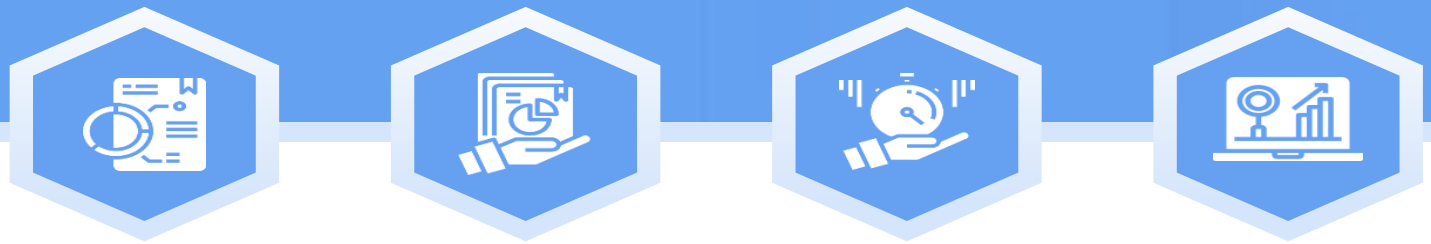
시스템 보안

#4 80x86 시스템 - 3

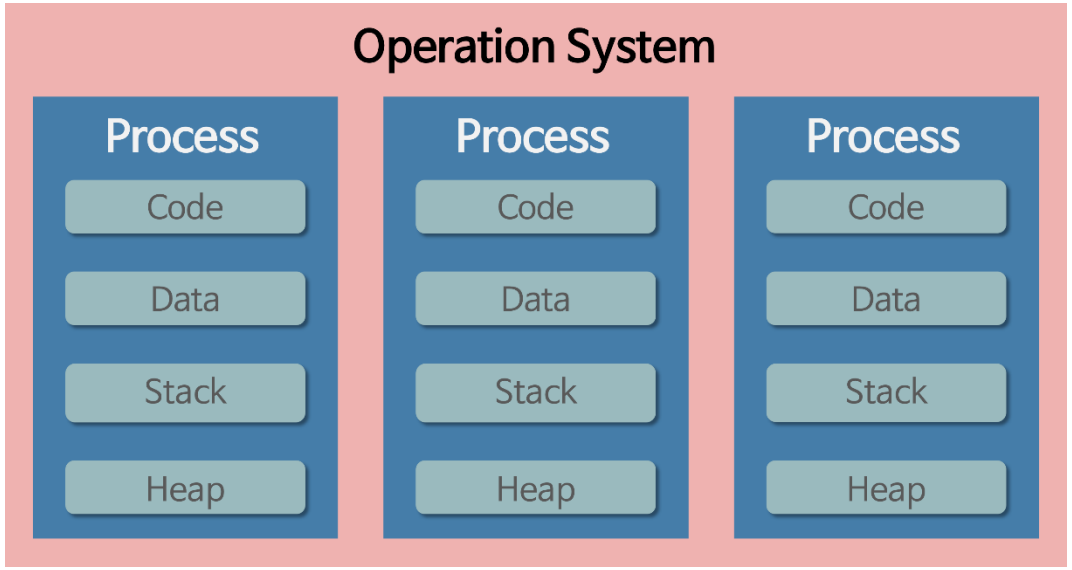


KOREA UNIVERSITY
SEJONG CAMPUS

프로세스(Process)와 스레드(Thread)

프로세스와 스레드

- **프로세스(Process) 란?**
  - 간단하게 말하면 실행중인 프로그램으로, 메모리에 올라와 실행되고 있는 프로그램의 인스턴스(독립적인 개체)
    운영체제로부터 시스템 자원을 할당 받는 작업의 단위
  - 프로세스는 사용 중인 파일, 데이터, 프로세서의 상태, 메모리 영역 주소 공간, 스레드 정보, 전역 데이터가 저장된 메모리 부분 등
    수 많은 자원을 포함하는 개념으로 종종 스케줄링의 대상이 되는 작업이라고 불리기도 함



