4장. 프로세스 (Processes)

이 장에서는 프로세스가 무엇이며 리눅스 커널이 어떻게 프로세스를 만들고 관리하고 없애 는지를 설명한다.

프로세스는 운영체제 안에서 작업을 수행한다. 프로그램은 디스크에 실행 가능한 형태로 저 장되어 있는 기계어 명령과 자료의 집합인데, 이 자체는 수동적인 존재이다. 한편 프로세스 는 동작중인 프로그램으로 생각할 수 있다. 즉 프로세서가 기계어 명령들을 실행함에 따라 끊임없이 변화하는 동적인 존재이다. 프로그램의 명령어와 데이터 뿐만 아니라, 프로세스는 프로그램 카운터, CPU 레지스터, 그리고 루틴 인자, 복귀 주소, 저장된 변수같은 일시적 데 이터를 포함하는 프로세스 스택도 함께 가진다. 현재 실행 중인 프로그램, 즉 프로세스는 현 재 마이크로프로세서 안에서 일어나는 모든 동작을 포함한다. 리눅스는 멀티프로세싱 운영 체제이다. 프로세스는 각각 고유의 권한와 책임을 갖는 별개의 태스크이다. 어떤 프로세스 하나가 비정상적으로 종료했다고 해서 이것이 시스템 내의 다른 프로세스까지 죽게 하지는 않는다. 개별 프로세스는 자신의 가상 주소공간에서 실행되며, 커널이 제공하는 안전한 방법 을 통하지 않고서는 다른 프로세스와 상호작용할 수 없다.

프로세스는 살아 있는 동안 많은 시스템 자원을 사용한다. 명령을 수행하기 위해서 CPU를, 명령어와 데이터를 저장하기 위해서는 물리적인 메모리를 사용한다. 파일 시스템의 파일들 을 열고 사용할 수도 있고, 시스템 내의 물리적인 장치들을 직접 또는 간접적으로 사용할 수도 있다. 리눅스는 여러 시스템 자원을 관리하고 프로세스들을 공평하게 관리하기 위해서 프로세스 자신과 프로세스가 가지고 있는 시스템 자원에 대해 계속 추적하고 있어야 한다. 하나의 프로세스가 시스템의 물리적인 메모리나 CPU의 대부분을 독점한다면, 다른 프로세 스들에게 공평하지 않을 것이다.

시스템에서 가장 중요한 자원은 CPU로, 대부분의 시스템에는 하나밖에 없다. 리눅스는 멀티 프로세싱(multiprocessing) 운영체제인데, 그 목적은 각각의 CPU가 언제나 실행 중인 프로세 스를 갖도록 하여 CPU의 활용을 극대화하는 것이다. 프로세스의 수가 CPU보다 많은 경우 (대부분의 경우가 이렇다), 나머지 프로세스들은 실행되기 위해서 CPU가 사용 가능할 때까 지 기다려야 한다. 멀티프로세싱이란 간단한 개념이다. 즉, 프로세스는 무언가 기달려야 하 기 전까지는 (보통은 시스템 자원을 기다린다) 계속 실행되며, 기다리고 있다가 자원을 얻게 되면 프로세스는 다시 실행될 수 있다. DOS와 같은 유니프로세싱(uniprocessing) 시스템에서 는 CPU는 그냥 아무것도 하지 않고 대기 시간을 낭비한다. 멀티프로세싱 시스템에서는 동 시에 많은 프로세스들이 메모리 내에 존재한다. 프로세스가 무언가 기다려야 할 때마다 운 영체제는 CPU를 빼앗아 다른 좀 더 적당한 프로세스가 사용하도록 한다. 어떤 프로세스가 다음에 실행될 가장 적당한 것인지 선택하는 일은 스케쥴러의 몫이고, 리눅스는 공정을 기 하기 위해 여러가지의 스케쥴링 정책을 사용한다.

리눅스는 여러가지 형태의 실행 파일을 지원하는데, ELF, JAVA 등이 그 중 하나다. 이들은 프로세스가 시스템의 공유 라이브러리(shared library)를 사용할 수 있도록 하는 것과 같은 일 을 위해 투명하게 관리해야 한다.

4.1 리눅스 프로세스

리눅스가 시스템 내의 프로세스들을 관리할 수 있도록, 각각의 프로세스는 task_struct라 는 자료구조로 표현된다 (태스크와 프로세스는 리눅스에서 같은 의미로 사용된다). task 벡 터는 시스템에 있는 task_struct 구조를 가리키는 포인터들의 배열이다. 이는 시스템이 가질 수 있는 프로세스의 수가 task 벡터의 크기로 제한되어 있다는 것을 의미한다. 이 크 기의 기본값은 512개이다. 프로세스가 만들어지면 시스템 메모리에서 새로운 task_struct 가 할당되어 task 벡터에 추가된다. 현재 실행되고 있는 프로세스를 찾기 쉽게 하기 위해 서, 이를 current 포인터가 가리키고 있다.

일반적인 프로세스 뿐 아니라 리눅스는 실시간(real time) 프로세스도 지원한다[1]. 이 프로세스 들은 외부에서 발생하는 사건(event)에 매우 빨리 반응해야 하므로 (다시 말하면 실시간으로), 스케쥴러는 이들을 일반 사용자 프로세스와는 다르게 취급한다. task_struct 자료구조는 방대하고 복잡하지만, 내부 항목들을 여러개의 기능 영역으로 구분할 수 있다.

- **상태(State)** 프로세스는 수행되면서 주변 상황에 따라서 상태를 변경한다. 리눅스 프로세스 들은 다음과 같은 상태를 가진다[2].

  - **실행중(Running)** 프로세스가 실행중이거나(현재 프로세스이거나), 언제든지 실행할 수 있 는 준비가 되었음(시스템의 CPU 중 하나에 할당되는 것을 기다리고 있는 것)을 나타 낸다.
  - **대기중(Waiting)** 프로세스가 이벤트나 자원이 할당되길 기다리는 중임을 나타낸다. 리눅 스는 두가지 종류 - 인터럽트 허용(interruptible)과 인터럽트 금지(uninterruptible) - 의 프로세스 대기상태를 가지고 있다. 인터럽트가 허용되는 대기상태의 프로세스는 시그 널에 의해 인터럽트될 수 있고, 인터럽트가 금지된 대기상태의 프로세스는 하드웨어 를 직접 기다리면서 어떤 환경하에서도 인터럽트되지 않는다[3].
  - **중단됨(Stopped)** 프로세스가 중단된 경우로, 대개 시그널을 받았을 경우이다. 프로세스를 디버그할 때 이런 상태에 있다.
  - **좀비(Zombie)** 이것은 정지된 프로세스이지만, 어떤 이유때문에 여전히 task_struct 자 료구조를 task 벡터에 가지고 있는 경우이다. 용어에서 느낄 수 있듯이, 죽은 프로세 스이다.

- **스케쥴링 정보** 스케쥴러는 시스템에 있는 프로세스 중 어느 것이 가장 실행되기에 적당한지 를 공정하게 판단하기 위해 이 정보를 필요로한다.

- **식별자(Identifier)** 시스템의 모든 프로세스는 프로세스 식별자를 가지고 있다. 프로세스 식별 자는 task 벡터에 대한 인덱스는 아니고, 그냥 단순한 숫자이다. 모든 프로세스는 또한 사용자 식별자와 그룹 식별자를 가지고 있는데, 이것들은 이 프로세스가 시스템에 있는 파일과 장치에 대한 접근하는 것을 제어하는 데 사용된다.

- **프로세스간 통신** 리눅스는 전통적인 유닉스의 IPC 메커니즘인 시그널, 파이프, 세마포어와 함께, 시스템 V IPC 메커니즘인 공유 메모리, 세마포어, 메시지 큐 등을 지원한다. 리눅 스에서 지원되는 IPC 메커니즘에 대해서는 5장에서 설명한다.

- **연결(Link)** 리눅스 시스템에서, 다른 프로세스와 무관한 프로세스는 없다. 시스템의 모든 프 로세스는 - 최초의 프로세스를 제외하고 - 부모 프로세스를 가진다. 새로운 프로세스는 생성되는 것이 아니라 이전의 프로세스로부터 복사(copy), 혹은 복제(clone)된다. 프로세스 를 나타내는 task_struct는 모두, 부모 프로세스, 형제(sibling, 부모가 같은 프로세스 들) 프로세스, 자신의 자식(child) 프로세스들에 대한 포인터를 가지고 있다. pstree 명령 을 실행하여 리눅스 시스템에 실행중인 프로세스들간의 가족 관계를 볼 수 있다.

```
init(1)-+-crond(98)
        |-emacs(387)
        |-gpm(146)
        |-inetd(110)
        |-kerneld(18)
        |-kflushd(2)
        |-klogd(87)
        |-kswapd(3)
        |-login(160)---bash(192)---emacs(225)
        |-lpd(121)
        |-mingetty(161)
        |-mingetty(162)
        |-mingetty(163)
        |-mingetty(164)
        |-login(403)---bash(404)---pstree(594)
        |-sendmail(134)
        |-syslogd(78)
        `-update(166)
```

  더불어, 시스템 내의 모든 프로세스들은 init 프로세스의 task_struct 자료구조에서 시작하는 이중 연결 리스트로 연결되어 있다. 이 리스트는 리눅스 커널이 시스템 내의 모든 프로세스들을 들여다볼 수 있게 한다. ps나 kill 등의 명령을 지원하려면 이렇게 할 필요가 있다.

