파이썬 - HW5

임베디드스쿨1기
Lv1과정
2020. 08. 24
박성환

1-1. 병렬처리-TLP(Task Function Parallelism)

1. 정의
- 각 여러 코어에서 서로 다른 작업을 동시에 실행하는 것

2. 특징
- 여러 코어에서 병렬로 작동하지만 각각 고유한 작업을 수행함
- 하나의 응용 프로그램내의 여러 스레드 또는 명령 시퀀스를 여러 프로세서에 분산하여 동시에 실행할 수 있음
- 작업 병렬 처리는 프로세스 또는 스레드에 의해 동시에 수행되는 작업을 서로 다른 프로세서에 분산하는데 중점을 둠
- 데이터의 서로 다른 구성 요소에서 동일한 작업을 실행하는 데이터 병렬 처리(DLP)와 달리 동일한 데이터에서 동시에 여러task를 실행함**(DLP와 차이)**
- 주로 데이터 베이스와 같은 상용 서버를 위해 작성된 응용 프로그램에서 발견되며 한 번에 많은 스레드를 실행함으로써 워크로드로 인해 발생할 수 있는 많은 양의 I/O 및 메모리 시스템 지연 시간을 허용할 수 있음
- 한 스레드가 메모리 또는 디스크 액세스를 하는 동안 다른 스레드가 유용한 작업을 수행

3. 제한
- Limited in practice by communication/synchronization overheads and by algorithm chara
  (통신이나 동기화 사용되는 경우 처리시간이라던지 타이밍이 중요하기 때문에 Task로 시간을 분할하여 작업하는 것은 어려움을 줄 수 있어서 이러한 경우는 해당 Task들을 최상위 우선순위로 처리하거나 DLP로 처리하거나 하는 것인가?)

  하나의 데이터의 접근과 다른 데이터의 접근의 정확한 의미를 모르겠음

```
program:
...
if CPU = "a" then
    do task "A"
else if CPU="b" then
    do task "B"
end if
...
end program
```

1-2. 병렬처리-DLP(Data Level Parallelism)

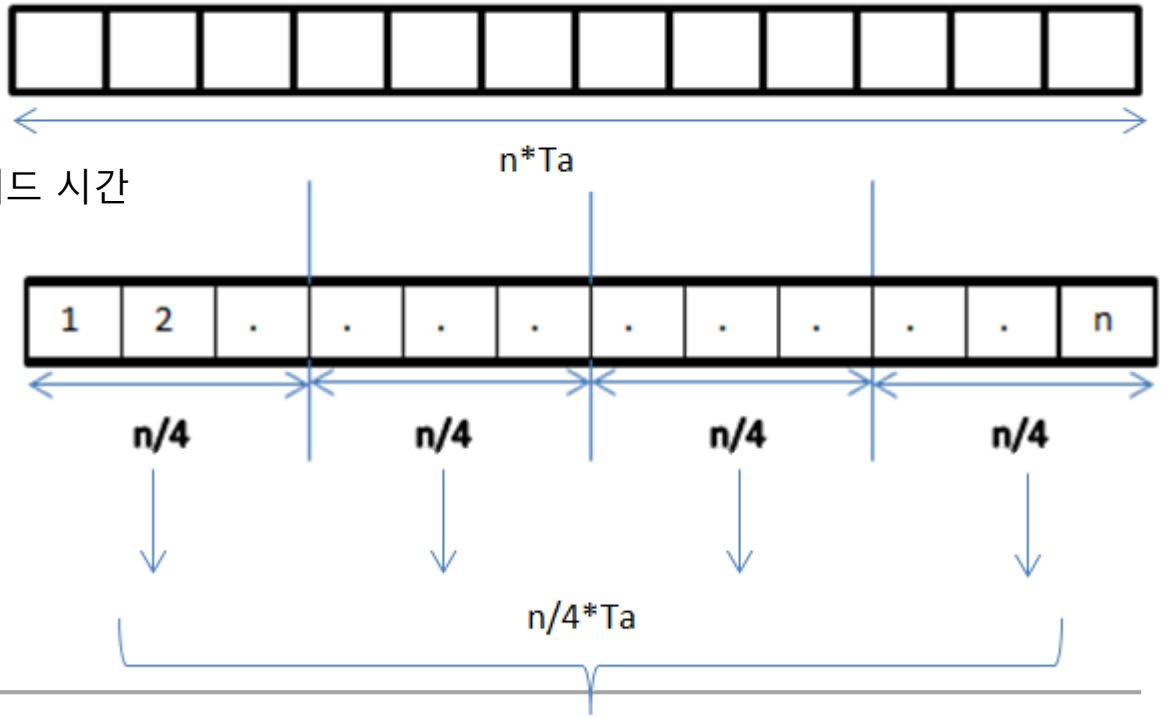
1. 정의
- 각 여러 코어에서 한 작업을 동시에 실행하는 것

2. 특징

- 데이터 병렬 처리는 데이터를 병렬로 처리하도록 데이터를 분산하는데 중점을 둠
- 단일 명령을 사용하여 동시에 여러 데이터에 대해 작업함으로써 연산의 성능 촉진 추구
- 각 요소를 병렬로 작업하여 배열 및 행렬과 같은 일반 데이터 구조에 적용 가능
- 큰 단위의 행렬, 배열연산의 경우 동일 데이터를 블록화하여 각각의 프로세서에서 동시 작업함으로써 시간을 향상 시키는 것

