운영체제 과제5 : 프로세스및 메모리 관리조사

14010384 설민욱

이번 과제5에서 현재 사용하는 운영체제 중 하나를 선택해서 조사를 해야 하는데, 저는 Linux시스템을 조사해보았습니다. 자료는 "운영체제 Operating System Concepts(10판)"을 참고하였습니다.

Linux에 대한 프로세스 관리기법과 메모리 관리 기법을 알아 보기 전에 Linux에 대해 설명을 먼저 하겠습니다. Linux는 Unix의 한 버전으로 다중사용자(Multiuser), 다중 케스팅(Multitasking)시스템이며, Unix와 호환되는 툴들을 전부 가지고 있습니다. Linux시스템의 구성도를 보게 되면,

<table>
<tr><td>시스템 관리 프로그램</td><td>사용자 프로세스</td><td>사용자 유틸리티 프로그램</td><td>컴파일러</td></tr>
<tr><td colspan="4">시스템 공유 라이브러리</td></tr>
<tr><td colspan="4">Linux커널</td></tr>
<tr><td colspan="4">적재 가능 커널 모듈</td></tr>
</table>

* Linux시스템 구조도

위와 같은 구조로 이루어져 있습니다. 그 중에서 Linux커널 코드는 3개의 주요 부분으로 구성되어 있는데, 1. 커널: 가상메모리와 프로세스 등을 포함하는 운영체제의 핵심을 다루는 부분 , 2. 시스템 라이브러리 응용프로그램이 커널과 소통하는 함수들을 정의하며, 커널모드의 특권이 필요하지 않은 기능들을 구현한 함수들로 구성 , 3. 시스템 유틸리티 : 개별적으고 특수한 관리 기능을 수행하는 프로그램이며, 시스템 내에서 계속 돌아가면서 네트워크 연결 요청에 대한 응답, 터미널로부터의 로그인 요청 처리, 로그 파일들을 업데이트하는 일 등등을 맡음 로 나누어져 있습니다.

마지막으로 Linux를 제외한 다른 운영체제는 커널 내부가 메세지 전달 아키텍쳐를 채텍하고 있지만, Linux는 Unix의 전통적인 모델인 커널이 단일화된 실행파일로 생성되는 형태를 유지하고 있습니다. 이와 같은 특징을 가진 Linux에 대한 프로세스 관리 기법과 메모리 관리 기법을 알아보도록 하겠습니다.

1. 프로세스 관리 기법

1.1 fork() /exec() 프로세스 모델

Linux에선 프로세스 관리의 원칙으로 프로세스의 생성과 새로운 프로그램의 실행을 분리한다는 점입니다. 이 모델을 채택함으로써, 새로운 프로그램을 수행시키기 위한 환경을 시스템 콜에 자세히 명세할 필요 없이, 새로운 프로그램이 기존 환정에서 실행될수 있다는 장점을 가지고 있습니다. Linux에선 하나의 프로그램이 수행되는데 필요한 모든 정보를 몇 개의 부분으로 구분할 수 있는데, 프로세스의 식별(identity), 프로세스 환경(environment), 그리고 프로세스의 문맥(context)로 나눌 수 있습니다.

또, Linux의 프로세스 식별을 위해 제공되는 요소들이 존재하는데,

1. 프로세스 ID(PID) : 각 프로세스마다 가지고 있는 고유의 식별자

2. 신임장(credentials) : 각 프로세스마다 시스템 자원과 파일에 접근할 수 있는 권리를 표현하기 위해서 가지고 있는 ID

3. personality : 특정 시스템 콜의 시맨틱을 다소 변경할 수 있게 하는 식별자, emulation libraries에 의해 주로 사용되어집니다.

4. 이름 공간(namespace) : 각 프로세스에는 파일 시스템 체계(hierarchy)에 대한 고유의 관점이 연관되어 있으며, 이 것을 이름 공간이라 부르고 각 이름 공간은 고유의 파일 시스템 체계, 즉 그들이 각자의 디렉터리와 마운트된 파일 시스템 집합을 가집니다.

총 4가지의 요소로 이루어져 있습니다,

프로세스는 프로세스 환경을 가지고 있는데, 부모 프로세스로부터 상속받게 되며 두 개의 벡터(인자 벡터, 환경 벡터)를 가지게 됩니다. 인자 벡터는 프로그램을 실행하는데 필요한 사용자 명령어의 인자들이며, 환경 벡터는 "이름=값" 형태의 쌍들의 목록이며, 지명된 환경 변수와 그에 대응하는 값을 서로 연관시킵니다.

