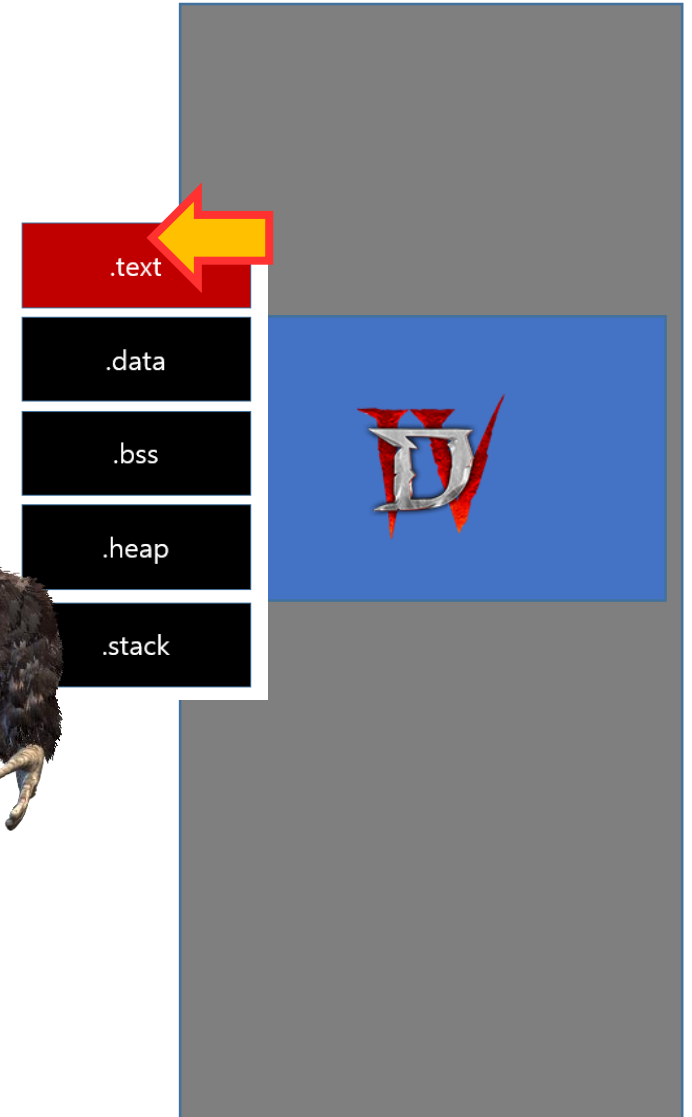


임베디드 제품을 만들 때,
프로세스들을 동시에 실행되는 기능이 필요하다면,

Firmware 개발 Level로는 어렵다.
RTOS 또는 Linux OS를 올려, 개발을 해야 한다.

멀티 쓰레드

한 프로세스 안에서
하나의 캐릭터가 뛰어 놀 수 있도록
개발하는 것은 쉽다.



한 프로세스 안에서,
캐릭터들이 동시에 움직이도록
코딩하는 것은 어렵다.

- 이럴 때 사용하는 것이 Thread 이다.

운영체제가
API로 지원해 주어야만 한다.



Thread를 위한
POSIX API가 존재하여,

이 Thread POSIX API만
사용할 줄 알면,
쉽게 동시작업 프로그래밍이 가능하다.



임베디드 제품을 개발시

- 여러 프로세스를 동시에 동작시키는 멀티태스킹을 구현해야 할 때,
- 한 프로세스 내, 쓰레드 프로그래밍이 필요할 때,

위와 같은 경우에

Firmware (직접개발) 보다는

POSIX API가 지원되는

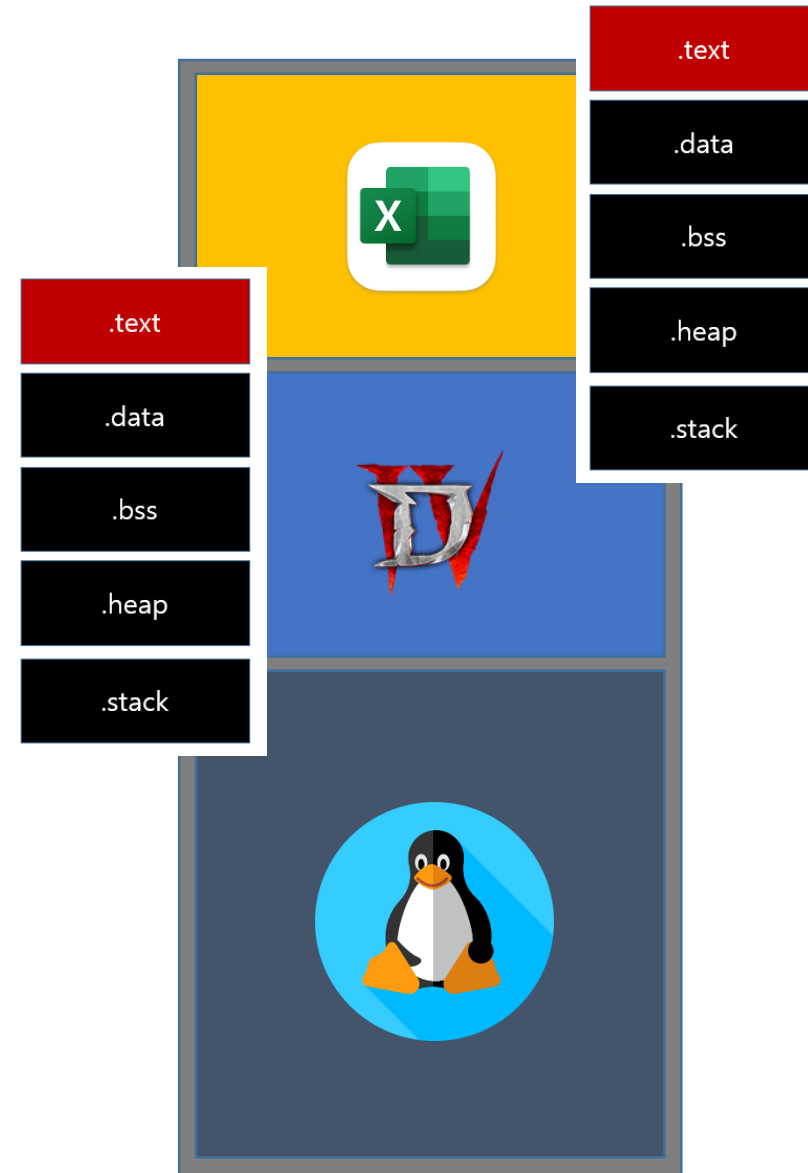
RTOS 또는 Linux 급의 OS를 임베디드 제품에 설치 후,
임베디드 개발을 한다.