3. 예시
- n요소의 배열 단일 덧셈 연산시간 Ta라 하면
  단일 프로세서 걸리는 시간 : n * Ta
  4개 프로세서 걸리는 시간 : (n/4) * Ta + 병합 오버 헤드 시간
  거의 4배 속도 향상

4. 제한
- 비정규적인 데이터 조작 패턴이 아닌 경우나
  메모리 bandwitdh에 의해 제한됨



1-3. 병렬처리 DLP vs TLP

- **핵심 차이**

Data Parallelisms	Task Parallelisms
1. Same task are performed on different subsets of same data.	1. Different task are performed on the same or different data.
2. Synchronous computation is performed.	2. Asynchronous computation is performed.
3. As there is only one execution thread operating on all sets of data, so the speedup is more.	3. As each processor will execute a different thread or process on the same or different set of data, so speedup is less.
4. Amount of parallelization is proportional to the input size.	4. Amount of parallelization is proportional to the number of independent tasks is performed.
5. It is designed for optimum load balance on multiprocessor system.	5. Here, load balancing depends upon on the e availability of the hardware and scheduling algorithms like static and dynamic scheduling.

1-4. 병렬처리-ILP(Instruction Level Parallelism)

1. 정의

- 얼마나 많은 operation(명령)이 동시에 실행할 수 있는가 재는 방법
- 명령어 수준에서의 병렬처리

2. 특징

- 싱글 스레드에서도 명령어 의존성을 분석하여 병렬성을 구현
- 컴파일러 및 프로세서에서 가능한 많은 ILP를 식별하고 활용하도록 하고 있음.

3. 예시

```
int main()
{

    int  x, y, a, b, c, d;
    int  myarray[4];

    ...
    x = myarray[4];     -- (가)
    y = x + 10;         -- (나)
    a = b - 4;          -- (다)
    c = d * 7;          -- (라)

    ...
}
```

1 cycle 에서 실행하는 명령: (가) (다) (라)

2 cycle 에서 실행하는 명령: (나)

4cycle을 2cycle만에 처리가 가능해짐, 이것이 ILP

결국, 지난번 학습한 슈퍼 스칼라 방식의 Pipelining이 ILP인 것인가?

cf) 슈퍼스칼라는 Data Dependancy만 없으면 fetch과정도 PipeLining 각 단계가 동시에 여러 개 가능함

2-1. inline함수

1. 정의
- C/C++에서 함수 호출시 별도로 분리된 위치의 레이블로 점프하여 실행되는 일반 함수와는 달리 호출 부분을 함수 전체 코드로 치환하여 컴파일함.

2. 사용범
- 함수 선언시 앞에 inline만 붙여주면 되는데 이는 컴파일러에게 주는 권고일뿐 항상 인라이닝을 보장하지 않음.
- 함수를 알고 있어야 하지만 컴파일러가 적절히 인라인화 하므로 inline 키워드를 사용할 필요는 없음.

3. 특징
- 최신의 컴파일러들은 대부분 최적화 기능이 잘 되어 있기에 inline이 붙어 있지 않더라도 인라이닝을 하는게 이득이 된다고 판단되면, 알아서 inline 처리를 함.

- 반대로 비용 분석을 통해 인라이닝이 손해라고 판단하면 코드에서 아무리 인라이닝을 붙여도 인라이닝을 포기함.

4. 권고가 아닌 강요하기/금지하기
- 컴파일러에게 특정 함수의 인라이닝을 강요하는 기능은 원래 없으나 방언으로 그러한 기능을 컴파일러들이 제공
  Gcc 의 경우는 아래와 같음
  강요하기: __attribute__((always_inline))를 함수 선언시 붙여줌
  금지하기: __attribute__((noinline))를 함수 선언시 붙여줌

5. 인라인 장점/단점
- 무차별적으로 쓸 경우 중복되는 부분이 컴파일러 결과 바이너리에 산재되어 크기가 커질뿐더러, 브랜칭 예측률 및 Instructio 캐시 적중률을 낮추는 효과가 있다.
- 즉, 복잡한 함수를 여러군데 인라이닝하면 프로그램 크기는 커질대로 커지고 성능은 오히려 감소하는 현상이 일어남

2-1. inline함수

5. 예시

- 단순히 함수 앞에 inline 만 붙여주면 됨
- gcc의 경우에서 강요하고 싶을 때는 __attribute__((always_inline))을 붙이면 됨

