파이썬 – HW4

임베디드스쿨1기

Lv1과정

2020. 08. 24

강경수

1. Thread

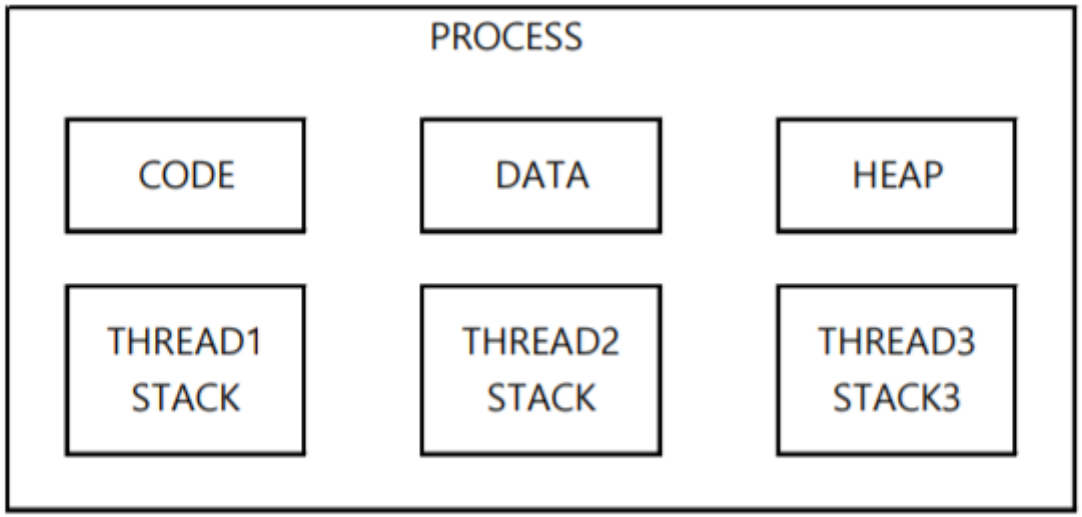
▣ THREAD란 무엇인가

2020.08.24 KKS

1. Process VS Thread

1)Process : 일반적으로 일컫는 프로그램 이는 내가 C로짠 구구단 프로그램도 해당되고
포토샵도 해당된다.

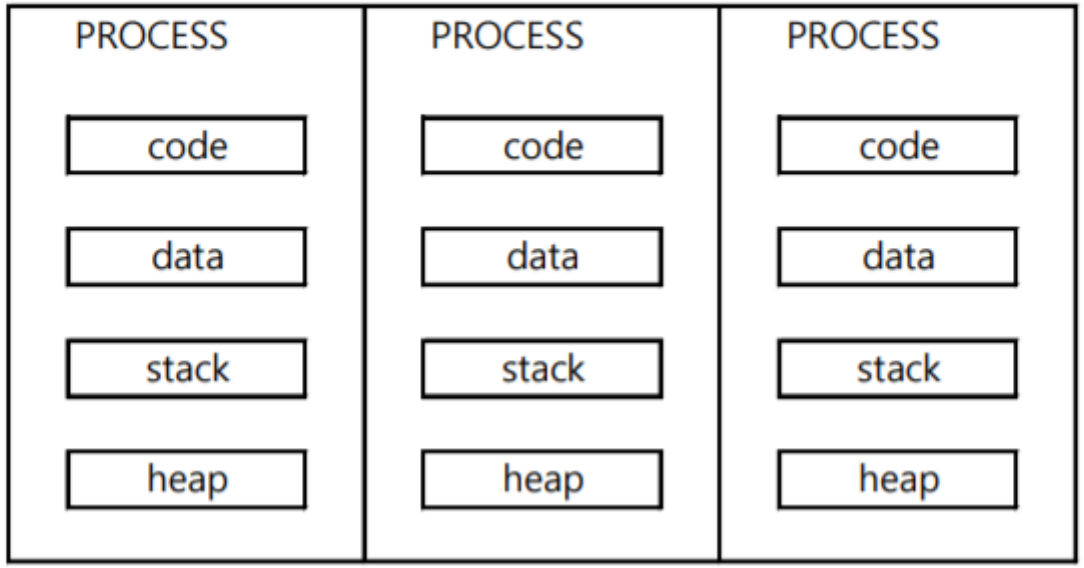
2) Thread : 한 Process 내부에서 Heap, Data, Code는 공유하며 Stack은 각각 할당받음
프로세스 내에서 실행되는 동작의 흐름