- 할당 받는 시스템 자원의 예

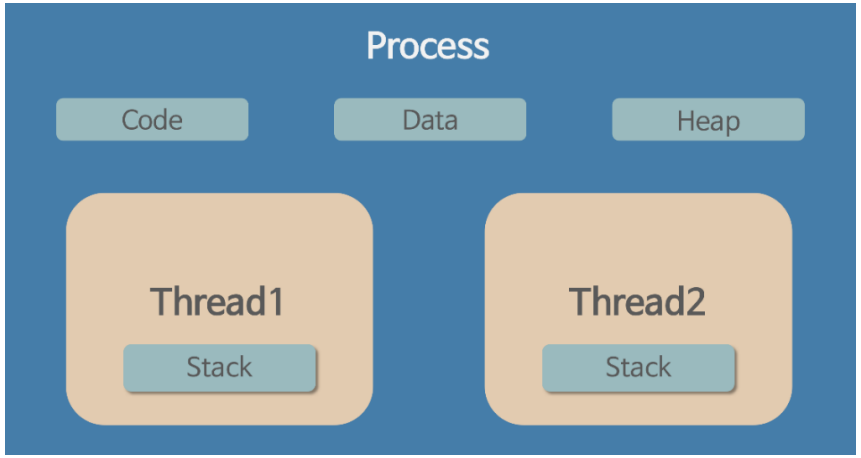
  - CPU 시간

  - 운영되기 위해 필요한 주소 공간

  - Code, Data, Stack, Heap의 구조로 되어 있는 독립된 메모리 영역

프로세스와 스레드

- **스레드(Thread) 란?**
  - 프로세스 내에서 실행되는 여러 흐름의 단위, 프로세스가 할당 받은 자원을 이용하는 실행 흐름의 단위로 프로세스에는 최소한 하나의 스레드가 존재
  - 하나의 프로세스에서 병렬적으로 여러 개의 작업을 처리하기 위해서는 각 작업을 독립적인 일의 단위인 스레드로 분리하여 실행시키는 멀티스레딩이 가능하도록 해야 함

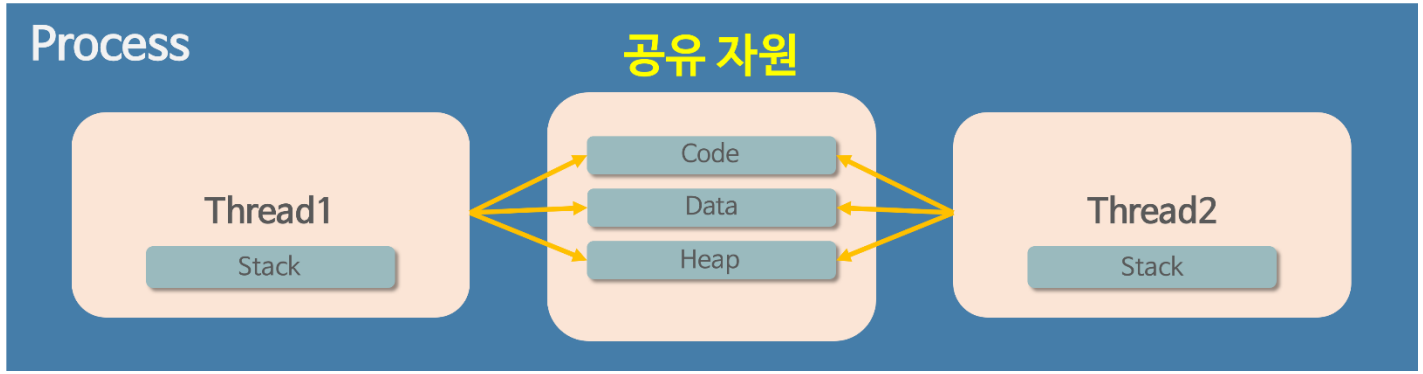


- 스레드는 프로세스 내에서 Stack만 따로 할당 받고
  Code, Data, Heap 영역은 공유

- 스택(Stack)을 독립적으로 할당하는 이유
  - 스택은 함수 호출 시 전달되는 인자, 되돌아갈 주소 값 및
    함수 내의 지역 변수 등을 저장하기 위해 사용되는 메모리 공간으로
    스택이 독립적이라는 것은 독립적인 함수 호출이 가능하다는 의미
  - 독립적인 실행 흐름의 추가를 위한 최소 조건이 독립된 스택을 제공 하는 것

프로세스와 스레드

- **멀티 프로세스 보다 멀티 스레드를 많이 사용하는 이유?**
  - 두 개의 프로세스는 완전히 독립되어 있기 때문에 컨텍스트 스위칭(Context Switching, 프로세스의 상태 정보를 저장하고 복원하는 작업)으로 인한 성능 저하가 발생하며, CPU 레지스터 교체 뿐만 아니라 RAM과 CPU 사이의 캐쉬 메모리에 대한 데이터까지 초기화 되므로 오버헤드가 큼
  - 스레드는 하나의 프로그램 내에서 여러 개의 실행 흐름을 두기 위한 모델로 프로세스처럼 완벽히 독립적인 구조가 아니고 스레드들 사이에는 공유하는 요소들이 있어서 컨텍스트 스위칭에 걸리는 오버헤드가 프로세스보다 적음
  - 스레드 간의 자원 공유는 전역 변수(데이터 세그먼트)를 이용하므로 함께 사용할 때 충돌이 발생할 수 있어서 동기화 문제를 고려하여 설계해야 함



- (실습) 파이썬에서의 병렬 처리 방법 (Thread)

```
import time
from threading import Thread

def work(start, end, result):
    total = 0
    for i in range(start, end):
        total += i
    result.append(total)
    return

if __name__ == "__main__":
    START, END = 0, 100000000
    result = list()
    th1 = Thread(target=work, args=(START, END, result))

    start_time = time.perf_counter()
    th1.start()
    th1.join()
    end_time = time.perf_counter()

    print(f"Result: {sum(result)}")
    print(f"Time: {end_time - start_time} second")
```