프로세스끼리는
독립된 메모리 공간을 갖는다. (O, X)

프로세스끼리
독립된 메모리 공간을 갖는다.

따라서,
변수 값들을 서로
공유할 수 없다.

- IPC를 이용하면 가능



Day2-2. Thread

챕터의 포인트

- void*
- Thread
- Thread와 매개변수
- Thread의 메모리 공유
- mutex

void*

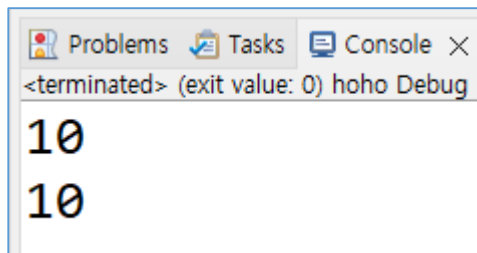
목표

- Thread 코드를 이해하기 위해 void 포인터를 먼저 학습한다.

만능 포인터 사용

- void * 를 사용한 예시

eclipse 로 코드를 실행한다.

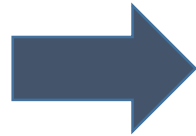


```
3  #include<stdio.h>
4  void abc(void* v){
5      int *p = (int*)v;
6      printf("%d\n", *p);
7
8      int q = *(int*)v;
9      printf("%d\n", q);
10 }
11
12 int main(){
13     int x = 10;
14     abc(&x);
15     return 0;
16 }
```

어떤 타입이든 상관 없이 저장할 수 있다. (만능)

- 저장만 가능하다.
- 함수에 타입 상관없이 전송할 때 유용
- 사용은 불가능하다.

```
int x = 0;  
char t = 'A';  
  
int* p1 = &x;  
char* p2 = &t;
```



```
int x = 0;  
char t = 'A';  
  
void* p1 = &x;  
void* p2 = &t;
```

*p1 을 바로 사용할 수 없다.

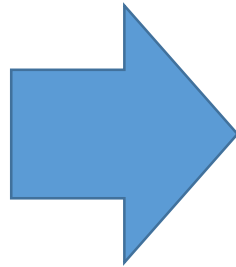
```
void * p1 = &x;  
void * p2 = &t;  
  
printf("%d", *p1);
```


void* 에 담긴 값을
사용하기 위해서는 형변환을 해야 한다.

```
int x = 0;
char t = 'A';

void * p1 = &x;
void * p2 = &t;

int* p = (int*)p1;
printf("%d", *p);
```



```
int x = 0;
char t = 'A';

void * p1 = &x;
void * p2 = &t;

//printf("%d", *p1); // 불가능
printf("%d", *(int *)p1); //가능
```

Thread

목표

- Thread 프로그래밍 기법에 대해 학습한다.
- Thread 의 메모리 공유에 대해 이해하고, mutex 를 이용해 Critical Section 을 방지한다.

어떤 프로그램 내에서,
특히 프로세스 내에서 실행되는 흐름의 단위를 말한다.

- 함수 단위로 구성된다.
- 동시 동작을 위해 꼭 필요한 기법 중 하나이다.



~/test 디렉토리 생성

- sample.c
- gcc sample.c -o ./gogo

sleep(s)

- s 초 만큼 delay
- Linux Library 지원 (man page 3)

해당 코드를 실행하면 어떤 결과가 나올까?

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  void abc(){
5      while(1) {
6          printf("ABC\n");
7          sleep(1);
8      }
9  }
10 void bts() {
11     while(1) {
12         printf("BTS\n");
13         sleep(1);
14     }
15 }
16
17 int main(){
18     abc();
19     bts();
20     return 0;
21 }
```

<https://gist.github.com/hoconoco/33ebca26b5b49be5f1f08b2c4a39a537>

abc() 에 있는 무한 반복을 실행하기 때문에,
bts() 가 실행되지 않는다.

- Thread를 이용해서 실행하자.

```
4 void abc(){  
5     while(1) {  
6         printf("ABC\n");  
7         sleep(1);  
8     }  
9 }
```

```
ssafy@ssafy-VirtualBox:~/test$ gcc ./sample.c -o ./gogo  
ssafy@ssafy-VirtualBox:~/test$ ./gogo  
ABC  
ABC  
ABC
```

~/test2 디렉토리 생성

- sample.c
- gcc sample.c -o ./gogo -lthread
 - 소문자 L

```
24  int main()  
25  {  
26      pthread_t t1, t2;  
27  
28      pthread_create(&t1, NULL, abc, NULL);  
29      pthread_create(&t2, NULL, bts, NULL);  
30  
31      pthread_join(t1, NULL);  
32      pthread_join(t2, NULL);  
33  
34      return 0;  
35  }
```

ABC, BTS 가 모두 출력된다!

- 이것이 Thread 프로그래밍의 핵심!

```
ssafy@ssafy-VirtualBox:~/test2$ ./gogo
ABC
BTS
BTS
ABC
ABC
BTS
```


Thread 를 사용할 때, 약속된 기본 형태가 존재한다.

- Interface → 이 방식으로 코드를 작성해 주세요~
- **void* 을 리턴 받는 형태** (생략 가능)
 - void 함수 → return 값 없음

```
pthread_create(&t1, NULL, abc, NULL);
```

```
5 → void *abc(){  
6     while(1) {  
7         printf("ABC\n");  
8         sleep(1);  
9     }  
10 }
```

<pthread.h>

- **POSIX thread** 의 줄임말
- 리눅스, Window 등 어떤 OS 든지 POSIX 를 지원한다면, Thread 를 사용할 수 있다.
- 위치 : /usr/include/pthread.h
- thread를 사용할 경우, 반드시 pthread librar를 같이 링킹 해야 한다.
 - gcc ./sample.c -o ./gogo -lpthread

```
ssafy@ssafy-VirtualBox:/usr/include$ ls -al | grep pthread.h
-rw-r--r--  1 root root  41701  7월  26 16:44 pthread.h
ssafy@ssafy-VirtualBox:/usr/include$
```

pthread_t

- Thread 핸들러
- Thread 를 제어하거나 정의할 때 사용한다. (구조체로 작성됨)
- 해당 구조체에는 채워 넣어야 할 멤버들이 있다.
- 이 멤버들은 pthread_create() 를 통해 채워 넣는다.