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int main()
{
    int num1;

    num1 = add(10, 20);

    printf("%d\n", num1);
}
```

호출

```
#include <stdio.h>

inline int add(int a, int b)
{
    return a + b;
}

int main()
{
    int num1;

    num1 = inline int add(10, 20)
    {
        return 10 + 20;
    }

    printf("%d\n", num1);
}
```

컴파일러가
함수를 복제하여 넣어줌

2-2. Macro vs inline 차이

1. 예시

아래와 같은 경우 인라인 함수와 매크로 (함수)의 차이는 다음과 같음

1) 인라인 함수는 타입체크를 해서 인자를 int형 정수로만 받지만, 매크로는 그런 것이 없이 무조건 치환함
2) 두 함수를 처리하는 주체가 다르다.
   - 전처리기(preprocessor)가 일괄적으로 치환하는 것이 **매크로**
   - 컴파일러가 일반 함수처럼 문법 검사 및 타입 체크 등을 하는 것인 **인라인 함수**
3) 매크로는 전처리기가 무조건 치환하기에 무시할 수 없지만, 인라인 함수의 경우는 진보된 컴파일러가 판단하여 교체해 넣는 것이 오히려 손해라고 판단되면, 일반적인 함수로서 작동할 수도 있음. 즉, 기본적으로 inline구문은 인라인이 선호될 뿐 강제는 아님
4) 매크로 함수는 과거에는 자주 쓰였지만 현재는 최대한 지양해야할 기능으로 여김. 타입 체크를 무시하고, 유지보수를 어렵게 만들기 때문

즉, inline 함수는 문법검사 및 타입체크를 하지만 매크로는 무조건 치환한다.

```
// 매크로 :
#define MUL(x,y) ((x)*(y))

// 인라인 :
inline int MUL(int x,int y) {
    return x*y;
}
```

참고. __attribute__(())

1. 정의

- __attribute__()는 GCC올바르게 동작하도록 설정가능

2. 예시

- 구조체가 5Byte 사이즈임에도 불구하고 32bit 머신인 경우 4Byte단위컴파일러가 제공하는 기능으로 (())안에 옵션을 보고 유닛이 최적화되어 있기 때문에(속도도 가장 빠름) 8Byte를 할당함.
- 이런 의미 없는 Padding Value를 삽입하고 싶지 않은경우에는 __attribute__((Packed))를 붙여줌. 즉, 5Byte만 할당