- **시간과 타이머** 커널은 프로세스의 생성시간과 살아있는 동안 소비하는 CPU 시간 등을 계속 추적한다. 커널은 매 클럭 틱(tick)마다, 현재 프로세스가 시스템 모드와 사용자 모드에서 사용한 시간의 양을 jiffies 단위로 갱신한다. 리눅스는 또한 간격 타이머(interval timer)도 지원하는데, 프로세스는 시스템 콜을 사용하여 타이머를 설정하고 지정한 시간이 지나면 자신에게 시그널을 보낼 수 있도록 한다. 이 타이머는 한번만 발생하는(single- shot) 타이머일 수도, 주기적으로 발생하는 타이머일 수도 있다.

- **파일 시스템** 프로세스는 원할 때 파일을 열고 닫을 수 있으며, task_struct에는 각 열린 파일의 기술자(descriptor)에 대한 포인터와, 두개의 VFS inode 포인터를 가지고 있다. VFS inode는 각각 파일 시스템에 있는 파일이나 디렉토리를 유일하게 기술하는 것으로, 하부 파일 시스템에 대한 동일한 인터페이스를 제공하는 것이다. 리눅스가 파일 시스템을

어 떻게 지원하는지는 9장에서 설명한다. 첫번째 VFS inode는 프로세스의 루트(홈 디렉토리) 를 가리키고, 두번째 것은 pwd 디렉토리라고도 불리는 현재 디렉토리이다. pwd는 유닉스 명령어인 pwd에서 유래된 것으로, print working directory(작업 디렉토리를 출력하라)의 약 자이다. 이 두 VFS inode에는 count 항목이 있어서, 몇 개의 프로세스가 그들을 참조하 고 있는지를 나타낸다. 따라서, 어떤 디렉토리나 그 디렉토리의 하위 디렉토리가 한 프로 세스의 pwd 디렉토리로 설정되어 있다면 그 디렉토리를 삭제할 수 없다.

- **가상 메모리** 대부분의 프로세스(커널 스레드와 데몬을 제외한)는 가상 메모리를 가지며, 리 눅스 커널은 이 가상 메모리가 시스템의 실제 메모리 어디와 연결되어 있는지를 추적해 야 한다.
- **프로세서 고유 컨텍스트(Processor Specific Context)** 프로세스는 시스템의 현재 상태의 총합 으로 생각할 수 있다. 프로세스는 실행될 때마다, 프로세서의 레지스터와 스택 등을 사용 한다. 이것이 프로세스 컨텍스트이며, 프로세스가 중단될 때 CPU 고유의 컨텍스트들은 모두 그 프로세스의 task_struct에 저장되어야 한다. 스케쥴러가 이 프로세스를 다시 시작할 때, 이 컨텍스트는 이 정보로부터 복구된다.

4.2 식별자(Identifiers)

리눅스는 다른 유닉스들과 같이 시스템에 있는 파일과 이미지에 대한 접근 권한을 검사하기 위해서 사용자 식별자와 그룹 식별자를 사용한다. 리눅스 시스템의 모든 파일들은 소유권과 접근 권한을 가지며, 접근권한은 사용자들이 파일이나 디렉토리에 대한 접근 방식을 다룬다. 기본적인 권한들은 읽기, 쓰기와 실행으로 파일의 소유자, 특정 그룹에 속하는 프로세스들, 시스템의 모든 프로세스들의 세가지 종류의 사용자에 할당된다. 각각의 사용자 계층은 각기 다른 권한을 가질 수 있다. 예를 들면, 어떤 파일에 대해서 소유자는 읽기와 쓰기를 할 수 있지만, 그룹은 읽기만 할 수 있고, 시스템의 다른 프로세스들은 접근하지 못하도록 할 수 있다.

REVIEW NOTE : 추가하여 비트를 할당하는 것을 (777) 설명하라.

그룹은 리눅스에서 한명의 개별 사용자나 시스템의 모든 프로세스들이 아닌, 사용자들의 모 임에 파일이나 디렉토리의 권한을 주는 방법이다. 예를 들면, 소프트웨어 프로젝트에 참가하 는 사람들을 하나의 그룹으로 만들고 이 사람들만 프로젝트의 소스 코드를 읽고 쓸 수 있도 록 할 수 있다. 하나의 프로세스는 여러 그룹에 속할 수 있고 (기본값은 최대 32개[4]) 이것들은 각 프로세스의 task_struct에 있는 groups 벡터에 저장되어 있다. 프로세스가 속해 있는 그룹 중의 하나가 파일에 접근 권한을 가지고 있다면, 그 프로세스는 그 파일에 대한 해당 그룹 접근 권한을 가지게 된다.

프로세스의 task_struct에는 네 쌍의 사용자 식별자와 그룹 식별자가 있다.

- **uid, gid** 프로세스를 실행시킨 사용자의 사용자 식별자, 그룹 식별자
- **효력(effective) uid, gid** 어떤 프로그램은 uid와 gid를 프로세스를 실행시킨 사용자의 것으로부 터 자신의 것(실행 이미지를 기술하는 VFS inode에 저장된 속성)으로 변화시킬 수 있다. 이러한 프로그램은 setuid 프로그램으로 알려져 있으며, 이런 프로그램은 특히 네트워크 데몬과 같이 다른 프로세스의 한켠에서 실행되고 있는 서비스의 권한을 제한하기 위한 유용한 방법이 된다[5]. 효력 uid와 gid는 setuid 프로그램의 uid와 gid이며, 원래의 uid와 gid 는 그대로 남는다. 커널은 특권 권한을 검사할 때 효력 uid와 gid를 검사한다.
- **파일 시스템 uid, gid** 이것은 효력 uid, gid와 거의 같으며, 파일 시스템의 접근 권한을 검사 할 때 사용된다. NFS 마운트된 파일시스템에서 사용자 모드인 NFS 서버가 파일을 접근 할 때 서버로서가 아니라 특정 프로세스로서 파일을 접근해야 하기 때문에 필요하다. 이 러한 경우에는 파일 시스템 uid와 gid만 변경된다 (효력 uid, gid는 변경되지 않는다). 이 렇게 함으로써 악의를 가진 사용자가 NFS 서버에게 kill 시그널을 보낼 수 있게 되는 것 을 막는다. kill 시그널은 특정 효력 uid와 gid를 가진 프로세스에게만 전달된다.
- **저장된(saved) uid와 gid** 이는 POSIX 표준의 요구사항에 따른 것이며 시스템 콜을 이용하여 프로세스의 uid와 gid를 바꾸는 프로그램이 사용한다. 원래의 uid와 gid가 바뀌어 있는 동 안 실제 uid와 gid를 저장하는데 사용된다.

4.3 스케쥴링(scheduling)

모든 프로세스는 어떨 때는 사용자 모드(user mode)로, 또 어떨 때는 시스템 모드(system mode)로 실행된다. 하드웨어가 이러한 모드를 지원하는 방법은 사용하는 하드웨어에 따라 다르지만, 일반적으로 사용자 모드에서 시스템 모드로 전환하거나 반대로 전환하는 안전한 메커니즘이 있다. 사용자 모드는 시스템 모드에 비하여 훨씬 적은 권한을 갖고 있다. 프로세 스는 시스템 콜을 할 때마다 사용자 모드에서 시스템 모드로 전환되어 계속 실행되게 된다. 이 시점에 커널은 프로세스의 다른 한편에서 실행된다. 리눅스에서 프로세스는 현재 실행 중인 프로세스를 선점하지 않는다 (non-preemptive). 즉, 자기가 실행되기 위하여 다른 프로세 스를 중단시킬 수 없다[6]. 각 프로세스는 어떤 시스템 이벤트가 발생하기를 기다려야만 할 때 CPU를 내놓아야겠다고 판단한다. 예를 들어, 프로세스는 파일에서 한 글자를 읽어오기 위하여 기다려야 할 때가 있다. 이 기다림은 시스템 콜 도중에 즉, 시스템 모드에서 발생한 다. 프로세스는 파일을 열고 읽기 위하여 라이브러리 함수를 사용하며, 이를 위하여 차례로 열린 파일에서 글자를 읽는 시스템 콜을 호출한다. 이 경우에 기다려야 하는 프로세스는 일 시 중단이 되고 다른 실행될 만한 프로세스가 선택되어 실행된다.

프로세스는 항상 시스템 콜을 호출하며 따라서 종종 기다리게 된다. 그럼에도 불구하고 어 떤 프로세스는 기다리게 될 때까지 너무 많은 CPU 시간을 사용할 수 있으며, 이러한 경우 에 리눅스는 선점형 스케쥴링(pre-emptive scheduling)을 사용한다. 이 정책에서는 각각의 프로 세스가 200ms 정도의 짧은 시간동안만 실행되며[7], 이 시간이 지나면 다른 프로세스가 선택되어 실행되며, 원래의 프로세스는 자신의 차례가 올 때까지 기다리게된다. 이런 작은 시간 의 단위를 타임 슬라이스(time-slice)라고 한다.

