프로그램 실행시 메모리 레이아웃

by. bOBaNa
Email : bOBaNa@wowhacker.org
Homepage : http://lazyhack.tistory.com

Wowhacker Team

http://www.wowhacker.org

Abstract

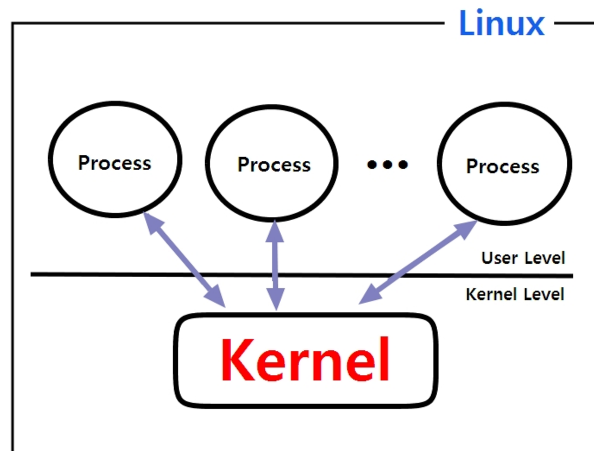
이 문서는 프로세스가 어떻게 생성되고, 어떻게 배치되어 있는지 알기 위해서 작성되었습니다. 개인적인 발표를 위해 공부하였지만, 나름대로 정리하여 보관하기 위해서 이렇게 문서화합니다. 혹시 이상한 부분이나 제가 틀린부분이 있다면 지적 부탁드립니다. 편의를 위해 말을 높이지 않은 점 양해바랍니다. ^^

1. 서론

이 문서는 리눅스 기반의 프로그램이 실행시 어떤 과정을 거쳐서 실행되고, 실제 메모리에 어떻게 자리잡는지 설명한다. 간단히 말하면, 프로세스에 대한 이야기라고 할 수 있다.
처음 이 주제에 대해서 발표를 준비 할 때에는 너무도 쉬운 내용이라 망설였지만, 깊이 들어갈수록 생각보다 어렵고 방대한 내용이었다. 이 문서에서는 커널 소스 분석이나 심도있는 설명은 배제한다.

2. 프로세스

먼저, 프로세스가 어떻게 실행되는지 보면 쉘에서 사용자의 입력을 기다리고, 사용자가 입력한 명령어가 실행된다. 예를들어 /bin/ls 를 실행하면 파일 및 디렉토리의 리스트를 출력하는 화면이 우리 앞에 보일 것이다. 이는 뒤에서 살펴보겠지만, 프로세스가 생성되고 생성된 프로세스가 /bin/ls 라는 프로그램을 메모리에 적재하여 해당 프로그램을 실행시킨 결과이다. 프로세스를 개념정의 해보자면, 프로그램이 메모리에 적재된 상태를 의미하며, 실행상태에 있는 프로그램의 인스턴스이다. 커널의 입장에서 보면 프로그램의 실행이 얼마나 진행되었는지를 기술하는 자료구조의 집합이라고 볼 수 있다. 즉, 메모리에 아직 적재되지 않은 상태를 프로그램이라 하고, 실행하여 메모리에 적재되면 프로세스라고 말한다.



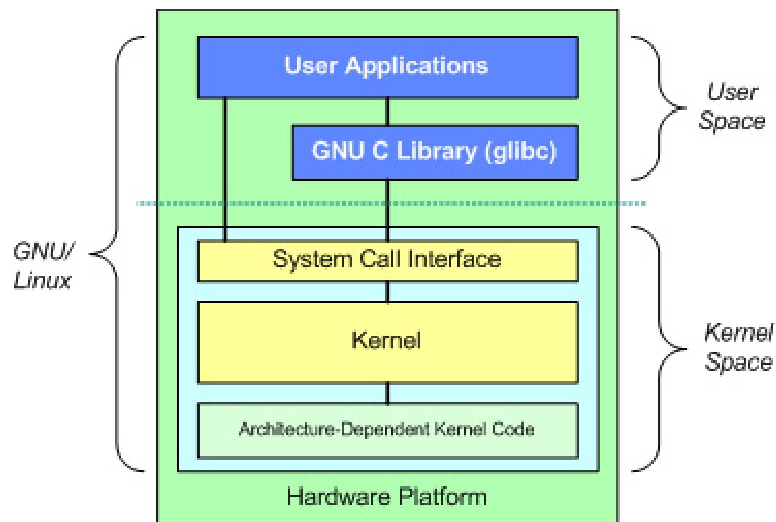
[그림 1]

위 그림은 리눅스에서 여러 프로세스가 실행되고 있는 상태를 간략하게 나타낸 것이다. 리눅스는 멀티태스킹을 지향하는 운영체제라서 여러 프로세스가 동시에 실행될 수 있다. 좀 더 자세히 말하면 동시에 실행되진 않지만, 사용자 입장에서 보면 동시에 실행되는 것처럼 보인다. 그러한 과정은 스케쥴러가 관리하는데 스케쥴러는 커널의 일부분이며, 운영체제에서 중요한 부분이다. 이에 대한 설명은 여기서는 생략하도록 하겠다. 위 그림에서는 각

프로세스와 커널이 유기적으로 관계를 맺고 있는 것을 볼 수 있으며, User Level 과 Kernel Level 로 구분되어 있는 것 또한 볼 수 있다. 이에 대한 설명도 잠시 뒤로 미루도록 하겠다.