```
typedef struct{
    char a;
    int  b;
}__attribute__((packed))
```

참고. Preprocessor

1. 정의
- 프로그램을 컴파일할 때 컴파일 직전에 실행되는 별도의 프로그램
- 전처리가 실행되면 각 코드 파일에서 지시자(directives)를 찾음.
- 지시자는 #으로 시작해서 줄 바꿈으로 끝나는 코드

2. 역할
- 컴파일러가 실행되기 직전에 단순히 텍스트를 조작하는 치환 역할을 하기도 하고(#define, #include등), 디버깅에도 도움을 주며 헤더 파일의 중복 포함도 방지해주는 기능을 가짐

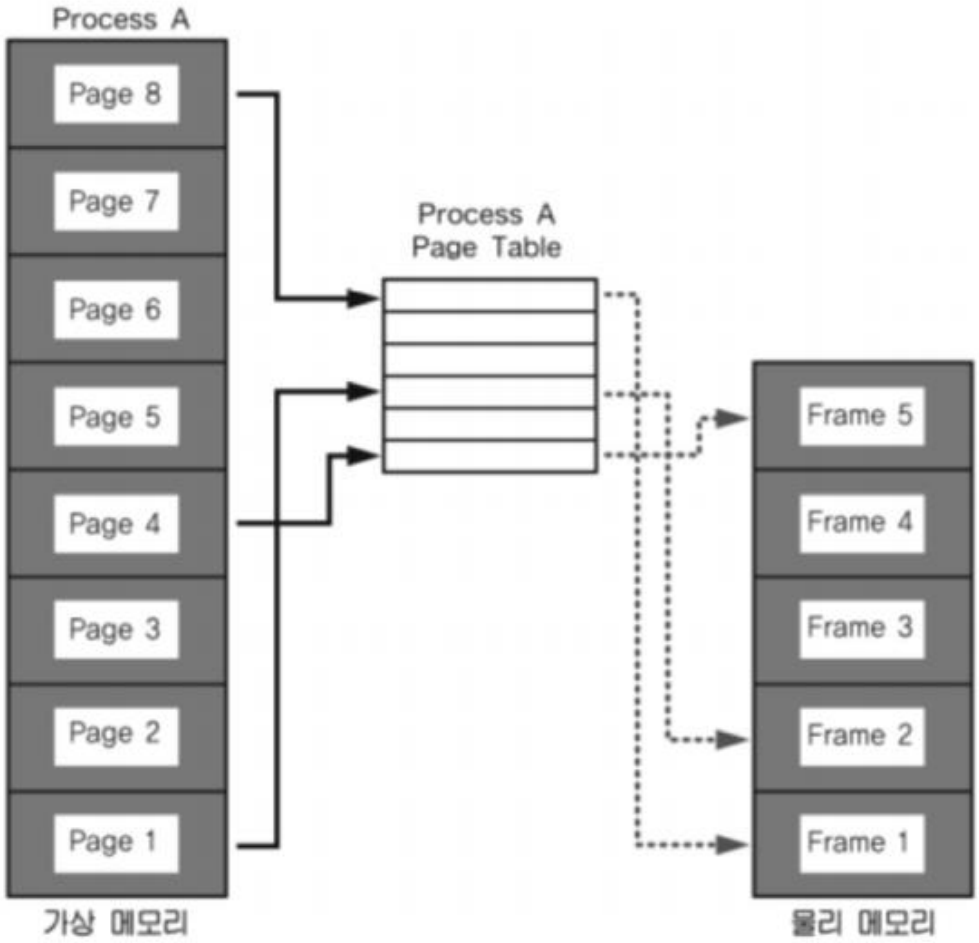
3. 예시
- #include 지시자 를 사용하면 전처리기는 include된 파일의 내용을 지시자의 위치에 복사함
- #define 지시자를 사용하면 macro를 만들어서 입력을 출력으로 변환함

3-1. Paging

1. Paging

- 크기가 동일한 페이지로 가상 주소 공간과 이에 매칭하는 물리 주소 공간을 관리
- 페이지 번호를 기반으로 가상주소/ 물리주소 매핑 정보를 기록/사용

리눅스 – 4KB로 Paging(단일)
Intelx86 – 4KB, 2MB, 1GB 지원.

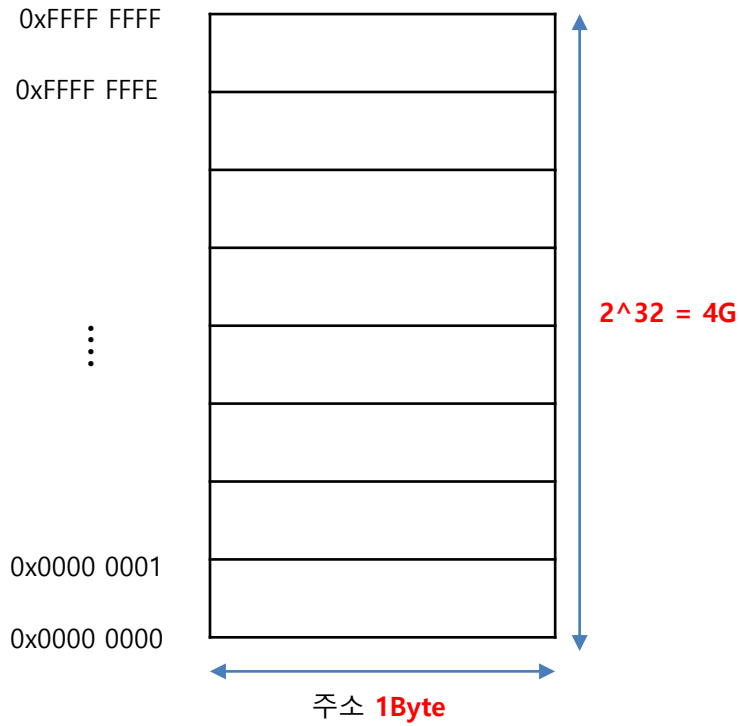


3-1. Paging

2. Page구조

- Page : 고정된 크기의 block(4KB – 리눅스)
- Paging System
  - 가상주소 v = (p, d)
    p : 가상 메모리 페이지 번호
    d : p안에서 참조하는 위치

3. 페이지 크기가 4KB 일 때
가상주소의 0bit – 11bit 가 변위(d)를 나타냄(2^12 = 2^2 * 2^10 = 4 * 1KB)
12bit 이상이 페이지 번호가 될 수 있음



3-2. Demand on Paging

1. 탄생배경