즉, Linux는 fork() /exec() 프로세스 모델을 채택하여 프로세스의 생성과 새로운 프로그램의 실행을 분리한다는 점을 지키고, 프로세스를 생성할 때, 프로세스 속성(로세스의 식별(identity), 프로세스 환경(environment), 그리고 프로세스의 문맥(context)), 프로세스 식별 정보(프로세스 ID(PID) ,신임장(credentials),personality,이름 공간(namespace)), 프로세스 환경(인자 벡터, 환경 벡터)를 부여받습니다.

1.2 프로세스와 스레드

Linux는 프로세스를 복사하는 전통적인 의미를 갖는 fork() 시스템 콜을 제공합니다. 또한 Linux는 clone() 시스템 콜을 사용하여 스레드를 생성할 수 있는 기능도 제공합니다. 하지만 Linux는 일반적으로 프로그램 내의 제어 흐름을 표현하는데 프로세스나 스레드가 아닌 태스크(task)용어

를 사용합니다. clone()이 호출될 때, 부모와 자식 간의 공유 정도를 결정하는 플래그의 집합을 전달받습니다. clone()에 대한 플래그 종류로 여러 개가 존재하는데,

1. CLONE_FS : 파일시스템 정보 공유

2. CLONE_VM : 동일한 메모리 공간 공유

3. CLONE_SIGHAND : 신호 처리기 공유

4. CLONE_FILES : 열린 파일 공유

총 4개가 존재하며, clone()이 위 4가지 플래그와 함께 호출이 되면 부모와 자식 task는 파일 시스템 정보, 같은 메모리 공간, 같은 신호 처리기, 그리고 동일한 열린 파일 등을 공유하게 됩니다. 위와 같은 방식으로 clone()을 사용하는 것을 다른 시스템에서 스레드를 생성하는 것과 동일합니다. 왜냐하면 부모와 자식은 대부분의 자원을 공유하게 되기 때문입니다.

Linux에서 프로세스와 스레드가 동시에 task로 불릴만큼 구분이 모호한 것은 Linux가 프로세스의 모든 문맥을 하나의 프로세스 자료구조에 모아두지 않고 독립된 하위문맥에 저장하기 때문에 task로 묶어서 부르는 것이 가능합니다.

1.3 스케줄링

Linux는 모든 UNIX와 마찬가지로 선점 가능 다중 태스킹을 지원합니다. 이러한 시스템으로 프로세스 스케줄러가 언제 어느 스레드를 수행할지 결정하며, 프로세스를 관리해줍니다. Linux에서는 다양한 커널 테스크의 실행이라는 면에서도 스케줄링이 매우 중요합니다. Linux에서 커널 태스크에는 일반 스레드에 의해 요청된 태스크와 장치 드라이버를 대신해 내부적으로 실행되는 커널 태스크 등이 포함됩니다. Linux는 좋은 스케줄링 성능을 내기 위해 여러 방법을 구현해두었습니다.

1.3.1 스레드 스케줄링

Linux는 두 종류의 프로세스 스케줄링 알고리즘을 가지고 있습니다. 하나는 시분할 알고리즘이고 다른 하나는 공정함보단 절대적인 우선순위가 훨씬 중요한 실시간 작업을 위해 고안된 알고리즘입니다. Linux는 UNIX 스케줄러가 가지고 있는 가장 큰 문제점인 SMP 시스템을 위한 자원이 부족하다는 점과 시스템의 태스크 개수가 많아지면 성능이 저하된다는 점과 대화형 태스크들에 공평성을 유지하지 못한다는 점을 해결하기 위해 완전 공평 스케줄러(CFS, Com-pletely Fair Scheduler)를 선도하였습니다.

Linux 스케줄러는 선점형, 우선순위 기반 알고리즘으로, 우선 순위를 실시간 영역(0~99), nice 영역(-20~19)로 두 가지 영역으로 나누었습니다. CFS는 UNIX 스케줄러가 가지고 있는 타임 슬라이스를 없에고 공평한 스케줄링(fair scheduling)이라는 스케줄링 알고리즘을 적용합니다.