3. About Process

위와 비슷한 그림이지만, 좀 더 자세히 나와있는 형태의 그림을 보자.



[그림 2]

위 그림을 보면 여러가지로 구성되어 있는 것을 볼 수 있다. 여기서 User Applications 이 실행되면 현재 시스템에 동적으로 할당되어 있는 GNU C Library 를 이용한다. Glibc 는 커널과 연결하는 시스템콜 인터페이스를 제공하며 Kernel Space 와 User Space 간의 전환 메커니즘을 제공한다. 그럼 Systel Call Interface 에 의해 커널과 교류하며, 커널은 디바이스를 제어하여 다시 사용자가 원하는 방향으로 이끌어 가는 것이다. 여기서도 Kernel level 과 User level 로 나뉘어져 있는데, 그 이유부터 알아보고 가도록 하자. 이에 대한 이야기를 시작하려면 무척 원론적인 이야기를 해야 한다. 운영체제에서의 프로세스 관련 내용을 보면 살펴 볼 수 있다. 앞서 말했듯이 프로세스는 여러 개가 동시에 메모리에 적재되어 실행될 수 있으며, 내부적으로 계속 각 프로세스간 전환(Context switching)을 통해 프로세스는 동시에 실행되는 것처럼 보인다. 프로세스는 단일주소 공간을 사용하며 그 크기는 4GB 이다. 여기서 하나의 의문이 생길 수 있다. 왜 4GB 인가? 이는 32bits 와 밀접한 관계를 갖는다. 컴퓨터는 모든 명령을 이진수 형태로 처리하며 이 진수는 0 과 1 의 두가지 형태를 갖는다. Bit 는 0 혹은 1 을 담을 수 있고 32bits 는 32 개의 0 혹은 1 을 담을 수 있다. 그래서, 이 것들의 경우의 수는 총 2 의 32 승으로 각각 특정 정보를 저장할 수 있는 공간이 4GB 가 되는 것이다. 이러한 32bits 의 0 과 1 의 모음을 선형 주소(Linear Address)라고 한다. 리눅스에서는 모든 주소는 32bits 로 되어있으며, 각 프로세스는 이러한 주소를 가지고 있다. 물론, 이러한 메모리는 RAM 이라고 불리는 주기억장치에서 가져오는 것이다. 상식적으로 생각하면 무언가 이상하다. 각 프로세스는 4GB 의 메모리를 갖는다고 했는데, 현재 우리가 사용하는 컴퓨터의

메모리는 보통 1GB 이며 많으면 4GB 이다. 많아야 4GB 까지의 메모리 밖에 존재하지 않는데 어떻게 하나의 프로세스가 4GB 의 크기를 가지며, 그러한 프로세스들이 제한적인 메모리에서 동시에 여러 개가 실행될 수 있는 것일까? 이에 대한 해답은 바로 Virtual Memory 이다.

(1) Virtual Memory

Virtual Memory 는 실행파일이 메모리보다 더 큰 문제를 해결하고 멀티프로그래밍에 따르는 작업간의 보호의 문제 등을 해결하기 위해 등장한 기법이다. Virtual Memory 의 기본적인 컨셉은 "Virtual Address"와 "Physical Address"의 분리이다. 즉, Vitural Address(이하 VM)은 어떤 가상 주소를 사용하지만, 이 가상주소에 대한 실제 매핑은 physical address 의 특정 부분으로 되는 것이다. 이 변환은 CPU 내의 MMU(Memory Management Unit)에 의해 가능해 지고 TLB(Translation Look-aside Buffer)에 의해 좀 더 빠른 변환이 가능해 진다.
이러한 과정에 의해 프로세스는 다른 프로세스들은 보이지 않는 자신만의 공간(4GB)를 가지게 되는 것이다. 좀 더 자세히 알아보자.

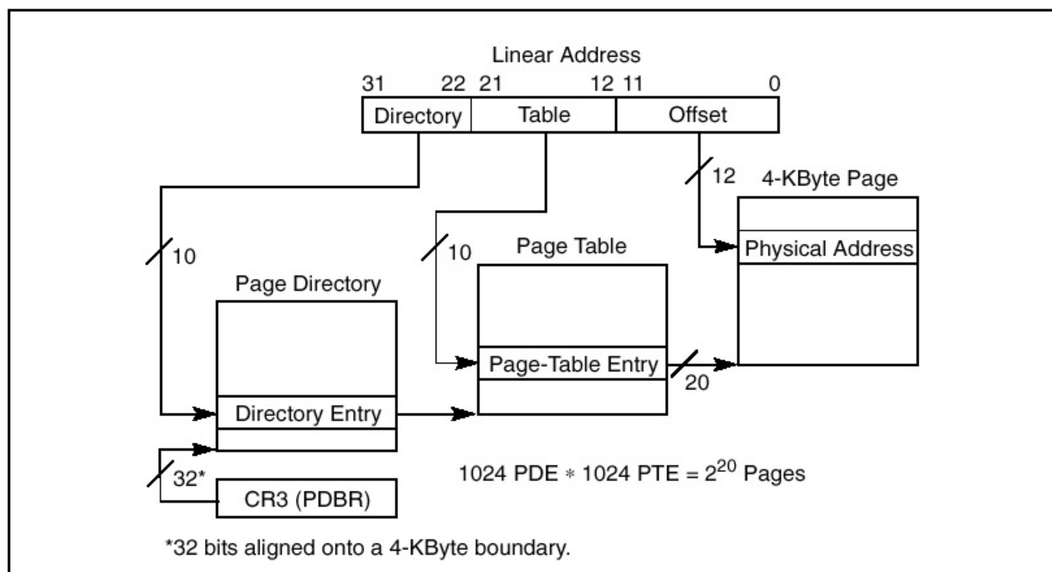
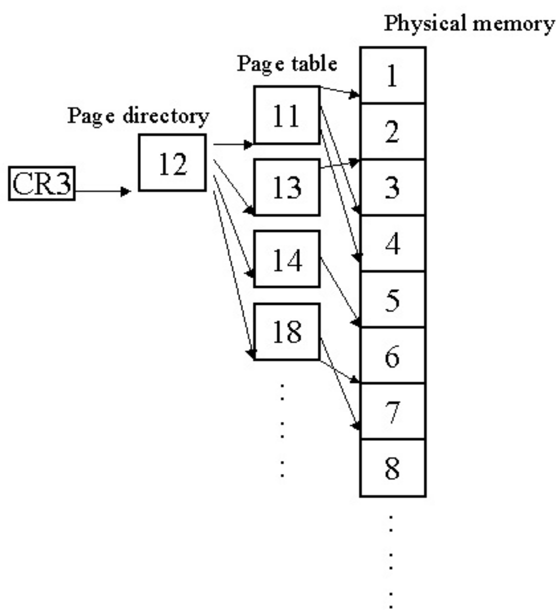