- 운영체제가 Paging을 통해서 가상 메모리를 물리 메모리에 맵핑함. 통째로 맵핑하면 분량이 많고 짧은시간에 모두 맵핑이 불가능하고 Memory Hierarchy 관점에서 봤을 때 우선 디스크에서 내용을 읽고 메모리에 로드해야하는데 굉장히 성능저하를 느낄 수 있음.

2. 정의
- 프로세스가 실제 필요로 하는 부분만 메모리로 올리는 기법
- 즉, 프로세스 모든 데이터를 메모리로 적재하지 않고, 실행 중 필요한 시점에서만 메모리로 적재함
- 더 이상 필요하지 않은 페이지는 다시 저장매체에 저장(페이지 교체 알고리즘 사용)
- 메모리 관리 메커니즘(MMU)을 사용해서 여러 프로세스가 시스템의 메모리를 효율적으로 공유할 수 있도록 하는 기술

3. 프로그램이 필요로 하는 모든 메모리가 Physical memory에 있어야 하는가?
그렇지 않음. 프로그램은 참조의 지역성(Locality of Reference)을 가지고 있기 때문임.

locality of reference 참조 지역성

동일한 값 또는 해당 값에 관계뛴 스토리지 위치가 자주 액세스 되는 특성.
참조지역성의 3가지 기본형 : 시간, 공간, 순차 지역성

1) 공간(spatial) 지역성 : 특성 클러스터들의 기억 장소들에 대해 참조가 집중적으로 이루어지는 경향으로, 참조된 메모리의 근처의 메모리를 참조.

2) 시간(temporal) 지역성 : 최근 사용되었던 기옥 장소들이 집중적으로 액세스 되는 경향으로, 참조했던 메모리는 빠른 시간에 다시 참조될 확률이 높다.

3) 순차(sequential) 지역성 : 데이터가 순차적 o로 액세스 되는 경향. 프로그램 내의 명령어가 순차적으로 구성되어있다는 것이 대표적인 경우.

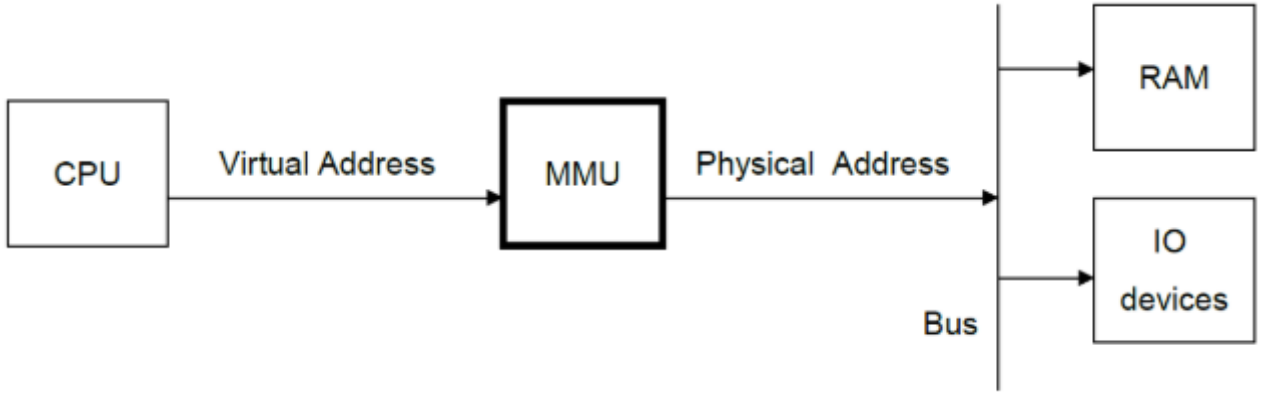
3-2. Demand on Paging

4. Logical Address Space에 있는 Page들 중에서 Physical Memory에 있지 않은 것들은 어디에 있어야 하나?
- Swap Device에 있어야 함

5. Swap Device란
