

# 메모리 관리

## \* 학습목표

- 리눅스의 메모리 관리 기법을 이해한다.
- 가상 메모리와 물리 메모리를 이해한다.
- 리눅스의 주소 변환 방법을 알아본다.
- 메모리 할당과 페이지 교체를 이해한다.

01. 메모리 관리의 이해와 기법 소개

02. 가상 메모리와 물리 메모리

03. 메모리 관리를 위한 자료구조

04. 페이징 기법과 주소 변환의 이해

05. 인텔 프로세스의 주소 변환

06. 리눅스의 3단계 페이징 기법

07. 메모리 할당과 해제

08. 페이지 교체

요약

연습문제



## 메모리 관리의 이해와 기법 소개

태스크는 실행되기 위해 궁극적으로 주기억장치인 메모리에 적재되어야 한다. 따라서 리눅스와 같이 다중 프로그래밍 환경을 지원하는 운영체제 상에서는 당연히 여러 태스크들이 동시에 메모리에 존재할 것이며, 태스크가 사용하는 데이터도 적재되고 태스크들을 관리하기 위한 커널도 메모리에 상주한다. 이러한 환경에서, 커널은 태스크를 메모리의 어디에 적재할 것인지, 언제 적재할 것인지 또 어떻게 적재할 것인지와 같은 관리가 필요하다.

메모리를 다루기 위해 크게 속도적인 문제와 메모리 용량 문제를 고려할 필요가 있다. 메모리는 CPU에 비해 상대적으로 속도가 느리다. 따라서 CPU의 속도가 아무리 빠르더라도 메모리의 속도가 느리다면 명령어나 데이터를 액세스하는 과정에서 많은 시간을 CPU가 기다려야 하므로 전체적인 시스템의 속도를 떨어뜨리는 요인이 된다. 이러한 단점은 하드웨어적으로 캐시 메모리를 둬으로써 보완할 수 있다. 마찬가지로 메모리보다 디스크의 속도는 현저히 느리므로 메모리 내에 버퍼 캐시를 둬으로써 이러한 속도 차이를 극복하고 있다.

두 번째 문제인 메모리의 크기에 대해 생각해보자. 리눅스를 포함한 현대의 대부분의 운영체제는 멀티태스킹 환경을 지원한다. 즉 다수의 태스크가 메모리에 적재되어 실행된다. 이때 태스크들의 전체 크기가 물리적인 메모리보다 크다면 시스템은 어떻게 반응할까? 단순한 물음으로 1MB 크기의 물리 메모리를 가진 시스템에서 600KB와 500KB 크기를 가지는 두 개의 태스크가 동시에 수행될 수 있을까? 리눅스는 하나의 태스크에 4GB 크기의 메모리 주소 공간을 할당한다. 아마도 현재의 컴퓨터 시스템에는 보통 512MB 또는 1GB 정도 크기의 물리 메모리를 일반적으로 장착하고 있을 것인데, 그러면 프로그램 하나도 제대로 수행할 수 없는 것 아닌가? 이 문제는 실행될 프로그램 전체가 메모리에 적재되어야 할 필요가 없으며, 부분 적재를 통해 프로그램의 수행이 가능하다는 개념을 통해 해결할 수 있다.

일상에서 일어날 수 있는 한 예를 들어보자. 도서관에서 자리 하나를 차지하고 시험공부를 하고 있다고 가정하자. 자신이 차지하고 있는 자리의 크기는 한정되어 있으므로 공부에 필요한 모든 책을 책상 위에 펴놓을 수 없을 것이다. 그러면 어떻게 하는가? 지금 당장 필요한 책만 펼쳐 놓고, 나머지는 책꽂이에 꽂아두든지 책상 밑에 잠시 두었다가 이후 다른 책을 봐야 할 때는 현재 보고 있는 책을 내려놓고 필요한 책을 가져다가 펼쳐볼 것이다. 어쩌면 한 번도 펼쳐보지 않는 책도 있을 것이다.

마찬가지로 컴퓨터 시스템에서도 하나의 프로그램이 수행되기 위해 반드시 프로그램 전체가 메모리에 존재할 필요가 없다는 것이다. 즉 똑같은 시간에 프로그램 전체의 내용이 실행에 필요하지 않으며, 어떤 루틴들은(예: 오류처리 루틴) 프로그램이 종료될 때까지 한 번도 사용되지 않을 수도 있다.

따라서 태스크의 가상 메모리 공간에 있는 페이지들 가운데 현재 수행에 필요한 부분만 물리 메모리에 올려 수행하고, 이후의 수행에 필요한 부분이 있으면 그때 메모리로 로드하고, 만약 물리 메모리가 부족해지면 일정 부분의 내용을 다시 하드디스크로 내려놓는다. 이렇게 함으로써 부족한 물리 메모리의 한계를 극복하고, 사용자 관점에서는 자신이 사용하는 전체 코드 및 데이터가 모두 메모리에 있는 것처럼 보이게 하는 것이다.

앞의 예에서 보았듯이 메모리 관리의 기본 핵심은 여러 태스크들이 동시에 실행되면서 각기 다른 태스크가 사용하는 메모리 영역을 침범하지 않고, 실제 시스템에 장착된 작은 크기의 물리 메모리의 한계를 극복하는 것이다. 리눅스는 가상 메모리, 페이징 기법 등을 적용하여 이러한 문제를 효율적으로 해결하고 있다.



## 가상 메모리와 물리 메모리

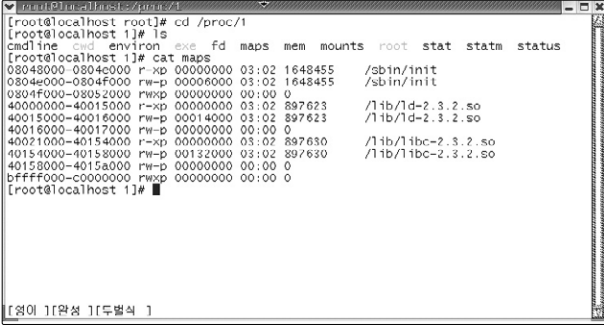
가상 메모리(Virtual Memory)는 물리 메모리(Physical Memory)의 한계를 극복하기 위한 한 방법이다. 여기서 물리 메모리는 시스템에 장착된 실제 메모리이며, 가상 메모리는 실제 존재하지 않지만 마치 아주 큰 메모리가 존재하는 것과 같은 효과를 준다. 따라서 물리 메모리의 크기에 구애받지 않고 큰 메모리 공간을 프로그래머에게 제공할 수 있다.