실행할 수 있는 프로세스 중에서 가장 실행할만한 가치가 있는 프로세스를 골라서 실행하는 것이 스케쥴러(scheduler)의 일이다. 실행가능한 프로세스는 CPU가 자신을 실행하길 기다린 다. 리눅스는 간단한 우선권에 기반한 스케쥴링 알고리즘을 사용하여, 현재 프로세스와 다른 프로세스 사이에서 실행할 놈을 고른다. 리눅스가 새로운 프로세스를 시키기로 하였다면, 현 재 프로세스의 상태와 프로세스와 관련있는 레지스터들, 다른 컨텍스트를 task_struct 자 료구조에 저장한다. 그리고 나서 실행할 새 프로세스의 상태를 복원(이것도 또한 프로세서 에 따라 다르다)하고 시스템의 제어권을 그 프로세스에게 넘겨준다. 스케쥴러가 시스템 내 의 실행가능한 프로세스들에게 공정하게 CPU 시간을 할당하기 위해서 각각의 프로세스에 대한 정보를 task_struct에 유지한다.

- **정책(policy)** 그 프로세스에 적용될 스케쥴링 정책이다. 리눅스 프로세스는 보통(normal) 프로 세스와 실시간(real time) 프로세스의 두 종류로 나누어 진다. 실시간 프로세스는 다른 모 든 프로세스들보다 높은 우선권을 갖고 있다. 만약, 실시간 프로세스가 실행 대기중이라 면, 이 프로세스가 항상 먼저 실행된다. 실시간 프로세스는 두 종류의 policy를 가질 수 있다. 하나는 라운드 로빈(round robin)이고, 다른 하나는 FIFO(first in first out)이다. 라운 드 로빈 스케쥴링에서는 실행가능만 각각의 실시간 프로세스들이 차례로 실행되고, FIFO 스케쥴링에서는 각각의 실시간 프로세스들이 실행 큐에 있는 순서에 따라서 실행되며 그 순서는 절대로 바뀌지 않는다.

- **우선권(priority)** 이값은 스케쥴러가 프로세스에 지정한 우선순위이다. 또한 이값은 프로세스 가 실행될 때, 프로세스가 실행될 수 있는 시간(jiffies 단위로)이다. 프로세스의 우선 순위는 시스템 콜이나 renice 명령을 사용해서 할 수 있다.

- **실시간 우선권(rt_priority)** 리눅스는 실시간 프로세스를 지원하며, 이것들은 시스템의 실시 간이 아닌 프로세스들보다 높은 우선순위를 갖도록 스케쥴링 된다. 이 항목은 스케쥴러 가 각각의 실시간 프로세스들간의 상대적인 우선순위를 지정할 수 있도록 한다. 실시간 프로세스들의 우선권는 시스템 콜을 사용해서 바뀔 수 있다.

- **카운터(counter)** 이 값은 프로세스가 실행될 수 있는 시간(jiffies 단위로)이다. 이 값은 프로세스가 처음 실행될 때 priority 값으로 설정되며, 클럭 틱에 따라서 줄어든다.

스케쥴러는 커널안에서 여러 몇몇 경우에 작동된다. 스케쥴러는 현재 프로세스를 대기큐에 넣은 다음이나, 시스템 콜이 끝난 직후, 프로세스가 시스템 모드에서 프로세스 모드로 돌아 오기 바로 전에 실행된다. 또 다른 경우는 시스템의 타이머가 현재 프로세스의 counter의 값을 0으로 설정한 경우이다. 스케쥴러는 실행될 때 다음과 같은 일들을 수행한다.

- **커널 작업(kernel work)** 스케쥴러는 하반부 핸들러(bottom half handler)를 실행하고, 스케쥴러 작업큐(task queue)를 처리한다. 이들 가벼운 커널 스레드들은 11장에서 자세하게 다루어 진다.

- **현재 프로세스(current process)** 현재 프로세스는 다른 프로세스가 선택되기 전에 처리되어야 한다.

  현재 프로세스의 스케쥴링 정책이 라운드 로빈이면 프로세스는 실행큐로 되돌아간다. 만약 태스크가 인터럽트를 허용(INTERRUPTIBLE)하고 이전에 스케쥴된 이후에 시그널 (signal)을 받았으면 실행중(RUNNING) 상태로 바뀐다. 현재 프로세스가 타임아웃되면, 이것은 실행중(RUNNING) 상태가 된다. 만약 현재 프로세스가 실행중(RUNNING) 상태이면, 그 상태가 유지된다. 프로세스들 중에서 상태가 실행중(RUNNING)이거나 인터럽트 허용(INTERRUPTIBLE)이 아닌 것들은 실행큐에서 삭제된다. 이것은 스케쥴러가 수행할 프로세스를 찾는 과정에서 이들을 제외한다는 의미이다.

- **프로세스 선택(process selection)** 스케쥴러는 실행큐에 있는 프로세스들 중에서 수행할만한 프로세스를 찾는다. (실시간 스케쥴링 정책을 따르는) 실시간 프로세스가 있으면, 이것들 이 보통의 프로세스들보다 높은 가중치를 갖는다. 보통 프로세스의 가중치는 counter의 값이지만 실시간 프로세스는 counter에 1000을 더한 값이다. 따라서, 시스템에 실행가 능한 실시간 프로세스가 있으면 항상 실행가능한 보통 프로세스보다 먼저 실행된다. 주 어진 타임 슬라이스를 어느 정도 소모한 (즉 counter값이 감소한) 현재 프로세스는 시 스템의 같은 우선순위를 가진 다른 프로세스들보다 불리한데 이것은 당연하다. 여러개의 프로세스가 똑같은 우선순위를 갖으면, 실행큐의 보다 앞쪽에 있는 것이 선택된다. 현재 프로세스는 다시 실행큐로 되돌아간다. 많은 프로세스들이 같은 우선순위의 갖는 균형 잡힌 시스템에서는, 각 프로세스가 차례로 실행된다. 이것이 라운드 로빈 스케쥴링이다. 물론, 프로세스들이 자원을 필요로 하게 되므로, 실행 순서는 바뀌게 된다.

- **프로세스 교체(swap process)** 가장 실행할만한 프로세스가 현재 프로세스가 아니라면, 현재 프로세스는 중단되고 새로운 프로세스가 실행되어야 한다. 프로세스는 실행중에 CPU 레 지스터와, 시스템의 물리적인 메모리를 사용하게 된다. 프로세스가 루틴을 호출할 때마다 레지스터에 있는 인자들을 넘겨주며, 호출한 루틴으로 돌아오기 위한 주소 등의 값을 스 택에 저장해 두기도 한다. 따라서 스케쥴러가 실행될 때는 현재 프로세스의 컨텍스트 안 에서 실행되는 것이다. 특권 모드인 커널 모드에 있기는 하지만, 실행중인 것은 아직 현 재 프로세스이다. 이 프로세스가 중지될 때는 프로그램 카운터(PC)와 프로세서의 레지스 터 전부를 포함하여 모든 기계적인 상태가 프로세스의 task_struct 자료구조에 저장 되어야 한다. 그리고 나면 새로운 프로세스의 모든 기계적인 상태를 로드해야 한다. 이것 은 시스템 종속적인 작업으로, 어떤 CPU도 정확히 동일한 방식으로 이 일을 처리하지는 않지만, 대개는 이 작업을 위한 하드웨어적인 도움이 있다.

  이 프로세스 컨텍스트 교체는 스케쥴러가 마지막으로 하는 작업이다. 따라서, 이전 프로 세스의 저장된 컨텍스트는 이 프로세스가 스케쥴러의 마지막에 있는 하드웨어 컨텍스트 에 대한 순간사진이다. 마찬가지로, 새로운 프로세스의 컨텍스트가 로드 되었을 때, 그것 은 그 프로세스의 프로그램 카운터와 레지스터 내용을 포함하여 스케쥴러의 마지막 상태 를 보여주는 순간사진일 것이다.

  만약, 이전 프로세스나 새로운 현재 프로세스가 가상 메모리를 사용한다면, 시스템의 페 이지 테이블 엔트리를 갱신할 필요가 있다. 물론 이 행동도 아키텍쳐에 따라 다르다. 알 파 AXP와 같은 프로세서는, 변환 참조 테이블(translation look-aside table) 즉 캐시된 페이 지 테이블 엔트리를 사용하므로, 이전 프로세스에 속하는 캐시된 테이블 엔트리를 지워 야만 한다.

4.3.1 멀티프로세서 시스템에서의 스케쥴링

여러개의 CPU를 가진 시스템은 리눅스 세계에서 그리 흔하지 않은 것이다. 그러나 리눅스 를 SMP(Symmetric Multi-Processing, 대칭형 멀티프로세싱) 운영체제로 만드려는 작업이 상당 히 진척되었다. 이는 시스템내의 여러 CPU간에 작업량을 공정하게 분배하는 것이다. 공정한 분배가 가장 뚜렸이 나타는 곳은 스케쥴러이다.