만일 N개의 수행 가능 스레드가 있다면 CFS는 각 처리기가 1/N으로 처리기 시간 비율을 갖

는다고 가정합니다. CFS는 이어서 각 스레드의 할당 시간의 무게를 각자의 nice값으로 조정합니다. 기준이 되는 스레드의 nice값은 1의 무게를 갖고 nice가 1보다 작으면 더 큰 가중치를 1보다 크면 더 작은 가중치를 부여합니다. 다음으로 CFS는 각 프로세스의 무게를 수행가능한 모든 프로세스의 무게 값을 더한 값으로 나누고 이에 비례하게 스레드를 수행합니다. 이 과정을 거치기 전에, 우선적으로 CFS는 목표 지연(target latency)이라 불리는 설정 가능한 변수를 갖습니다. 목표지연이란 모든 수행가능 태스크가 최소한 한 번 수행되어애 하는 간격을 의미합니다. 그리고 최소 입도(minimum granularity)라는 두 번째 설정 가능 변수를 사용합니다. 최소 입도란 임의의 스레드가 받는 최소의 시간 할당량을 의미합니다. CFS는 스위칭 비용이 수용할 수 없을 정도로 커지는 것을 방지하도록 최소 입도를 설정합니다.

1.3.2 실시간 스케줄링

Linux의 실시간 스케줄링은 두 가지 실시간 스케줄링 클래스들을 구현하는데 FCFS와 라운드로빈을 구현합니다. 이 둘의 공통점으론, 스레드가 여러 개 있다고 가정을 한다면, 가장 오랫동안 기다린 스레드를 스케줄러는 실행합니다. 이 둘의 차이점으론, FCFS 스레드들은 종료하거나 봉쇄하기 전까지는 실행을 계속하나, 라운드-로빈 스레드들은 어느 정도의 시간이 지나면 선점되어 스케줄링 큐의 맨 끝으로 간다. Linux의 실시간 스케줄링은 연성 실시간 방식입니다. Linux의 실시간 스케줄링은 스레드들 간의 상대적인 우선운위에 대해서는 엄격한 보장을 제공하나, 스레드가 실행 가능하게 된 후, 그 실시간 스레드가 얼마나 빨리 스케줄 될지에 대해서는 보장을 하지 않습니다.

1.3.3 커널 동기화

2.6버전 이후부턴, Linux는 완전 선점형 커널로 동작합니다. 따라서 커널에서 실행 중일 때에도 태스크는 선점될 수 있습니다. 커널 안에 락킹을 위해서 Linux 커널은 spinlock과 세마포를 제공합니다. 하지만 처리기의 갯수에 따라 다릅니다. 단일 처리기에서는 spinlock을 사용하는 것이 부적합하여 커널 선점을 허용하고 금지하는 방법을 사용하고 있습니다. 즉, 단일 처리기에서는 스핀락을 획득하는 것이 아니라 태스크는 커널 선점을 금지합니다. 위 패턴을 아래 표에 나타나 있습니다.

단일 처리기	다중 처리기
커널 선점 금지	스핀락 획득
커널 선점 허용	스핀락 방출

Linux에서 커널 선점을 허용하고 금지하는데 preempt_disable()과 preempt_enable()의 간단한 시스템 콜을 사용합니다. 이 시스템 콜을 사용하기 위해서 각 태스크는 thread_info 자료구조에 preempt_count라는 필드를 가지고 있고 이 필드는 점유한 락의 개수를 나타냅니다. preempt_count가 0보다 큰 경우는 태스크가 락을 점유하고 있다는 것을 의미하고 preempt_count가 0이고 preempt_disable() 호출이 없다면 커널은 안전하게 선점될 수 있습니다.

1.3.4 대칭형 다중 처리(symmetric Multiprocessing)

Linux2.0 커널은 대칭형 다중 처리(SMP) 하드웨어를 지원하는 최초의 안전된 Linux커널입니다. 이로 인해 여러 스레드가 여러 처리기상에서 병렬적으로 실행될 수 있게 되었습니다. Linux2.2 커널에서는 다른 처리기에서 실행 중인 여러 스레드가 동시에 커널 내에 활성화 될 수 있도록 하기 위해, 단일 커널 스핀락(big kernel lock)을 도입합니다. 하지만 BKL는 입도 면에서 범위가 너무 넓어서 버전이 업그레이드 되면서 SMP 구현의 확정성을 향상시켰습니다.

이와 같은 여러 프로세스 관리 기법들을 사용해서 Linux에서의 프로세스는 관리받고 있습니다.