Figure 3-12. Linear Address Translation (4-KByte Pages)

Virtual Address(IA 에서 Linear Address 라고 부름)는 위 그림과 같은 구조를 통해 Physical Address 로 변환된다. 위 그림을 이해하면 프로세스가 왜 4GB 의 공간을 가질 수 있는지 알 수 있다. Virtual Address 는 32 비트로 구성되는지 앞의 10 비트는 페이지 디렉토리에서의 인덱스이며, 중간의 10 비트는 페이지 테이블에서의 인덱스, 그리고 마지막 12 비트는 오프셋으로 사용된다. 그리고 CPU 내에서 페이지 디렉토리를 가리키는 레지스터가 필요한데 그것이 바로 CR3 라는 레지스터로서 아주 중요한 역할을 가지고 있다. 운영체제에서 Physical Memory 는 모두 4Kb 의 페이지로 구성되어 있다. 때문에 각각의 페이지 디렉토리와 페이지 테이블은 모두 1 페이지를 나타낸다. Virtual Address 에서 10 비트, 10 비트, 12 비트의 인덱스를 사용하기 때문에 페이지 디렉토리를 가리키는 앞의 10 비트는 2 의 10 승으로 총

1024 개의 엔트리를 가질수 있다. 또한 중간의 페이지 테이블을 가리키는 10 비트 또한 1024 개의 엔트리를 가질 수 있다. 이런식으로 하나의 메모리를 참조하게 될 경우에는 CR3 가 가리키는 페이지에서 Virtual Address 의 앞 10 비트를 인덱스로 사용해서 해당하는 entry 를 참조하고, 이렇게 얻은 entry 는 다음 page table 로의 base address 를 제공하게 된다. 여기서 다시 중간의 10 비트를 인덱스로서 사용하여 page-table entry 가 physical page 의 물리적인 주소를 제공하게 되며, 이 주소에 virtual address 의 마지막 12bit(오프셋)를 더해주면 최종적인 Physical Address 를 얻게 되는 것이다.



그럼 왼쪽과 같은 형태의 Tree 구조를 취하게 된다. 여기서 CR3 는 하나의 프로세스의 주소를 가리킨다고 볼 수 있다. 즉, Context Swtiching 이 일어날 때마다, CPU 는 저 CR3 레지스터가 가지고 있는 페이지 디렉토리의 주소를 변경한다. 페이지 디렉토리가 이와 같이 페이지 디렉토리는 총 1024*1024 개의 페이지를 가리킬 수 있게 되므로 프로세스의 주소는 4GB 를 가질 수 있게 되는 것이다.
하지만, 왼쪽의 그림은 개념적인 형태일 뿐 실제의 메모리는 선형적으로 이루어져 있다는 것을 명심해야 한다.