메모리에 접근하기 위해 메모리 영역은 주소로 구분된다. 여기서 주소는 가상 메모리의 가상 주소(Virtual address)와 물리 메모리의 물리 주소(Physical address)로 구분할 수 있으며, 물리 주소는 시스템에 장착된 물리적인 메모리의 직접적인 위치를 가리키는 것이고, 범위는 0번지부터 장착된 메모리의 크기까지 해당된다. 가상 주소는 물리 주소와는 상관없이 각 태스크마다 할당하는 논리적인 주소를 의미하며, CPU가 생성하는 주소다. 프로그램 상에서 포인터 변수 등을 통해 프로그래머에게 보여주는 주소도 바로 가상 주소이며, 실제 물리 메모리의 주소가 아니다.

앞에서 리눅스는 하나의 태스크마다 4GB 주소 공간을 할당한다고 하였는데, 이는 바로 가상 주소를 의미한다. 실제로 리눅스는 각 태스크에게 3GB를 할당하고, 나머지 1GB는 모든 태스크들의 공통 영역으로서 커널 공간으로 사용한다. [그림 6-1]은 init 태스크에 할당된 가상 주소 영역으로서 /proc/1/maps 파일을 cat 명령으로 읽은 화면이다. 주소가 0x08048000번지에서 시작하고, 0xC0000000로 끝나는 것을 볼 수 있다. 즉 리눅스에서는 모든 태스크가 0x08048000번지부터 시작되고 마지막 주소는 3GB를 가리키는 것을 의미한다.

이러한 가상 주소는 내부적으로 메모리 관리 기능을 통해 물리 주소로 변환되어 실제 물리 메모리에 매핑된다. 따라서 가상 주소를 물리 주소로 변환하는 기법이 필요하며, 리눅스에서는 기본적으로 페이징(paging) 기법을 사용한다. 페이징 기법은 다음 절에서 다룬다. 가상 메모리를 사용하는 것이 물리 메모리의 한계를 극복하는 좋은 방법이지만, 물리 주소로 변환하는 주소 변환 과정이 필요하므로 프로그램의 수행을 지연시키는 오버헤드가 발생한다.

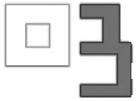
다. 따라서 인텔 프로세서와 같은 CPU에는 이를 줄이기 위한 방법으로 CPU 내에 MMU(Memory Management Unit)와 같은 하드웨어 모듈을 포함시켜 지원하고 있다. MMU 모듈의 처리 방식은 뒤에서 다시 다룬다.



```

[root@localhost ~]# cd /proc/1
[root@localhost 1]# ls
cmdline  cwd  environ  exe  fd  maps  mem  mounts  root  stat  statm  status
[root@localhost 1]# cat maps
08043000-0804e000 r-xp 00000000 03:02 1648455 /sbin/init
0804e000-0804f000 rw-p 00006000 03:02 1648455 /sbin/init
0804f000-08052000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:02 897623 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:02 897623 /lib/ld-2.3.2.so
40016000-40017000 rw-p 00000000 00:00 0
40021000-40154000 r-xp 00000000 03:02 897630 /lib/libc-2.3.2.so
40154000-40158000 rw-p 00132000 03:02 897630 /lib/libc-2.3.2.so
40158000-4015a000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
[root@localhost 1]#
  
```

[그림 6-1] init 태스크가 사용하는 가상 주소 영역



## 메모리 관리를 위한 자료구조

메모리를 사용하는 주체는 태스크다. 따라서 태스크가 사용하는 메모리에 대한 정보를 유지하기 위한 자료구조가 필요하고, 이는 `task_struct` 구조체에 `struct mm_struct *mm`으로 선언되어 있다. 이 `mm_struct` 구조체는 `include/linux/sched.h`에 다음과 같이 정의되어 있으며 메모리 디스크립터라고 한다.

```
struct mm_struct {
    struct vm_area_struct * mmap;
    rb_root_t mm_rb;
    struct vm_area_struct * mmap_cache;
    pgd_t * pgd;
    atomic_t mm_users;
    atomic_t mm_count;
    int map_count;
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock;
    struct list_head mmlist;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long rss, total_vm, locked_vm;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_address;
    unsigned dumpable:1;
    mm_context_t context;
};
```

위 구조체의 주요 멤버 변수들을 살펴보자.

① struct vm\_area\_struct \*mmap

vm\_area\_struct 구조체는 태스크가 사용하는 가상 메모리 영역을 관리하기 위해 사용된다. 각각의 vm\_area\_struct들은 리스트로 연결되며, mmap 변수는 이 리스트의 시작을 가리킨다. [그림 6-1]의 수행 결과로 나온 각 행은 vm\_area\_struct 하나에 대응된다. vm\_area\_struct 구조체는 include/linux/mm.h에 다음과 같이 선언되어 있다.