멀티프로세서 시스템에서는 모든 프로세서가 바쁘게 어떤 프로세스들을 실행하고 있길 바란 다. 각 프로세서는 현재 프로세스가 타임 슬라이스를 다 소모하였거나, 어떤 시스템 자원을 기다려야 할 때마다, 독립적으로 스케쥴러를 실행한다. SMP 시스템에서 맨 먼저 주목할 점 은 시스템에 있는 idle 프로세스[8]가 단 하나가 아니라는 것이다. 하나의 프로세서가 있는 시스템에서는 task 벡터의 첫번째 태스크가 idle 프로세스이다. 반면에 SMP 시스템에서는 CPU마다 하나의 idle 프로세스가 있으며, 따라서 하나 이상의 idle CPU가 있을 수 있다. 게 다가 CPU마다 하나씩의 현재 프로세스가 있으므로, SMP 시스템에서는 각 프로세서별로 현 재 프로세스와 idle 프로세스를 관리하여야 한다.

SMP 시스템에서 각 프로세스의 task_struct에는 자신이 현재 실행되고 있는 프로세서 번호(processor)와 마지막으로 실행하였던 프로세서의 번호(last_processor)가 들어있 다. 어떤 프로세스를 실행하도록 선택할 때마다 다른 CPU에서 실행하지 못할 이유는 없지 만, 리눅스는 processor_mask를 이용하여 그 프로세스가 시스템의 특정 프로세서 또는 몇개의 프로세서에서만 실행되도록 제한할 수 있다. 만약 N비트가 설정되어 있으면 그 프로 세스는 프로세서 N에서만 실행될 수 있다. 스케쥴러가 실행할 새로운 프로세스를 고를 때 processor_mask에 현재 프로세서의 번호가 설정되어 있지 않은 프로세스는 고려하지 않 는다. 스케쥴러는 마지막으로 현재 프로세서에서 실행되었던 프로세스에게 약간 유리하게 해준다. 이는 한 프로세스를 다른 프로세서로 옮길 때 성능상의 오버헤드가 발생하는 경우 가 종종 있기 때문이다.

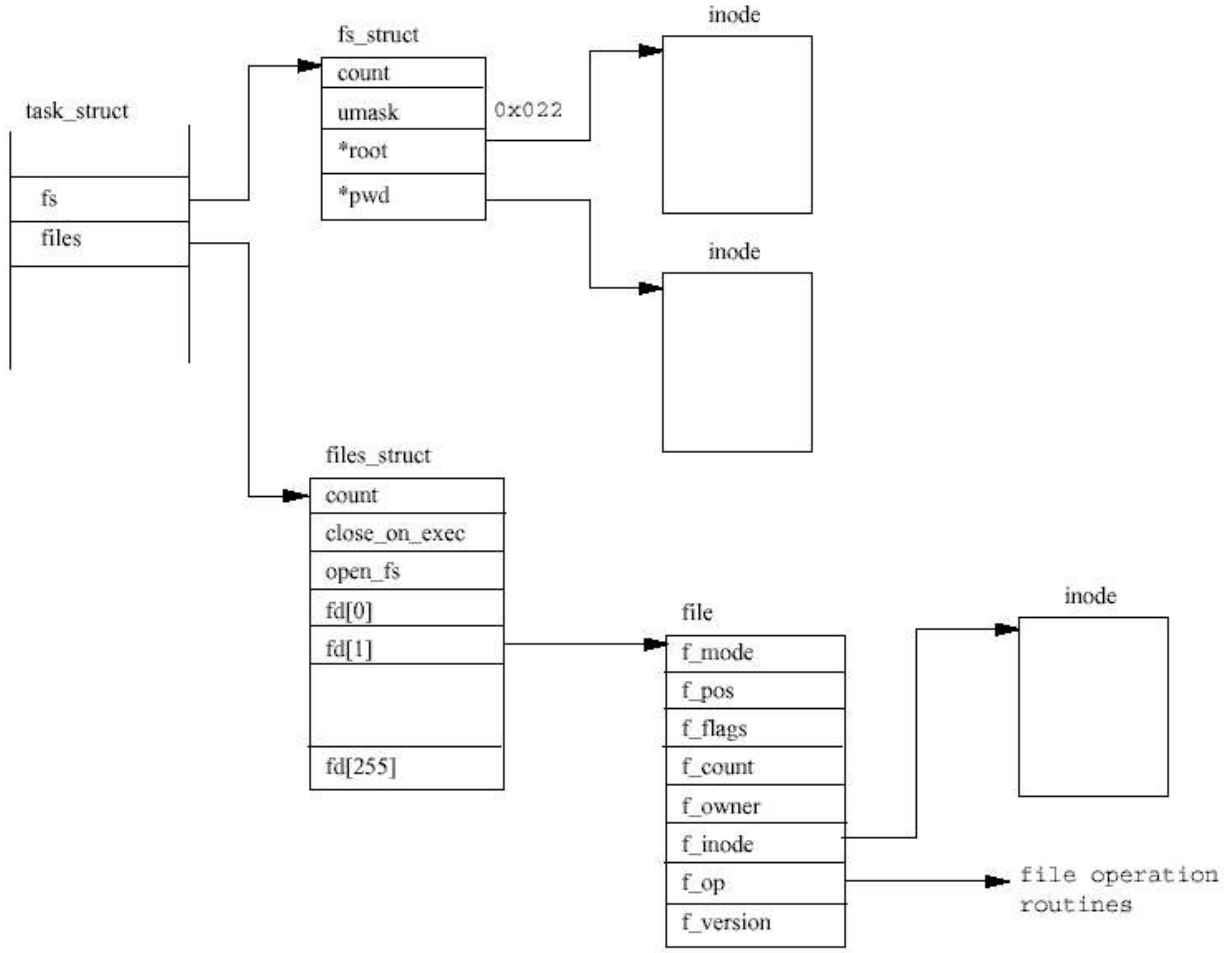

그림 4.1 : 프로세스의 파일

4.4 파일

그림 4.1은 각 프로세스의 파일 시스템 관련 정보를 저장하는 두가지 자료구조를 보여준다. 첫번째로, fs_struct는 이 프로세스의 VFS inode에 대한 포인터와 umask를 저장하고 있다. umask는 새로운 파일이 만들어질 때의 기본 모드이며 시스템 콜에 의해 바뀔 수 있다.

두번째 자료구조인 files_struct는 현재 프로세스가 사용하고 있는 모든 파일들에 대한 정보를 가지고 있다. 프로그램은 표준 입력(standard input)에서 읽고, 표준 출력(standard output)으로 쓴다. 에러 메시지는 모두 표준 에러(standard error)로 가게 된다. 이들은 파일일 수도 있고, 단말 입/출력이나, 실제 장치일수도 있으나, 프로그램에 있어서 이들 모두는 파 일로 처리된다. 각 파일은 자신을 나타내는 기술자(descriptor)를 가지며, files_struct는 이 프로세스가 사용하는 파일을 기술하는 file 자료구조에 대한 포인터를 256개까지 가진 다. f_mode 항목은 파일이 만들어질 때의 모드(읽기 전용, 읽고 쓰기, 쓰기 전용)를 나타낸 다. f_pos에는 다음 번에 읽거나 쓸 위치가 들어 있다. f_inode는 그 파일에 해당하는 VFS inode를 가리키고 있으며, f_ops는 그 파일에 대하여 무언가 하려고 할 때 사용할 수 있는 루틴들의 주소의 벡터를 가리킨다. 이런 함수로 데이터 쓰기 함수를 들 수 있다. 이렇 게 인터페이스를 추상화하는 것은 매우 강력하며 리눅스가 방대한 종류의 파일 유형을 지원 할 수 있도록 해준다. 뒤에서 살펴보겠지만 리눅스에서 파이프는 이러한 메커니즘을 통하여 구현되었다.

하나의 파일이 열 때마다 files_struct에 있는 빈 file 포인터 중 하나가 새로운 file 자료구조를 가리키기 위해 사용된다. 리눅스 프로세스는 처음 시작할 때 세개의 파일 기술 자가 열려 있다고 생각한다. 표준 입력, 표준 출력, 표준 에러가 그 세가지로, 이들은 대개 그 프로세스를 만든 부모 프로세스로부터 상속된다. 파일에 대한 모든 접근은 표준 시스템 콜을 통하며, 여기에 파일 기술자를 넘겨주거나 되돌려 받게 된다. 이들 기술자는 프로세스 의 fd 벡터에 대한 인덱스 값으로, 표준 입력, 표준 출력, 표준 에러는 각각 0, 1, 2 의 기술 자를 갖고 있다. 파일에 대한 접근은 file 자료구조의 파일 연산 루틴과 VFS inode를 같이 사용한다.