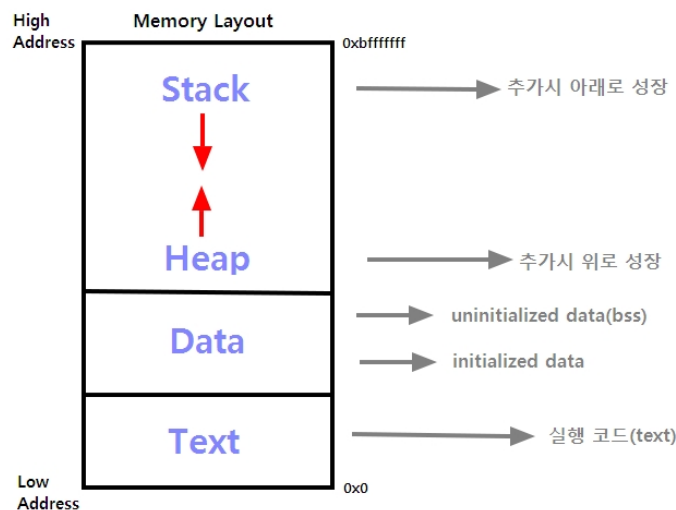
(2) 프로세스의 주소공간

[그림 2]를 다시 보자. Kernel Level 과 User Level 로 나눠져있는 것을 볼 수 있다. 현재 CPU 는 대부분 CPU 의 동작 모드를 Kernel Mode 와 User Mode 로 구분하여 사용한다. Kernel Mode 에서는 모든 Instruction 의 실행에 제한이 없지만, User Mode 에서는 특정 Instructiong 을 수행할 수 없다. 따라서 Kernel Mode 를 특권 모드(Privileged mode)라고도 한다. 이를 이용하여 다른 프로레스에 의한 침입을 보호할 수 있다. 이런 방식으로 CPU 는 동작모드를 여러단계로 나눈다. 흔히 말하는 ring 0 가 Kernel mode 이며 ring 3 를 user mode 라고 한다. 리눅스에서는 프로세스의 주소공간을 만들 때 User Level 과 Kernel Level 을 나누어서 만들게 되는데 Kernel Level 은 4GB 중 1GB 를 사용하며 User Level 의 경우 3GB 를 사용하게 된다. 여기서 중요한 점은 Kernel Level 인 1GB 는 모든 프로세스가 공유한다는 사실이다. 이러한 구조 때문에 시스템이 Kernel Mode 일 때는 두 공간을 모두 넘나들 수 있지만, User Mode 일때는 Kenel Level 에 접근할 수 없다. 일반적인 응용 프로그램이 User Level 에서 실행될 때는 대부분 System Call 을 이용하게 된다. 만약 Kernel Level 이 공유되어 있지 않고 다른 곳에 존재한다면, System Call 이 발생할 때마다 Context Switching 이 이루어져 속도면에서 매우 비효율적일 것이다. 이러한 이유로 Kernel Space 는 각 프로세스가

똑 같은 영역을 공유하고 있는 것이다. 그리고 Kernel 이 User Space 를 마음대로 제어하기 위해서는 Kernel Space 가 공유되는 것이 좀 더 효율적이다.

(3) Process 의 구성

프로그램을 실행하면, 커널은 그 Image 를 메모리에 올리고(load), 실행을 시작하는데 그 실행환경(context)와 메모리에 올라온 이미지를 프로세스라고 하며, 리눅스에서는 이를 task 라고 부른다. 리눅스에서는 일종의 Segmentation 기법인 VMA(Virtual Memory Area)를 구현하여 한 프로세스가 Address space 를 사용할 수 있도록 한다. 이러한 Area 들은 Stack, bss, Data, Text 가 존재하며 다음과 같은 개념으로 존재한다.



[그림 3]

Text 는 Code Section 이라고 부르며, Start_routine 과 실제 프로그램 코드(실행되는 이미지)가 적재된다. 이 Section 은 읽기만 가능하며, 로더에 의해서 메모리에 로드된다.

Data 는 두가지 영역으로 볼 수 있는데 .data 와 .bss 영역이다. .data 의 경우는 초기화 되어 있는 전역 변수들이 해당되며 Text 가 적재될 때 로더에 의해서 같이 로딩된다. .bss 영역은 초기화되어 있지 않은 변수들이 저장되는 장소이며 .data 와 다르게 실행파일 내에 포함되지 않고 로더에 의해서 로드후 메모리가 할당되고 0 으로 채워지게 된다. Stack 영역은 프로그램 실행 시 지역변수나 환경변수와 같은 변수들이 위치하며, 프로그램 실행과 중요한 영역이다. 앞서서 프로세스는 4GB 의 메모리 공간을 가진다고 설명하였다. 하지만, 엄밀히 말하면, 프로세스가 4GB 의 메모리를 사용할 수 있는 것은 맞지만, 4GB 의 메모리를 모두 가지고 있는 것은 아니다. 일반적으로 프로세스가 생성될 때에는 프로그램의 실행에 필요한 최소한의 Code, Data 만이 메모리에 적재된다. 메모리의 효율성을 증가시키고, 효과적인 실행을 위함이다. 운영체제는 최소한의 것들만 가지고 필요할 때마다 추가시키는 메커니즘을 사용한다. 이와 같이 효율적인 페이지 사용을 위한 운영체제의 메커니즘이 몇가지 있는데 Demand Paging 과 Copy on Write 가 바로 그것들이다. 앞서 말했듯이, 프로세스는

VMA 영역을 생성한다, 그러한 VMA 들은 기본적으로 각각 시작점과 끝점을 가진다. 예를들면, 스택의 경우는 bffff000-c0000000 까지가 그 범위이다. 특정 주소에 대한 메모리 참조가 일어났을 때 올바른 참조라면 해당 메모리를 주겠지만, 올바르지 않다면 해당하는 페이지가 없다는 의미로 Page Fault 가 일어나게 된다. 이를 이용한 것이 Demand Paging 과 Copy On Write 이다. Demand Paging 은 최소한의 Address mapping 만으로 프로세스를 시작시키고, 프로세스가 어느 시점에서 자신의 특정 메모리 영역으로 접근하려고 시도했을 때 page fault 가 일어나고 해당 영역이 VMA 의 범위에 해당하는 경우에 Page Fault Handler 가 실제 매핑을 추가하게 되는 것이다. 예를들어, Stack 이 Demand Paging 의 사용의 대표적인 예이다. 우리는 Stack 이 메모리 아래 방향으로 자란다는 사실을 알고있다. 하지만, 운영체제(커널)는 이 Stack 이 프로그램 실행시 어디까지 자랄지 알지 못한다. 그렇다고해서 사용가능한 모든 메모리를 미리 할당하면 사용되지 못하는 공간이 생길 수 있어 효율성이 떨어지게 된다. 결국 Demand Paging 을 이용하여 요구가 있을 때마다 페이지를 추가하여 매핑시켜주는 것이다. Copy On Write 방식은 fork()에서 유용하게 사용될 수 있는데, 부모 프로세스의 Text 와 같은 읽기만 가능한 페이지들은 복사할 필요가 없는데, 해당 Physical Page 를 공유함으로써 빠르게 fork 한다. 즉, 특정 페이지를 복사할 필요 없이 두 프로세스가 해당 페이지를 공유하고, 두 프로세스 중 어떤 프로세스가 해당 페이지에 대한 쓰기가 필요할 때(이때 Page Fault 가 일어난다) 공유하고 있던 페이지를 두 개로 분리하는 방식이 Copy on write 이다.