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    unsigned long vm_flags;
    .....
};
```

위에서 vm\_mm은 태스크의 메모리 디스크립터(mm\_struct)를 가리키는 포인터 변수이며, vm\_start는 이 영역의 시작 주소, vm\_end는 끝 주소다. [그림 6-1]의 각 행의 첫 번째 두 필드의 주소를 이 변수에서 가져오는 것이다. vm\_flags는 접근 제어 등을 담은 플래그다. vm\_next는 다음 가상 메모리 블록을 가리키는 포인터 변수로서, 이를 따라가면 태스크가 사용하는 전체 가상 메모리 공간을 알아낼 수 있다. 마지막 블록의 vm\_next는 NULL 값을 가진다.

② pgd\_t \*pgd;

pgd는 페이지 글로벌 디렉토리의 시작 주소를 가진다. 페이지 글로벌 디렉토리는 가상 주소를 물리 주소로 변환하기 위한 최상위 테이블이다. 주소 변환 부분은 다음 절에서 설명한다.

③ start\_code, end\_code

코드 세그먼트의 영역을 가리킨다. 코드 세그먼트는 프로그램의 명령들이 들어가는 영역을 의미하며, 이 영역의 시작 위치와 끝 위치를 담는다.

## ④ start\_data, end\_data

데이터 세그먼트 영역을 가리킨다. 데이터 세그먼트는 프로그램에 선언된 전역 변수들로 구성되며, 다시 초기화된 변수가 들어가는 데이터 영역과 초기화되지 않은 전역 변수가 들어가는 BSS(Block Started by Symbol) 영역으로 구분된다. 두 변수가 데이터 세그먼트 영역의 시작과 끝 위치를 가진다.

## ⑤ start\_brk, brk

힙(heap)의 시작과 끝을 가지는 변수다. 힙 공간은 프로그램 수행 중에 malloc() 및 free() 같은 라이브러리를 통해 동적으로 메모리를 할당받거나 메모리 공간을 해제하기 때문에 동적으로 크기가 변한다. 새로운 메모리가 할당되면 brk가 가리키는 위치부터 위로 자란다.

## ⑥ start\_stack

스택의 시작 위치를 가진다. 함수를 호출할 때 전달되는 인자들과 복귀 주소 및 지역 변수들로 구성되며, 프로그램을 수행하거나 함수를 호출할 때 인수나 호출된 함수의 지역 변수가 저장되면서 동적으로 범위가 늘어난다. 물론 함수에서 리턴되면 스택의 크기는 줄어든다. 스택이 자라는 방향과 힙이 자라는 방향은 반대다.

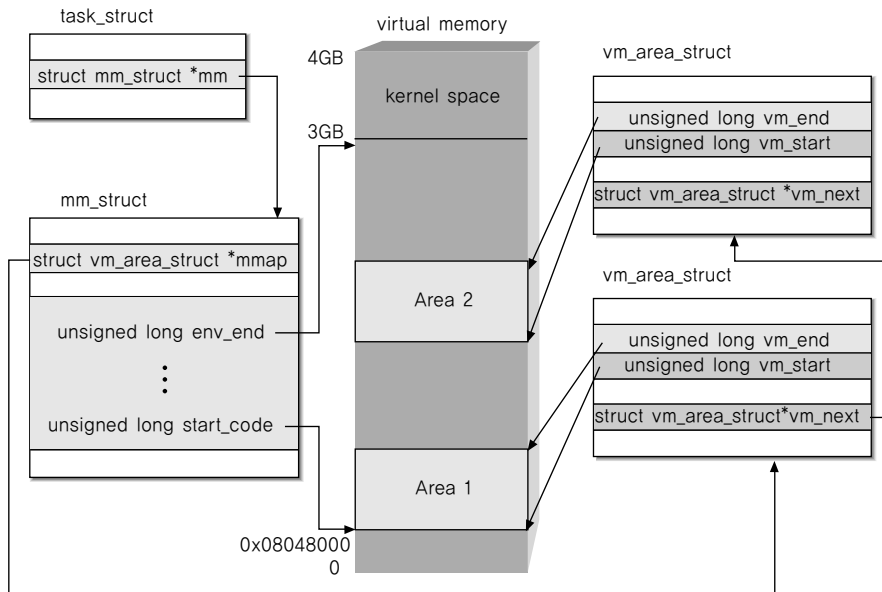
## ⑦ env\_start, env\_end, arg\_start, arg\_end

3GB부터 시작하는 env\_start와 env\_end 사이에 가장 먼저 환경 변수가 자리를 차지하고, 그 다음 arg\_start와 arg\_end 사이를 태스크의 초기 인자가 차지한다. 환경 변수와 초기 인자는 main 함수가 호출될 때 인자로 전달되는 env와 argv를 의미한다. arg\_end 아래에는 main 함수의 지역 변수가 들어가며, 그 이후에는 다른 함수들이 호출될 때 호출된 지역 변수들이 차지하면서 스택은 아랫방향으로 자라게 된다.

위에서 ③~⑦까지의 변수들은 할당받은 가상 메모리 주소를 배치하기 위해 사용되며, 공유 라이브러리 및 공유 메모리 등은 스택과 힙 공간 사이에 위치한다. 예를 들어, sys\_shmget() 시스템 콜을 통해 공유 메모리를 할당받으면 이 공유 메모리는 스택과 힙 사이에 맵핑된다.

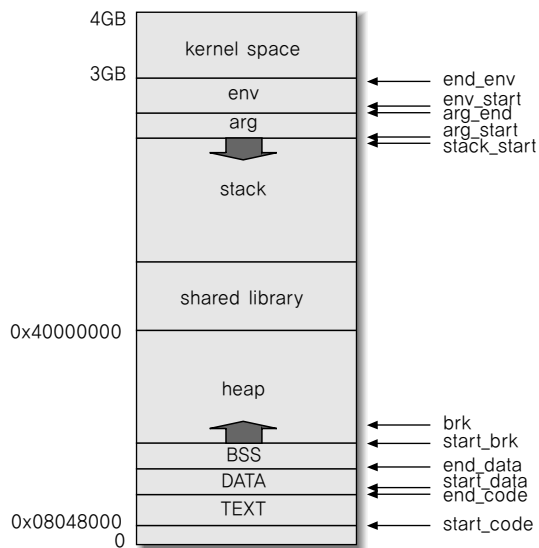
[그림 6-2]는 위에서 설명한 자료구조들을 사용하여 가상 메모리 공간을 어떻게 관리하고 영역을 어떻게 가리키는지를 보여준다.





[그림 6-2] 가상 메모리 관리

[그림 6-3]은 ③~⑦에서 설명한 각 변수들이 가상 메모리의 어느 부분을 가리키는지 자세히 표현한 그림이다.



[그림 6-3] 가상 메모리 구조

## 실습하기 [6-1]

## 응용 프로그램에 할당된 가상 메모리 주소 확인하기

응용 프로그램이 실행될 때 프로그램에 포함된 변수를 비롯하여 힙의 위치 등 가상 메모리의 어느 주소에 배치되는지 알아보자. 이를 위해 다음의 항목을 포함한 응용 프로그램을 하나 작성한다. 또한 본문에서 설명한 각 영역의 시작 주소와 끝 주소를 담은 mm\_struct 내의 변수 값을 돌려받을 수 있는 시스템 콜을 작성하여 각 영역의 범위도 함께 출력하도록 한다.

- 초기화된 전역 변수
- 초기화되지 않은 전역 변수
- 지역 변수
- 힙 위치 지정 변수

**1** 응용 프로그램 작성

파일명은 pp\_getvminfo.c로 작성한다.

**[소스 6-1]** [실습 6-1]의 응용 프로그램 작성