- Physical Memory에 저장되지 못한 Page들을 저장하는 디스크의 공간으로 File에 비해 운영체제가 직접 빠르게 접근할 수 있음(logically swap devic라고도 말함)

6. 목적
- Physical Memory와 Swap Device 사이의 데이터 전송이 최소한으로 발생하도록 함

7. Physical Address가 Ram에만 있는 것이 아닌 아래 그림처럼 I/O Device 즉, Swap Device로 가는 Address도 존재



참고. 페이지 폴트(Page Fault)

1. 페이지 폴트

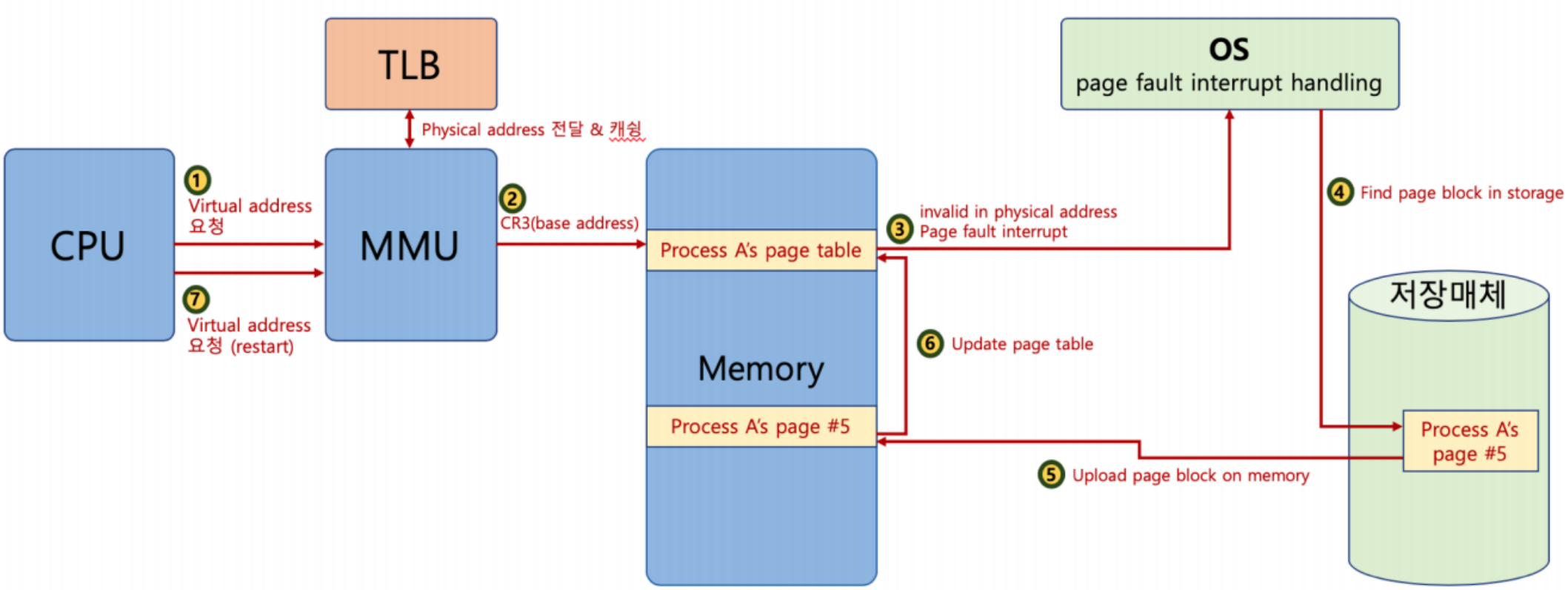
- 어떤 페이지가 실제 물리 메모리에 없을 때 일어나는 인터럽트
- 운영체제가 Page Fault가 일어나면 , 해당 페이지를 물리 메모리에 올림

***페이지 폴트가 자주 일어나면?**

- 실행되기 전에, 해당 페이지를 물리 메모리에 올려야함 -> **시간 소모**

***페이지 폴트가 안 일어나게 하려면?**

- 향후 실행/ 참조될 코드/ 데이터를 미리 물리 메모리에 올리면 됨 -> **신의 영역 말이 안됨**



3-1. Static Link Library

1. 라이브러리(Library)
표준화할 수 있는 함수를 미리 만들어서 모아 놓은 것으로 한 번 구축해 놓기만 하면 다시 만들 필요없이 불러서 사용할 수 있으므로 개발 속도도 빨라지고 신뢰성 확보도 가능

이러한 라이브러리를 언제 메인 프로그램에 연결하느냐에 따라서 Static Link와 Dynamic Link로 나뉘며 DLL은 이 중 후자를 뜻함.

2. Static Link Library
정적 링크라고 하며 컴파일을 하면 링커가 프로그램이 필요로 하는 부분을 라이브러리에서 찾아 실행 파일에다가 바로 복사

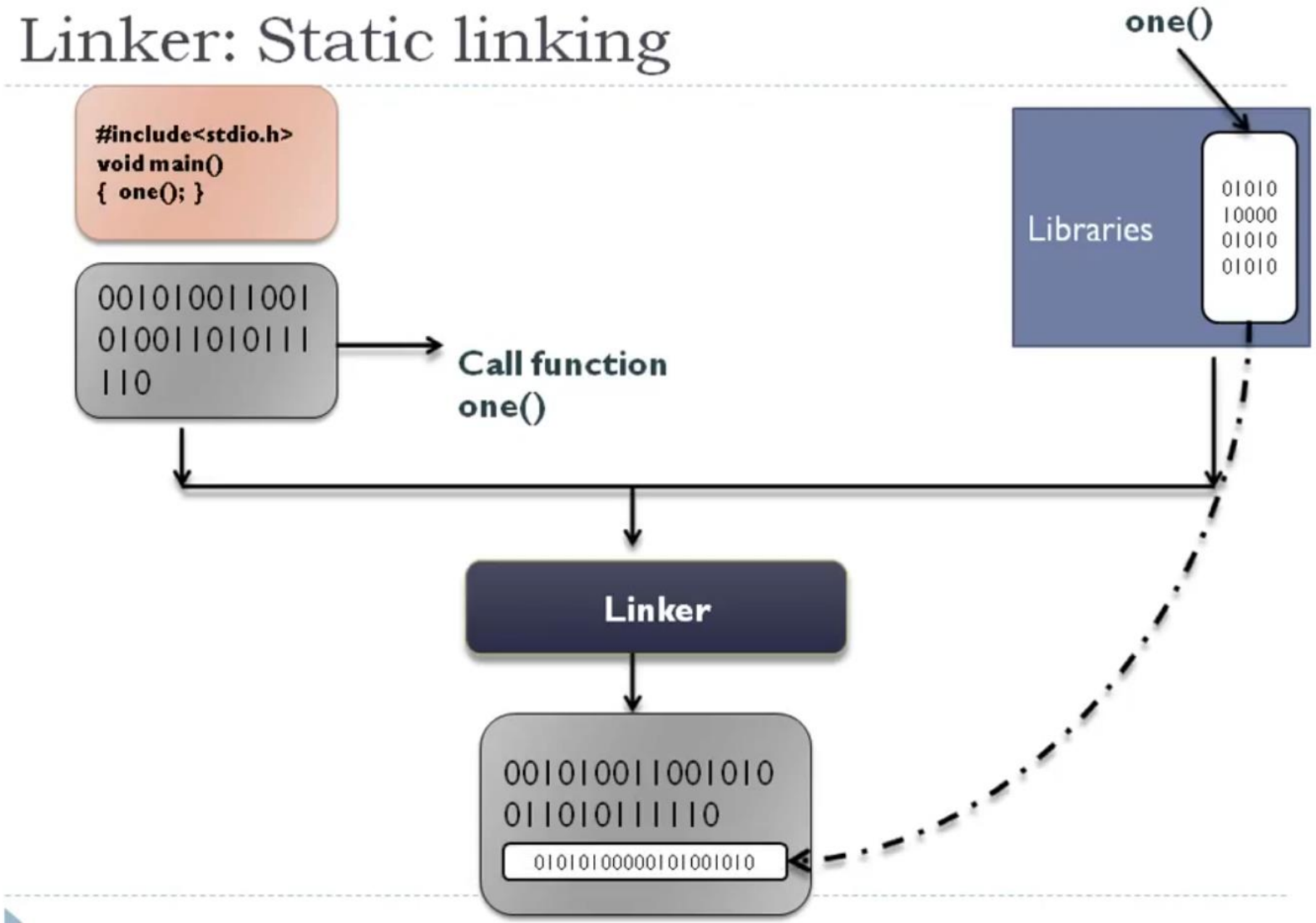
2-1 정적링크의 이점
실행 파일에 다 들어가 있기 때문에 라이브러리가 따로 필요 없음.

미리 컴파일되어 있기 때문에 **컴파일 시간도 단축됨**

직접 구현한 코드를 라이브러리화 시켜서 **기술 유출 방지로** 사용