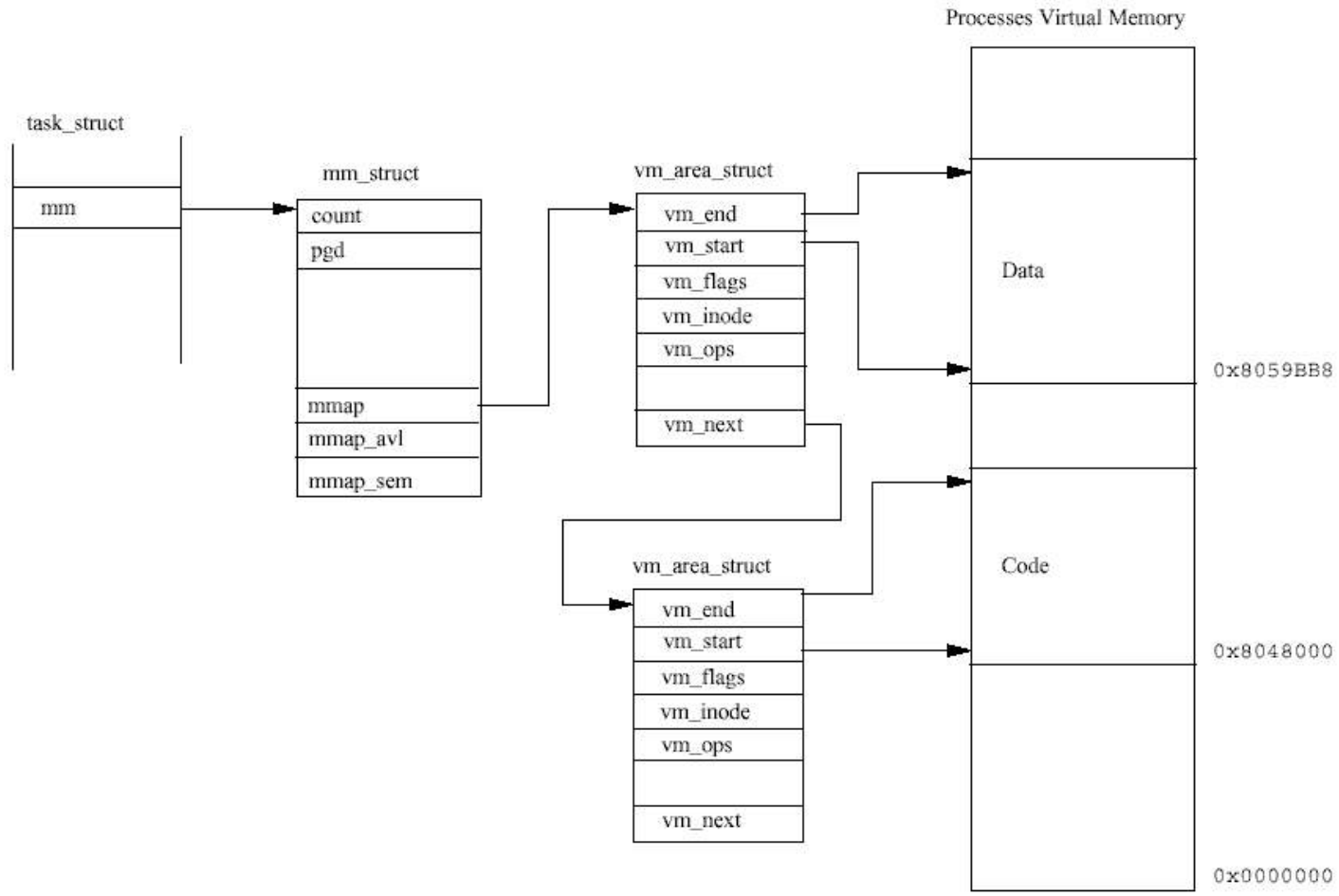

그림 4.2 : 프로세스의 가상 메모리

4.5 가상 메모리(Virtual Memory)

프로세스의 가상 메모리에는 여러 소스에서 나온 실행가능한 코드와 데이터가 들어 있다. 첫번째로, 로드된 프로그램의 이미지가 있다. ls 같은 명령을 예로 생각해보자. 이 명령은 다 른 실행 이미지와 마찬가지로 실행가능한 코드와 데이터로 구성되어 있다. 이미지 파일에는 실행가능한 코드와 해당되는 프로그램 데이터를 프로세스의 가상 메모리에 로드하기 위해 필요한 모든 정보가 들어 있다. 두번째로, 프로세스는 처리 과정에서 필요에 의하여 - 예를 들어, 읽고 있는 파일의 내용을 담기 위하여 - (가상) 메모리를 할당받을 수 있다. 이렇게 새 로 할당된 가상 메모리를 실제로 사용되기 위해서는 프로세스의 가상 메모리와 연결되어야 한다. 세번째로, 리눅스 프로세스는 파일 처리 루틴과 같이 공통적으로 유용하게 쓰이는 코드의 라이브러리를 사용하고 있다. 모든 프로세스가 똑같은 라이브러리의 복사판을 한개씩 갖고 있는다는 것은 말이 안되며, 리눅스는 실행되고 있는 여러 프로세스가 동시에 사용할 수 있는 공유 라이브러리를 사용한다. 이들 공유 라이브러리에 있는 코드와 데이터는 이 프 로세스의 가상 주소 공간에 연결되어야 할 뿐만 아니라, 그 라이브러리를 공유하는 다른 모 든 프로세스의 가상 주소 공간과도 연결되어야 한다.

어떤 주어진 시간 동안 한 프로세스는 가상 메모리에 들어 있는 코드와 데이터를 모두 사용 하지는 않는다. 코드 중에는 어떤 특정한 경우, 예를 들어 프로세스가 시작될 때 또는 어떤 이벤트가 발생할 때에만 필요한 코드가 있다. 그리고 공유 라이브러리의 루틴도 모두 사용 하는 것이 아니라 일부만 사용한다. 따라서 안 쓰일 수도 있는 코드를 실제 메모리에 모두 로드하는 것은 낭비가 될 수 있다. 이러한 낭비가 시스템내의 프로세스 수만큼 반복된다면 시스템은 매우 비효율적으로 실행될 것이다. 대신에 리눅스는 요구 페이징(demand paging)이 라는 기법을 사용한다. 요구 페이징에서는 프로세스의 가상 메모리를 사용하려고 하는 순간 에, 가상 메모리를 실제 메모리로 가져온다. 따라서 리눅스 커널은 프로세스의 코드와 데이 터를 곧바로 실제 메모리에 로드하는 대신, 프로세스의 페이지 테이블을 수정하여 가상 영 역에는 존재하고 있지만 실제로는 메모리에 있지는 않다고 표시한다. 만약 프로세스가 코드 나 데이터에 접근하려고 하면, 시스템은 페이지 폴트를 발생하고, 리눅스 커널로 하여금 그 상황을 해결하라고 제어권을 넘겨준다. 이러한 페이지 폴트를 해결하려면, 리눅스는 프로세 스의

주소 공간에 있는 모든 가상 메모리 영역에 대해, 그 가상 메모리가 어디에서 왔으며 어떻게 메모리에 로드할 수 있는 지를 알아야만 한다.

리눅스 커널은 이들 가상 메모리의 모든 영역을 관리할 필요가 있다. 각 프로세스의 가상 메모리의 내용은 task_struct에서 가리키고 있는 mm_struct라는 자료구조에 설명되어 있다. 프로세스의 mm_struct 자료구조는 로드된 실행 이미지에 대한 정보와 프로세스의 페이지 테이블에 대한 포인터도 갖고 있다. 여기에는 그 프로세스의 각 가상 메모리 영역을 나타내는 vm_area_struct 자료구조의 리스트에 대한 포인터도 들어 있다.

이 연결 리스트는 가상 메모리에서 오름차순으로 되어 있으며, 그림 4.2는 간단한 프로세스 에서 가상 메모리의 배치상황과 그것을 관리하기 위한 커널 자료구조를 보여준다. 가상 메 모리의 영역들은 여러 소스로부터 나오므로, 리눅스는 여러개의 가상 메모리 처리 루틴을 vm_area_struct에 있는 vm_ops를 통하여 가리키게 함으로써 인터페이스를 추상화 하였 다. 이렇게 함으로써 하부 서비스가 메모리를 여러가지 다른 방식으로 관리하는 것과 상관 없이 프로세스의 가상 메모리를 일관성 있게 처리할 수 있게 된다. 예를 들어, 여기에는 어떤 프로세스가 메모리에 접근하는데 그 메모리가 존재하지 않을 때 불리는 루틴이 들어 있다. 페이지 폴트는 이러한 방식으로 처리된다.

리눅스 커널은 이 프로세스 용으로 가상 메모리에 새로운 영역을 만들거나, 물리적 메모리 상에 있지 않은 가상 메모리에 대한 참조를 해결할 때, 이 프로세스의 vm_area_struct 자료구조 집합을 자주 액세스하게 된다. 따라서 올바른 vm_area_struct를 찾는 데 걸리 는 시간은 시스템의 성능에 큰 영향을 미친다. 이 액세스를 빠르게 하기 위하여 리눅스는 vm_area_struct 자료구조를 AVL(Adelson-Velskii and Landis) 트리의 형태로 정리해둔다. 이 트리에서는 각각의 vm_area_struct(즉, 노드)의 왼쪽 포인터와 오른쪽 포인터는 인접 하는 vm_area_struct에 대한 포인터이다. 왼쪽 포인터가 가리키는 노드는 더 낮은 시작 가상 주소를 갖고 있으며, 오른쪽 포인터가 가리키는 노드는 더 높은 시작 가상 주소를 갖고 있다. 맞는 노드를 찾을 때는 트리의 루트로부터 시작하여 찾으려는 vm_area_struct 를 찾을 때까지 왼쪽 또는 오른쪽 포인터를 따라간다. 물론 세상에는 공짜가 없기 때문에 새로운 vm_area_struct를 이 트리에 집어 넣는데에는 추가적인 처리 시간이 필요하다.