2. 메모리 관리 기법

Linux에서의 메모리 관리는 크게 두 부분으로 나누어져 있습니다. 하나는 물리 메모리(페이지, 페이지 그룹, 메모리의 작은 블록)을 할 당하고 반납하는 것을 다룹니다. 다른 하나는 가상 메모리를 다루는 일인데, 이는 실행 중인 프로세스들의 주소 공간으로 매핑(mapping)되는 메모리를 말합니다. Linux에서 메모리 관리 기법을 물리 메모리의 관리 부분과 가상 메모리 관리 부분으로 나누어서 설명하겠습니다.

2.1 물리 메모리의 관리

Linux에서의 물리 메모리는 4가지 zone 또는 영역들로 구분되어져 있습니다.

1. ZONE_DMA

2. ZONE_DMA32

3. ZONE_NORMAL

4. ZONE_HIGHMEN

총 4가지로 구분되어져 있으며, 이 ZONE들은 아키텍쳐마다 다릅니다.

Linux 커널에서 물리메모리 주 관리자는 페이지 할당기(page allocator)입니다. 각 zone은 자신의 할당기를 가지고 있는데, 이 할당기는 zone의 모든 물리 페이지들의 할당과 반납을 담당하고 있고, 요청에 따라 물리적으로 연속한 페이지들의 영역을 할당해줄 수 있습니다. 각ㄱ각의 할당 가능한 메모리 영역은 인접한 짝(partner), 즉 Buddy를 가지고 있는데, 할당된 두 ㅉ가이 모두 반납될 때(free up)마다 합쳐서 더 큰 영역(Buddy heap)을 형성합니다. 그 커진 영역은 다시 짝을 가지고 있어 합쳐져서 다 큰 가용 영역(free region)을 형성할 수도 있습니다. 또한, 존재하는 작은 가용 영역을 할당해서 메모리 요청을 만족시켜 줄 수 없다면 더 큰 가용 영역을 두 개로 자르게 됩니다.