```
pthread_t t1, t2;
```

int pthread_create(thread id, thread 속성, 함수, 파라미터)

- thread id : Thread 핸들러 주소, pthread_t*
 - thread 생성 시 속성 : Thread 우선순위 같은 것 설정 가능, const pthread_attr_t*
 - 함수 : Thread로 동작 시킬 함수의 주소 (함수 포인터), void*
 - 파라미터 : Thread로 동작할 함수에 인자 값을 전달 가능, void*
-
- Thread를 생성한다.
 - 생성된 Thread는 파라미터 값을 함수에 전달하며, 함수를 실행시킨다.
 - Thread가 언제 실행될 지는 전적으로 스케줄러가 정한다. (BTS 와 ABC 출력이 제멋대로인 이유!)

```
28      pthread_create(&t1, NULL, abc, NULL);  
29      pthread_create(&t2, NULL, bts, NULL);
```

`int pthread_join(pthread_t th, void **thread_return)`

- Thread 가 종료되는 것을 기다린다.
- 종료될 경우, 해당 Thread가 사용하던 자원 등을 정리한다.
- `thread_id` : Thread 핸들러
- `void** thread_return` : Thread 종료 시, 동작한 함수의 return 값
- `return` : error 처리 기능

`pthread_create()`와 `pthread_join()`은 깐부다!

- 꼭! 같이 사용한다.

Thread를 생성할 때,
반드시 첫번째 Thread 가 동작하는 것이 아니다!
→ 무엇이 먼저 동작할 지에 대한 것은 스케줄러가 정한다.

```
ssafy@ssafy-VirtualBox:~/test2$ ./gogo
ABC
BTS
BTS
ABC
ABC
BTS
```

현재 코드에서 Thread는 몇 개 일까?

```
24  int main()
25  {
26      pthread_t t1, t2;
27
28      pthread_create(&t1, NULL, abc, NULL);
29      pthread_create(&t2, NULL, bts, NULL);
30
31      pthread_join(t1, NULL);
32      pthread_join(t2, NULL);
33
34      return 0;
35  }
```

정답! 총 3개의 Thread 가 동작 중이다.

Confidential

t1, t2 그리고, main() 3개의 Thread 가 동작 중이다.

- pthread_join 에서 멈춰 있는 것은 main Thread 이다.
- abc, bts 가 종료될 때까지 pthread_join 에서 멈춰 있다!

기다리고 또 기다린다.

```
24  int main()  
25  {  
26      pthread_t t1, t2;  
27  
28      pthread_create(&t1, NULL, abc, NULL);  
29      pthread_create(&t2, NULL, bts, NULL);  
30  
31      pthread_join(t1, NULL);  
32      pthread_join(t2, NULL);  
33  
34      return 0;  
35  }
```


Thread 와 매개변수

~/test3 디렉토리 생성한다.

- sample.c
- gcc sample.c -o ./gogo -lpthread

```
5 void* abc(void* p){
6     int a = *(int*)p;
7     while(1){
8         printf("%d\n", a);
9         a++;
10        usleep(300*1000);
11        if( a>10 ) break;
12    }
13 }
14
15 int main(){
16     pthread_t tid;
17     int gv = 1;
18
19     pthread_create(&tid, NULL, abc, &gv);
20     pthread_join(tid, NULL);
21     return 0;
22 }
```

pthread_create() 로 함수에 인자 값을 전달 할 수 있다.

- void* 타입
- 형변환해서 사용한다.

변수 하나를 전달하였다. 다음은 구조체 data를 전달하자.

```
ssafy@ssafy-VirtualBox:~/test3$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test3$ ./gogo
1
2
3
4
5
6
7
8
9
10
```

~/test4 디렉토리 생성한다.

- sample.c
- gcc sample.c -o ./gogo -lpthread

```
5  struct Node{
6      int y;
7      int x;
8  };
9
10 void *abc(void *p){
11     struct Node *val = (struct Node *)p;
12     printf("%d %d\n", val->x, val->y);
13 }
```

void* 는 만능이다.

변수, 구조체 상관없이 모두 받는다.

- 역참조 하는 코드도 모두 같은 것이다.

```
ssafy@ssafy-VirtualBox:~/test4$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test4$ ./gogo
val1 4 2
val2 4 2
```

~/test5 디렉토리 생성한다.

- sample.c
- gcc sample.c -o ./gogo -lpthread

실행 하기 전,
해당 코드의 결과는 무엇일까?

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *run(void *arg){
5      int a = *(int*)arg;
6      printf("%d", a);
7  }
8
9  int main(){
10     pthread_t t[4];
11     for (int i = 0; i<4; i++) pthread_create(&t[i], NULL, run, &i);
12     for (int i = 0; i<4; i++) pthread_join(t[i], NULL);
13     return 0;
14 }
```

<https://gist.github.com/hoconoco/2b989ecaefbe16e55e3a4d78fb69d900>

예상과 전혀 다른 결과!

- 분명, 해당 코드의 개발 의도는 0~3까지 숫자를 출력하는 것 같았는데, 무엇이 잘못된 걸까?



```
2442ssafy@ssafy-VirtualBox:~/test5$ ./gogo
2442ssafy@ssafy-VirtualBox:~/test5$ ./gogo
3424ssafy@ssafy-VirtualBox:~/test5$ ./gogo
2244ssafy@ssafy-VirtualBox:~/test5$ ./gogo
2243ssafy@ssafy-VirtualBox:~/test5$ ./gogo
3344ssafy@ssafy-VirtualBox:~/test5$ ./gogo
2334ssafy@ssafy-VirtualBox:~/test5$ ./gogo
4434ssafy@ssafy-VirtualBox:~/test5$ ./gogo
2324ssafy@ssafy-VirtualBox:~/test5$
```

Thread의 동작은 스케줄러가 정한다.

- 언제 동작할 지 알 수 없는 상태

그리고 Thread 는 동작할 때,

i의 주소를 보고 그 곳에 저장된 값을 가져간다.

- 가져가려고 봤더니, main Thread 가 for문을 다 완료하여, i에 3이 저장되어 있다면?

```
pthread_create(&t[i], NULL, run, &i);
```


~/test6 디렉토리 생성한다.