```

01 #include<stdio.h>
02 #include<malloc.h>
03 #include<unistd.h>
04 #include<linux/unistd.h>
05 #include<errno.h>
06 #define __NR_getvminfo 259
07 int vbss;
08 int vdata=1;
09
10 struct vminfo {
11     unsigned long startcode;
12     unsigned long endcode;
13     unsigned long startdata;
14     unsigned long enddata;
15     unsigned long startbrk;
16     unsigned long brk;
17     unsigned long startstack;
18 };
19
20 _syscall2(int, getvminfo, int, pid, struct vminfo*, vm)

```

```

21
22 int main()
23 {
24
25     int i, pid, vstack=2;
26     struct vminfo vm;
27
28     pid = getpid();
29     i = getvminfo(pid, &vm);
30
31     printf("process's memory information\n");
32     printf("Location of main Function in Code Segment:
33           %p(%lx~%lx)\n",main,    vm.startcode, vm.endcode);
34     printf("=====\n");
35     printf("Location of Initialied Data in Data Area:
36           %p(%lx~%lx)\n",&data,   vm.startdata, vm.enddata);
37     printf("=====\n");
38     printf("Location of uninitialized Data in BSS Area:
39           %p(%lx~%lx)\n",&vbss,  vm.enddata, vm.startbrk);
40     printf("=====\n");
41     printf("Stack Location:%p(%lx~)\n",&vstack,
42           vm.startstack);
43     printf("=====\n");
44     sbrk(3);
45     char*b = sbrk(0);
46     i = getvminfo(pid, &vm);
47     printf("Heap Location:%p(%lx~%lx)\n",b, vm.startbrk,
48           vm.brk);
49     printf("=====\n");
50     return 0;
51 }

```

- ① 29행 : 자신의 PID와 가상 메모리 영역을 돌려받기 위한 멤버 변수들로 구성된 구조체를 인자로 하여 작성한 getvminfo( ) 시스템 콜을 호출한다.

② 31행~37행 : 응용 프로그램에서 사용된 요소들이 TEXT, DATA, BSS 영역 및 STACK 영역 중 어느 곳에 위치하는지 출력한다. 이때 시스템 콜을 통해 알아낸 각 영역의 시작과 끝도 함께 출력한다.

③ 41행~45행 : sbrk( )와 brk( )를 이용하여 힙의 영역을 변화시킨 후 힙의 정보를 출력한다.

## 2 시스템 콜 작성

시스템 콜은 프로그래밍 편의를 위해 모듈로 작성하며, 파일명은 getvminfo.c로 한다.

### [소스 6-2] [실습 6-1]의 시스템 콜 작성

```

01  #include <linux/kernel.h>
02  #include <linux/module.h>
03  #include <sys/syscall.h>
04  #include <asm/uaccess.h>
05  #include <linux/mm.h>
06
07  #define __NR_getvminfo 259
08
09  asmlinkage int (*saved_entry)(void);
10  extern void *sys_call_table[];
11
12  struct vminfo {
13      unsigned long startcode;
14      unsigned long endcode;
15      unsigned long startdata;
16      unsigned long enddata;
17      unsigned long startbrk;
18      unsigned long brk;
19      unsigned long startstack;
20
21  };
22
23  asmlinkage int sys_getvminfo(int pid, struct vminfo *uvm)
24  {
25
26      struct vminfo vm;
27      struct task_struct *t;
```

```

28     struct mm_struct *mm;
29
30     t = find_task_by_pid(pid);
31     mm = t->active_mm;
32
33     vm.startcode = mm->start_code;
34     vm.endcode = mm->end_code;
35     vm.startdata = mm->start_data;
36     vm.enddata = mm->end_data;
37     vm.startbrk = mm->start_brk;
38     vm.brk = mm->brk;
39     vm.endenv = mm->env_end;
40     vm.startstack = mm->start_stack;
41
42     copy_to_user(uvm, &vm, sizeof(struct vminfo));
43     return 0;
44 }
45
46 int module_start()
47 {
48
49     saved_entry = sys_call_table[__NR_getvminfo];
50     sys_call_table[__NR_getvminfo] = sys_getvminfo;
51     return 0;
52 }
53
54 void module_end(void)
55 {
56     sys_call_table[__NR_getvminfo] = saved_entry;
57 }
58
59 module_init(module_start);
60 module_exit(module_end);
61
62 MODULE_LICENSE("GPL");

```

- ① 7행 : 시스템 콜의 이름을 getvminfo로, 시스템 콜 번호를 259로 할당한다.
- ② 23행~44행 : 시스템 콜 처리 함수다. PID를 이용해 태스크의 task\_struct를 찾아 메모리 정보를 유지하는 mm\_struct 구조체를 통해 각 영역의 시작과 끝 정보를 찾고 이를 응용 프로그램에게 반환한다.
- ③ 30행 : find\_task\_by\_pid( ) 함수는 시스템 내에서 제공하는 PID를 인자로 하여 task\_struct를 찾는 내부 함수다. 이 함수는 include/linux/sched.h에 구현되어 있으며 해시 테이블을 사용하여 찾는다. 우리는 4장에서 이 함수를 사용하는 대신 task\_struct를 찾는 부분을 직접 구현하였다.

### 3 Makefile 작성과 컴파일 수행

다음과 같이 Makefile을 작성하고 컴파일한다.

#### [소스 6-3] [실습 6-1]의 Makefile 작성

```
TARGET = getvminfo
INCLUDE = -isystem /usr/src/linux-2.4.32/include
CFLAGS = -O2 -D__KERNEL__ -DMODULE $(INCLUDE)
//CFLAGS = -O2 $(INCLUDE)
CC = gcc

${TARGET}.o: ${TARGET}.c

clean:
    rm -rf ${TARGET}.o
```

또한 다음과 같이 응용 프로그램도 컴파일한다.