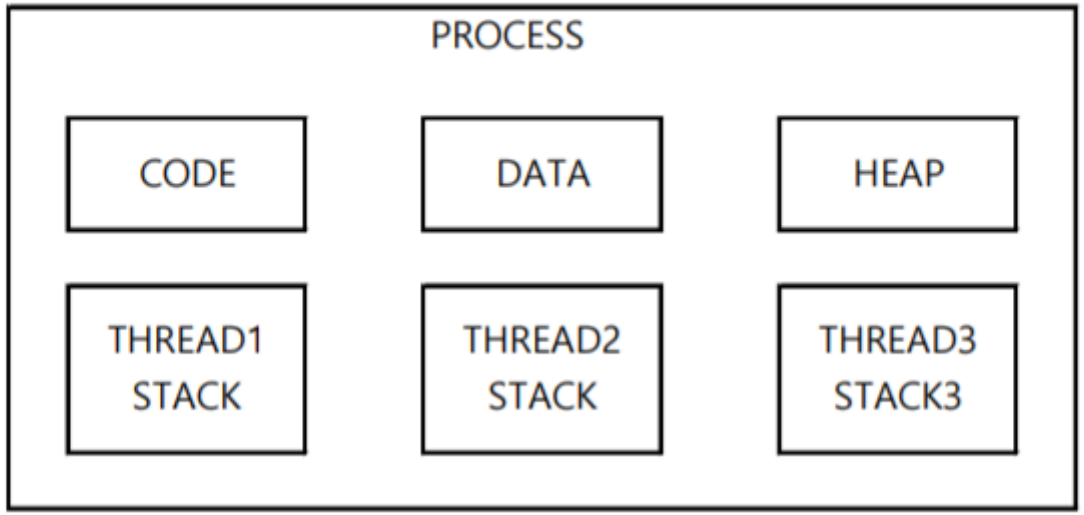
1. Thread

2. Multi Process VS Multi Thread

1) Multi Process: 하나의 컴퓨터에 여러 복수의 CPU를 장착→프로세스를 동시에 처리



2)Multi Thread: 여러 개의 스레드가 PROCESS를 실행



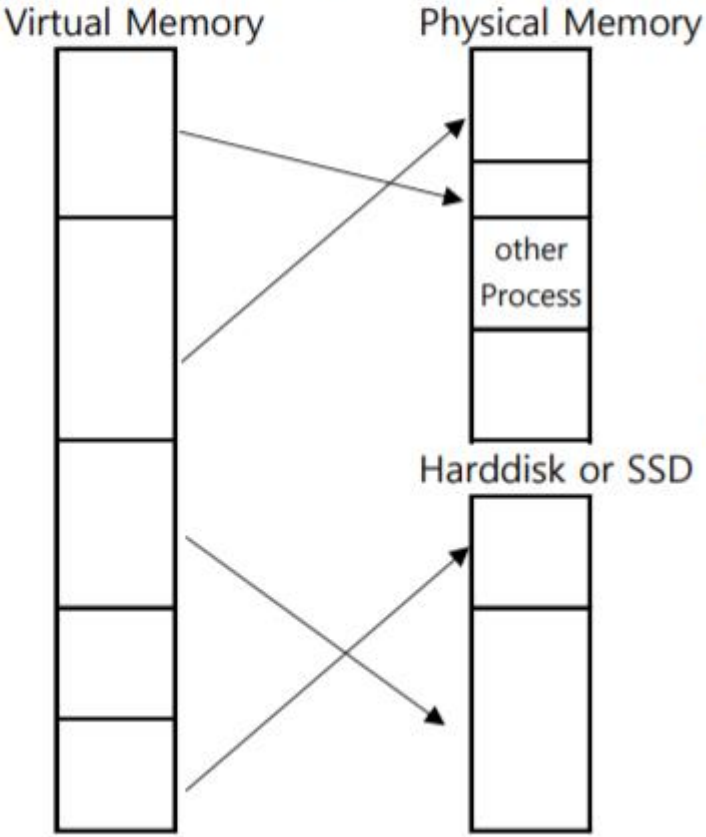
2. 가상메모리

▣ 가상메모리

2020.08.24

1. 가상메모리란 무엇인가?
메모리에 로드된 즉 실행된 프로세스가 가상의 공간을 참조하여 마치 커다란
물리 메모리를 갖고 있는것처럼 사용 할 수 있도록 하는것

2.프로그램의 용량 5GB? RAM용량 4GB? → HDD,SSD를 사용하기에 가능하다.