- sample.c
- gcc sample.c -o ./gogo -lpthread

i 값을 배열에
미리 담아두면 된다.

```
11  int main(){
12      int id[4];
13      for (int i = 0; i<4; i++) {
14          id[i] = i;
15          pthread_create(&tid[i], NULL, run, &id[i]);
16      }
17      for (int i = 0; i<4; i++) pthread_join(tid[i], NULL);
18      return 0;
19  }
```

의도대로 출력되는 것을 볼 수 있다.

```
ssafy@ssafy-VirtualBox:~/test6$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test6$ ./gogo
0321ssafy@ssafy-VirtualBox:~/test6$ ./gogo
0213ssafy@ssafy-VirtualBox:~/test6$ ./gogo
0231ssafy@ssafy-VirtualBox:~/test6$ ./gogo
1032ssafy@ssafy-VirtualBox:~/test6$ ./gogo
3201ssafy@ssafy-VirtualBox:~/test6$ ./gogo
0213ssafy@ssafy-VirtualBox:~/test6$ ./gogo
```

Thread의 메모리 공유

여러 Thread가 동작 중일 때, Thread 끼리 메모리를 공유한다.

- .text, .bss, .data, .heap → 공유 가능
- .stack → 공유 안됨

우선, 코드를 이용해 메모리가 공유되는 지 확인한다.

- ~/test7 디렉토리 생성
 - sample.c
 - gcc sample.c -o ./gogo -lpthread

```
5  int a = 100; //.data
6  int b;          //.bss
7  void* abc(){
8      int c = 10; //stack
9      printf("=====\n");
10     printf(".data  : 0x%p\n", (void*)&a);
11     printf(".bss   : 0x%p\n", (void*)&b);
12     printf(".stack : 0x%p\n", (void*)&c);
13     printf(".heap  : 0x%p\n", (int*)malloc(4));
14     return 0;
15 }
```

공유하고 있는 메모리 : .data , .bss, .heap

공유하지 않는 메모리 : .stack

```
ssafy@ssafy-VirtualBox:~/test7$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test7$ ./gogo
=====
.data : 0x0x564285d49010
.bss : 0x0x564285d49018
.stack : 0x0x7f77e6a4ee4
.heap : 0x0x7f77e8000f70
=====
.data : 0x0x564285d49010
.bss : 0x0x564285d49018
.stack : 0x0x7f77ede3ee4
.heap : 0x0x7f77e8000f90
```

메모리 공유가 좋은 것이 아니다.
전혀 예상치 못한 결과를 초래하게 된다.

이를 확인하고, 해결하자.

- ~/test8 디렉토리 생성
 - sample.c
 - gcc sample.c -o ./gogo -lpthread

```
4  int cnt=0; //.data 공유되는 메모리
5  void* run(void* arg){
6      for(int i=0; i<10000; i++) cnt++;
7  }
8
9  int main(){
10     pthread_t tid[4];
11     for(int i=0; i<4; i++) pthread_create(&tid[i], NULL, run, NULL);
12     for(int i=0; i<4; i++) pthread_join(tid[i], NULL);
13
14     printf("%d\n", cnt);
15     return 0;
16 }
```

<https://gist.github.com/hoconoco/a10c5ce80f862aa96caf4cfb62833c28>

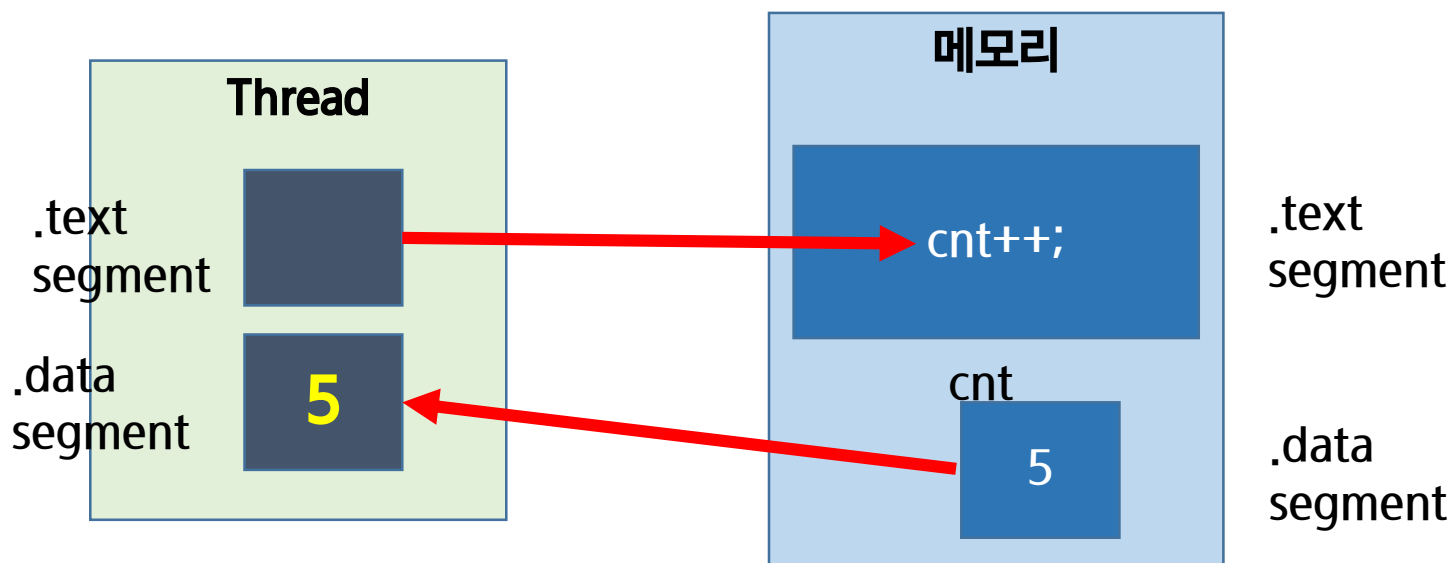
예상치 못한 결과!

