2000 년 한국정보처리학회 춘계 학술발표논문집 제 7 권 제 1 호

임베디드 시스템을 위한 LINUX 메인 메모리에 관한 연구

최지원*, 이동근*, 유재필* 김기천*
*건국대학교 컴퓨터공학과
e-mail : mailto:jackeroo@kkucc.konkuk.ac.kr; willow@kkucc.konkuk.ac.kr; dklee@kkucc.konkuk.ac.kr; kevinkckim@kkucc.konkuk.ac.kr

A Study on Linux Main Memory for Embedded System

Ji-Won Choi*, Dong-keun Lee*, Jae-Pil Yoo*, Kee-Cheon Kim*
*Dept. of Computer Science & Engineering, Kon-Kuk University

요 약

리눅스는 무료로 사용 및 배포가 가능한 유닉스 계열의 운영 체제이다. 본 논문에서는 리눅스 커널의 최신 버전 소스 중 메모리에 관련된 부분만을 페이징, 컨텐트와 메모리와의 메핑, 그리고 페이지 폴트와 페이징 폴트에 따른 스와핑에 관한 부분으로 나누어 분석하고 메모리상에서만 리눅스 운영체제가 동작할 수 있는 방안을 제시한다.

1. 서론

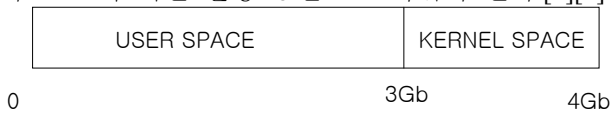
리눅스(LINUX) 시스템에서 메인 메모리 부분의 기능은 커널(kernel) 혹은 사용자에게 메모리라는 자원(resource)을 효율적으로 활용 가능하게 하는 환경을 제공하는데 있다. 리눅스 운영체제는 다중 플랫폼(Multi Platform)을 지원하고 있다. 따라서 실질적인 다양한 하드웨어를 수용하기 위하여 많은 커널 코드들이 다 기종에서 수용되는 추상화된 코드를 제공하고 있고 그 하부 적으로 플랫폼에 종속적인 코드들이 플랫폼 종속적인 특징을 수렴하고 있으며 상위 추상화된 코드에게 일관적인 인터페이스를 제공하고 있는 형태이다.[1] 분석 대상이 되는 코드는 크게 기능별로 구분하게 되면 메모리의 페이징(paging)과 컨텐트(contents)와 메모리와의 매핑, 그리고 페이지 폴트(Page fault)와 페이징 폴트에 따른 스와핑(swapping), 그리고 마지막으로 성능향상을 위한 캐싱(caching)부분으로 크게 나뉠 수 있다.

본 논문에서는 임베디드 시스템을 위한 리눅스의 메인 메모리를 구성하기 위하여 먼저 리눅스가 지원하는 다중 플랫폼 중 특별히 인텔계열(i386 : intel 386)에 의존적인 코드를 분석하였다.

2. 리눅스 메모리 구조 분석

2.1 부팅 과정

paging 이란 리눅스에서 실제 물리 메모리보다 더 큰 가상메모리(4GB)를 사용할 수 있도록 하는 메모리 관리 기법이다. i386 기반의 리눅스에서는 모든 프로세스는 각각 4GB 의 가상메모리 공간을 가질 수 있다. 가상 메모리는 보통 선형(linear) 메모리 공간으로 불려진다. 4GB 의 가상 메모리는 3GB 의 유저(user)공간과 1GB 의 커널 선형 공간으로 나뉘어 진다.[1][2]



<그림 1> 프로세스의 가상 메모리

4GB 의 가상 메모리의 실제 물리 메모리를 매핑 시키기 위해서는 paging 기법을 사용하며 이를 위해선 모든 프로세스는 각자 자신의 page table 을 가지게 된다. 리눅스 커널도 역시 page table 을 가지며 부팅과정에서 이를 초기화 한다. 리눅스 커널의 페이지 테이블은 1 페이지 4KB 의 크기로 되어 있다. 하나의 페이지 테

이블에는 모두 1024 개(1K)의 엔트리가 들어가게 된다. <그림 2>에서 0x1000 부터 0x2000 까지의 1 페이지크기의 페이지 테이블을 페이지 디렉토리라고 한다. 페이지 디렉토리는 모든 프로세스 당 1 개씩 있으며 각 프로세스의 4GB 의 선형 메모리 공간(가상 메모리)을 나타낸다. 페이지 디렉토리는 2-level 페이징에서 첫 단계의 페이지 테이블에 해당하며 두 번째 단계의 페이지 테이블은 그냥 페이지 테이블이라고 한다.