어떤 프로세스가 가상 메모리를 할당받을 때 리눅스는 실제 메모리를 진짜로 확보해 두지는 않는다. 대신 새로운 vm_area_struct 자료구조를 만들어 가상 메모리를 나타낸다. 이 자 료구조는 프로세스의 가상 메모리 리스트에 연결된다. 프로세스가 새로운 가상 메모리 영역 안의 어떤 주소에 값을 쓰려고 하면 페이지 폴트가 발생하게 된다. 프로세서는 가상 주소를 해석하려고 하지만, 이 메모리에 대해서 페이지 테이블 엔트리가 존재하지 않기 때문에, 프 로세서는 이를 포기하고 페이지 폴트 예외를 발생하며, 리눅스 커널이 이를 수정하도록 한 다. 리눅스는 참조된 가상 주소가 현재 프로세스의 가상 주소 공간에 있는지 찾는다. 그렇다 면 리눅스는 해당하는 PTE를 생성하고, 물리적 메모리 페이지를 할당한다. 코드나 데이터는 파일시스템이나 스왑 디스크로부터 물리적 페이지로 가져와야 할 수도 있다. 이제 프로세스 는 페이지 폴트를 발생한 명령에서부터 다시 시작할 수 있으며, 이제 메모리가 물리적으로 존재하므로 작업을 계속할 수 있다.

4.6 프로세스 생성하기

시스템이 처음 시작될 때 시스템은 커널 모드에 있으며, 초기 프로세스라는 단 하나의 프로 세스만 존재한다. 다른 프로세스들과 같이 초기 프로세스는 스택과 레지스터 등으로 대표되 는 기계 상태를 갖고 있다. 이것들은 시스템의 다른 프로세스들이 만들어지고 실행될 때, 초 기 프로세스의 task_struct 구조에 저장된다. 시스템 초기화의 마지막 단계에서, 초기 프 로세스는 init라고 하는 커널 쓰레드를 시작하고 아무일도 하지 않는 루프로 들어간다. 언 제나 다른 할 일이 없으면 스케쥴러는 이 idle 프로세스를 실행한다. idle 프로세스의 task_struct는 유일하게 동적으로 할당된 것이 아니고 커널이 생성될 때 정적으로 정의 된 것으로, 조금 혼란스럽겠지만 init_task라고 한다.

init 커널 쓰레드(또는 프로세스)는 시스템의 첫번째 진짜 프로세스로, 프로세스 식별자로 1을 갖는다. 이 프로세스는 시스템 초기화의 일부를 담당하고 (시스템 콘솔을 열고, 루트 파 일 시스템을 마운트하는 것 등), 시스템 초기화 프로그램을 실행한다. 시스템에 따라서 다르 지만 /etc/init, /bin/init, /sbin/init 중의 하나이다. init 프로그램은 시스템에 서 새 프로세스들을 만들기 위해서 /etc/inittab이라는 스크립트 파일을 사용한다. 이 새 프로세스들은 또 다른 프로세스들을 만들기도 한다. 예를 들면, getty 프로세스는 사용자가 로그인을 시도할 때 login 프로세스를 만들기도 한다. 시스템내의 모든 프로세스들은 init 커널 쓰레드의 자손이다.

새 프로세스들은 예전의 프로세스들을 복제하거나 현재의 프로세스를 복제하면서 생성된다. 새 태스크는 시스템 콜(fork나 clone)에 의해서 만들어 지며, 복제는 커널이 커널 모드에서 한다. 시스템 콜의 마지막에는 스케쥴러가 자신을 선택하여 실행하길 기다리는 새로운 프로 세스가 있게 된다. 새 task_struct 자료구조가 시스템의 실제 메모리에서 할당되고, 하나 또는 몇 개의 페이지가 복제된 프로세스의 스택(사용자와 커널) 용으로 할당된다. 시스템에 있는 식별자들 중에서 유일한 새로운

식별자가 만들어진다. 그리고 복제된 프로세스는 당연 하게도 부모 프로세스의 식별자를 가지고 있다. 새 task_struct가 task 벡터에 할당되고, 예전 (current) 프로세스의 task_struct의 내용이 복제된 task_struct에 복사된다.

프로세스를 복제할 때, 리눅스는 두 프로세스가 별도의 복사본을 사용하는게 아니라 자원을 공유하도록 한다. 프로세스의 파일들, 시그널 핸들러와 가상 메모리가 여기에 해당된다. 자 원을 공유할 때 이들의 count 값을 증가시켜 두개의 프로세스 모두가 자원 사용을 마치기 전에는 할당을 해제하지 못하도록 한다. 그래서, 예를 들어 복제된 프로세스와 가상 메모리 를 공유할 때, 이 프로세스의 task_struct는 원래 프로세스의 mm_struct에 대한 포인터 를 갖고, mm_struct의 count 값은 증가되어서 이를 공유하고 있는 프로세스의 개수를 나 타낸다.

프로세스의 가상메모리를 복제하는 데에는 좀 더 트릭을 사용한다. 새 vm_area_struct 자료구조들은 이들을 포함하는 mm_struct 자료구조와 복제된 프로세스의 페이지 테이블과 함께 만들어 져야 한다. 프로세스의 가상 메모리는 이 시점까지는 전혀 복사되지 않는다. 가 상 메모리의 일부는 실제 메모리에 있고, 또 다른 부분은 현재 실행중인 프로세스의 실행 이미지에 있으며, 어떤 부분은 스왑 파일에 있을 수 있으므로, 이것은 상당히 어렵고 시간을 소요하는 일이다. 대신에 리눅스는 "기록시 복사(copy on write)"라는 기술을 사용하는데, 이것 은 두 프로세스 중 하나가 기록을 시도할 때만 가상 메모리를 복사하는 것이다. 가상 메모 리 중에서 기록되지 않은 부분은 (설사 그것이 쓸 수 있는 영역이라고 하더라도) 아무 문제 없이 두 프로세스 사이에서 공유된다. 실행 코드와 같은 읽기 전용 메모리는 항상 공유된다. "기록시 복사"가 동작하기 위해서, 쓸 수 있는 영역들의 페이지 테이블 엔트리는 읽기 전용 으로 표시되고, 이를 나타내는 vm_area_struct 자료구조에는 "기록시 복사"라고 표시한다. 그러면 프로세스 중 하나가 이 가상 메모리에 쓰려고 하면 페이지 폴트가 발생한다. 이 때 리눅스는 메모리의 복사본을 만들고 두 프로세스의 페이지 목록과 가상 메모리 구조를 조정 한다.

4.7 시간과 타이머

커널은 각 프로세스의 생성 시간과, 프로세스가 사용한 CPU 시간을 관리한다. 각 클럭 틱마 다 커널은 현재 프로세스가 시스템 모드와 사용자 모드에서 사용한 시간의 양을 jiffies 단위로 계산하여 갱신한다.

이들 요금계산용 타이머에 외에도, 리눅스는 프로세스가 지정하여 사용할 수 있는 간격 타 이머를 지원한다. 프로세스는 이들 타이머를 이떤 시간이 지났을 때 자신에서 여러가지 시 그널을 보내는데 사용할 수 있다. 리눅스는 세가지 종류의 간격 타이머를 지원한다.

- **실제(Real)** 실제 시간으로서의 타이머 틱으로, 타이머가 만료되면 프로세스는 SIGALRM 시 그널을 받는다.
- **가상(Virtual)** 프로세스가 수행한 시간으로서의 타이머 틱으로, 만료되면 SIGVTALRM 시그널 을 받는다.
- **일람(Profile)** 프로세스가 수행한 시간과 프로세스의 다른 한편에서 시스템이 수행한 시간을 합친 타이머 틱으로, 만료되면 SIGPROF 시그널을 받는다.

하나 또는 모든 간격 타이머가 실행될 수 있으며, 리눅스는 프로세스의 task_struct 자료 구조에 필요한 모든 정보를 간직한다. 시스템 콜을 사용하여 이들 간격 타이머를 설정하고, 시작하고, 멈추고, 현재 값을 읽을 수 있다. 가상 타이머와 일람 타이머는 똑같은 방법으로 처리된다. 각 클럭 틱마다 현재 프로세스의 간격 타이머는 감소하며, 만료되면 해당하는 시 그널을 받는다.

실제 시간 간격 타이머는 다른 타이머들과는 약간 다르며, 리눅스는 이들을 위해 11장에서 설명하고 있는 타이머 메커니즘을 사용한다. 각 프로세스는 자신의 timer_list 자료구조 를 가지고 있으며, 실제 간격 타이머가 실행되고 있으면, 이를 시스템 타이머 리스트 큐에 넣는다. 타이머가 만료되면 타이머 하반부 핸들러는 이를 큐에서 제거하고 간격 타이머 핸 들러를 부른다. 이 핸들러는 SIGALRM 시그널을 발생하고, 새로 간격 타이머를 시작하여 이 를 다시 시스템 타이머 큐에 넣는다.