2-2 정적링크의 단점
메모리를 어마하게 잡아먹음(실행 파일 내에 라이브러리 코드가 저장되기 때문에)

3-1. Static Link Library



포기하면 얻는 건 아무것도 없다.

3-2. Dynamic Link Library(DLL, Shared Library)

1. 배경

- 정적 링크를 써서 했더니 메모리에 쓸데 없이 똑같은게 너무 많이 올라가니 자주 많이 쓰이는 라이브러리는 메모리에 하나만 올려서 사용하자는 것에서 출발

- printf와 같이 프로그램마다 자주 사용하는 함수를 실행 프로그램에 포함시킬 경우(정적링크 방식) 프로그램의 덩치가 커지고 외부 라이브러리가 업그레이드 됐을 경우 이를 사용하는 프로그램을 다시 컴파일해야하는 부담이 있음

- 라이브러리를 Shared Library라는 형식으로 만들어 놓고 컴파일 시점에 사용할 라이브러리를 연결만 하는 방법을 사용

2**. 정의**

- 동적 링크라고 하며 실행 파일에서 해당 라이브러리의 기능을 사용 시에만, 라이브러리 파일을 참조하여 기능을 호출함

- 정적 링크와는 다르게 컴파일 시점에 실행 파일에 함수를 복사하지 않고, 함수의 위치정보만 갖고 그 함수를 호출할 수 있게 함

- 현재 Linux에서 프로그램 개발시 별다른 옵션을 주지 않으면 Linking을 동적으로 Linking함

- 프로그램이 실행되는 Runtime시 링크(정적 링크는 컴파일 타임에 링킹과정에서)

***원래 코드와 별도로 라이브러리가 존재**

리눅스 – Shared Library(~~~.so, ~~~.sa)
윈도우 – DLL (~~~.dll)

3-2. Dynamic Link Library(DLL, Shared Library)

3. DLL의 이점

1) 더 적은 리소스 사용

- 한 코드를 여러 프로그램이 동시에 사용하기 때문에 메모리가 절약됨
- 사용되는 디스크 공간을 줄일 수 있음(정적링크는 실행 파일에 라이브러리의 함수가 모두 포함되어 실행파일이 커짐)

2) 모듈식 아키텍처 활용