```
#gcc -o app_getvminfo app_getvminfo.c
```

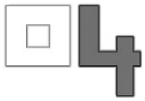
### 4 실행 및 결과 확인

insmod 명령으로 모듈을 적재한 후 응용 프로그램을 실행하여 결과를 확인한다.

```
#insmod getvminfo.o
#./app_getvminfo
```

[실행 화면 예]

```
[root@localhost memory]# ./app_getvminfo
process's memory information
Location of main Function in Code Segment: 0x804800b(8048000~8048790)
=====
Location of Initialed Data in Data Area:0x804979c(8049790~80498a0)
=====
Location of uninitialized Data in BSS Area:0x80498a4(80498a0~80498a8)
=====
Heap Location:0x80498ab(80498a8~80498ab)
=====
Stack Location:0xbffffae4(bffffb40~)
=====
[root@localhost memory]# _
```



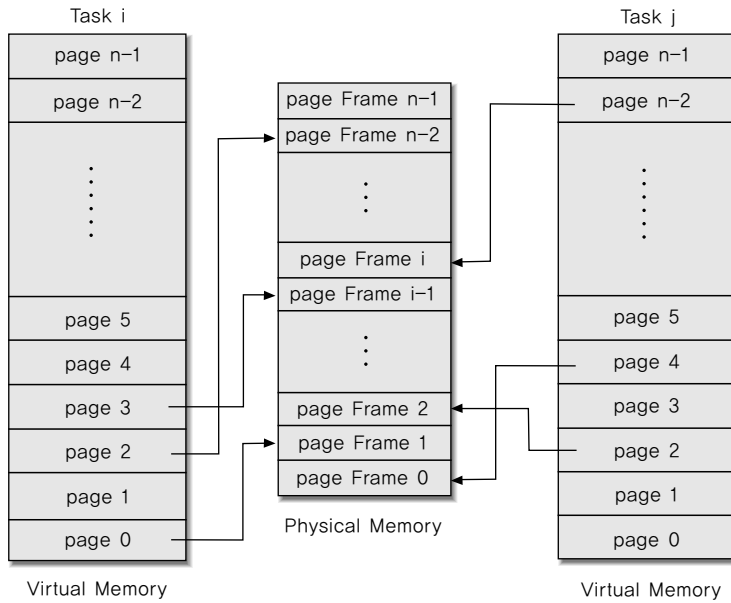
## 페이징 기법과 주소 변환의 이해

앞에서 태스크에서 할당되는 가상 메모리를 살펴보았고, 하나의 태스크마다 4GB 크기의 가상 메모리를 사용한다고 하였다. 궁극적으로 태스크는 물리 메모리에 저장된다. 따라서 가상 주소를 사용하는 태스크를 적절히 물리 메모리에 배치할 수 있어야 하며, 물리 주소도 알 수 있는 방법이 필요하다. 다시 말해서 가상 주소를 물리 주소로 변환하여 실제 물리 메모리의 어디에 적재되어 있는지 알아야 하는 것이다. 이러한 가상 주소를 물리 주소로 바꾸는 것을 주소 변환이라고 한다.

이제 태스크를 물리 메모리로 어떻게 적재하는지 알아보자. 리눅스는 페이징 기법을 사용하여 가상 메모리의 페이지를 물리 메모리에서 현재 사용되고 있지 않는 빈 페이지 프레임에 적재시킨다. 여기서 페이지는 가상 메모리를 일정한 고정 크기로 분할하는 단위를 의미하며, 페이지 프레임은 물리 메모리를 일정한 고정 크기(예: 인텔 펜티엄 프로세서에서는 4KB, 알파칩의 경우에는 8KB)로 분할한 단위를 의미한다. 일반적으로 페이지와 페이지 프레임의 크기는 동일하다. 예를 들어, 인텔 펜티엄 프로세서를 사용하고 장착된 메모리의 크기가 512MB라고 가정하면 물리 메모리는 512MB/4KB개의 페이지 프레임 단위로 분할되고, 가상 메모리의 4GB도 마찬가지로 4KB 크기로 분할된다. 이때 분할된 각각의 영역을 하나의 페이지라고 한다.

다중 프로그래밍 환경에서 여러 태스크가 동시에 메모리에 적재되어야 하고, 페이지의 수는 페이지 프레임의 수보다 훨씬 많으므로 한꺼번에 모든 페이지를 물리 메모리에 적재하는 것은 불가능하다. 따라서 가상 메모리의 페이지들 가운데 현재 수행에 필요한 페이지만 물리 메모리에 적재하는 방식으로 이 문제를 해결한다. 앞 절에서 어느 순간에 모든 태스크의 내용이 물리 메모리에 있을 필요가 없다고 하였다. 즉 태스크의 현재 실행에  $n$ 개의 페이지 프레임만큼의 메모리 공간이 필요하다면 커널은 물리 메모리의 비어있는 페이지 프레임 가운데  $n$ 개를 할당하고, 가상 메모리의  $n$ 개 페이지를 적재한다. 이후 더 이상 비어있는 페이지 프레임이 존재하지 않으면 시스템의 정책에 따라 할당된 페이지 프레임을 비우고, 필요한 페이지를 다시 적재하는 방식으로 진행한다.





[그림 6-4] 페이지 적재 예

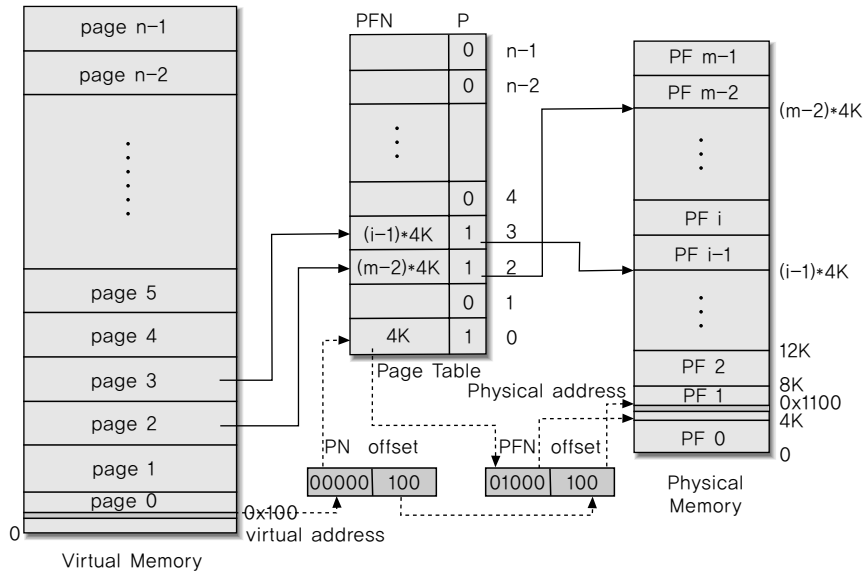