- 4개의 Thread 로 전역 변수 cnt 값을 총 4만번 ++ 했지만, 결과는 다르다.
- 왜 이런 결과가 나오는 지 그림으로 이해하자.

```
ssafy@ssafy-VirtualBox:~/test8$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
18244
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
18606
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
31539
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
39892
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
34396
ssafy@ssafy-VirtualBox:~/test8$ ./gogo
27502
```

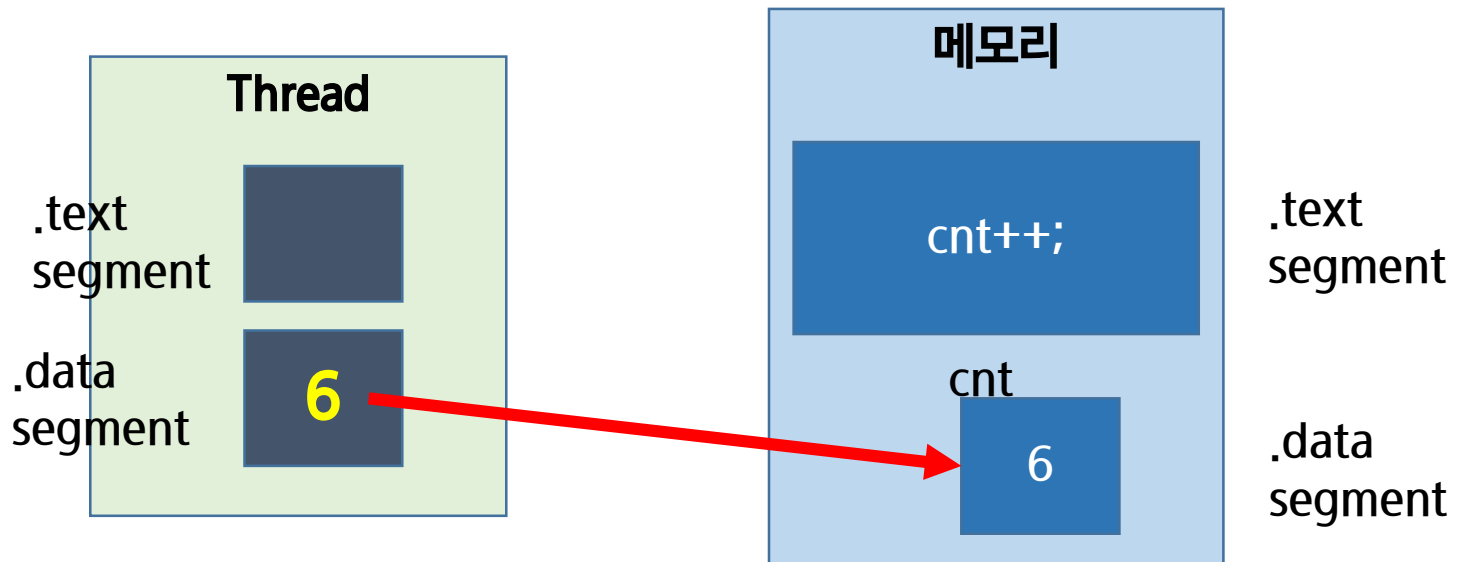
Thread가 하나일 때, 메모리의 변수 값 변경

- Thread 는 공유하고 있는 명령어 코드(.text) 를 확인
- cnt++ 을 하기 위해 전역 변수 (.data) 의 값을 복사



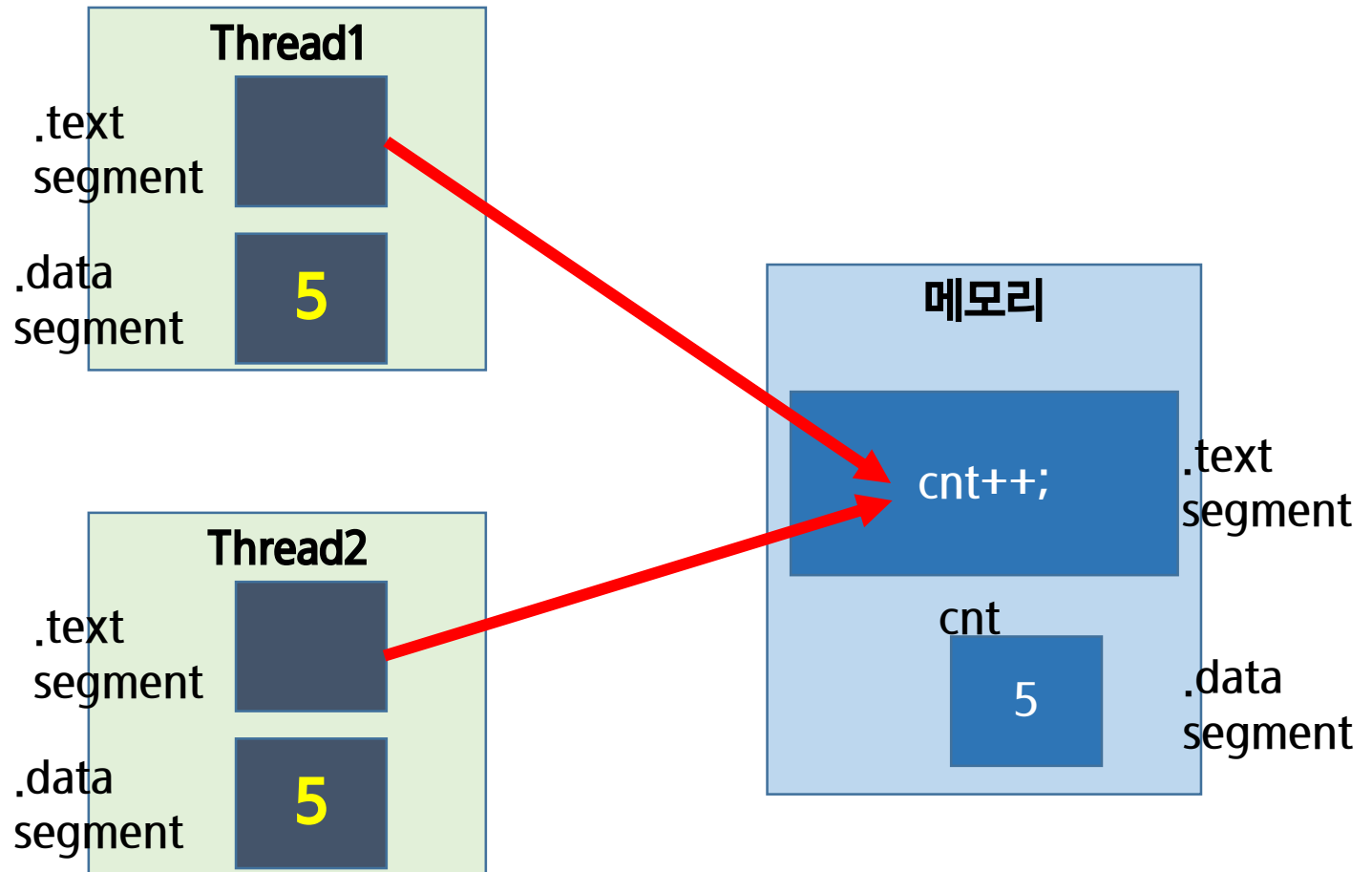
Thread가 하나일 때, 메모리의 변수 값 변경

- `cnt++` 을 수행하여 6 으로 증가
- 이제 `cnt` 값을 메모리에 공유한다.