페이지 디렉토리의 각 엔트리에 들어가 있는 값들은 각 페이지 테이블의 시작 주소를 가리킨다. 페이지 테이블도 역시 1 개의 물리 메모리 페이지로 이루어져 1024 개의 엔트리를 갖는다. 하나의 페이지 테이블이 4MB 의 메모리 공간을 나타내므로 4GB 의 메모리 공간을 다 나타내기 위해서는 1024 개의 페이지 테이블이 필요하며 이 숫자는 바로 페이지 디렉토리의 모든 엔트리 개수와 같다.

<그림 2>에서 pg0 는 커널의 첫 번째 페이지 테이블의 시작 주소를 가리킨다. 즉, <그림 2>에서 0x1000 번지부터 0x2000 번지까지의 페이지(swapper_pg_dir)는 커널의 페이지 디랙토리가 되고, 0x2000 번지부터 0x3000 번지(pg0)까지는 커널의 첫 번째 페이지 테이블이 된다. 페이지 테이블의 각 엔트리에는 실제 매핑되는 물리 메모리의 페이지 넘버가 들어가며, 하위 12 비트에는 해당 페이지의 속성을 나타내는 값들이 들어간다. 보다 자세한 내용은 가상메모리 파트에서 다룬다.

커널의 paging 은 두 단계로 이루어진다. 첫 번째로 부팅 초기 단계에서 head.S 에 의해 물리 메모리의 처음 4MB 까지의 매핑이 이루어지며, 두 번째로 역시 부팅 과정에서 paging_init() 함수에 의해 나머지 물리 메모리에 대한 매핑이 이루어 진다. 초기 4MB 에 대

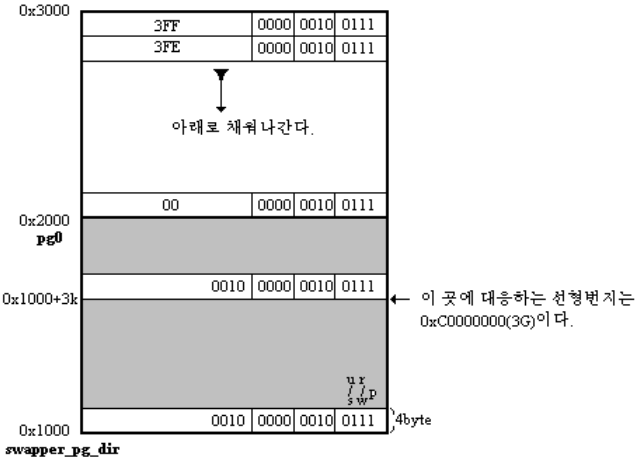
```
ENTRY(pg0)
.longx000007,0x001007,0x002007,0x003007,0x004007,0x00
5007,0x006007,0x007007
.longx008007,0x009007,0x00a007,0x00b007,0x00c007,0x00
d007,0x00e007,0x00f007
.longx010007,0x011007,0x012007,0x013007,0x014007,0x01
5007,0x016007,0x017007
                    .
                    .
                    .
.longx3f8007,0x3f9007,0x3fa007,0x3fb007,0x3fc007,0x3fd0
07,0x3fe007,0x3ff007
```

한 paging 의 과정은 어셈블리 코드로 되어 있으며 이에 대한 내용이 위에 나타나 있다.

아래의 <그림 2>은 head.S 에서 paging 이 일어난 후의 메모리 구조를 나타낸 것이다. swapper_pg_dir 은 커널의 페이지 디렉토리의 시작 주소를 가리킨다.

페이지 매핑을 위한 페이지 테이블도 역시 하나의 페이지로 나타나게 되는데 i386 에서는 page 의 크기는 4Kbyte 가 된다. 따라서 페이지 테이블의 크기도 4KB 가 된다. swapper_pg_dir 부터는 4 바이트의 크기의 엔트리가 들어가는데 보다 자세한 내용은 가상 메모리와 물리 메모리를 매핑하는 부분에서 자세히 다루도록 하겠다. head.S 에서는 물리 메모리의 처음 4MB 만 매핑하므로 페이지 디렉토리 테이블에는(swapper_pg_dir) 커널을 위한 엔트리와 유저 공간을 위한 엔트리, 이렇게 2 개의 엔트리가 들어가게 된다. 각 엔트리의 크기가 4 바이트이므로 하나의 페이지 테이블에는 1024 개의 엔트리가 들어간다. 커널 공간과 유저 공간의 구분은 1024 개의 페이지 디렉토리 테이블 엔트리 중에서 0 부터 767 개의 엔트리는 유저 공간을 나타내고(3GB), 768 개부터 1023 개 까지는 커널 공간을 나타낸다. 따라서 <그림 2>와 같은 형태로 페이지 디렉토리 테이블이 초기화 된다.