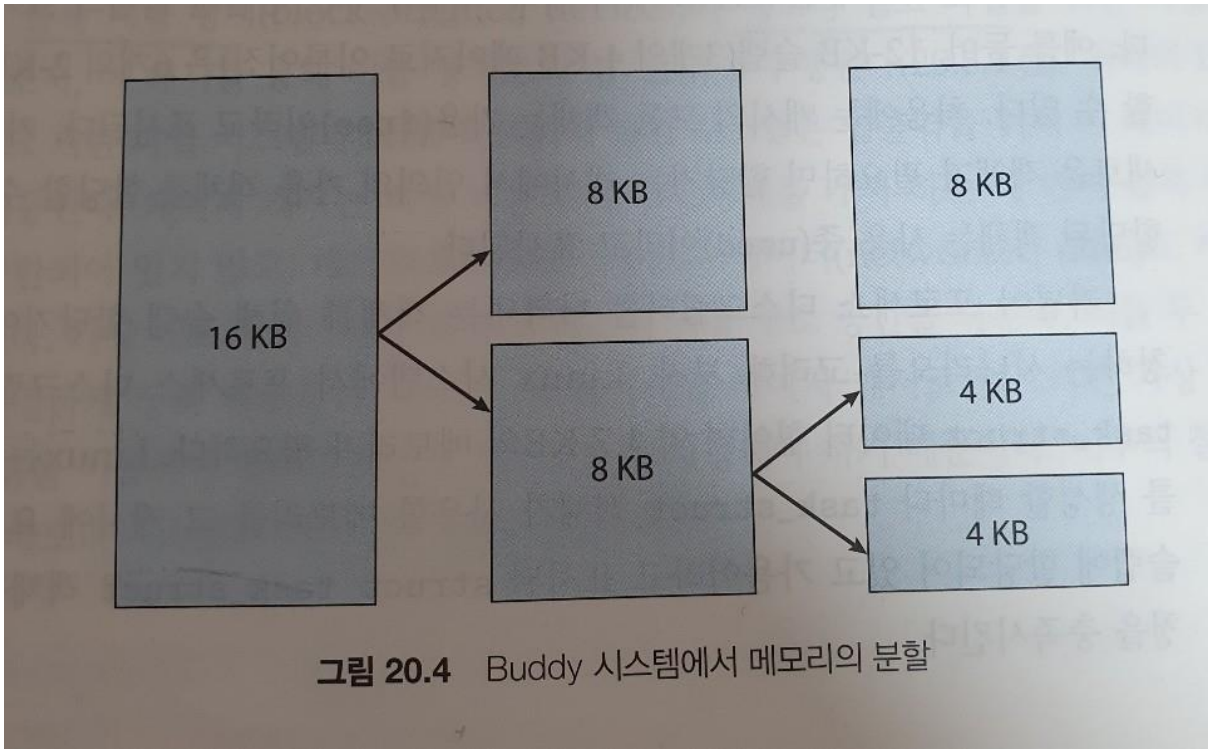

그림 20.4 Buddy 시스템에서 메모리의 분할

* Buddy 시스템에서 메모리의 분할 예 :

위 사진으로 예를 들지면 buddy-heap할당으로 8KB만 할당하려고 할 때면, 그 영역을 절반으로 나누어 메모리 요청에 만족시켜줄 수 있습니다. 만약 4KB만 할당하려고 할 때면, 그 영역을 두번 절반으로 나누어 메모리 요청에 만족시켜줄 수 있습니다.

Linux 운영체제 요소 대부분은 페이지 단위로 할당하기를 요구하지만, 더 작은 메모리 블록들을 요구하는 경우도 존재합니다. 메모리 할당 요청은 C에서의 malloc()과 비슷한 kmalloc()을 사용합니다. 메모리 할당이라는 것은 이처럼 적절한 리스트를 찾고, 리스트에서 사용 가능한 첫번째 가용 조각(free piece)을 가져오거나 또는 새 페이지를 할당하고 그것을 쪼개는 것까지 포함합니다. 하지만 kmalloc()시스템은 이 영역을 재할당(reallocate), 재사용(reclaim)할 수 없습니다.

커널 메모리를 할당하는데 다른 방법인 슬랩이 존재합니다. 슬랩은 커널 자료구조를 할당하는데 시용되며, 커널 자료구조마다 하나의 캐시가 존재합니다. Linux에서 슬랩은 3가지 상태를 가지게 됩니다.

1. Full : 슬랩의 모든 객체가 사용 중이라고 표시됩니다.

2. Empty : 슬랩의 모든 객체가 가용이라고 표시됩니다.

3. Partial : 슬랩은 가용인 객체와 사용 중인 객체로 이루어져 있습니다.

슬랩은 3가지 상태를 가지고 있게 됩니다.