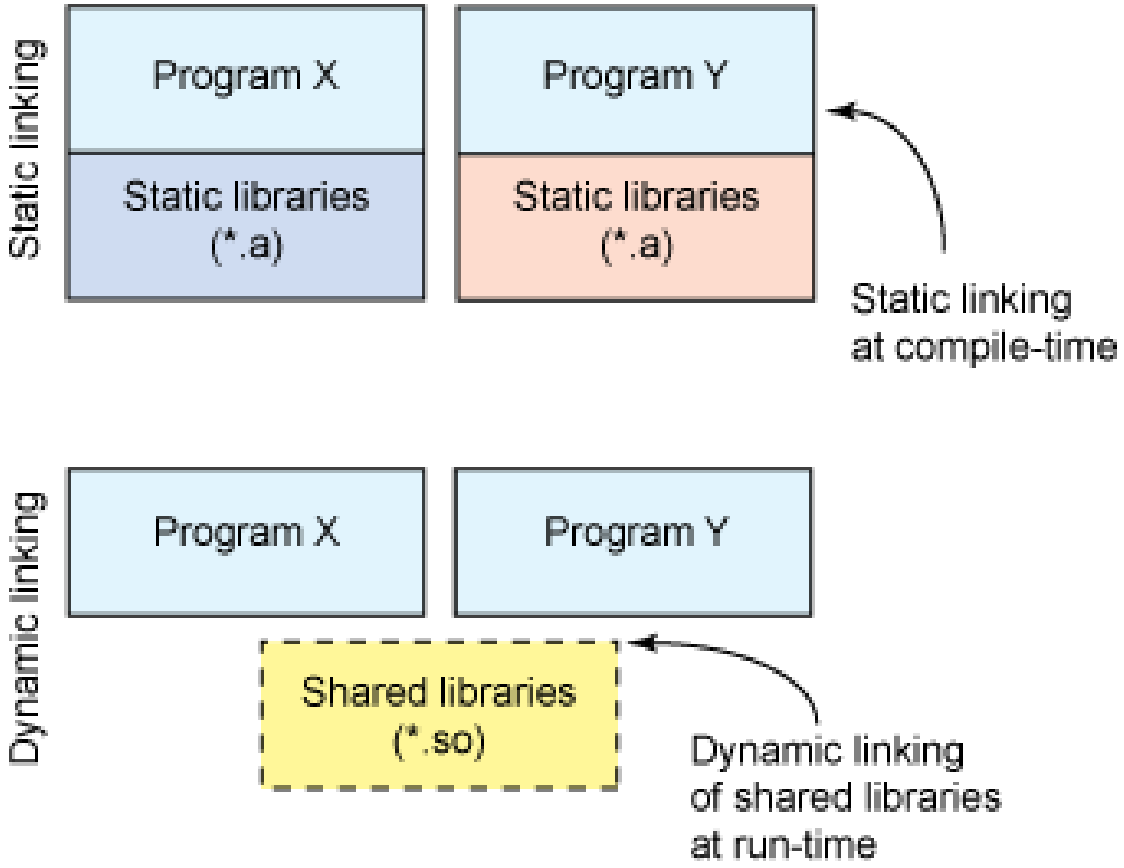
- 모듈식 프로그램을 효율적으로 개발가능
- 여러 언어 버전이 필요한 큰 프로그램이나 모듈식 아키텍처가 필요한 프로그램 개발용이

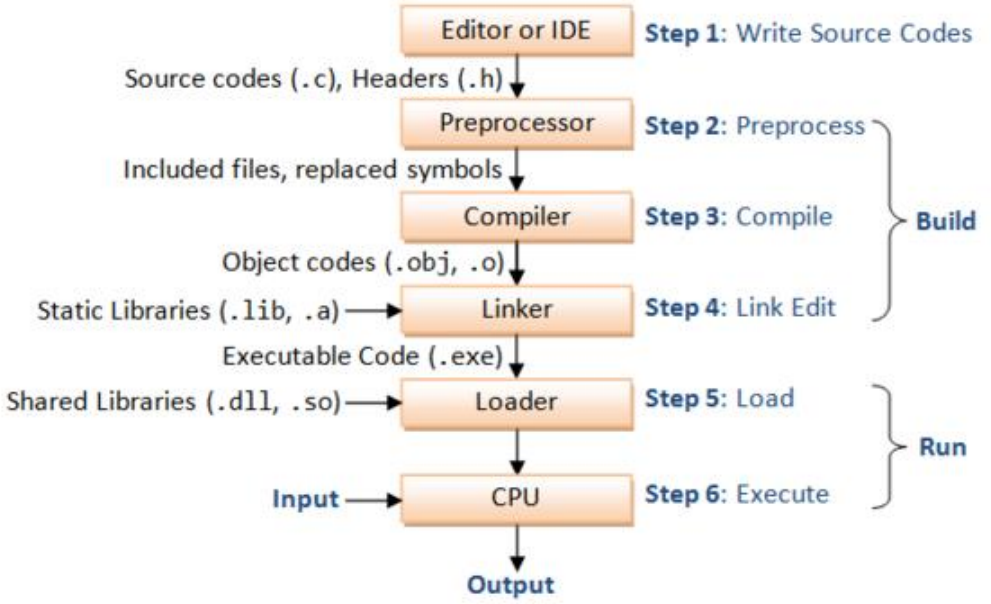
3) 손쉬운 배포와 설치

- DLL 내의 함수를 업데이트하거나 수정해야 하는 경우 DLL을 배포하고 설치할 때 프로그램을 DLL과 다시 연결하지 않아도 됨
- 여러 프로그램이 같은 DLL을 사용하는 경우에는 모든 프로그램에 업데이트나 수정 내용이 적용됨

4) 프로그래머들의 분담 작업이 용이하며 재사용성도 뛰어남

3-3. Static Library vs Dynamic Library





4. Load & Store Architecture

1. Load & Store Architecture

- Load(메모리 to 레지스터)명령과 Store(레지스터 to 메모리)명령만 메모리 액세스 가능하도록 설계된 구조

- 예를 들어 덧셈 연산을 하더라도 메모리끼리 바로 더하는 것이 안되고 Load를 통해 레지스터에 값들을 불러와서 더한 다음에 결과를 다시 레지스터에 저장하고 이를 Store를 통해 메모리에 저장하는 식의 구조

예) ADD 연산에서 피연산자가 하나는 레지스터 하나는 메모리에 있는 것은 절대 가능하지 않음

2. Register Memory Architecture

- 레지스터뿐만 아니라 메모리에서 연산을 수행할 수 있는 명령어 세트 구조

예) ADD 연산에서 모든 피연산자가 메모리나 레지스터에 있을 수 있음

Review)

- 일단 기본적인 RISC 아키텍처는 메모리 2 메모리 연산이 불가능하다.
  그렇기 때문에 Load 이후에 Store를 통해서 메모리를 제어한다.
  반면 CISC 아키텍처는 메모리 2 메모리 연산이 가능하다.
  그렇기 때문에 mov가 처리할 수 있는 케이스가 여러가지가 된다.
  레지스터 2 메모리, 메모리 2 레지스터, 레지스터 2 레지스터, 메모리 2 메모리

감사합니다.