페이지 디렉토리 테이블의 엔트리들의 값은 각 페이지 테이블의 시작 주소를 가리킨다. 페이지 테이블의 엔트리에는 물리 메모리를(4MB) 페이지 크기인 4096 으로 나눈 값(주소)이 들어간다. 각 페이지 테이블의 엔트리에 있는 주소 값에서 하위의 12 비트는 실제 값이 아닌 속성 값이 된다. 즉 물리 메모리 페이지가 가지는 속성에 대한 정보를 이곳에서 가지고 있는 것이다. 따라서 엔트리의 값을 실제 물리 주소로 바꾸기 위해서는 하위 12 비트의 값을 모두 0 으로 놓고 계산한다. 예를 들면 0x1000 번지에 있는 이진수 0010000000100111 은 이진수 0010000000000000 으로 계산되어 0x2000 번지를 가리키게 되는 것이다.[2]



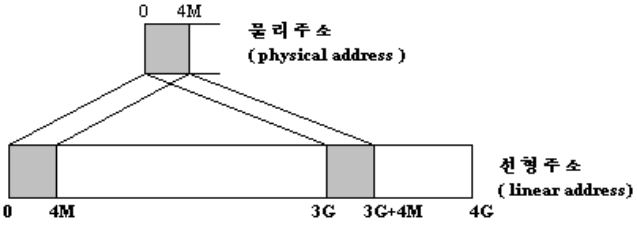
<그림 2> paging 초기화(head.S)

메모리에 대한 정보를 나타내는 변수는 다음과 같이 지정된다.

memory_end : bios 에 의해 얻은 전체 메모리의 양
memory_start : 컴파일된 커널에서 text 와 data 영역의 끝.(&end)
low_memory_start : 사용가능 메모리 시작.

kernel 이 실메모리 0x100000 에 있으므로 paging 후에는 커널은 선형번지 0xC0100000 에 있는 것이 된다. 그러나 커널은 여전히 선형번지 0x100000 에 있기도 하다. <그림 2>에서 보이는 것처럼 head.S 에서 초기 4MB 의 물리 메모리에 대하여 페이지 테이블을 생성하고 초기화 한 후에는 커널의 페이지 디렉토리 테이블(swapper_pg_dir)에 2 개의 엔트리(1, 769)가 들어간다.

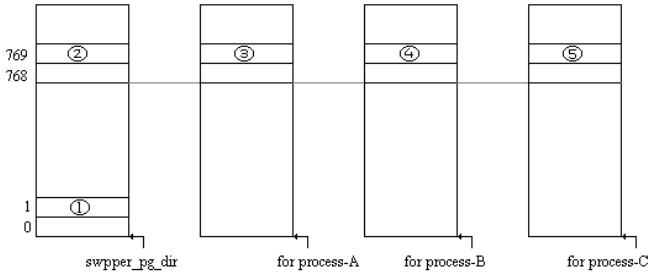
이 두 개의 엔트리는 모두 같은 페이지 테이블을 가리키는 주소 값이 들어가 있다. 즉, 커널의 선형 메모리 공간의 아래쪽 4MB 와 커널영역의 초기 4MB 가 모두 물리 메모리의 처음 4MB 와 매핑되는 것이다. 다시 말하면, 커널 선형 메모리 공간에는 2 개의 커널이 존재하는 것이다.



<그림 3> 물리주소와 선형주소의 매핑

paging 이후의 커널은 선형공간에서 수행된다. 즉 kernel 내부의 모든 번지값들은 선형번지가 되는 것이다. 그것은 kernel 이 paging 이후에도 컴파일할때 번지값들을 그대로 가지고 있기 때문이다. 따라서 선형공간에서도 여전히 물리메모리에서와 같은 위치에 있어야 한다. 즉 커널 세그멘트에서의 물리 주소와 선형주소는 같다.

모든 프로세스의 페이지 디렉토리 테이블의 커널 엔트리들은 모두 동일한 한 개의 커널을 가리키고 있다. 즉, swapper_page_dir 의 내용이 바뀌게 되면(①) 모든 프로세스의 커널 페이지 디렉토리의 엔트리들은 똑같이 바뀌게 된다. (②③④⑤) 다시 말하면, 시스템의 모든 프로세스들은 같은 커널 세그먼트 영역을 참조하게 된다.



<그림 4> 커널과 프로세스들의 page directory

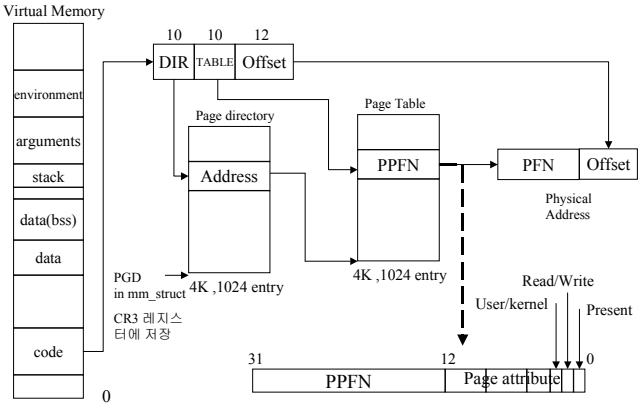
커널 모드에서 메모리를 억세스하게 되면 CS, DS, ES 등의 레지스터에 저장된 세그먼트의 주소가 커널 세그먼트를 가리키도록 하여 커널 세그먼트의 데이터를 읽고 쓸 수 있게 된다. 세그먼트를 지원하지 않는 시스템에서는(ex:Alpha) 물리 주소와 가상(선형)주소 사이에 PAGE_OFFSET 을 제공한다. PAGE_OFFSET 은 가상 메모리 공간에서의 커널 영역의 시작 주소를 가지고 있다. (0xC0000000)[1][2][3]