[그림 6-4]는 두 개의 태스크(Task i, Task j)의 현재 실행에 필요한 가상 메모리의 페이지를 물리 메모리의 페이지 프레임에 적재하는 예다. 그림에서 알 수 있듯이 모든 페이지가 한꺼번에 적재되지 않으며, 하나의 페이지는 어느 위치의 페이지 프레임에든 적재할 수 있다. 즉 페이지를 적재할 때 연속적인 페이지 프레임을 할당할 필요가 없다는 의미다.

가상 메모리 주소는 연속적인 공간으로 보이지만, 실제 물리 메모리상에서는 임의의 페이지 프레임에 적재되므로 페이지가 적재된 페이지 프레임의 위치와 함께 임의의 물리 메모리 공간에 저장되어 있는 데이터의 위치를 알 수 있는 방법이 필요하다. 이를 위해 페이지 테이블(page table)을 사용한다. 이 테이블에는 태스크의 페이지가 적재된 페이지 프레임 정보를 담는다.

리눅스에서는 페이지 테이블을 3단계로 나누어 사용한다. 그 이유는 하나의 페이지 테이블을 사용하는 경우 테이블의 크기가 커지므로 이를 유지하기 위한 메모리 공간이 많이 필요하기 때문이다.

우선 하나의 페이지 테이블을 이용하는 경우를 통해 실제 물리 주소를 알 수 있는 방법과 비효율성에 대해 알아보자. 이때는 페이지 테이블을 페이지의 개수만큼 엔트리로 분할하고,

각 엔트리에는 각 페이지가 적재된 페이지 프레임의 위치 정보를 담도록 한다.



[그림 6-5] 한 개의 페이지 테이블을 사용할 때의 주소 변환 예

[그림 6-5]는 하나의 페이지 테이블을 이용하여 [그림 6-4]의 Task i의 페이지가 물리 메모리의 페이지 프레임으로 맵핑되는 예다. 페이지 테이블의 각 엔트리는 하나의 페이지에 각각 하나씩 대응된다. 현재 분할된 페이지 가운데 페이지 번호 0, 2, 3의 페이지가 실제 물리 메모리에 적재되어 있으며, 페이지 테이블의 엔트리에는 페이지가 적재되어 있는 페이지 프레임의 시작 위치(PFN(Page Frame Number) 필드)가 담겨 있다. P(Present) 필드는 페이지가 페이지 프레임에 적재되지 않은 경우에는 0으로, 적재된 경우는 1로 표시되는 플래그다. 그림을 통해 페이지 0은 페이지 프레임 1에 적재되어 있고, 페이지 2는 페이지 프레임 (i-1)에 적재되어 있음을 알 수 있다.

하나의 페이지 테이블을 사용하는 경우 실제 물리 주소를 알아내는 방법을 알아보자. 가상 주소는 CPU에서 생성되는 주소다. 이 주소는 페이지 테이블의 엔트리를 찾기 위한 페이지 번호와 오프셋 값으로 나뉘는데, 32bit 주소를 가진다고 가정하면 상위 20bit는 페이지 번호에 해당하며, 나머지 12bit는 오프셋 값이 된다. 오프셋은 하나의 페이지 프레임의 전체 주소를 가리킬 수 있는 값이 된다.

예를 들어, [그림 6-5]에서와 같이 가상 주소 0x100번지가 물리 메모리의 어디에 저장되어 있는지 찾아보자. 가상 주소 0x100은 2진수로 0000 0000 0000 0000 0000 0001 0000 0000<sub>2</sub>이 되며, 따라서 페이지 번호는 0, 오프셋은 0x100이 된다. 이제 페이지 테이블의 0번 엔트리에서 페이지 프레임의 위치가 4K(0x1000)임을 알게 되고, 오프셋 값을 더하여 실제 물리 주소인 0x1100을 얻게 된다. 이 값은 4KB 위치에서 0x100만큼의 오프셋만큼 이동한 값과 같다.

또 다른 예로, 만약 0x1001번지에 접근하면 어떻게 될까? 0x1001은 2진수로 0000 0000 0000 0000 0001 0000 0000 0001<sub>2</sub>이 되며, 상위 20bit를 이용해 페이지 테이블의 인덱스 1을 알아낸다. 그러나 이 엔트리의 P 필드는 0으로 표시되어 있으며, 이는 해당 페이지가 물리 메모리에 적재되지 않았음을 의미한다. 이러한 상황을 페이지 폴트(page fault)가 발생했다고 한다. 따라서 커널은 물리 메모리의 빈 페이지 프레임 중 하나를 요구하여 할당받고, 해당 페이지를 적재한 후 페이지 테이블에 적재된 페이지 프레임의 위치를 기록하고, 다시 변환 과정을 거쳐 접근한다. 이와 같은 방식으로 페이지징을 처리하는 기법을 요구 페이지징(Demand Paging) 기법이라고 한다.