1개 Thread 이용

```
Result: 4999999950000000
Time: 2.9693838999992295 second
```

```
if __name__ == "__main__":
    START, END = 0, 100000000
    result = list()
    th1 = Thread(target=work, args=(START, int(END/2), result))
    th2 = Thread(target=work, args=(int(END / 2), END, result))

    start_time = time.perf_counter()
    th1.start()
    th2.start()
    th1.join()
    th2.join()
    end_time = time.perf_counter()

    print(f"Result: {sum(result)}")
    print(f"Time: {end_time - start_time} second")
```

2개 Thread 이용

```
Result: 4999999950000000
Time: 2.886001499999111 second
```

80x86 시스템

- **(실습) 파이썬에서의 병렬 처리 방법**

```
import time
from multiprocessing import Process, Queue


def work(start, end, result):
    total = 0
    for i in range(start, end):
        total += i
    result.put(total)
    return

if __name__ == "__main__":
    START, END = 0, 100000000
    start_time = time.perf_counter()

    result = Queue()
    arg_list = [(START, int(END / 2), result),
                (int(END / 2), END, result)]
    p1 = Process(target=work, args=arg_list[0])
    p2 = Process(target=work, args=arg_list[1])
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    end_time = time.perf_counter()
```

```
total = 0
   while True:
       if result.empty():
           break
       total += result.get()

   print(f"Result: {total}")
   print(f"Time: {end_time - start_time} second")
```

※ get() 함수

Queue에서 맨 처음 원소를 삭제하고

해당 원소를 리턴

```
Result: 4999999950000000
Time: 1.5605901999988419 second
```

- 1개의 스레드와 2개의 스레드를 이용했을 때는 걸리는 시간이 거의 동일 했지만,

- 2개의 프로세스를 이용했을 때는 병렬처리 효과로 빠르게 실행됨

  프로세스 생성 부하가 있기 때문에 시간이 1/2이 되지는 않음

80x86 시스템

- **(실습) 파이썬에서의 병렬 처리 방법(멀티 스레드가 속도가 빠르지 않은 이유)**

  - GIL(Global Interpreter Lock)
    프로그래밍 언어에서는 자원을 보호하기 위해 다양한 방식의 락(Lock) 정책을 사용하는데, 파이썬에서는 하나의 프로세스 안에서는
    모든 자원의 락(Lock)을 글로벌(Global)하게 관리함으로써 한번에 하나의 스레드 만 자원을 컨트롤하여 동작하도록 함

  - 위의 코드에서 result 라는 자원을 공유하는 두 개의 스레드를 동시에 실행시키지만, 결국 GIL 때문에 한번에 하나의 스레드만 계산을 실행하여
    실행 시간이 비슷한 것

  - GIL 덕분에 자원 관리(예를 들어 가비지 컬렉션)를 더 쉽게 구현할 수 있지만, 지금처럼 멀티 코어가 당연한 시대에서는 조금 아쉬운 것이 사실

  - 파이썬의 스레드가 쓸모 없는 것은 아니며, GIL이 적용되는 것은 cpu 동작에서 이고 스레드가 cpu 동작을 마치고 I/O 작업을 실행하는 동안에는
    다른 쓰레드가 cpu 동작을 동시에 실행할 수 있기 때문에 cpu 동작이 많지 않고 I/O동작이 더 많은 프로그램에서는 멀티 스레드만으로도
    성능에서 큰 효과를 얻을 수 있음

DLL(Dynamic Linked Library)

DLL(Dynamic Linked Library)

DLL(Dynamic Linked Library)

- **DLL(Dynamic Linked Library, 동적 연결 라이브러리) 이란?**

  - Windows OS는 Multi-tasking을 지원하는 운영체제이고, 수 많은 공용 라이브러리 함수(process, memory, window, message 등)를 사용
  - 이런 공용 라이브러리 함수들을 각각의 실행 파일 안에 모두 포함하고 있으면 디스크나 메모리 사용에 있어서 비효율 적일 수 밖에 없음
  - 32bit Windows 부터 설치 프로그램의 수와 동시에 실행하는 프로그램의 수가 크게 늘어나면서 디스크나 메모리의 낭비 문제가 심각해 졌고, 문제를 해결하기 위해서 Windows OS 설계자들은 DLL 개념을 고안해 냄

"프로그램내에 라이브러리를 포함시키는 대신 별도의 파일(DLL)로 구성하여 필요할 때마다 로드해서 사용하기"