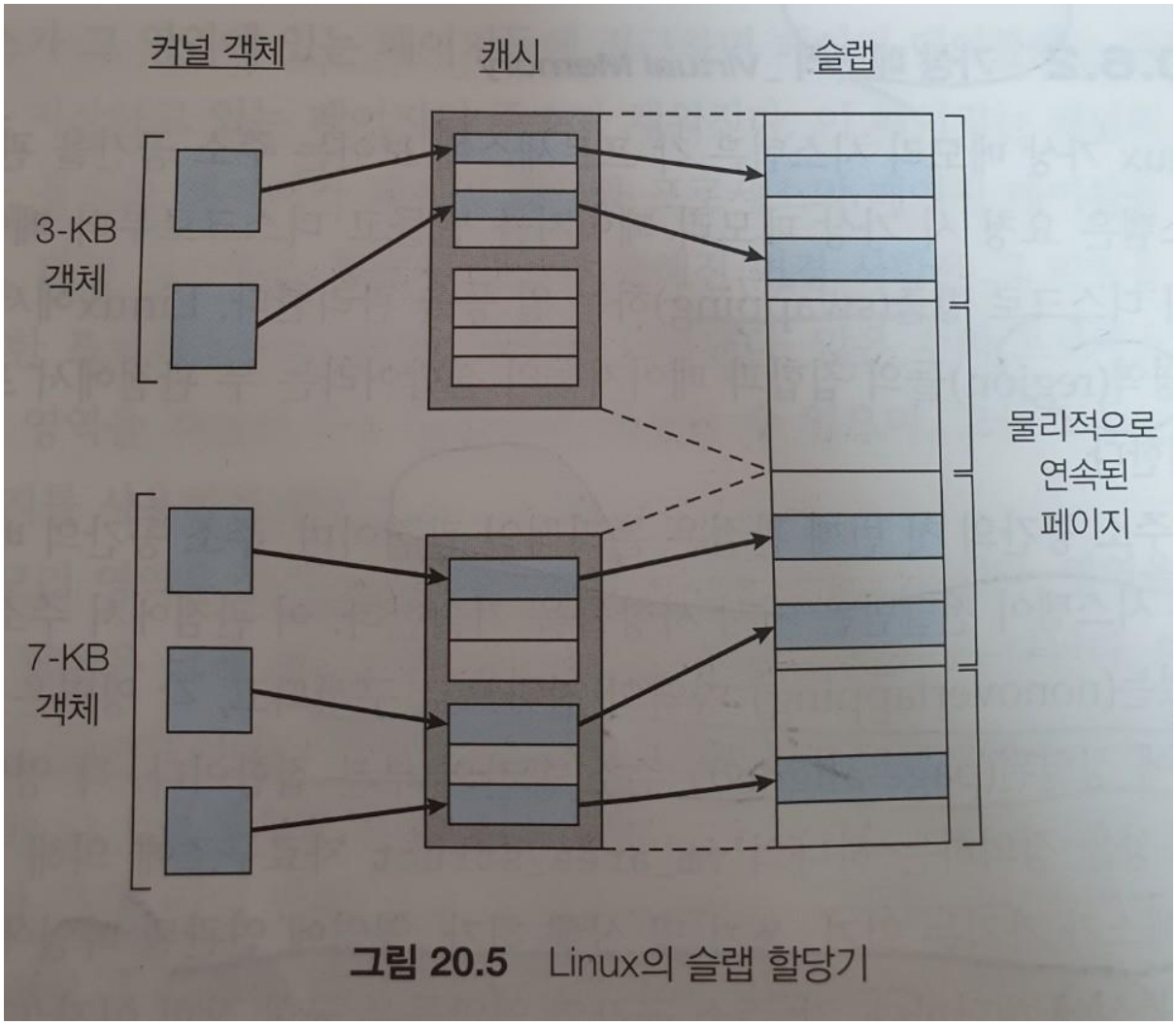

그림 20.5 Linux의 슬랩 할당기

위 사진은 크기가 3KB인 두 개의 커널 객체와 크기가 7KB인 세 개의 커널 객체가 존재하는 경우 Linux의 슬랩 할당기가 어떻게 구성이 되는지 보여줍니다.

슬랩 할당기의 실행 순서를 알아봅니다. 우선, 슬랩 할당기는 처음에 partial 슬랩에서 가용 객체를 찾으려고 시도합니다. 만일 가용 객체가 없으면 empty 슬랩에서 배정됩니다. 만일 empty슬립이 없으면 물리적으로 연속 페이지에서 새로운 슬랩이 할당되어 캐시에 배정됩니다. 객체를 위한 메모리는 이 슬랩에서 할당되게 됩니다.

Linux에서 물리 페이지를 자신이 직접 관리하는 나머지 두 서브 시스템이 있는데 이것은 페이지 캐시와 가상 메모리 시스템입니다. 페이지 캐시는 블록 지향 장치와 메모리 맵드 파일을 위한 커널의 주 캐시로서, 이 캐시를 통해 이들 장치로의 입출력이 수행됩니다. 가상 메모리 시스템은 각 프로세스의 가상 주소 공간을 다룹니다.

슬랩, 페이지 캐시, 가상메모리 시스템, Buddy 등등와 같은 여러 물리 메모리 관리 기법들을 통해 Linux에서의 물리 메모리가 관리되어 집니다.

2.1 가상 메모리의 관리

Linux에서 가상 메모리 시스템은 디스크로부터 페이지들을 적재하거나 다시 디스크로 방출(swapping)하는 일 등을 관리하며 각 프로세스에 보이는 주소 공간을 관리합니다. Linux에서 가상 메모리 관리자는 영역(region)들의 집합과 페이지들의 집합이라는 두 가지 관점에서 프로세스 주소 공간을 관리합니다.

주소 공간의 첫 번째 관점은 논리적인 관점이며, 이 관점에서는 주소 공간은 서로 겹쳐질 수 없는(nonoverlapping) 영역의 집합으로 구성되고, 각 영역은 연속적이면서 페이지 위치에 정렬된(page-aligned) 주소 공건의 부분 집합입니다. 각 영역은 특성을 정의하는 vm_area_struct 자료구조에 의해 기술되고 영역에서의 프로세스가 가지는 읽기, 쓰기, 및 실행 허가, 영역에 관련된 파일에 관련된 정보들은 균형 잡힌 이진트리로 연결되어 있어서 쉽게 영역을 빠르게 찾을 수 있습니다.