2.2 리눅스 가상 메모리

리눅스는 가상메모리를 사용하기 때문에 가상 메모리를 물리 메모리로 변환 하는 과정이 필요하다. i386 에서의 페이지 크기는 4K 이고 페이지 테이블의 크기도 페이지 크기와 같은 4K 이다. 따라서 페이지 테이블은 하나의 페이지로 나타낼 수 있다. 페이지 테이블의 엔트리의 크기는 4 바이트이므로 페이지 테이블에는 1024 개의 엔트리가 들어가게 된다.

페이지 테이블 엔트리에는 가상 메모리 페이지와 대응되는 실제 메모리 페이지의 주소가 들어 있어서 페이지 테이블을 통해 물리 메모리에 접근할 수 있다. 가상 메모리 공간에서의 모든 페이지들은 주소 순서대로 존재하지만 실제 물리 메모리에 로드되는 페이지들은 그때 그때의 필요에 따라 메모리로 로드되고 다시 언로드되므로 주소 순서대로 존재하지 않는다.

메모리 변환을 위해 1-레벨의 페이지 테이블, 즉 프로세스 당 하나의 페이지 테이블만을 두면 페이지 테이블에 의한 메모리 낭비가 심하므로 i386 에서는 2-레벨 페이징 기법을 사용한다. 2-레벨 페이징 기법이란 프로세스당 2 개의 페이지 테이블을 가지게 하여 첫 번째 테이블은 4GB 의 프로세스 가상 메모리 공간을 1024 개의 영역으로 나누어 가리키도록 하고, 두 번째 테이블은 4MB 의 영역을 가리키도록 하는 것이다. 다시 말하면 4GB 의 가상 메모리 전체를 매핑하기 위한 페이지 테이블을 두게 되는데 페이지 테이블에는 1024 개의 엔트리가 들어가므로 하나의 엔트리는 4MB 의 가상 메모리 영역을 매핑하게 된다. 이 테이블을 글로벌 페이지 테이블 또는 카탈로그 테이블, 또는 페이지 디렉토리라고 부른다. 4MB 의 영역을 매핑하기 위해서 중간에 페이지 테이블을 하나 더 두어 페이지 디렉토리의 엔트리가 이 테이블을 가리키도록 한다. 이 중간 테이블은 페이지 테이블이라고 부른다. 페이지 테이블이 4GB 의 영역을 매핑하기 위해서는 1024 개의 테이블이 필요하다. 따라서 각 프로세스는 1 개의 페이지 디렉토리와 이론상 최대 1024 개의 페이지 테이블을 가질 수 있다. 페이지 테이블의 엔트리들은 4MB 를 1024 로 나눈값, 즉 4K 의 물리 페이지를 가리킨다. 2-레벨 페이징을 사용하면 페이지 디렉토리를 위한 페이지 하나와 실제 프로세스가 사용하는 페이지영역을 나타낼 수 있는 만큼의 페이지만이 필요하게 된다. 예를 들어 어떤 프로세스가 6MB 의 메모리를 사용한다고 하면, 페이지 디렉토리를 위한 페이지, 처음 4MB 를 매핑하는 페이지 테이블을 위한 페이지, 그리고 다음 2MB 를 매핑하는 페이지 테이블을



<그림 5> 가상 메모리와 물리 메모리 변환

위한 페이지, 이렇게 3 개의 페이지만이 필요하게 된다. 가상 메모리와 물리 메모리의 변환 과정은 <그림

5>에 나타내었다. 메모리 변환은 물리 장치인 MMU 에서 이루어지며 CR3(Control Register)에는 페이지 디렉토리의 시작 주소를 가리키는 주소 값이 들어가 있다. MMU 는 CR3 의 값과 가상 메모리 주소 값을 이용하여 매크로 계산을 통해 물리 페이지 주소를 얻는다.