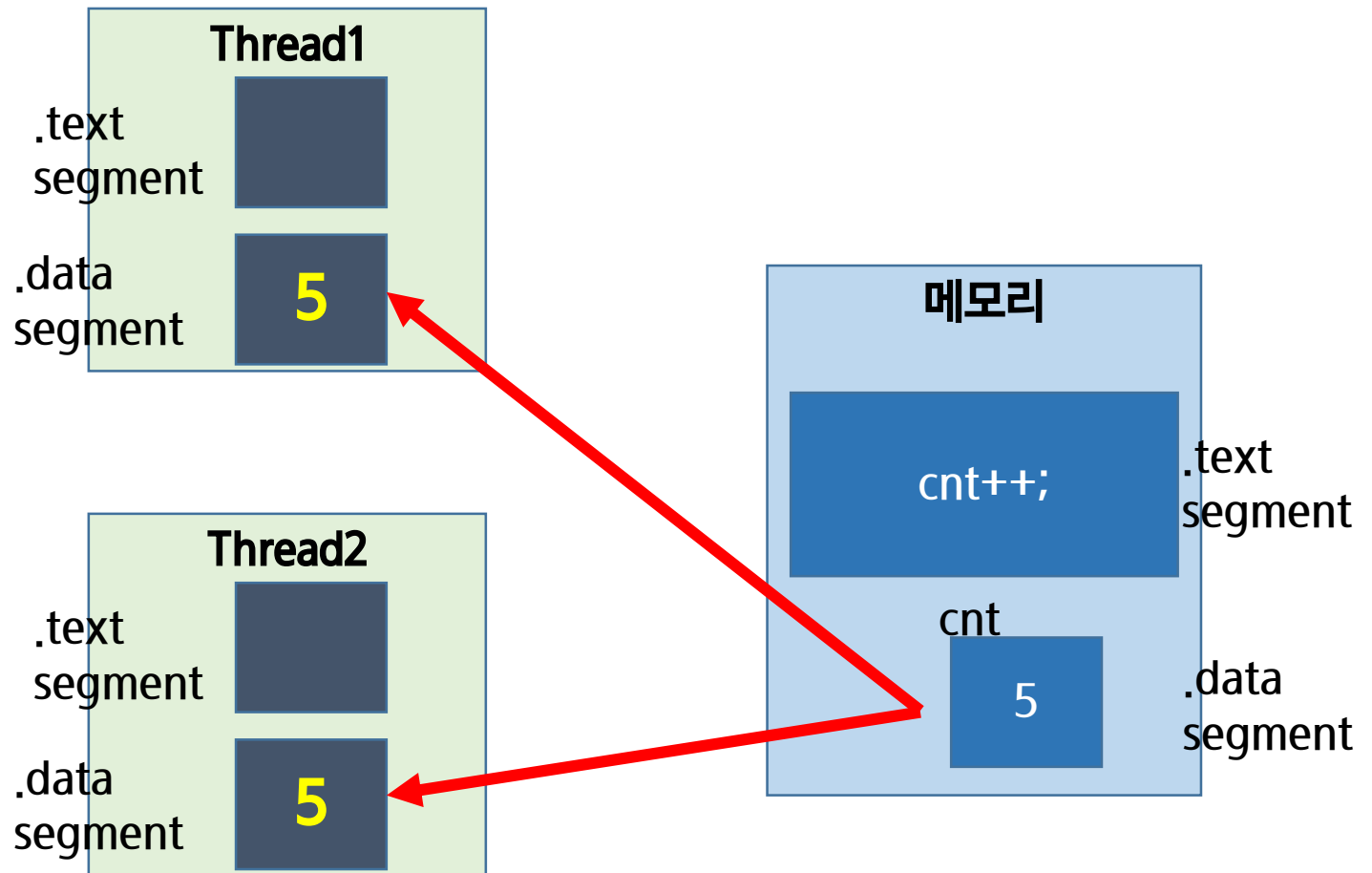
Thread가 둘 이상일 때, 메모리의 변수 값 변경

- 우선, Thread의 동작은 스케줄러가 정한다.
- 언제 동작할 지 모른다.
- 그래도 각자, 공유되고 있는 명령어 (.text) 를 보고 cnt++ 를 하려고 준비한다.