가상 메모리는 각 프로세스당 메인메모리와
동일한 크기로 하나씩 할당된다.
그리고 그 공간은 보조기억장치를 활용한다.

MMU에 의해 유저는 메모리매핑을 의식하지
않고 사용할 수 있다.

2. 가상메모리

3. 가상메모리 관리하는 기법 2가지

1) 세그멘테이션 기법

- 메모리를 크기가 다른 블록단위인 세그먼트로 분리하는 방법

메모리를 옆과 같은 세그먼트 블록으로 나누어 관리

필요한 만큼만 프로세스가 할당하기 때문에
내부 단편화는 존재하지 않는다.
외부 단편화는 존재한다.

메모리를 기능별로 쌓아서 프로그램을 관리함.

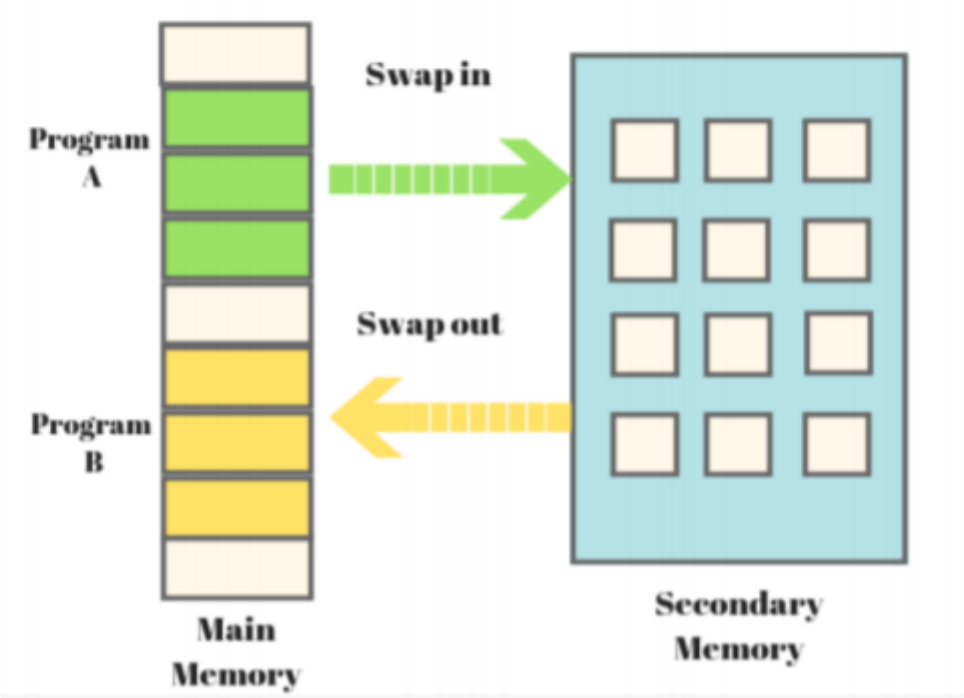
2) 페이징기법

- 세그멘테이션을 고정된 크기로 나누어 관리
  즉, 커다란 크기의 작업을 고정된 일정한 크기로 나누어 잘게 쪼개어 처리하는 것.