위에서 보는 바와 같이 한 개의 페이지 테이블을 사용하는 경우 주소 변환 방법이 아주 간단하다. 그러나 페이지 테이블도 메모리에 위치해야 하므로 각 태스크마다 페이지 테이블 하나를 사용하는 경우 페이지 테이블이 차지하는 메모리 공간이 많이 필요하다. 예를 들어, 인텔 펜티엄 프로세서를 사용하는 시스템의 경우, 위의 예와 같이 하나의 태스크가 4GB 크기의 가상 메모리 공간과 4KB의 페이지 크기를 가지므로 페이지 테이블의 엔트리 수가 1,048,756개가 되며, 한 엔트리는 주소를 가리키기 위한 한 워드, 즉 4byte 크기를 가지므로 각 태스크마다 페이지 테이블을 위해 4MB의 공간을 필요로 한다. 하지만 1M 크기의 태스크를 위해 4M의 페이지 테이블을 사용한다는 것은 당연히 있을 수 없는 일이다. 따라서 이러한 문제를 극복하기 위해 인텔 프로세서에서는 페이지징을 2단계로 나누어 처리하며, 리눅스에서는 페이지 테이블을 3단계(커널 2.6에서는 4단계의 페이지징을 지원한다)로 나누어 처리한다.

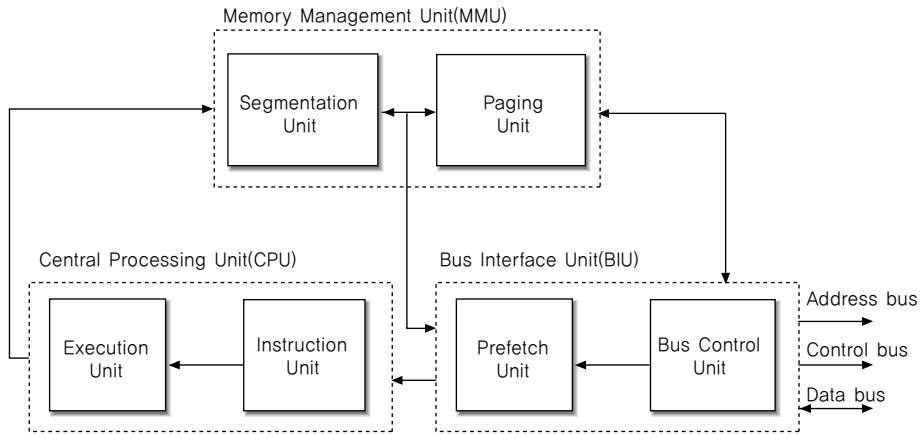


## 인텔 프로세스의 주소 변환

앞 절에서 가상 메모리와 물리 메모리의 관계를 배웠으며, 이때 주소 변환에 따른 오버헤드가 발생한다고 했다. 또한 하나의 페이지 테이블을 사용하는 경우 페이지 테이블의 크기 문제가 발생하며, 이를 해결하기 위해 인텔 프로세서에서는 CPU 내에 MMU라는 하드웨어 장치를 포함시켜 주소 변환에 따른 시간 지연을 줄이고 페이지 테이블의 크기 문제를 2단계 페이징 기법을 사용하여 해결한다고 하였다. 인텔 x86 계열의 CPU에서는 80386부터 이러한 메모리 아키텍처를 제공하였다. 이 절에서는 인텔 프로세서에서의 주소 변환에 대해 알아본다.

초기의 8086에서는 보호 모드(protected mode)가 별도로 존재하지 않았으며, 16bit 세그먼트 레지스터와 16bit 오프셋을 중첩하여 20bit, 즉 1MB의 주소 공간을 제공하였다. 이후 80286에서 8086과 같은 주소 공간을 제공하는 실제 모드(real mode)와 함께 보호 모드가 도입되었다. 80286도 세그먼테이션 메커니즘을 사용하였으며, 24bit의 기준 주소와 16bit의 오프셋을 더하여 16MB의 주소 공간을 제공하였다. 또한 셀렉터(selector)를 사용하는 개념을 도입하였고, 이 셀렉터를 24bit의 기준 주소(base address)로 변환시켜 주는 디스크립터 테이블(descriptor table)이 등장하였다. 그러나 80286도 8086과 마찬가지로 16bit 프로세서기 때문에 세그먼트 크기가 64KB로 제한되었다.

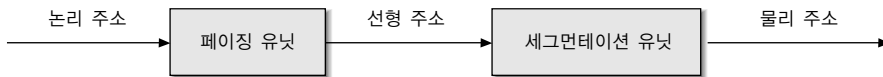
80286에서 등장한 보호 모드의 개념은 80386에서 완성되었다고 할 수 있다. 80386에는 세그먼테이션 기법 외에 메모리 관리에 필수적인 페이징 메커니즘이 추가되고, 메모리 공간도 32bit, 즉 4GB로 확장되었다. 따라서 8086이 비록 프로그래밍 모델 측면에서 현재 펜티엄 프로세서까지의 기반이 되었지만, CPU 내부의 구조적인 면에서는 80386이 기반이 되었다고 볼 수 있다. 흔히 인텔 i386 아키텍처를 기반으로 한다는 말의 의미도 바로 이 때문이다. 참고로 80386의 내부 구조는 [그림 6-6]과 같으며, 내부에 MMU를 포함하고 있음을 알 수 있다.



[그림 6-6] 80386 내부 구조 블록도

## 1 주소 변환 과정

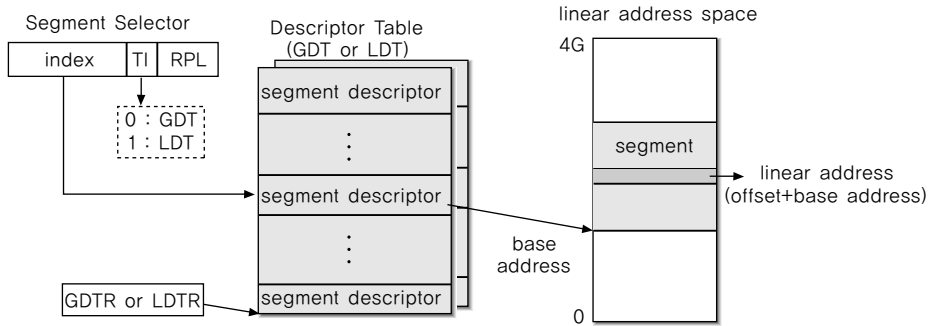
인텔 i386에서는 물리 주소를 생성하기 위해 [그림 6-7]과 같이 MMU의 세그먼테이션 유닛과 페이징 유닛을 각각 거친다. 이때 세그먼테이션 유닛으로 입력되는 주소를 논리 주소라 하며, 이 유닛에서 선형 주소(linear address)를 생성한다. 다시 이 선형 주소는 페이징 유닛으로 입력되고, 여기서 실제 물리 주소를 얻는다.



[그림 6-7] 인텔 i386에서의 주소 변환 과정