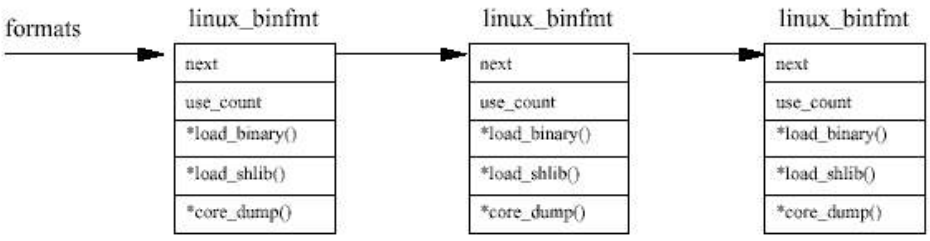

그림 4.3 : 등록된 이진 포맷들

4.8 프로그램 실행하기

유닉스와 마찬가지로 리눅스에서는 프로그램과 명령어들은 보통 명령어 해석기(command interpreter)에 의해 수행된다. 명령어 해석기는 다른 프로세스처럼 사용자 프로세스이며, 쉘 (shell)[9]이라고 불린다. 리눅스에는 여러가지 쉘이 있는데 가장 대중적인 것으로는 sh, bash, tcsh가 있다. cd나 pwd같이 적은 수의 내부에 직접 구현된 명령어들을 제외하고, 명령어들은 실행할 수 있는 이진 파일이다. 명령어가 입력되면 쉘은 환경변수 PATH에 저장된 프로 세스의 찾기 경로(search path)에서 같은 이름을 가진 실행 이미지를 찾는다. 파일을 찾으면 이를 로드하고 실행한다. 쉘은 앞에서 설명한 fork 메커니즘을 이용하여 자기자신을 복제한 후, 이렇게 만들어진 새로 만들어진 자식 프로세스는 이전에 실행하고 있던 이진 이미지를 (여기서는 쉘) 찾은 파일의 실행 이미지로 교체한다. 보통 쉘은 명령이 완료되길, 즉 자식 프로세스가 종료되기를 기다린다. 여기서 쉘이 이 자식 프로세스를 백그라운드로 돌려 실행 되게 할 수 있는데, 먼저 control-Z를 눌러서 자식 프로세스에게 SIGSTOP 시그널을 보내 멈추게 한다. 그리고 쉘 명령어인 bg를 사용하면 이를 백그라운드로 돌리고 SIGCONT 시그 널을 보내 다시 시작하게 한다. 이 프로세스는 종료하거나 터미널 입출력이 필요할 때까지 그대로 남아 있을 것이다.

실행 파일은 여러가지 포맷으로 되어 있을 수 있으며, 심지어 스크립트 파일도 가능하다. 스 크립트 파일로 인식했다면, 이를 처리할 수 있는 올바른 해석기를 실행해야 한다. 예를 들어 /bin/sh는 쉘 스크립트를 해석한다. 실행할 수 있는 오브젝트 파일은 실행 코드와 데이터 와 함께, 운영체계가 이를 메모리에 올리고 실행할 수 있도록 하는데 필요한 정보를 가지고 있다. 리눅스에서 가장 일반적으로 사용하는 파일 포맷은 ELF이지만, 리눅스는 어떤 오브젝 트 파일 포맷도 다룰 수 있을만큼 유연하게 되어 있다.

파일 시스템처럼 리눅스는 커널을 컴파일할 때 이진 포맷을 지원하는 것을 커널에 포함할 수도 있고 모듈로 로드할 수도 있다. 커널은 지원하는 이진 포맷의 목록을 관리하고 있다가 (그림 4.3참조), 파일을 실행하려고 하면 동작하는 것을 찾을 때까지 하나씩 각 이진 포맷을 시도해본다. 일반적으로 리눅스에서 지원하는 이진 포맷은 a.out과 ELF이다. 파일을 실행 할 때 파일을 모두 다 메모리로 읽어들일 필요는 없으며, 요구시 로딩(demand loading) 기술 을 사용하여, 프로세스가 실행 이미지의 각 부분을 사용할 때 이것을 메모리로 가져온다. 이 미지에서 안쓰이는 부분은 메모리에서 폐기된다.

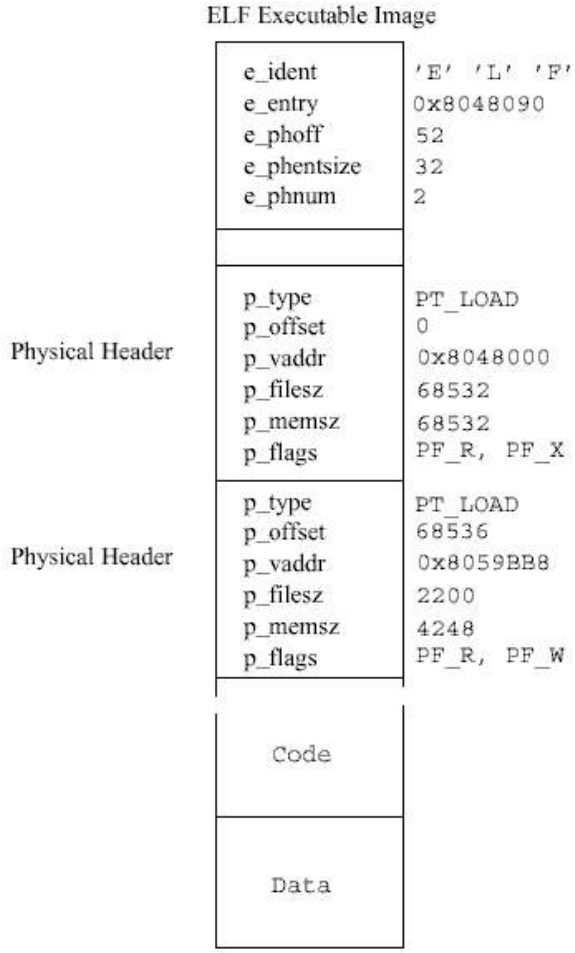

그림 4.4 : ELF 실행 파일 포맷

4.8.1 ELF

ELF (실행가능하고 링크할 수 있는 포맷 : Executable and Linkable Format) 오브젝트 파일 포맷 은 유닉스 시스템 연구소 (Unix System Laboratories)에서 디자인한 것으로, 이제는 리눅스에서 가장 일반적으로 사용하는 포맷이 되었다. ECOFF나 a.out같은 다른 오브젝트 파일 포맷과 비교하면 약간의 성능상의 오버헤드가 있지만, ELF는 좀 더 유연하다. ELF 실행 파일은 텍 스트(text)라고 부르는 실행 코드와 데이터(data)를 가지고 있다. 실행 이미지 안에 있는 테이 블은 어떻게 프로그램이 프로세스의 가상 메모리에 들어가야 하는지를 기술한다. 정적으로 링크된 이미지는 링커(ld)나 링크 편집기(link editor)같은 것을 이용하여, 하나의 이미지에 실 행하는데 필요한 모든 코드와 데이터를 가지고 있다. 이와 함께 이미지는 자신의 메모리에 서의 배치도와 처음 수행할 코드의 이미지 내의 주소를 지정하고 있다.

그림 4.4는 정적으로 링크된 ELF 실행 이미지의 배치도를 보여준다. 이것은 "hello world"를 출력하고 종료하는 간단한 C 프로그램이다. 헤더는 이것이 두개의 물리적 헤더(e_phnum이 2이다)가 이미지 파일의 처음을 기준으로 52바이트(e_phoff)에 위치하는 ELF 이미지라는 것을 이야기한다. 첫번째 물리적 헤더는 이미지에서 실행 코드를 기술한다. 이는 가상 주소 0x8048000에서 시작하고 65532 바이트를 갖는다. 이렇게 큰 이유는 이것이 정적으로 링크된 이미지여서, "hello world"를 출력하는 printf() 함수에 대한 라이브러리 코드를 모두 가지고 있기 때문이다. 이미지의 진입점(entry point), 즉 프로그램에서 처음 실행하는 명령은 이미지 의 시작주소가 아니라 가상 주소 0x8048090 (e_entry)이다. 이 코드는 두번째 물리적 헤더를 로드한 직후에 바로 시작된다. 이 두번째 물리적 헤더는 프로그램에서의 데이터를 나타 내고, 가상 메모리의 0x8059BB8 위치에 로드된다. 이 데이터는 읽거나 쓸 수 있다. 여기서 파일에서 데이터의 크기는 2200바이트(p_filesz)인데 반해, 메모리에서의 크기는 4248바이 트인 것을 눈치챈 사람도 있을 것이다. 이는 처음 2200바이트는 미리 초기화된 데이터를 가 지고 있지만, 다음에 있는 2048바이트는 실행 코드가 초기화할 데이터를 가지고 있기 때문 이다.