2. 가상메모리

3) Demand Paging

- 요구 페이징이란 초기에 필요한 것들만 적재하고, 페이지들이 실행 과정에서 실제로 필요할 때 적재하는 기법이다.
  즉, 한번도 접근되지 않는 페이지는 물리 메모리에 전혀 적재되지 않는다.



2. 가상메모리

4) 내부단편화
메모리를 할당할 때 프로세스가 필요한 양보다 더 큰 메모리가 할당되어서
프로세스에서 사용하는 메모리 공간이 낭비 되는 상황
메모장 os 할당 : 4kB, 실제 사용 : 1kB, 내부단편화 : Kb

5) 외부단편화
- 프로세스 메모리가 종료되고 남은 메모리들이 조각조각 존재하는 상황
  메모장 os 할당 :4kB FREE시 4kB가 조각으로 따로 존재

6)페이징 폴트
- 페이지가 램메모리에는 존재하지 않고 가상메모리(SSD.HDD에 매핑된 메모리)에
  존재하는 경우, 이때 운영체제는 DATA를 스왑한다.

6) 스레싱
- 페이징 기법에서 페이징 폴트가 빈번하여 성능 및 속도가 저하되는 현상

→ 이 메모리 단편화에 대한 해결방법이 페이징 기법
1) 페이징 기법 2) 세그멘테이션 기법 3) 메모리 풀 기법

3. TLP,DLP,ILP

▣ TLP, DLP, ILP

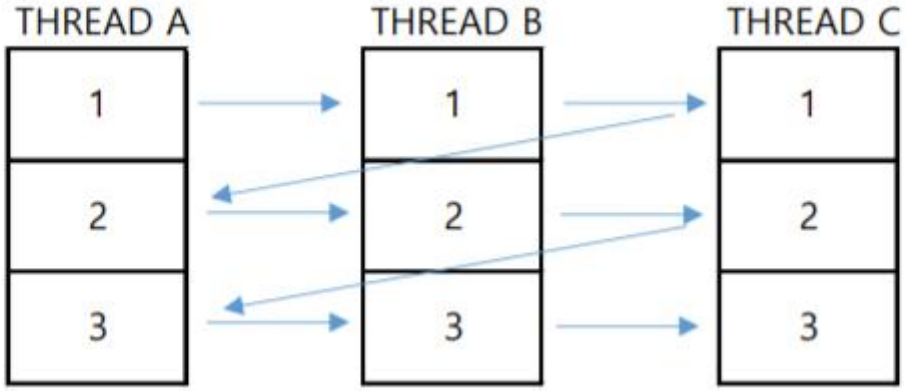
2020.08.24

1. TLP

스레드를 여러개만들어 프로세스를 병렬 처리하는 것.

멀티스레드는 멀티코어에서만 성능을 발휘한다. 아무리 멀티스레드로 프로그램을 작성해도 멀티코어 환경이 아니면 제 역할을 하지 못한다.

→ but os 스케쥴링 차원에서 스레드를 분할처리하여 병렬로 처리되는것처럼 할 수 있다.



(싱글코어에서의 스케쥴링 병렬처리)

3. TLP,DLP,ILP

2. ILP

비순차 실행에서 사용되는 기술, 순차적으로 처리하는게 아닌

명령어 간의 상관관계를 따져서 병렬처리가 가능하게 하는 기술

```
int main(void)
{
    .....                          엽과 같은 코드가 있을때
    x = 3;          ---a           acd를 1cycle에 처리하고 2cycle에 y=x+5를
    y = x+ 5;       ---b           처리하게 함.
    a = 0.01;       ---c
    b= 10           ---d
}
```

CPU차원에서 뿐만이 아니라 COMPILER차원에서 수행할 수 있다.

3. TLP,DLP,ILP

3. DLP

- DATA를 한 INSTRUCTION에 처리하는것을 이야기 한다.