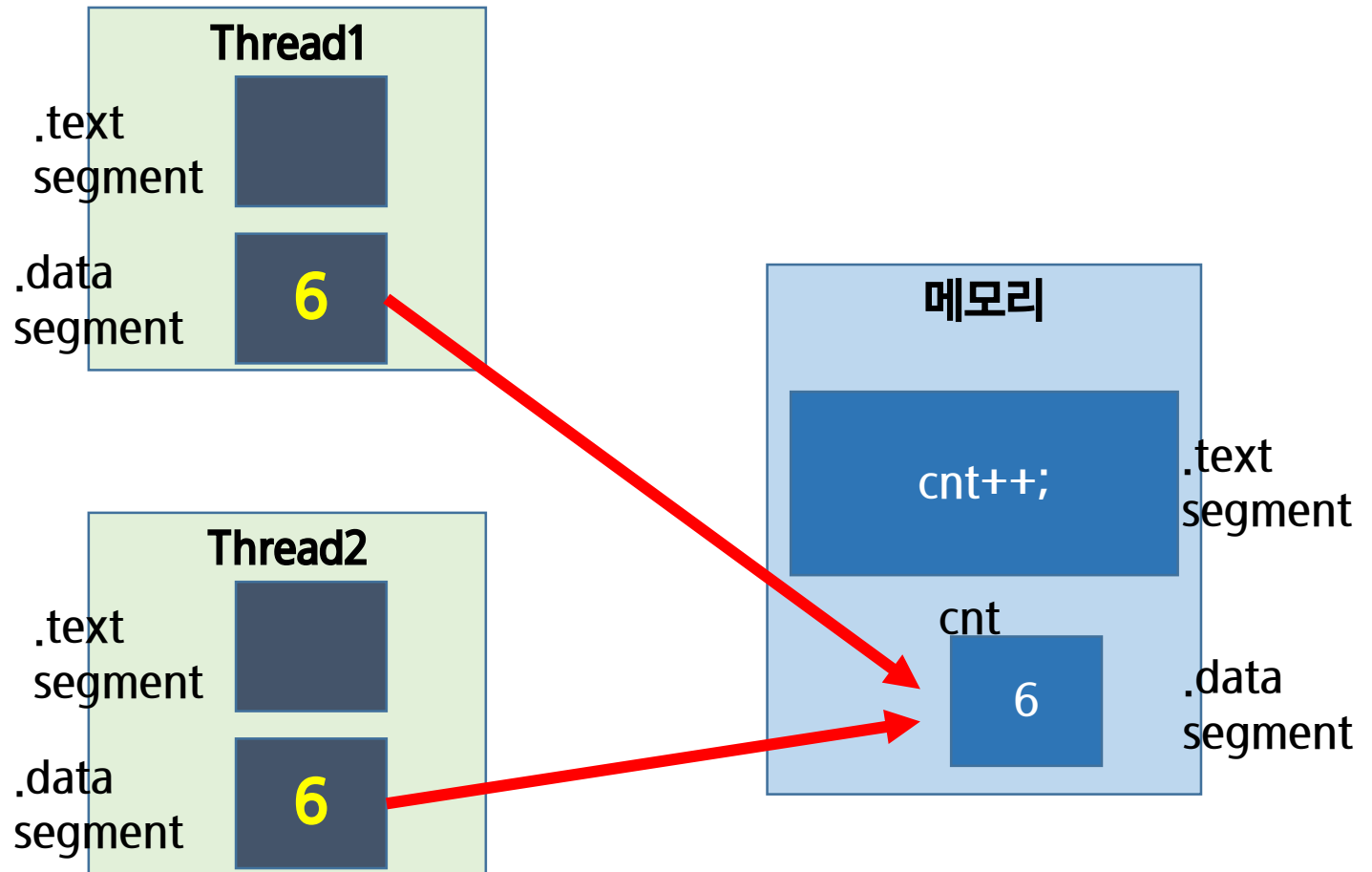
Thread가 둘 이상일 때, 메모리의 변수 값 변경

- 각각의 Thread가 전역변수(.data) 값을 복사해 왔다.



Thread가 둘 이상일 때, 메모리의 변수 값 변경

- 복사한 cnt 값을 ++ 해서 6으로 만든 뒤, 서로 업데이트를 한다.
- 2개의 Thread 가 cnt++ 을 해서, 7이 되어야 하지만, 6이 되는 이유



Race Condition

- Thread / Process 의 타이밍에 따라 결과 값이 달라질 수 있는 상태
- 공유 메모리로 인해, 결과가 달라질 수 있는 상태를 Race Condition 이라 한다.

Critical Section

- 공유되는 메모리로 인해, 문제가 생길 수 있는 코드 영역을 Critical Section 이라 한다.
- Thread / Process가 동시에 접근해선 안되는 곳
 - ex) HW 장치를 사용하는 곳 / Shared Resource (전역 변수 등)

아주 작은 sleep을 사용하여,
동작 타이밍을 바꾼다.

- 유지보수 측면 좋지 않은 방법

```
10     pthread_t tid[4];
11     for(int i=0; i<4; i++){
12         pthread_create(&tid[i], NULL, run, NULL);
13         usleep(100);
14     }
```


Thread / Process Synchronization

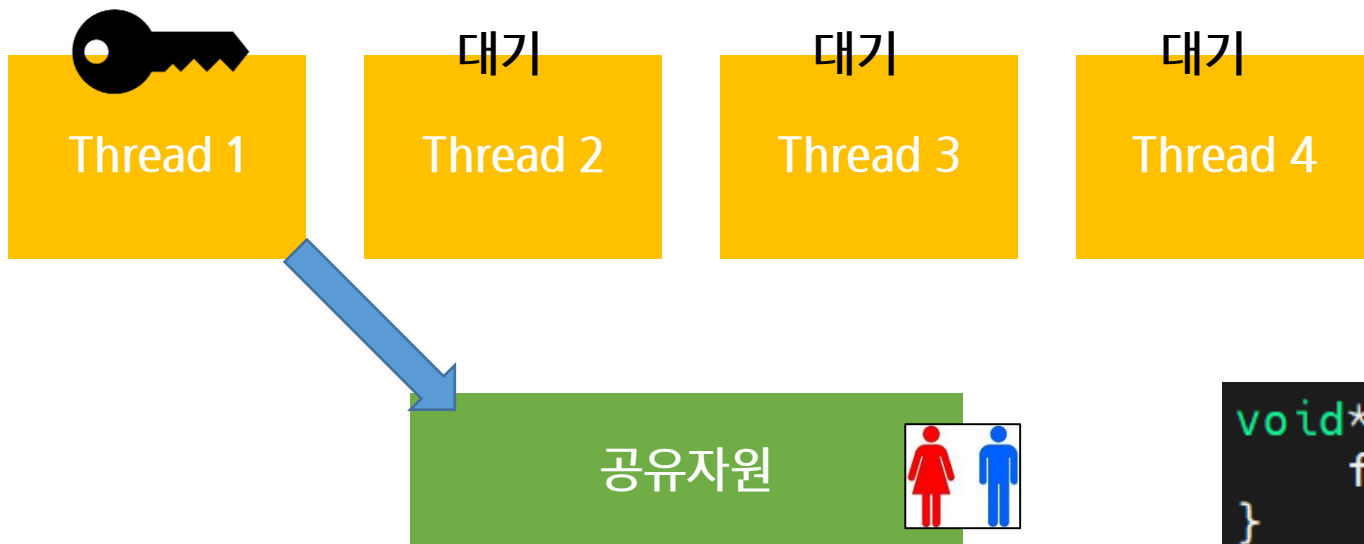
- Race Condition 이 발생하지 않도록, 예방하기 위한 도구, 기능
- 다양한 알고리즘을 이용해서 해결한다.
 - Mutex (뮤텝스)
 - Semaphore
 - Spinlock
 - Barrier 등등

우리 수업에서는 Mutex 에 대해 알아보자.

mutex

키를 기반으로 하는 상호 배제 기법

- 공유 자원에 접근할 때, 키를 이용해서 다른 Thread / Process 의 접근을 막는다.
- Critical Section 을 진입할 때, 잠그고, 코드를 다 수행한 뒤, 푼다.
- 동기화 하고자 하는 대상이 1개 일 때 사용
 - 키를 잃어 버리면, 대참사가 발생한다.