리눅스가 프로세스의 가상 주소 공간에 ELF 실행 이미지를 로드할 때, 실제로 이미지를 올 리는 것은 아니다. 리눅스는 단지 가상 메모리 자료구조인 프로세스의 vm_area_struct 트리와 여기에 속한 페이지 테이블들을 셋업하는 것이다. 그리고 프

로그램이 실행되면서 페 이지 폴트가 발생하면 프로그램의 코드와 데이터를 물리적인 메모리로 가져온다. 프로그램 에서 안쓰이는 부분은 절대 메모리에 로드되지 않는다. ELF 이진 포맷 로더는 자신이 실행 할 이미지가 ELF 실행 이미지가 맞다는 것을 확인하면, 프로세스의 가상 메모리에서 현재 실행 이미지를 쫓아낸다. 이 프로세스는 복제된 이미지이므로 (모든 프로세스가 마찬가지지 만), 이 옛날 이미지는 부모 프로세스가 실행했던 프로그램 - 예를 들어 bash같은 명령어 해석 쉘 - 일 것이다. 이렇게 옛날 실행 이미지를 쫓아내는 것은 옛날 가상 메모리 자료구 조를 없애고 프로세스의 페이지 테이블들을 리셋한다. 또한 설정되어 있는 모든 시그널 핸 들러를 지우고, 열려진 파일들을 모두 닫는다. 이 쫓아내기 과정이 끝나면 프로세스는 새로 운 실행 이미지를 받아들일 준비가 된다. 실행 이미지가 어떤 포맷이냐에 관계없이 프로세 스의 mm_struct 는 똑같은 정보로 셋업이 된다. 여기에는 이미지의 코드와 데이터의 시작 과 끝을 나타내는 포인터가 있다. 이 값들은 ELF 실행 이미지 물리적 헤더를 읽는 중에 발 견하게 되고, 이 헤더에서 기술하는 프로그램 섹션들은 프로세스의 가상 주소 공간에 맵핑 이 된다. 이는 vm_area_struct 자료구조를 셋업하고 프로세스의 페이지 테이블들을 수정 할 때도 마찬가지다. mm_struct 자료구조 또한 프로그램에 전달될 인자들에 대한 포인터와 프로세스의 환경 변수에 대한 포인터도 가지고 있다.

ELF 공유 라이브러리

한편, 동적으로 링크되는 이미지는 실행하는데 필요한 모든 코드와 데이터를 가지고 있진 않는다. 이들 중 일부는 실행시에 이미지와 링크되는 공유 라이브러리(shared library)에 들어 있다. ELF 공유 라이브러리의 테이블들은 실행시에 동적 링커가 공유 라이브러리를 이미지 와 연결할 때 사용한다. 리눅스는 여러개의 동적 링커를 사용한다. ld.so.1, libc.so.1, ld-linux.so.1. 이들 모두는 /lib에서 찾을 수 있다. 이 라이브러리는 언어 서브루틴 같 이 공통으로 사용하는 코드를 가진다. 동적 링크를 사용하지 않는다면 모든 프로그램은 이 들 라이브러의 복사본을 가지고 있어야 할 것이며, 훨씬 많은 디스크 공간과 가상 메모리를 필요로 할 것이다. 동적 링크에서 정보들은 ELF 이미지에 있는 참조하는 모든 라이브러리 함수들의 테이블에 들어 있다. 이 정보는 동적 링커에게 어떻게 라이브러리 루틴을 위치시 키고 프로그램의 주소 공간에 링크시킬지를 알려준다.

REVIEW NOTE : 실행 예제를 가지고 이를 더 자세히 설명할 필요가 있는가?

4.8.2 스크립트 파일(Script File)

스크립트 파일은 실행하는데 인터프리터(interpreter)를 필요로하는 실행파일이다. 리눅스에는 아주 다양한 인터프리터가 있다. 예를 들어 wish, perl이나 tcsh같은 명령쉘이 모두 인터프 리터이다. 리눅스는 인터프리터의 이름을 스크립트 파일의 첫번째 줄에 가지고 있는 표준 유닉스 표기법을 따른다. 따라서, 전형적인 스크립트 파일은 다음과 같이 시작한다.

```
#!/usr/bin/wish
```

스크립트 이진 로더는 이 스크립트를 처리할 인터프리터를 찾으려고 한다. 이것은 스크립트 의 첫번째 줄에서 말한 실행파일을 열려고 하는 것이다. 만약 이를 열 수 있다면, 이 프로그 램의 VFS inode에 대한 포인터를 가지고 스크립트 파일 해석을 시작할 수 있을 것이다. 스 크립트 파일의 이름은 인자 0번(프로그램에 전달되는 첫번째 인자)에 설정되고, 다른 모든 인자들도 한 칸씩 이동하게 된다 (원래 첫번째 인자였던 것이 두번째 인자가 되는 식이다). 인터프리터를 로드하는 것은 리눅스가 모든 실행 파일을 로드하는 것과 같은 방법으로 한다. 리눅스는 각 이진 포맷을 차례로 시도하여 동작하는 것을 찾는다. 이는 이론적으로 여러개 의 인터프리터와 이진 포맷들을 쌓아 올릴 수 있게 하며, 리눅스 이진 포맷 핸들러를 매우 유연한 소프트웨어로 만든다.

번역 : 윤경일, 고양우, 서창배, 이호, 정직한, 김기용
정리 : 이호

역주 1) 실시간이라는 의미는 어떤 사건이 발생하였을 때 이것이 어느 시간 이내에 처리되 어야 한다는 것이다. 즉 더 중요한 사건이 발생하면 덜 중요한 일은 그만두고 이를 빠른 시간 내에 처리하는 것이다. 이를 위해 실시간 처리를 하는 운영체제(real time operating system, RTOS)는 우선순위(priority)를 사용하여, 어떤 프로세스를 수행하고 있더라도 우선 순위가 더 높은 프로세스가 등장하면 하던 일을 멈추고 해당 프로세스를 수행하게 되며, 이 프로세스가 종료되거나 우선순위가 낮아지거나 더 높은 우선순위를 갖는 프로세스가 등장하지 않는 이상 계속 이 프로세스를 수행하게 된다. 이런 점에 있어서 리눅스는 실 시간 프로세스가 일반 프로세스보다 먼저 수행되긴 하지만, 실시간 프로세스를 위해 프 로세스를 중단하지 않고, 더 높은 우선순위의 프로세스라도 할당된 시간이 지나면 스케 쥴링이 되므로 RTOS라고 할 수는 없다. (flyduck)

2) REVIEW NOTE : SWAPPING 상태는 사용되지 않는 것 같아 제외했다.

역주 3) 세마포어를 기다리거나 파일을 읽을 수 있게 되길 기다리는 것처럼 자원을 기다리 는 일반적인 대기상태는 대개 인터럽트 가능한 상태이다. 인터럽트가 금지되는 대기상태 는 스왑파일에서 메모리로 페이지를 읽어들이는 것과 같이 임계지역에서 일이 끝마치길 기다리고 있는 상태이다. (flyduck)

역주 4) 이 값은 아래나오는 groups 벡터 크기에 의해 제한된다. 이 값은 NGROUPS로 정의 되어 있다. include/linux/sched.h 참조 (flyduck)

역주 5) setuid는 passwd같은 프로그램이 일반 사용자가 실행하였더라도 root의 권한을 획득 하여 /etc/passwd 또는 /etc/shadow 파일을 수정할 수 있게도 하지만, 반대로 웹서 버(httpd)같은 프로그램을 root가 실행하였더라도 nobody의 권한으로 바꾸어 다른 시스템 파일에 접근하지 못하게 하기도 한다. (flyduck)

역주 6) 이 말은 오해를 낳을 수 있는 말이다. 이 말은 리눅스가 비선점형 스케쥴링을 한다 는 것이 아니다. 뒤에 나오듯이 한 프로세스가 정해진 타임 슬라이스를 초과해서 사용하 면, 그 프로세스를 중단시켜 다른 프로세스를 실행하는 선점형 스케쥴링을 한다. 여기서 선점하지 않는다는 의미는 기다려야 하는 상황이 발생하여 멈추어야 하는 경우가 발생하 여 자발적으로 CPU를 내놓지 않은 이상 정해진 타임 슬라이스동안 계속 실행된다는 것 이다. 또한 리눅스는 커널 모드에서는 비선점형이다. 이는 커널 코드가 재진입가능하지 않게 만들어졌기 때문이다. 일단 시스템 콜이 불리면 시스템 콜이 자발적으로 CPU를 내 놓지 않은 이상 (schedule(), sleep_on(), interruptible_sleep_on() 등의 함수를 불러 스케쥴링이 일어나게 하지 않는 이상), 시스템 콜이 다른 프로세스에 의해 중단되지 않는다. (flyduck)

역주 7) 200ms는 0.2초로 CPU 입장에서 결코 짧은 시간이 아니다. 하지만 대개의 경우 프로 세스가 실행되는 동안 여러 I/O에서뿐만 아니라, 스왑파일에서 페이지를 읽는 것이나, 메 모리 맵된 파일을 디스크에서 메모리로 읽어들이는 것처럼 기다려야 하는 경우가 많이 발생하여 이 시간을 다 쓰는 경우는 많지 않다. (flyduck)

역주 8) idle 프로세스는 CPU가 할 일이 아무것도 없을 때 실행하는 프로세스이다. idle 프로 세스는 말 그대로 아무일도 하지 않고, CPU에서 가장 전력을 적게 소모하는 명령을 하염 없이 수행한다. (flyduck)

9) 땅콩을 생각해보면, 커널은 가운데 먹을 수 있는 부분이고, 쉘은 이를 둘러 싸고 있는 것 으로 인터페이스를 제공한다.