2-레벨 페이징을 위해 가상 메모리 주소는 3 부분으로 나뉘어 진다. 첫 번째 10 비트는 페이지 디렉토리에서의 offset 으로 사용되어 엔트리의 값을 알 수 있다. 페이지 디렉토리 엔트리에는 해당 페이지 테이블을 가리키는 물리 주소가 들어있다. 가상 메모리 주소의 두번째 10 비트는 페이지 테이블에서의 offset 이 되며 이것을 이용해 페이지 테이블 내에서의 엔트리를 알 수 있다. 페이지 테이블의 엔트리는 물리 메모리에서의 하나의 페이지를 가리킨다. 가상 메모리의 마지막 12 비트는 4K 의 물리 페이지 내에서의 offset 이 된다. 따라서 페이지 테이블의 엔트리 값과 이 offset 을 합쳐서 실제 물리 메모리 주소를 얻게 되는 것이다. 페이지 테이블의 엔트리인 메모리의 주소를 나타내는 32 비트의 값 중에서 20 개의 비트만이 페이지의 프레임 넘버를 나타내기 위해 사용되고 나머지 12 비트는 페이지의 속성을 나타내는 비트로 사용된다. (그림 7 참조) 페이지 속성에는 읽기/쓰기, 커널/유저 모드, 페이지의 존재, 공유 여부 등 페이지를 관리하기 위해 필요한 정보가 들어간다.[1][2][3]

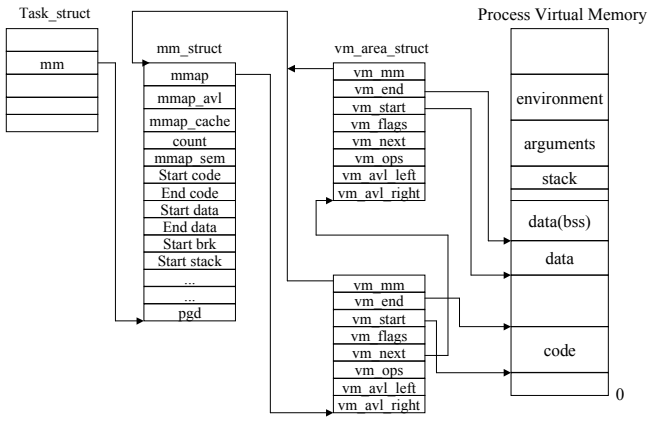
프로세스의 전체 가상 메모리 공간을 나타내는 mm_struct 에는 pgd 필드가 있는데 이 필드가 프로세스의 페이지 디렉토리를 가리킨다. CR3 에는 현재 실행 중인 프로세스의 pgd 값이 들어가게 된다.

2.3 메모리 매핑(Memory Mapping)

이미지를 실행하기 위해서는 실행 이미지의 내용을 프로세스의 가상 주소 공간으로 가져와야 한다. 실행 이미지뿐만 아니라 이미지가 사용하는 라이브러리 역시 가상 주소 공간으로 가져와야 한다. 이처럼 실행 이미지를 프로세스의 가상 주소 공간과 연결하는 것을 메모리 매핑이라고 한다.

모든 프로세스의 가상 메모리 공간은 mm_struct 자료구조와 vm_area_struct 자료구조로 표현된다. mm_struct 는 각 프로세스가 하나씩 가지고 있는 자료구조로 프로세스의 전체 가상 메모리 공간을 나타낸다. 프로세스의 task_struct 구조체의 mm 필드가 mm_struct 를 가리킨다. mm_struct 는 현재 이미지의 정보와 가상 메모리 공간의 일부 영역을 가리키는 여러 개의 vm_struct_area 자료구조를 가리키는 포인터로 구성되어 있다. vm_area_struct 자료구조는 가상 메모리 공간에서 일부 영역을 가리키며 가상 메모리 영역의 시작과 끝, 프로세스의 접근 권한, 가상 메모리 연산 루틴 등에 대한 정보를 가지고 있다.

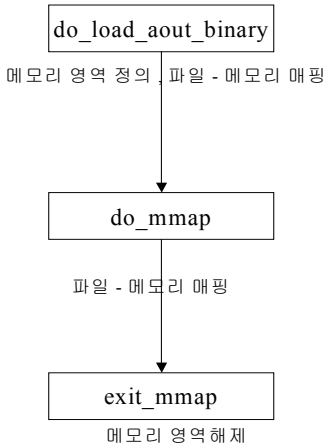
프로세스의 모든 vm_area_struct 자료구조들은 검색 및 할당 속도를 빠르게 하기 위해 AVL 트리 형태로 관리되어진다. 따라서 가상 메모리 영역을 탐색하거나 할당하기 위한 루틴들은 내부적으로 AVL 트리를 다룰 수 있도록 되어 있다. <그림 6>은 mm_struct 와 vm_area_struct, 그리고 가상 메모리 영역간의 관계를 나타낸 그림이다.



<그림 6>프로세스의 가상 메모리 구조

리눅스에서의 프로세스는 fork() 시스템 콜이나 do_load_aout_binary() (fs/binfmt_aout.c) 또는 do_load_elf_binary() (fs/binfmt_elf.c) 함수에 의해서 생성된다.