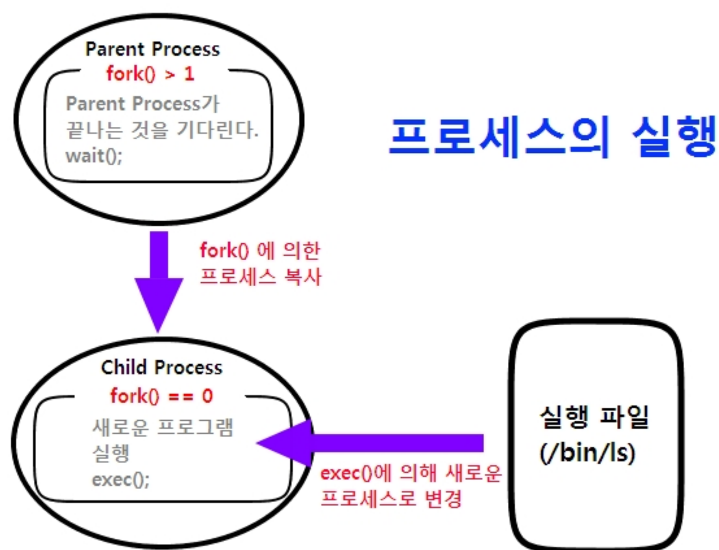
여기까지가 프로세스의 레이아웃을 알기 위한 기본지식이다. 이제는 본격적인 프로세스의 실행으로 넘어가자.

4. 프로세스의 실행

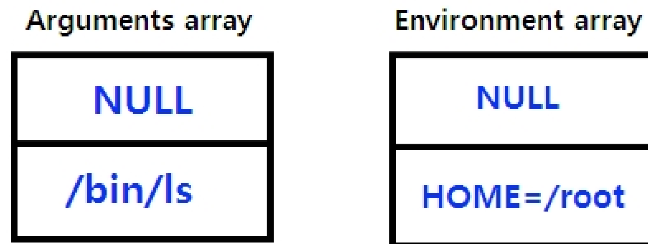
프로세스가 실행되기 전에는 대부분 쉘을 통하여 명령어를 받고 해당 명령어에 의해 특정 프로그램이 실행되게 된다. 다음 코드가 기본적인 쉘의 형태이다.

```
#define TRUE 1
        while(TRUE)
        {
        read_cmd_line(command, parameters);
        pid_t = fork();                → ①
        if(pid_t != 0) {
                /* parent section */
                waitpid(-1, &status, 0);
        }
        else {  /* Child Section */
                execve(command, parameters, 0);     → ②
        }
}
```

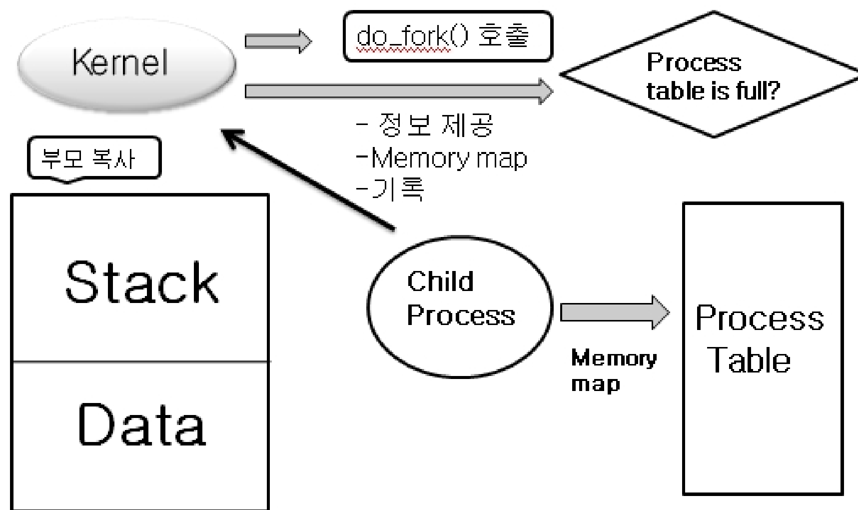
무한루프를 통해 쉘은 사용자로부터의 입력을 기다리며, 입력을 받으면 fork() 시스템 콜 함수를 이용하여 새로운 프로세스를 생성하고, 생성된 자식 프로세스는 exec() 시스템 콜을 이용하여 사용자로부터 입력받은 명령어를 실행하기 위해 프로세스를 새로운 프로세스로 변경한다. 부모 프로세스는 이러한 자식 프로세스가 끝날 때까지 기다리며, 자식프로세스가 종료되면 다시 원래의 무한루프로 돌아가게 된다. 이러한 과정은 다음 그림과 같다.