주소 공간의 두 번째 관점은 물리적인 면을 유지하는 것에 대한 관점이며, 프로세스가 읽거나 쓰려고 할 때 그 페이지가 페이지 테이블에 존재하지 않으면 소프트웨어-인터럽트 처리기에 의해 vm_area_struct 테이블에 있는 핸들러 함수로 dispatch됩니다. 존재하지 않는 페이지 테이블에 접근하려고 하면 소프트웨어-인터럽트 처리기에 의해 호출되는 루틴들의 집합에 의해 관리되어집니다.

Linux에서의 가상 메모리 영역을 구현하는데, 백업 저장 장소를 한 요소로 구현합니다. 가상 메모리 영역은 쓰기(write)의 처리 방식에 의해 구분되어지는데 개인전용(private)과 공유(shared)로 나눠집니다. 개인전용(private)으로 매핑된 영역을 쓰려면 페이저(pager)가 페이지 내용에 일어난 변화를 그 프로세스만 보게 하기 위해 쓰기 시 복사(copy-on-write)가 필요한지 파악해야 합니다. 하지만 공유(shared) 영역에서 사용할 때에는 그 영역으로 매핑되었던 객체 자체가 갱신됨으로써 그 객체를 매핑 중인 다른 모든 프로세스도 그 내용의 변화를 알 수 있게 됩니다.

Linux에서 커널이 새로운 가상 주소 공간을 만드는 경우는 두가지 상황 밖에 없습니다. 하나는 exec()시스템 콜을 통해 새 프로그램 코드를 실행할 때이고 새 프로그램이 수행될 때, 프로세스는 완전히 비어있는 새 가상 주소 공간을 받게 됩니다. 다른 하나는 fork() 시스템 콜을 통해 새 프로세스를 생성할 때이고 이 때에는 부모 프로세스의 가상 주소 공간을 완전히 복사하여 새로운 프로세스를 생성하게 됩니다.

2.1.1 스와핑과 페이징

초창기 UNIX시스템들은 swapping out으로써 이러한 이동 작업을 수행했으나, Linux는 프로세스 전체를 스와핑하는 방식을 사용하지 않고 오로지 새로운 페이징 기법만 사용하고 있습니다. 페이징 시스템은 두 부분으로 나뉘어집니다. 하나는 정책 알고리즘(policy algorithm)으로 어느 페이지를 디스크로 내보내야 할지와 언제 디스크로 내보내야 할지를 결정해야 하는 알고리즘입니다. 다른 하나는 페이징 기법(paging mechanism)으로 이 페이지들을 실제로 물리 메모리에서

디스크로 이동하는 작업도 담당하고 이 페이지들이 다시 필요하게 될 때, 이 페이지들을 물리 메모리로 옮겨오는 작업도 수행합니다.

Linux의 페이지아웃 정책은 다중-패스 클록(multiple-pass-clock)을 사용하는데, 모든 페이지는 이 클록이 통과함에 따라 조정되는 에이지(age)를 가지게 됩니다. age는 최근에 그 페이지가 얼마나 자주 사용되었느냐를 말해주는 척도입니다. 이러한 에이지를 사용함으로써 LFU(least frequently used)정책을 기반으로 하여 페이지 아웃 대상 페이지들을 선택하게 해줍니다.

2.1.2 사용자 프로그램의 로딩과 실행

Linux 커널은 exec() 시스템 콜을 통해 사용자 프로그램을 실행합니다. exec()를 호출하면 현좋나는 프로세스가 새 프로그램 문맥으로 바꾸어 새 프로그램 코드를 실행할 수 있게끔 만들어줍니다. Linux는 새로운 프로그램을 적재하기 위해 하나의 적재 루틴만을 가지고 있지는 않다. 대신 Linux는 여러 로더 함수들의 테이블을 가지고 있으며, exec() 시스템 콜이 수행될 때, 이들 각 로더 함수는 지정된 파일을 적재할 기회를 갖게 되는 것입니다. 이러한 로더 테이블이 필요한 이유는 Linux 바이너리 파일의 형식이 변경되었기 때문입니다. 최근에는 Linux 시스템은 더 최신의 포맷인 ELF를 사용합니다.