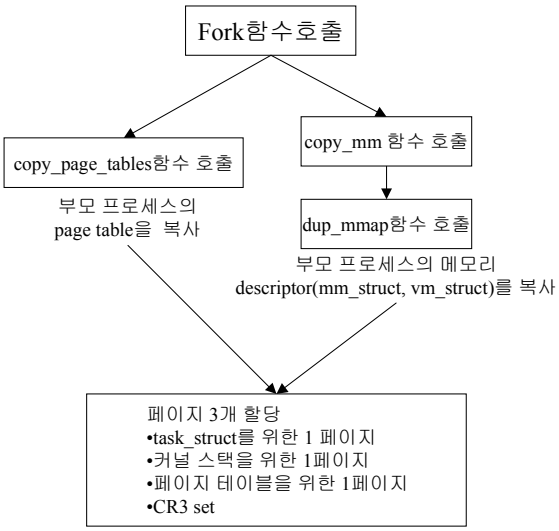
fork 시스템 콜이 수행되면 자식 프로세스는 부모 프로세스의 page table 과 가상 메모리 공간을 복사하여 물려받게 된다. 이때 물론 부모 프로세스의 코드 세그먼트는 복사가 되지 않지만 리눅스에서는 코드 세그먼트뿐만 아니라 데이터 세그먼트도 복사하지 않는다. 프로세스가 복사될 때, 데이터 세그먼트의 모든 페이지들은 단지 두 프로세스의 페이지 테이블에서 읽기 모드로만 억세스 가능하도록 표시만 된다. 다시 어느 한 프로세스가 데이터를 변경하려고 하면, 즉 읽기 모드로 되어있는 데이터 세그먼트의 페이지들을 변경하려고 하면 커널에 의해 메모리 트랩이 발생하게 된다.



<그림 7> do_load_aout_binary()에 의한 메모리 매핑

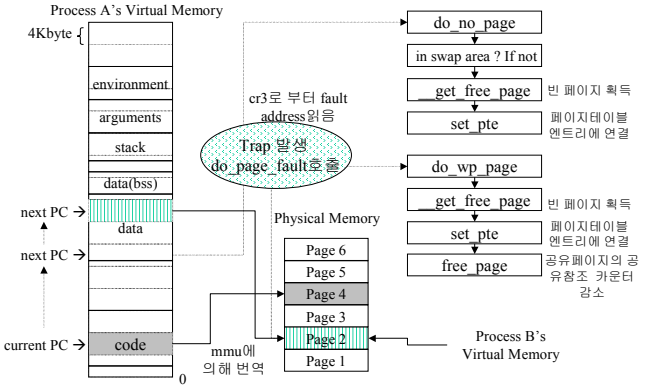
메모리 트랩과정에서 리눅스는 트랩이 발생한 프로세스에게 새로운 페이지를 할당하고 변경하려는 페이지를 복사해준다. 이런 방법으로 메모리의 페이지들은 자신의 내용이 변경될 때만 물리적으로 복사되는 것이다. 내용이 변경되지 않는 페이지는 두 프로세스에

의해 공유될 수 있으므로 물리 메모리 공간을 절약하게 된다. 이것이 Copy-On-Write 라고 불리는 방식으로 부모와 자식 프로세스는 물리 페이지를 공유하고 있다가 어느 한쪽이 페이지에 쓰기 작업을 시도하게되면 그때서야 비로소 새로운 페이지가 할당되는 것이다. do_load_aout_binary()함수가 호출되면 내부적으로 do_mmap() (mm/mmap.c)함수를 호출하여 실행 이미지를 프로세스의 가상 메모리 공간으로 매핑시킨다.[3][4] 위의 <그림 7>과 아래의 <그림 8>은 두 가지 방법에 대한 진행과정을 나타낸다.



<그림 8> fork 에 의한 메모리 매핑

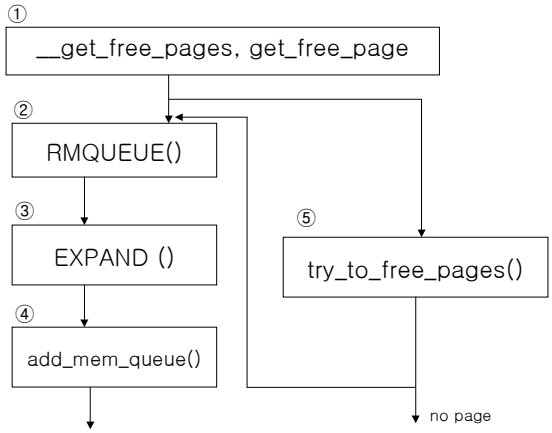
2.4 메핑(mapping)된 프로세스의 진행과 트랩(trap)
<그림 9>는 프로세스가 트랩이 일어났을 때의 진행과정을 나타낸다.