방금까지 프로세스의 실행의 개략적인 형태를 살펴보았다. 이제부터는 Step by Step 으로 조금씩 깊이들어가 보자. 일반적으로 사용자는 쉘을 통해 특정 명령어를 입력받는다. 여기서는 /bin/ls 를 실행하였다고 가정하자. 사용자가 /bin/ls 를 입력하면 쉘은 그것을 받아서 Argument array 형태로 저장하고, 프로그램 실행에 도움을 주기 위해 환경변수 내용도 Environment array 형태로 저장한다. 다음 그림과 같다.



위처럼 두 개의 정보를 배열형태로 만들어 놓는다. 그 후 fork() 시스템 콜이 호출되는데, 다음 그림과 같은 과정을 통해 fork()가 수행된다.

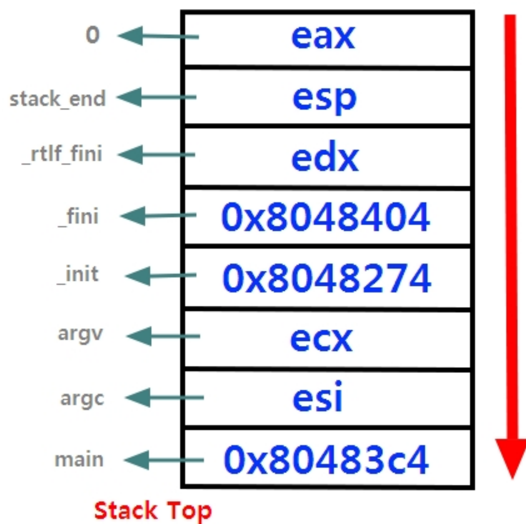


우선 커널은 do_fork() 시스템 콜을 호출하고, 프로세스 테이블이 가득찼는지 확인한다. 그 후 자식의 Stack, data 공간을 마련하고 마련된 공간에 부모의 Stack, data 를 자식으로 복사한다. 이 후 자식의 메모리 맵을 프로세스 테이블에 입력하고 getpid()를 이용하여 자식프로세스에 pid 값을 부여하며, 커널에 자식프로세스에 대한 정보를 제공한다. 이러한 과정을 통해서 fork()는 최종적으로 자식프로세스의 pid 를 반환하는 것이다.

여기까지의 과정은 쉘로부터 명령어를 받고 fork()를 수행하였으며, exec()를 호출하기 전이다. 하지만, 우선적으로 알아야 할 것이 있다. 보통 gcc 를 이용하여 특정 코드가 컴파일 되었을 때는 다음과 같은 start_routine 이 프로그램에 포함되게 되어 프로그램 실행시 main 이 실행되기 전 실행되게 된다.

```
080482e0 <_start>:
 80482e0:       31 ed                   xor    %ebp,%ebp
 80482e2:       5e                      pop    %esi
 80482e3:       89 e1                   mov    %esp,%ecx
 80482e5:       83 e4 f0                and    $0xfffffff0,%esp
 80482e8:       50                      push   %eax
 80482e9:       54                      push   %esp
 80482ea:       52                      push   %edx
 80482eb:       68 04 84 04 08          push   $0x8048404
 80482f0:       68 74 82 04 08          push   $0x8048274
 80482f5:       51                      push   %ecx
 80482f6:       56                      push   %esi
 80482f7:       68 c4 83 04 08          push   $0x80483c4
 80482fc:       e8 cb ff ff ff          call   80482cc <__libc_start_main@plt>
 8048301:       f4                      hlt
 8048302:       90                      nop
 8048303:       90                      nop
```

이 코드를 살펴 보면 특정 주소를 stack 에 push 하는 것을 볼 수 있는데, 이를 도식화하면 다음 그림과 같으니 비교하여 이해하자.



이 그림을 보면 argv, argc, main 의 주소등 여러가지가 stack 으로 저장이 되는 것을 알 수 있다. start_routine 은 argument stack 을 만들고 __libc_start_main 을 호출한다.
물론, 여기까지의 과정에서는 argv, argc, envp 는 argument stack 상에 적용되지 않은 상태이다. 프로그램을 실행하기 전, 컴파일 할 때 이러한 과정이 미리되어 있다는 것은 염두하자.

이제 다시 본론으로 넘어가서 fork() 수행 후 쉘은 argv, envp 를 가지고 exec() 시스템 콜을 호출한다. exec()가 호출되면 Kernel System call handler 가 처리하게 된다. (sys_execve())
x87 의 User-mode 프로그램은 ebx, ecx, edx 레지스터를 통해 각각 프로그램명 문자열의 포인터, argv 배열 포인터, 환경변수 배열 포인터를 커널에 넘기게 된다. 여기까지의 과정이 지나게 되면 바로 위의 그림에서 맨위의 stack 에 환경변수 등이 추가되어 execve()가 호출되게 된다. 이제 execve()가 호출될 때 과정을 살펴보자.
execve()가 호출되면 do_execve()가 호출되고, 메모리를 검사하여 해당파일이 실행가능한지 확인하며 segment 와 total size 를 얻기 위해 프로그램의 헤더를 읽는다. 그 후 이전에 생성된 argument, environment stack 을 불러온다(fetch). 불러온 stack 을 할당하기 위해서 새로운 메모리를 할당하고 필요없는 메모리를 제거하며 불러온 Stack 을 할당된 새로운 메모리에 복사하고, Data segment 또한 새로운 메모리 이미지에 복사한다.(data segment 는 컴파일 시 프로그램 이미지에 기록된다), 이후 프로세스 테이블을 수정하여 프로세스가 작동가능하다고 커널에 전달하게 되면 do_execve()의 역할은 끝나게 되고, 원하는 프로그램이 실행되는 것이다. 최종적으로 [그림 3]과 같은 형태의 메모리 레이아웃이 메모리에 적재된다.