Critical Section

```
void* run() {  
    for(int i=0; i<10000; i++) cnt++;  
}
```

- ~/test9 디렉토리 생성
 - sample.c
 - gcc sample.c -o ./gogo -lpthread

```
4  pthread_mutex_t mlock;
5  int cnt=0;
6  void *run(){
7      pthread_mutex_lock(&mlock);
8      for(int i=0; i<10000; i++) cnt++;
9      pthread_mutex_unlock(&mlock);
10 }
11
12 int main(){
13     pthread_mutex_init(&mlock, NULL);
14
15     pthread_t tid[4];
16     for(int i=0; i<4; i++) pthread_create(&tid[i], NULL, run, NULL);
17     for(int i=0; i<4; i++) pthread_join(tid[i], NULL);
18
19     printf("%d\n", cnt);
20
21     pthread_mutex_destroy(&mlock);
22
23     return 0;
24 }
```

<https://gist.github.com/hoconoco/869c0a7635f7d797ed2aae261141ed1a>

정상 동작한다!

- pthread_mutex_lock() / pthread_mutex_unlock() 을 이용해, Critical Section 인 for 문을 동기화 하였다.

```
6  void *run(){
7      pthread_mutex_lock(&mlock);
8      for(int i=0; i<10000; i++) cnt++;
9      pthread_mutex_unlock(&mlock);
10 }
```

```
ssafy@ssafy-VirtualBox:~/test9$ gcc ./sample.c -o ./gogo -lpthread
ssafy@ssafy-VirtualBox:~/test9$ ./gogo
40000
ssafy@ssafy-VirtualBox:~/test9$ ./gogo
40000
```

pthread_mutex_t

- mutex 핸들러 객체 생성

int pthread_mutex_init (핸들러 , 속성)

- mutex 객체 초기화
- 핸들러 : pthread_mutex_t*
- 속성 : const pthread_mutex_attr*
 - NULL을 사용하면, default 로 동적 초기화

int pthread_mutex_destroy(핸들러)

- mutex 핸들러 제거, 동적 초기화일 때만 사용 가능
- 잠금 상태일 때 파괴하면, error 발생

int pthread_mutex_lock(핸들러)

- mutex 를 잠근다.
- 이미 잠겨 있는 데, 잠그면 오류

int pthread_mutex_unlock(핸들러)

- 잠근 mutex 를 푼다.
- 이미 풀려 있으면 오류가 발생

Thread 의 종료

Thread를 생성하고 난 뒤,
꼭 종료를 잘 해야, 사용한 메모리가 정리 된다.

- 메모리 정리는 주로 OS 의 역할이다.

int pthread_cancel(pthread_t)

- 특정 Thread 를 종료하기 위한 목적으로 사용
- pthread_cancel() 을 받은 Thread는 pthread_exit() 를 실행한다.
- return : 성공 여부

~/test10 디렉토리 생성

- sample.c
- gcc sample.c -o ./gogo -lpthread

```
12 void *killABC(void *arg) {
13     int count = 0;
14     while (count < 10) {
15         count++;
16         printf("[T2] %d\n", count);
17         sleep(1);
18     }
19     // 스레드1을 종료
20     printf("Thread1 Kill\n");
21     pthread_cancel(*(pthread_t *)arg);
22 }
23
24 int main() {
25     setbuf(stdout, NULL);
26     pthread_t t1, t2;
27
28     pthread_create(&t1, NULL, abc, NULL);
29     pthread_create(&t2, NULL, killABC, (void *)&t1);
```

<https://gist.github.com/hoconoco/c082400c15ffb65a36e099d854c085b9>

Thread 2 에서 Thread 1 원격 종료 가능

- pthread_cancel() 을 위해
t1의 정보를 t2를 생성할 때, arg 로 전송

```
pthread_create(&t2, NULL, killABC, (void *)&t1);
```

```
ssafy@ssafy-VirtualBox:~/test10$ ./gogo
[T1] BTS
[T2] 1
[T1] BTS
[T2] 2
[T1] BTS
[T2] 3
[T1] BTS
[T2] 4
[T1] BTS
[T2] 5
[T2] 6
[T1] BTS
[T1] BTS
[T2] 7
[T1] BTS
[T2] 8
[T2] 9
[T1] BTS
[T1] BTS
[T2] 10
Thread1 Kill
[T1] BTS
```

Day2-4. 도전

다음 기능을 가진 App을 제작한다.
기능

- 3개의 Thread 사용
 - Thread 1 → A,B,C 문자를 0.3초에 하나씩 반복 출력
 - Thread 2 → 0~5 숫자를 0.2초에 하나씩 반복 출력
 - Thread 3 → A~Z 문자를 0.3초에 하나씩 반복 출력
-
- usleep(us) 사용 : 1초는 몇 us 일까?

```
ssafy@ssafy-VirtualBox:~$ ./gogo
[DD1] A
[DD2] 1
[DD3] A
[DD2] 2
[DD3] B
[DD1] B
[DD2] 3
[DD3] C
[DD2] 4
```

다음 기능을 가진 App을 제작한다.

기능

- Thread 37개 사용
- 각각의 Thread는 실행할 때, 1~37 까지의 숫자를 순서대로 받아간다.
- Thread 동작 시, 받아간 숫자를 출력한다.
- 모든 Thread의 동작이 종료되면, “VVCC END” 라고 출력

```
ssafy@ssafy-VirtualBox:~$ ./gogo
1 2 4 6 8 3 7 9 5 10 11 12 13 14 15 16 17 18 19 20 21 23 22 25 24 26 27 28 29 30
31 33 34 32 35 36 37 VVCC END
```

다음 기능을 가진 App을 제작한다.

기능

- Thread 4개 사용
- 모든 Thread는 하나의 함수 `abc()` 를 사용한다.
- Thread 1 → ABC 출력
- Thread 2 → MINMIN 출력
- Thread 3 → COCO 출력
- Thread 4 → KFCKFC 출력

```
ssafy@ssafy-VirtualBox:~$ ./gogo
2 MINMIN
3 COCO
4 KFCKFC
1 ABC
```


아래 기능을 만족하는 채굴기를 제작한다.

- 시작

- Prompt Message : “ssafy>>”
- 프로그램 실행 시, 채굴기가 자동으로 하나 생성되어 채굴 시작됨 (0.1초에 코인은 1씩 증가)

- 생성한 코인 출력 명령어 : **show**

- 현재 채굴한 코인 개수 출력

- 채굴기 생성 명령어 : **add**

- 채굴기가 하나 더 추가 됨(Thread)
- 0.1초에 코인 1씩 증가

- 채굴기 제거 명령어 : **del**

- 가장 오래된 채굴기 하나 제거 (Queue 자료구조)



내일 방송에서 만나요!

삼성 청년 SW 아카데미