0	1	2	3	4	5	6	7	8	9	10	11

위와같은 12개의 데이터를 더한다고 가정해보자
한가지 데이터를 읽어드리는데에 T라는 시간이 걸린다 가정하면
데이터를 읽는 시간은 12*T가 소요된다.(덧셈과정 별도)

0	1	2	3	4	5	6	7	8	9	10	11
CORE1			CORE2			CORE3			CORE4		

이때 이를 4코어 프로세서에서 DLP처리하게 되면 시간은 3T가 소요되게 된다.

4. Flynn's Taxonomy

	single instruction	multiple instruction
single data	sisd	misd
multiple data	simd	mimd

MISD, MIMD → ILP
SIMD,MIMD → DLP

4. INLINE

▣ inline

2020.08.24

1. 사용법
- 함수 선언문 앞에 inline 키워드 사용 ex) inline sum(int a, int b);

2. 이점
- 별도의 함수 호출이 필요없다. 함수문이 복사된것과 같다. 속도에서의 이점이 존재한다.

3. 단점
- 과도하게 사용시 오히려 성능이 저하 될 수 있다.
- 코드의 크기가 커진다.
- 페이징크기를 초과하여(기본4k) 추가적인 페이징을 유발 → 성능감소

5. C FUNCTION 모듈화

▣ C에서의 함수 사용

2020.08.24

1. 명제
- C 코드에서 코드를 함수로 잘게 쪼개는 것이 좋다.

2. 이점
- 생산성 직관성 면에서 좋다.

3. 단점
- 펌웨어에서의 잦은 함수호출은 클록수가 빠르지 않은 MCU특성상 성능저하로 이어질 수 있다.
  범용 OS의 경우 신경안써도 된다. 그냥 무조건 잘게 쪼개라.

6. OBJDUMP

▣ OBJDUMP

2020.08.24

1. 목적
- 실제 어셈블리어의 기계어 확인 가능

2. 사용법
- gcc -g *.c → objdump -d a.out

3. 사용예시
  - 소스코드

```
int sum(int num1, int num2);

int main(void)
{
        int a = 3;
        int b = 5;
        sum(a,b);
        return 0;

}
```

6. OBJDUMP

- GDB MAIN

```
0x0000000000001129 <+0>:     endbr64
0x000000000000112d <+4>:     push   %rbp
0x000000000000112e <+5>:     mov    %rsp,%rbp
0x0000000000001131 <+8>:     sub    $0x10,%rsp
0x0000000000001135 <+12>:    movl   $0x3,-0x8(%rbp)
0x000000000000113c <+19>:    movl   $0x5,-0x4(%rbp)
0x0000000000001143 <+26>:    mov    -0x4(%rbp),%edx
0x0000000000001146 <+29>:    mov    -0x8(%rbp),%eax
0x0000000000001149 <+32>:    mov    %edx,%esi
0x000000000000114b <+34>:    mov    %eax,%edi
0x000000000000114d <+36>:    callq  0x1159 <sum>
0x0000000000001152 <+41>:    mov    $0x0,%eax
0x0000000000001157 <+46>:    leaveq
0x0000000000001158 <+47>:    retq
```

- GDB SUM

```
0x0000000000001159 <+0>:     endbr64
0x000000000000115d <+4>:     push   %rbp
0x000000000000115e <+5>:     mov    %rsp,%rbp
0x0000000000001161 <+8>:     mov    %edi,-0x14(%rbp)
0x0000000000001164 <+11>:    mov    %esi,-0x18(%rbp)
0x0000000000001167 <+14>:    mov    -0x14(%rbp),%edx
0x000000000000116a <+17>:    mov    -0x18(%rbp),%eax
0x000000000000116d <+20>:    add    %edx,%eax
0x000000000000116f <+22>:    mov    %eax,-0x4(%rbp)
0x0000000000001172 <+25>:    mov    -0x4(%rbp),%eax
0x0000000000001175 <+28>:    pop    %rbp
0x0000000000001176 <+29>:    retq
```