Linux에서는 이진 파일을 물리메모리로 적재하는 일이 이진 로더(binaary loader)에 의해 수행되지 않았습니다. 이진 파일의 페이지들은 먼저 가상 메모리 영역으로 매핑된 후 프로그램이 어떤 페이지에 접근하려 해서 페이지 폴트가 발생하면 그때 그 페이지가 메모리에 적재됩니다.

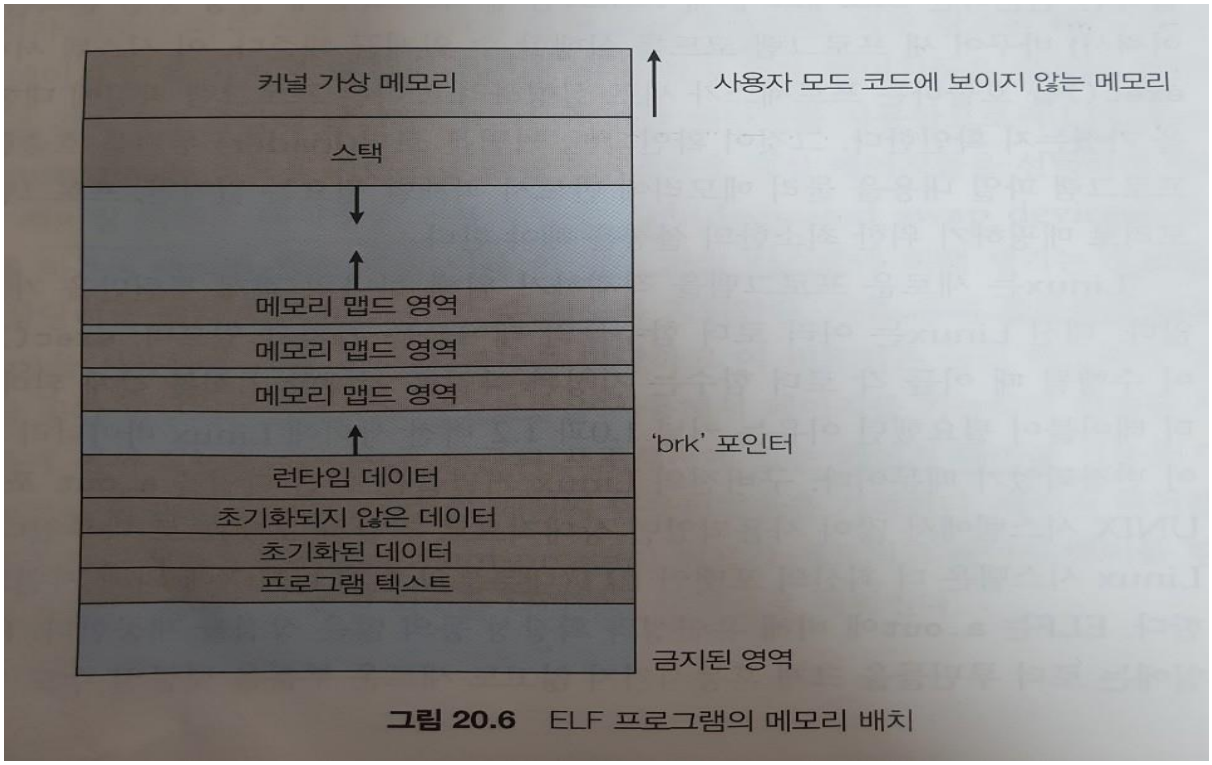

그림 20.6 ELF 프로그램의 메모리 배치

* ELF프로그램의 메모리 배치에 대한 예시 그림

위 사진은 ELF 로더에 의해 셋업된 메모리 영역의 전형적인 배치를 보여줍니다. 주소공강의 한쪽 끝의 영역에는 커널이 존재하는데, 일반 사용자 모드 프로그램은 이 영역에 접근할 수 없습니다. 가상 메모리의 나머지 부분은 응용 프로그램에 할당되는데, 응용 프로그램은 커널의 메모리 매핑 함수들을 사용하여 파일의 한 부분으로 매핑이 되는 영역이나 응용 프로그램 데이터를 쓸 수 있는 영역들을 생성할 수 있습니다. 가상메모리에서 top과 bottom이 생성되고 그 영역 안에서 자유롭게 매핑이 이루어질 수 있습니다.

위와 같은 많은 가상 메모리 관리 기법으로 Linux에서의 가상메모리 관리가 이루어지고 있습니다.