이제는 실제로 메모리에 어떻게 적재되어 있는지 그 레이아웃을 눈으로 확인해 본다.

5. 실제 Memory Layout

실제 메모리 레이아웃을 확인하기 위한 과정은 아주 간단하다. 우선 gdb 를 통해 파일을 실행하고, breakpoint 를 특정 위치(아무곳이나 상관없다)에 잡아주고 shell 명령어를 실행 후 ps 를 통해 우리가 실행한 프로세스의 아이디를 확인한다. 마지막으로 /proc/<pid>/maps 를 확인하면 다음과 같은 정보를 볼 수 있을 것이다.

```
[lazyhack@localhost main]$ cat /proc/18899/maps
08048000-0809d000 r-xp 00000000 03:02 294917     /home/lazyhack/main/memory
0809d000-080a0000 rw-p 00055000 03:02 294917     /home/lazyhack/main/memory
080a0000-080a1000 rwxp 00000000 00:00 0
bfff f000-c0000000 rwxp 00000000 00:00 0
```

메모리 범위
해당영역에 대한 권한
Offset
디바이스 번호
I-node 번호
경로명

위 정보를 보면 맨 앞의 08048000-0809d000 는 읽기와 실행이 가능한 영역으로 보아 .text 영역임을 알 수 있다. 두번째는 .data 영역이며, 세번째는 .bss 영역, 마지막은 stack 영역을 의미한다. 참고로 Stack 영역의 위 c0000000 부터는 Kernel Space 다.
이제 실제 코드를 작성하여 각각의 변수들이 어떤 위치에 자리잡고 있는지 확인해 보자.

```
 1 #include<stdio.h>
 2
 3 int no_value_glob_int;
 4 int init_glob_int = 5;
 5
 6 char *string ="This is a string in .data section!";
 7 char *dynamic_val;
 8
 9 main()
10 {
11     int local_int = 5;
12     int no_value_local_int;
13     dynamic_val = (char*)malloc(24);
14     strcpy(dynamic_val, "U.U.U Workshop");
15
16     return 0;
17 }
```

실제로 수행되는 것이 없어 보이지만, 여러 변수들을 선언함으로써 각 변수들의 위치를 확인할 수 있게끔 작성하였다.

- 리눅스의 실행파일은 모두 ELF 의 형태를 갖는다. ELF 는 리눅스 시스템에서 사용하는 공유 라이브러리와 실행 파일을 위한 기본파일 형식이다. Executable and Linking Format 의 약어이다.

이러한 ELF 는 파일을 여러가지 형태로 분류하고 세그먼트를 나누는데, 그 중 몇가지 세그먼트들이 메모리에 그대로 올라간다. ELF 를 기준으로 하여 각 메모리 영역을 살펴보겠다.

@ .text section

모든 실행파일과 공유 라이브러리는 "텍스트" 세그먼트 하나와 "데이터" 세그먼트 하나를 포함한다. 이 섹션의 이름이.text 인 이유는 메모리에 올라올 때 읽기 전용 접근 허가만 허용하는 섹션이기 때문이다. 이 .text 세그먼트의 경우는 프로그램 실행시에 로더에 의해 .data 세그먼트와 같이 메모리에 올라가게(loading) 된다. 프로그램의 실행 코드가 이 영역에 포함되고, 모든 컴파일된 함수가 이 섹션에 포함된다. (objdump –d <파일이름>을 통해 내용을 확인할 수 있다.) 이제는 실제 프로그램의 VMA 를 확인하여 어떻게 메모리에 적재되어 있는지 확인해보자.

```
[lazyhack@localhost main]$ readelf -S mem | egrep text
  [12] .text             PROGBITS        08048350 000350 000174 00  AX  0   0 16
```

```
[lazyhack@localhost main]$ cat /proc/19381/maps
08048000-08049000 r-xp 00000000 03:02 294928     /home/lazyhack/main/mem
08049000-0804a000 rw-p 00000000 03:02 294928     /home/lazyhack/main/mem
40000000-40015000 r-xp 00000000 03:01 424482     /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 424482     /lib/ld-2.2.4.so
4001b000-4001c000 rw-p 00000000 00:00 0
4001c000-40139000 r-xp 00000000 03:01 473434     /lib/i686/libc-2.2.4.so
40139000-40140000 rw-p 0011c000 03:01 473434     /lib/i686/libc-2.2.4.so
40140000-40144000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

그림을 보면 08048350 이 .text 영역의 위치인 것을 알 수 있다.

@ .data section

이 세그먼트는 쓰기가능한 초기화된 전역 변수와 정적변수를 저장한다. .text 세그먼트와 마찬가지로 프로그램이 실행될 때 .text 세그먼트와 함께 메모리에 적재된다.