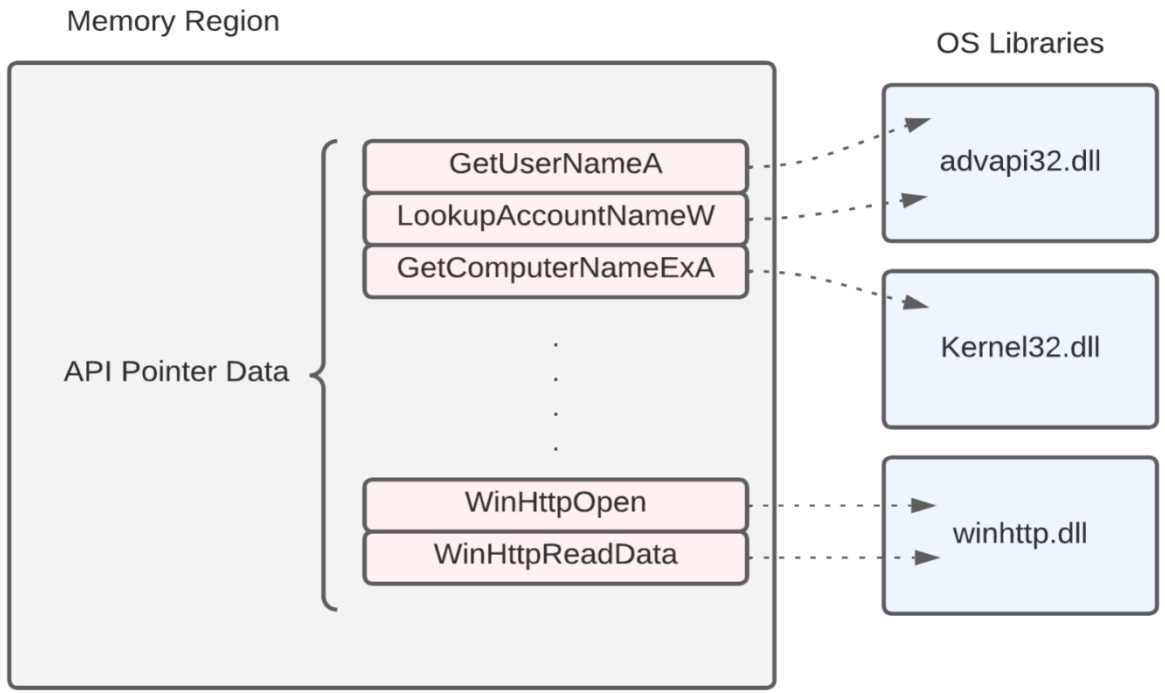
"일단 한번 로딩된 DLL 의 코드와 리소스는 메모리 맵핑 기술로 여러 프로세스에서 공유하기"

"라이브러리가 업데이트 되었을 때 해당 DLL 파일만 교체하면 되기 때문에 쉽고 편함"

DLL(Dynamic Linked Library)

- **DLL(Dynamic Linked Library) 이란?**

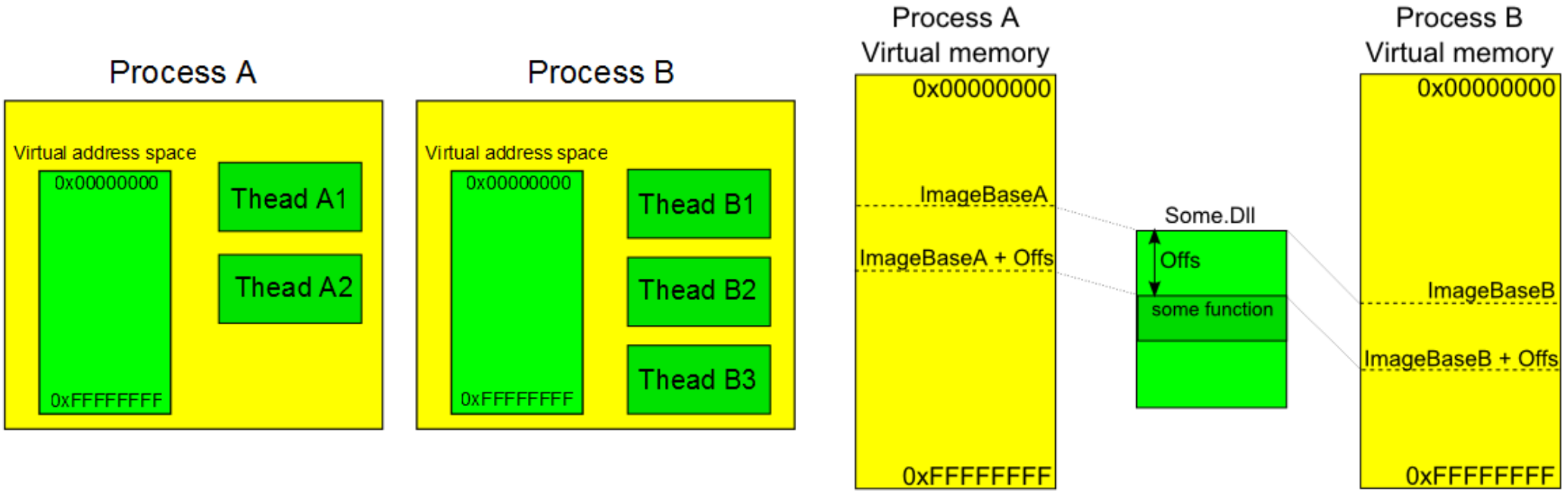
  - 프로그램에서는 다양한 DLL(Dynamic Linked Library)의 함수들을 호출