<그림 9> 프로세스의 진행과 트랩

PC 가 가상 메모리 영역의 페이지들 사이에서 이동하면 가상 메모리 주소는 MMU 에 의해 물리 메모리 주소로 변환되어 물리 페이지를 참조하게 된다. 이때 해당 페이지가 메모리에 로드되지 않았거나 스왑(swap)되었을 때, 또는 잘못된 억세스를 시도하려고 할 때는 page fault 가 발생하게 되고 이를 처리하기 위해 do_page_fault()가 호출된다. <그림 9>의 경우를 보면 첫 번째 페이지 fault 는 첫 번째 next PC 가 가리키는 곳에서 발생한다. 이 경우는 물리 메모리에 페이지가 없으므로 do_no_page()가 호출되어 새로운 페이지를 할당받는다. 두 번째 페이지 fault 는 그 다음 next PC 에서 발생한다. 두 번째 fault 는 페이지가 메모리에는 존재하지만 공유되어 있는 페이지이다. 그리고 프로세스가 이 페이지에 대하여 write 를 시도한 경우이므로 do_wp_page()가 호출된다. 첫 번째 페이지 fault 에 대한 결과로 물리 페이지 3 이 새로 할당되었고, 두 번째 fault 의 결과로 물리 페이지 5 가 새롭게 할당되어 현재 프로세스가 write 권한을 가지고 억세스할 수 있게 된다.

2.5 페이지 할당과 해제
리눅스에서의 페이지 할당 및 해제는 buddy 알고리듬을 따른다. 리눅스 커널은 free 페이지를 관리하기 위해서 free_page_list 를 만들고 각 리스트의 엔트리는 2n 개의 물리적으로 연속적인 free 페이지를 가리키도록 하였다. 예를 들어 free_area[0]은 20 개(=1)의 free 페이지들의 리스트를 가리키고 있고, free_area[1]은 21 개(=2)의 물리 페이지가 연속적으로 free 인 것들의 리스트를 가리킨다. 소스 코드에서는 2n 에서 n 을 order 로 부른다. free 페이지 리스트의 개수는 10 또는 12 를 가지는데 i386 은 10 개를 가진다. 따라서 order 는 이보다 작은 값을 가져야 한다. 페이지를 할당받기 위해 필요한 페이지 수를 요청할 때는 order 에 맞게 요청한다. 요청된 페이지 order 에 맞는 free 페이지가 있으면 바로 할당되고, 없으면 그 다음 order 에서 free 페이지를 찾는다. 요청된 페이지 order 보다 더 큰 order 의 페이지가 할당 되면, 나머지 페이지들은 차례로 바로 아래 order 의 free_page 리스트에 삽입된다. free_page 리스트가 2n 형태를 유지하므로 이 과정은 단순히 free 페이지들을 반씩 잘라서 삽입하면 된다. 페이지 할당과 삽입은 각 리스트의 첫 번째 요소에서부터 일어난다. 페이지 할당은 __get_free_pages()나 __get_free_page(), get_free_page()를 호출하는 것으로 이뤄진다. get_free_pages()는 order 를 인자로 가지고 나머지 두 함수는 order 가 0 이다.



<그림 10> 페이지 할당 순서

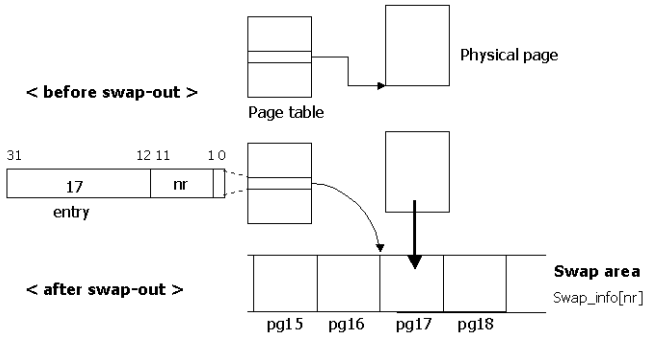
buddy 알고리듬에서의 페이지 해제는 다음과 같다. 페이지가 반환되면 일단 같은 order 를 가지는 free_page 리스트를 검색한다. 비트맵을 검색하여 반환될 페이지

와 물리적으로 인접한 같은 order 의 free 페이지들이 있으면 그 둘을 하나로 합쳐서 한 단계 상위의 order free_area 리스트에 집어넣는다. 그리고 다시 비트맵 검색과정을 거쳐 인접한 free 페이지 리스트가 있으면 또 하나로 합치게 된다. 이 과정을 처음에 주어진 ordre 로부터 가장 큰 order 까지 반복한다.[1][2][3][4]