```
[lazyhack@localhost main]$ readelf -S mem | egrep '₩.data'
  [21] .data             PROGBITS        08049670 000670 000014 00  WA  0   0  4
```

```
[lazyhack@localhost main]$ cat /proc/19381/maps
08048000-08049000 r-xp 00000000 03:02 294928     /home/lazyhack/main/mem
08049000-0804a000 rw-p 00000000 03:02 294928     /home/lazyhack/main/mem
40000000-40015000 r-xp 00000000 03:01 424482     /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 424482     /lib/ld-2.2.4.so
4001b000-4001c000 rw-p 00000000 00:00 0
4001c000-40139000 r-xp 00000000 03:01 473434     /lib/i686/libc-2.2.4.so
40139000-40140000 rw-p 0011c000 03:01 473434     /lib/i686/libc-2.2.4.so
40140000-40144000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

그림에서 2 번째의 메모리 영역이 .data 세그먼트이며 읽기와 쓰기가 가능한 영역인 것을 확인할 수 있다. 그럼 이 공간에 실제로 우리가 선언했던 변수가 있는지 확인해볼 수 있다. 이 공간은 초기화된 전역변수가 저장되므로 이전의 작성된 코드에서는 init_glob_int 와 string 이 이에 해당된다. 확인해보자. 다음과 같은 명령어로 확인가능했다.

: nm –v –f s mem | egrep '/.data'

```
init_glob_int      |0804967c|   D  |        OBJECT|00000004|     |.data
string             |08049680|   D  |        OBJECT|00000004|     |.data
```

주소값을 확인해보면 8049000-804a000 사이에 두 변수가 위치해 있는 것을 알 수 있다.

@ .bss section

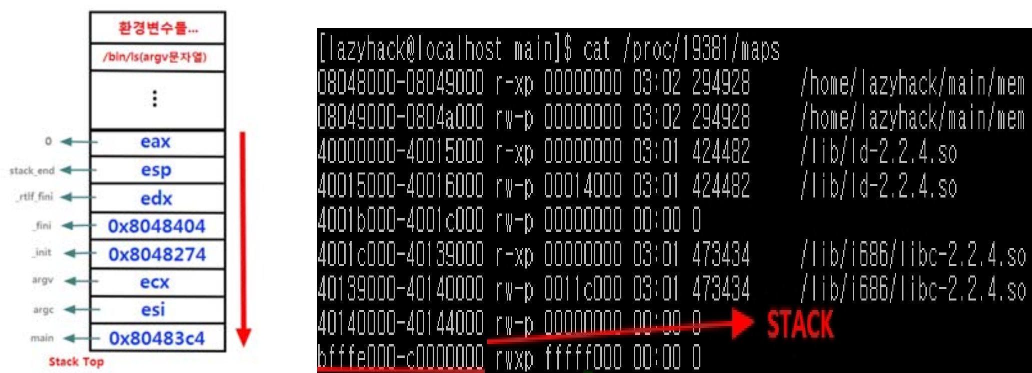
.bss 섹션은 Block Started by Symbol 의 약어이다. 이 공간에는 초기화되지 않은 전역변수가 저장되며, 프로세스가 시작할 때 0 으로 채워져 .bss 섹션에 속한 모든 변수 초기값은 0 이 된다.

```
[lazyhack@localhost main]$ readelf -S mem | egrep '\.bss'
  [22] .bss            NOBITS          08049684 000684 000020 00  WA  0   0  4
[lazyhack@localhost main]$ nm -v -f s mem | egrep '\.bss'
object.11          |08049684|   b  |        OBJECT|00000018|     |.bss
dynamic_val        |0804969c|   B  |        OBJECT|00000004|     |.bss
no_value_glob_int  |080496a0|   B  |        OBJECT|00000004|     |.bss
```

위의 실행 결과를 보고 dynamic_val, no_value_glob_int 가 .bss 영역에 포함되어 있는 것을 확인할 수 있다. (object.11 은 아직까지 무슨 역할을 하는 변수인지 확인하지 못했다.)

@ Stack section

일반적으로 스택은 지역변수, 환경변수를 저장하고, 프로그램 실행 시에 함수에서 사용하는 데이터 등을 임시적으로 저장하는 매우 중요한 공간이다. Stack 은 LIFO(Last In First Out)방식으로 동작한다. 다음 그림과 같이 확인할 수 있다.



Heap 공간에 대한 내용은 생략하겠다. 와우스토리의 'Heap-based Overflow for baby'에 Heap 에 관한 내용이 자세히 설명되어 있다.

6. 참고문서

[1] Operating System Inside by 이민 (http://osinside.net/)
[2] 와우스토리 'Heap-based Overflow for baby – by me'
[3] Operating System Principles by Silberschatz Galvin Gagne
[4] 리눅스 문제 분석과 해결 by 마크 윌딩 [에이콘 출판사]
[5] 리눅스 커널의 이해 [한빛미디어]
[6] 여러 인터넷 문서들;;

7. 끝으로

하루만에 후다닥 쓴 문서라 정리가 안되어있고 산만한 감이 있지만, 발표를 준비하면서 공부한 내용을 정리하고자 한 일이라 괜찮을 것 같습니다(응??). 과연 어디까지 깊게 들어갈까에 대한 의문이 많았지만, 그저 훑어보는 정도로 이정도에 그치는 것도 나쁘지 않을 것 같습니다. 정말 공부를 하면 할수록 끝이 나오지 않았습니다. 그래서 많은 부족함도 느꼈고, 흥미도 느꼈습니다. 운영체제는 왜이리 어렵고 방대한지...^^ 공개하고자 쓴 문서는 아니지만, 혹시 모르니;;
많은 도움 되셨으면 좋겠습니다.