DLL(Dynamic Linked Library)

- **DLL(Dynamic Linked Library) 이란?**

  - 프로세스 간에 DLL을 공유함으로써 효율적인 메모리 관리가 가능

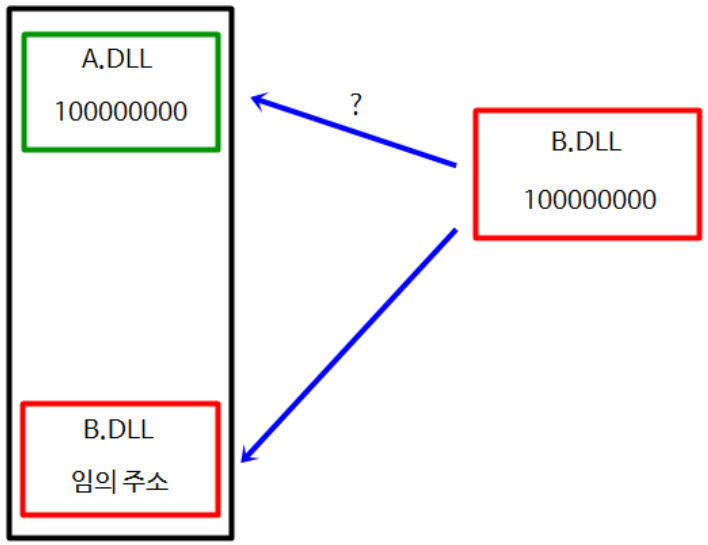


DLL(Dynamic Linked Library)

● DLL Relocation(DLL 재배치)

- PE loader가 PE 헤더에 지정된 ImageBase 주소에 DLL을 로드 하려고 할 때, 해당 위치에 이미 다른 DLL이 로드되어 있다면 비어 있는 임의의 주소 공간에 DLL을 로드하는 것

* 참고 : EXE 파일의 경우 DLL과는 달리 가장 먼저 메모리에 로드되기 때문에 재배치가 필요 없지만, 메모리 취약점 공격을 어렵게 하기 위해서 도입한 기능인 ASLR(Address Space Layout Randomization)로 인해 재부팅 시 마다 로드되는 주소가 바뀜



DLL 재배치

```
Base address    : 0x7ff666470000
main() address  : 0x7ff66647e350
```

- 첫 번째 실행 시 로드 된 기본 주소와 main() 함수 주소

```
Base address    : 0x7ff71f390000
main() address  : 0x7ff71f39e350
```

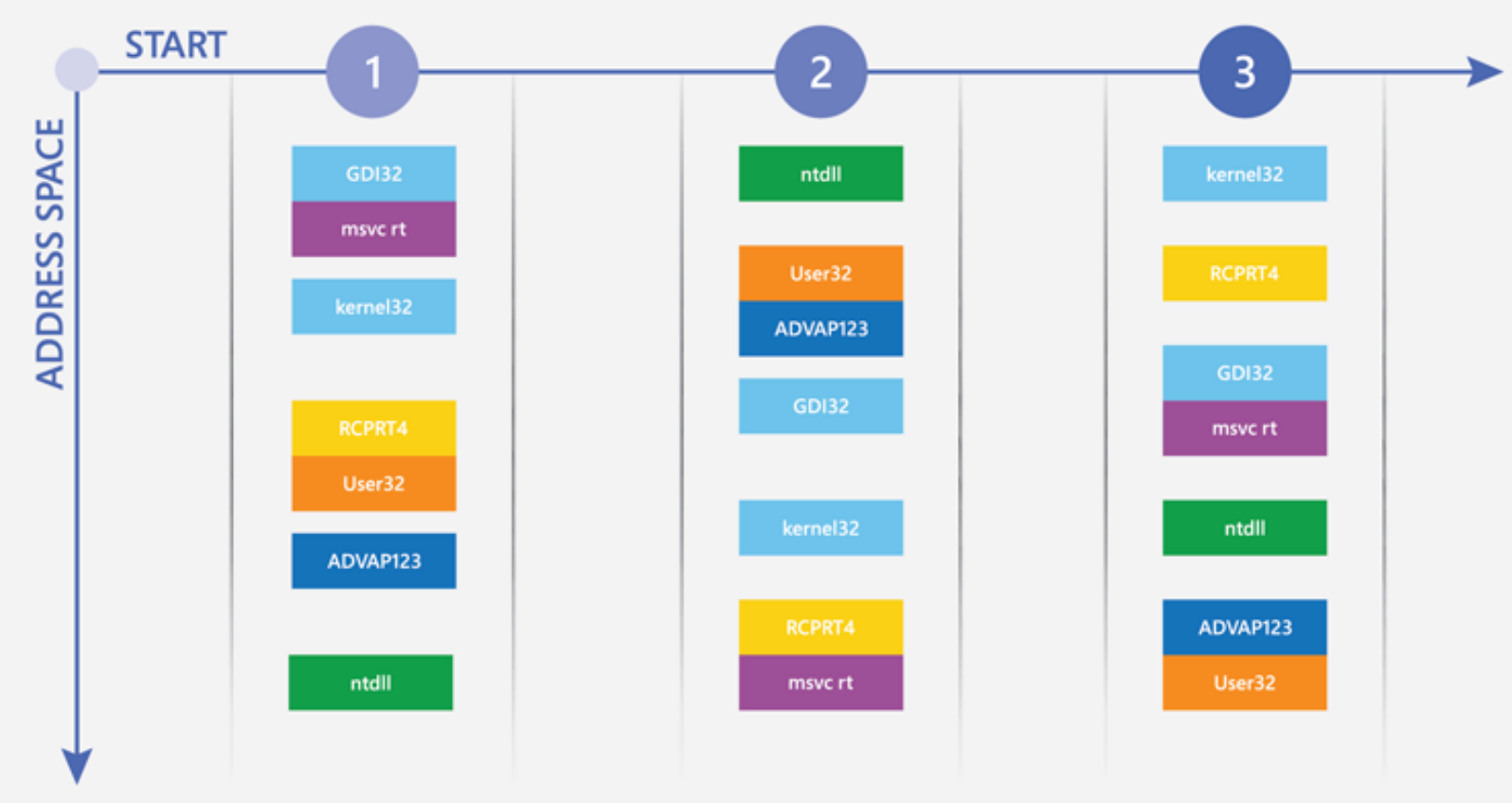
- 재 부팅 후 실행 시 로드 된 기본 주소와 main() 함수 주소 달라짐