3. 임베디드 시스템을 위한 리눅스 메모리 구조

서론에서 언급한 바와 같이 본 논문에서는 분석메모리의 페이징(paging)과 컨텐트(contents)와 메모리와의 매핑, 그리고 페이지 폴트(Page fault)와 페이징 폴트에 따른 스와핑(swapping), 그리고 마지막으로 성능 향상을 위한 캐슁(caching)부분으로 나누었다. 임베디드 시스템을 구축하기 위해서는 메인 메모리에서 모든 작업을 수행하기 때문에 스와핑과 캐슁이 필요없다. 우선 스와핑 작업에 대해 조금 설명하자면 현재 수행되고 있는 프로세스들이 메모리에 꽉 차서 새로운 프로세스를 생성시키기에 충분한 메모리가 확보되질 못할 때 실행 중에 있지 않은 프로세스를 일단 디스크로 대피시키기 위한 장소이고 이때 스왑 스페이스(Swap Space)로 옮기는 작업을 Swap out, 다시 메모리로 옮기는 작업을 Swap in 이라한다.

Swap space 는 mkswap 에 의해 초기화 되어진다. Swap space 의 Catalogue 를 담고 있는 첫 페이지를 가리키는 첫 bit 는 1 로써 사용이 불가하고 다음 1023bits 는 0 으로 상응하는 page 들이 사용 가능을 표시한다. 그리고 다음에 오는 bit 들은 1 로 File 안에서 다음 페이지가 없음을 알린다.



<그림 11> Swap out 과정

Kernel swap daemon (kswapd)은 커널 스레드 라는 특별한 종류의 프로세스 시스템에 충분한 free page 가 있도록 하고 커널의 init 프로세스에 의해 시작된다. 커널 스레드는 가상메모리 없이 물리적 메모리 공간에서 커널 모드로 실행되는 프로세스이다. do_try_to_free_pages( )는 공유 메모리 page 를 swap-out 한다. pages 는 저장된 data 를 얻어올 수 없을 때 물리 메모리로부터 swap out 된다. swap 가능한 page 를 모두 swap-out 하지 않고 몇 개의 page 만 제거한다. 초기에는 6 개, 그렇지 않으면 3 개의 page 를 해제하고 충분한 page 가 해제될 때까지 차례로 시도된다. page 에 lock 이 되어 있으면 swap 하지 않는다.(swap algorithm 은 page aging 을 사용한다.) nr_async_pages( )(mm/swap.c)은 Swap file 에 씌어지고 있는 페이지의 수를 count 값으로 유지한다. 이 값은 page 가 swap file 에 씌어지기 위해 queue 에 들어 갈 때마다 증가하고 swap device 에 씌여질 때마다 감소한다. 각 page 마다 count 가 있어서 어떤 page 가 swap 될 것인지 도움을 준다. mm_struct 에 정의되어 있다. 처음 할당된 page 의 count 는 3 이고 사용될 때 마다 count 는 3 씩 증가하고 kswapd 이 실행 될 때마다 count 는 1 씩 감소 count 가 0 이 되면 dirty page 로 간주하고 swap out 하게 된다. Shmid_ds( )은 가상 메모리 영역을 공유하는 프로세스 마다 하나씩 대응되는 vm_area_struct 에 대한 포인터를 가지고 있다. vm_area_struct 는 가상 메모리의 어느 공간이 공유되는지를 나타낸다. 매핑되어 있는 실제 페이지와 페이지 테이블 엔트리의 List 를 갖고 있다. shm_swap( )은 kswapd 이 실행될 때 마다 마지막으로 swap out 된 공유 메모리 page 를 기억한다.

4. 결론

요즘 리눅스는 차세대 운영체제로 주목 받고 있으며 많은 중대형 업체들이 리눅스의 대중화를 위해서 노력하고 있으며 중대형컴퓨터 업체들의 지원으로 리눅스의 신뢰성이 높아지게 되고 다양한 응용소프트웨어(SW) 개발을 촉진하고 있으며 사용자들이 저렴하게 시스템을 도입할 수 있는 계기가 될 것이다.

지금까지 데스크 탑 컴퓨터에만 탑재해오던 리눅스 운영체제를 PDA 나 휴대폰에 장착하기 위해서는 디스크가 필요 없도록 리눅스 소스가 수정되어져야 한다.

본 논문에서는 메인 메모리 상에서만 동작할 수 있는 리눅스를 고안하기 위하여 먼저 리눅스의 메인 메모리를 분석하였다. 캐슁을 하는 부분은 아직 분석이 덜 끝난 상태이고 향후 메인 메모리에서 동작하는 데이터 베이스를 구현할 것이다.

참고문헌

[1] Michael Bock, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner, “Linux Kernel Internals”, 2nd Ed. Addison-Wesley, 1999

[2] 박장수, “Linux hacker 들을 위한 Unix kernel 완전 분석으로 가는길”, KLDP, 1995

[3] Remy Card, Eric Dumas, Frank Mevel, "The LINUX KERNER book" WILEY, pp. 121-248 269-322, December 1998,

[4] Torvalds Linus, The Linux 2.1.11 kernel sources 1999