6. OBJDUMP

-OBJDUMP

```
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       48 83 ec 10             sub    $0x10,%rsp
    1135:       c7 45 f8 03 00 00 00    movl   $0x3,-0x8(%rbp)
    113c:       c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
    1143:       8b 55 fc                mov    -0x4(%rbp),%edx
    1146:       8b 45 f8                mov    -0x8(%rbp),%eax
    1149:       89 d6                   mov    %edx,%esi
    114b:       89 c7                   mov    %eax,%edi
    114d:       e8 07 00 00 00          callq  1159 <sum>
    1152:       b8 00 00 00 00          mov    $0x0,%eax
    1157:       c9                      leaveq
    1158:       c3                      retq

0000000000001159 <sum>:
    1159:       f3 0f 1e fa             endbr64
    115d:       55                      push   %rbp
    115e:       48 89 e5                mov    %rsp,%rbp
    1161:       89 7d ec                mov    %edi,-0x14(%rbp)
    1164:       89 75 e8                mov    %esi,-0x18(%rbp)
    1167:       8b 55 ec                mov    -0x14(%rbp),%edx
    116a:       8b 45 e8                mov    -0x18(%rbp),%eax
    116d:       01 d0                   add    %edx,%eax
    116f:       89 45 fc                mov    %eax,-0x4(%rbp)
    1172:       8b 45 fc                mov    -0x4(%rbp),%eax
    1175:       5d                      pop    %rbp
    1176:       c3                      retq
    1177:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
    117e:       00 00
```

6. OBJDUMP

- 어셈블리 명령어 옆에 16진수가 실제 기계어를 나타냄.

- 기계어가 똑같은 어셈명령에 여러 개인 이유
RISC 아키텍쳐는 메모리 TO 메모리 연산 불가 → LOAD 이후 STORE 를 통해 메모리 제어
CISC 메모리 TO 메모리 연산 가능 → 그러므로 mov가 처리할 수 있는 케이스 많아짐
레지스터 to 메모리, 메모리 to 레지스터, 레지스터 to 레지스터
하나하나에 대응하는 기계어임

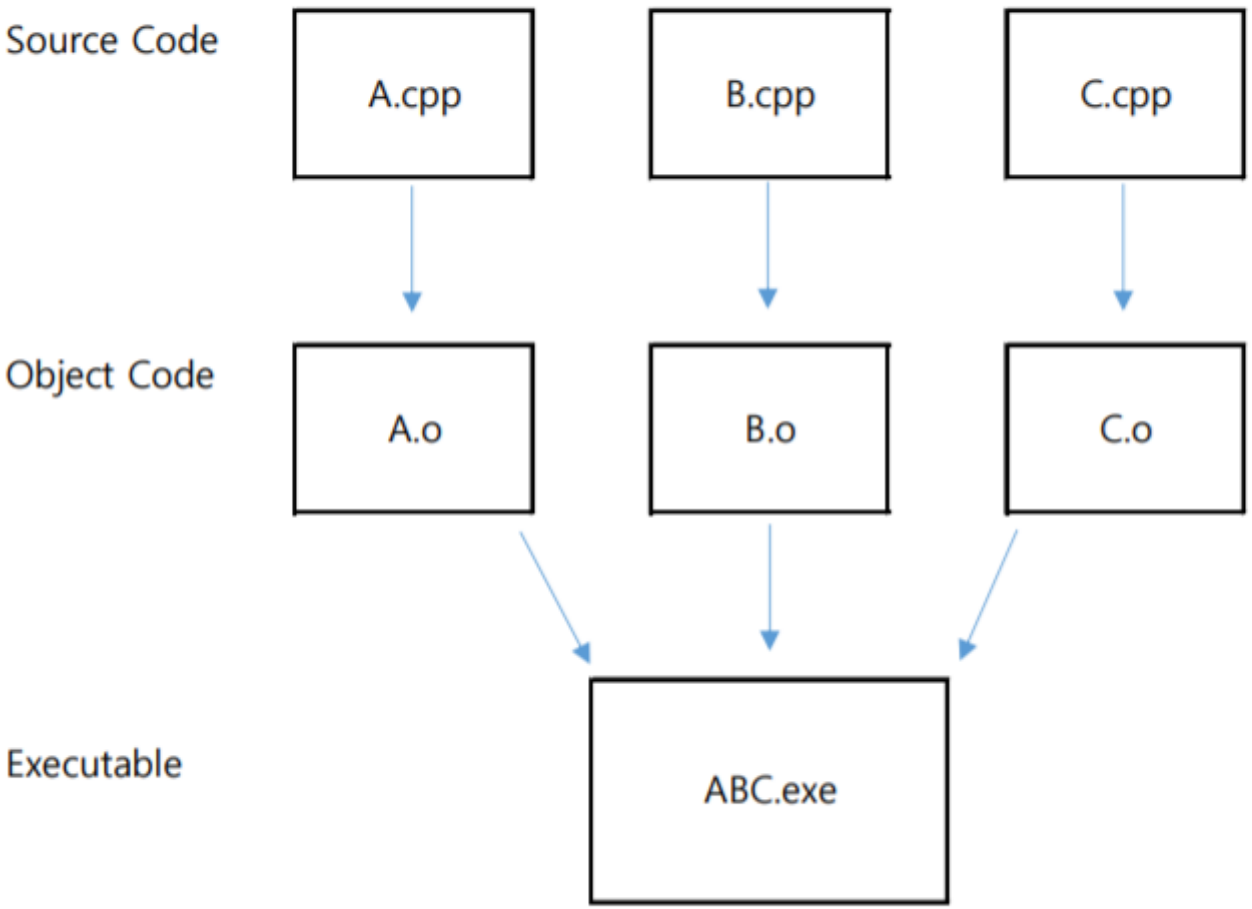
맨 왼쪽 숫자의 경우 메로리 상대주소를 가르키는 섹션 오프셋을 의미한다.