ASLR로 인한 EXE 재배치

DLL(Dynamic Linked Library)

- **ASLR(Address Space Layout Randomization)과 시스템 DLL 재배치**

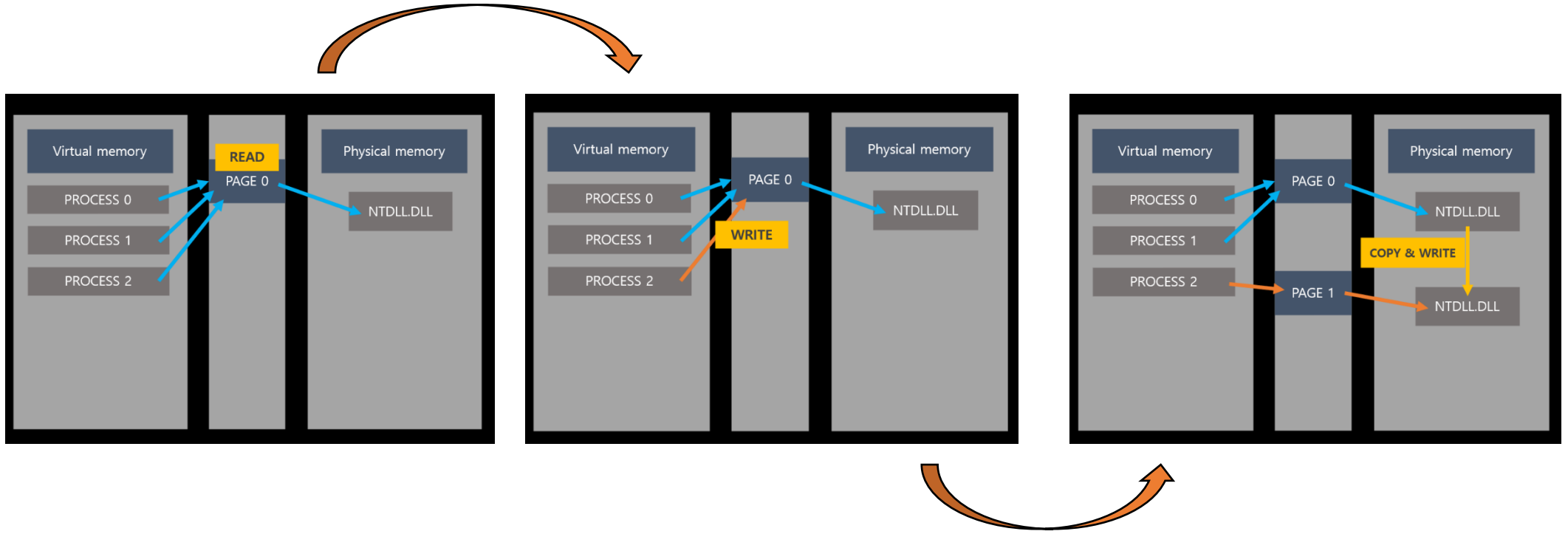
  - kernel32.dll과 같은 시스템 DLL들은 기본적으로 각자의 고유한 영역에 로드되기 때문에 로드되는 주소가 바뀌지 않지만 ASLR기능은 이런 시스템 DLL의 경우에도 임의 주소로 재배치를 수행(시스템 DLL의 경우 재부팅 전까지는 유지)



DLL(Dynamic Linked Library)

- **COW(Copy-On-Write)**

  - 특정 프로세스에서 로드 한 DLL의 특정 함수를 후킹(Hooking)하거나 코드를 수정해도 다른 프로세스에 영향을 주지 않음
    COW(Copy-On-Write) 기법으로 인해서 DLL에 write하려고 하면, OS는 physical memory DLL을 그대로 복사하여 해당 write를 복사한 DLL에 진행한 다음 write를 요청한 user-mode process의 DLL을 새로 복사한 DLL로 update



DLL(Dynamic Linked Library)

- **DLL 로딩 방식 : 암시적 로딩(Implicit Loading), 암시적 링킹(Implicit Linking)**

  - 정적 로드 방식, 함수가 정적으로 연결되고 실행 파일 내에 포함된 것과 동일한 방식으로 DLL에서 내보낸 함수를 호출할 수 있음
  - DLL 컴파일시 생성되는 lib 파일을 이용해 연결되며, lib에 담겨진 정보를 토대로 런타임에 DLL의 함수 코드를 참조
  - 프로그램 시작할 때 DLL이 로드되고, 프로그램이 종료될 때 메모리에서 해제 됨 (IAT, Import Address Table 사용)

  * 지연된 DLL 로드(delay load dll) : 암시적 로딩과 유사하지만, 프로그램 실행 시 무조건 DLL이 로드되는 것이 아니라 실제 DLL의 함수가 호출될 때 로드되는 방식

```
#include <windows.h>

#pragma comment(lib, "delayimp")                  // delayimp.lib 지연로딩을 위해 필요한 라이브러리
#pragma comment(linker, "/DELAYLOAD:user32.dll") // user32.dll 을 지연로딩한다는 설정
#pragma comment(lib, "user32")                    // user32.dll implicit linking

int APIENTRY _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR    lpCmdLine,
    int       nCmdShow)
{
    MessageBoxA(NULL, "Hello", "Hello", MB_OK);
}
```

DLL(Dynamic Linked Library)

- **DLL 로딩 방식 : 명시적 로딩(Explicit Loading)**

  - 동적 로드 방식, 실행 중에 필요한 DLL을 로드하는 방식으로 DLL의 각 함수에 접근할 함수 포인터를 설정해 함수를 호출해야 함
  - 사용이 끝나면 DLL을 언로드 해주어야 하며, LoadLibrary(), GetProcAddress(), FreeLibrary()의 세 가지 함수를 통해 구현

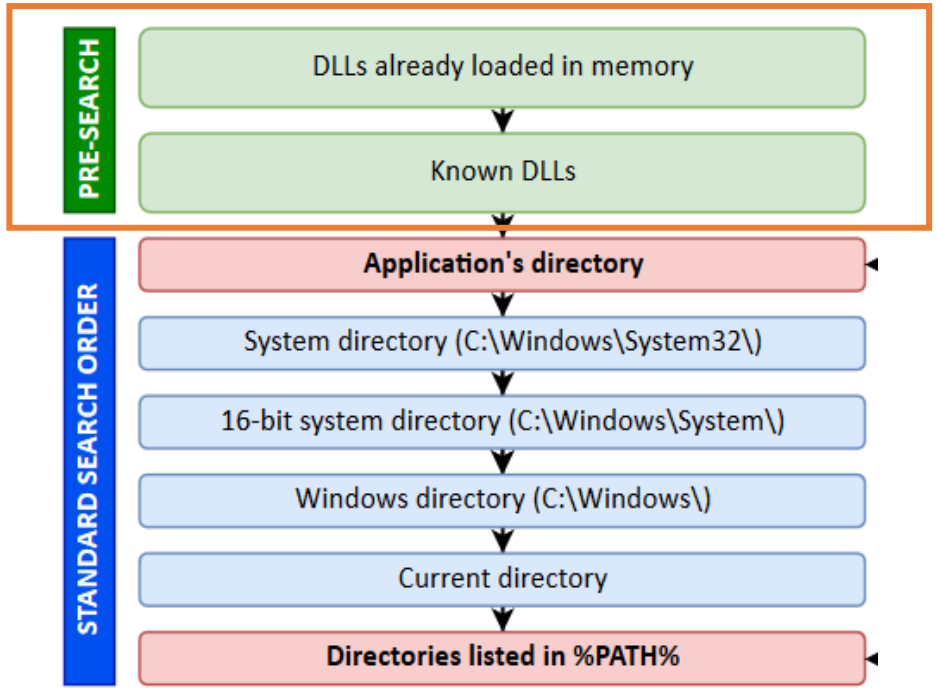
```c
#include <windows.h>
#include <stdio.h>

// DLL function signature
typedef double (*importFunction)(double, double);

int main(int argc, char** argv)
{
    importFunction addNumbers;
    double result;

    // Load DLL file
    HINSTANCE hinstLib = LoadLibraryA("Example.dll");
    if (hinstLib == NULL) {
        printf("ERROR: unable to load DLLWn");
        return 1;
    }

    // Get function pointer
    addNumbers = (importFunction)GetProcAddress(hinstLib, "AddNumbers");
    if (addNumbers == NULL) {
        printf("ERROR: unable to find DLL functionWn");
        FreeLibrary(hinstLib);
        return 1;
    }

    // Call function.
    result = addNumbers(1, 2);

    // Unload DLL file
    FreeLibrary(hinstLib);

    // Display result
    printf("The result was: %fWn", result);

    return 0;
}
```