7. LINK

▣ LINKING, LINKER

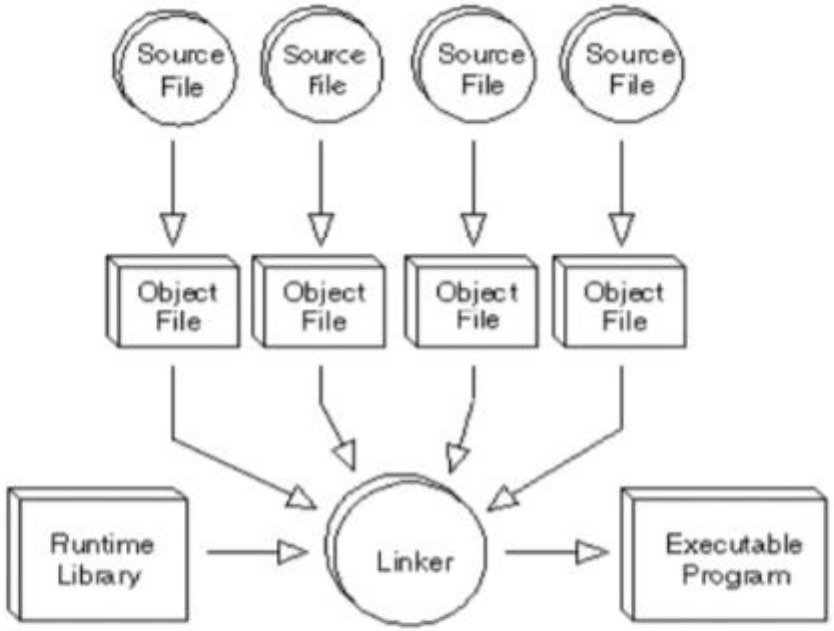
2020.08.24

1. LINKING



소스코드와 실행파일 중간에 컴파일을 하면 object파일이 생성됨

7. LINK



오브젝트 파일들과 라이브러리를 링커가 링킹해서 실행파일을 만듦

2. Static Linking vs Dynamic Linking

1) Static Linking

- 컴파일러가 컴파일시 실행파일에 직접 실행라이브러리를 복사함.
  별도의 컴파일 필요 없음. 하지만 메모리가 커지게 된다.

2) Dynamic Linking

- DLL의 해당 함수를 호출하면 메모리에 존재하는 그 함수로 JUMP하여 실행
  메모리 요구사항이 적으나, 약간의 OVERHEAD 발생