DLL(Dynamic Linked Library)

● DLL (Dynamic Linked Library) Search Order

- Windows OS에서 DLL을 로드하기 위해 DLL을 찾는 순서
- DLL Side-Loading(DLL Preloading, DLL Hijacking, DLL Planting, DLL Proxying) 공격 기법에서 악용

* 참고 : IsWow64Process() 함수는 64비트 OS에서 프로그램이 32비트 호환모드(Wow64)로 돌아가는지 확인해주는 함수
64비트 시스템에서 동작하는 32bit 프로세스인 경우, 실제로는 시스템 폴더가 C:₩Windows₩SysWOW64로 연결되어 있음(하위 호환성 유지를 위한 기능)



- Windows에서 dll 위치를 찾는 순서 (dll search order)

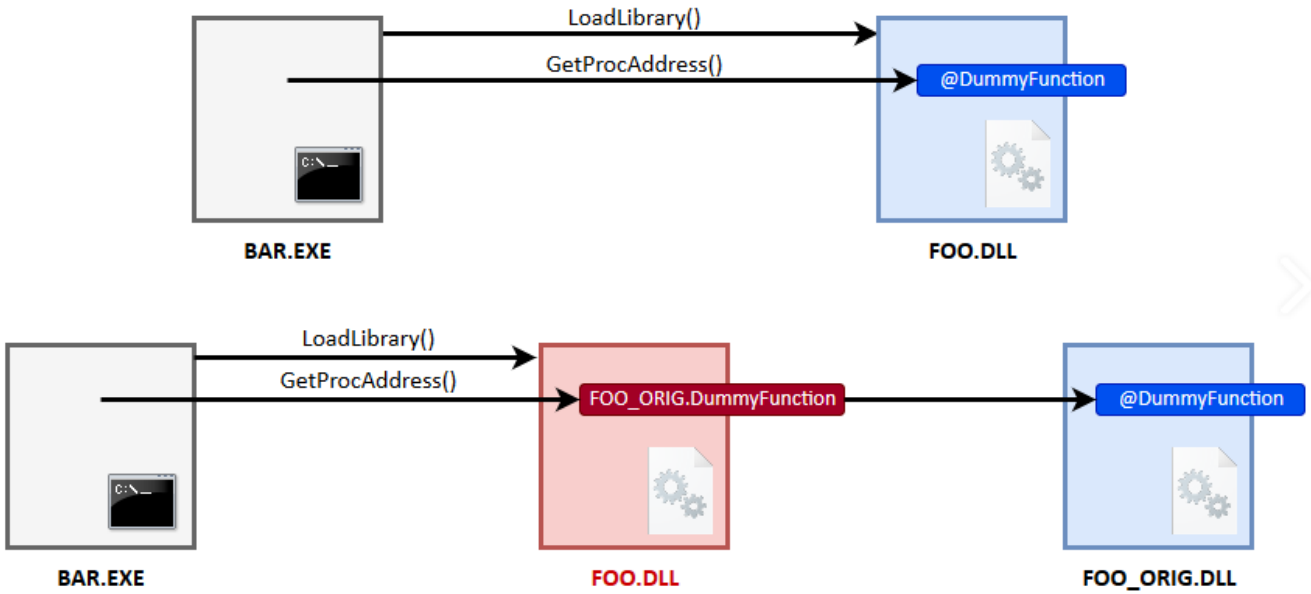
0. “Known Dlls”에 등록된 dll 들은 시스템 디렉토리를 먼저 검색
   : 이런 dll 들은 대부분 이미 메모리에 로드 되어 있음
1. 현재 프로세스의 실행 파일이 있는 디렉토리
2. 현재 디렉토리
3. Windows 시스템 디렉토리
   : C:₩Windows₩System32
4. Windows 디렉토리(일반적으로 대부분의 시스템에서 C:₩Windows)
5. PATH 환경 변수에 나열된 디렉토리

DLL(Dynamic Linked Library)

- **DLL (Dynamic Linked Library) Proxy Hijacking**

  - Proxy DLL : DLL에서 후킹하려는 특정 함수를 제외한 나머지 함수는 기존 DLL의 함수를 랩핑하여 그대로 호출하도록 만든 DLL
    이렇게 Proxy DLL을 만드는 것을 DLL Proxying이라고 함
  - 원래의 FOO.dll의 Copy() 라는 함수를 랩핑하려면 Proxy DLL을 만들 때 아래 내용을 추가해주면 됨
    #pragma comment(linker, "/export:Copy=FOO.Copy")

  * 이러한 기능은 하위 호환을 고려한 새로운 버전의 DLL 을 만들고 싶을 경우에 사용하기도 하지만 악성 DLL을 만드는데 활용되기도 함



Q A