### [논리 주소] → [선형 주소]

논리 주소를 선형 주소로 변환하기 위해서 16bit의 셀렉터와 32bit의 오프셋을 사용한다. 셀렉터는 [그림 6-8]과 같이 디스크립터 테이블에 대한 인덱스, 어떤 디스크립터 테이블을 가리키는지를 나타내는 TI(Table Indicator) 항목, 접근 권한을 나타내는 RPL(Requested Privilege Level)의 3가지 부분으로 구성되며, 세그먼트 레지스터에 로드된다.



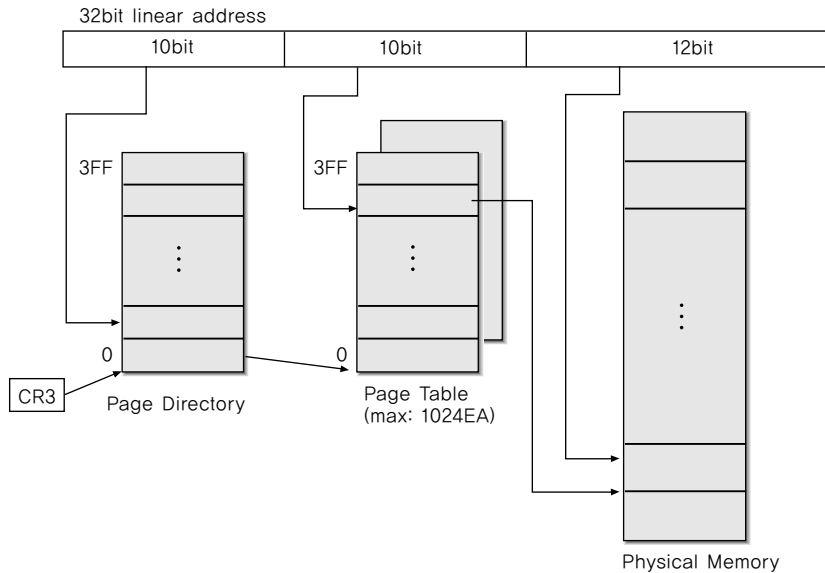
[그림 6-8] 논리 주소→선형 주소 변환 과정

디스크립터 테이블은 각 세그먼트의 정보를 기술한 세그먼트 디스크립터들로 구성되며, 각 디스크립터들은 64bit 크기이며 32bit 크기의 기준 주소와 20bit 크기의 범위 그리고 세그먼트의 속성으로 구성된다. 이때 기준 주소는 4GB의 선형 주소 공간에서의 시작 위치이며, 범위는 기준 주소에서부터 접근 가능한 메모리 범위를 나타낸다. 디스크립터 테이블은 GDT(Global Descriptor Table)와 LDT(Local Descriptor Table)로 구분되는데, GDT는 커널 모드에서 사용하는 테이블로서 테이블의 시작 위치를 GDTR 레지스터가 가리킨다. 반면 LDT는 사용자 모드에서 사용되는 테이블로서, 시작 위치를 LDTR 레지스터가 가리킨다. 세그먼트 속성 중에는 2bit의 DPL(Descriptor Privilege Level)을 포함하고 있으며, 이는 디스크립터의 접근 권한을 명시한다.

잠시 특권 레벨에 대해 알아보자. 특권 레벨은 어떤 명령을 실행할 수 있는 권한이나 메모리의 어떤 영역에 접근할 수 있는 권한의 레벨을 의미한다. 특권 레벨은 0~3까지의 4단계로 구분되며, 리눅스에서는 0과 3을 사용하여 커널 모드와 사용자 모드로 구분한다. 이때 CPU가 현재 가지고 있는 특권 레벨을 CPL(Current Privilege Level)이라고 한다. 세그먼트 셀렉터가 가지고 있는 RPL은 세그먼트 레지스터로 로드되면 이 값이 CPL이 된다. 세그먼트 디스크립터의 DPL은 CPL과 비교되어 접근 가능 여부를 판단하는 데 사용된다.

### [선형 주소] → [물리 주소]

선형 주소는 페이징 유닛을 거쳐 물리 주소로 변환된다. 페이징 유닛은 32bit의 선형 주소를 3가지 영역으로 분리하는데, 상위 10bit를 페이지 디렉토리, 다음 10bit를 페이지 테이블, 나머지 12bit를 오프셋으로 나누어 [그림 6-9]와 같이 두 단계의 페이징을 수행한다.



[그림 6-9] 2단계 페이징 과정

[그림 6-9]에서 CR3 레지스터는 물리 메모리에 위치한 태스크의 페이지 디렉토리의 시작점 주소를 가리키는 레지스터다. 이를 참조하여 페이지 디렉토리의 시작 주소를 알아내는 것에서부터 주소 변환이 시작된다.

페이지 디렉토리는 페이지 테이블들에 대한 테이블이라 생각하면 된다. 페이지 디렉토리는 각 태스크마다 하나씩 할당되며, 10bit를 가지므로 0~3FF까지의 1,024개 엔트리를 갖는다. 여기서 각 엔트리는 32bit 주소를 가리키기 위한 4byte 워드의 크기를 가지므로 페이지 디렉토리 하나의 크기는 4KB이며, 이는 페이지 프레임 하나의 크기와 일치한다. 즉 태스크가 생성되면 하나의 페이지 프레임을 할당하여 페이지 디렉토리를 저장하는 것이다. 페이지 디렉토리의 각 엔트리는 하나의 페이지 테이블을 가리킨다. 따라서 태스크 하나는 최대 1,024개의 페이지 테이블을 가질 수 있다. 하지만 태스크가 생성된다고 무조건 1024개의 페이지 테이블을 만드는 것이 아니며, 사용하지 않는 영역에 대해서는 페이지 테이블을 생성하지 않는다.

하나의 페이지 테이블의 크기도 페이지 디렉토리와 마찬가지로 10bit로 구성되므로 4KB이며, 1,024개의 엔트리를 갖는다. 페이지 테이블의 각 엔트리는 물리 메모리의 페이지 프레임의 주소 정보를 가진다. 이렇게 페이지 디렉토리와 페이지 테이블을 거치면 물리 메모

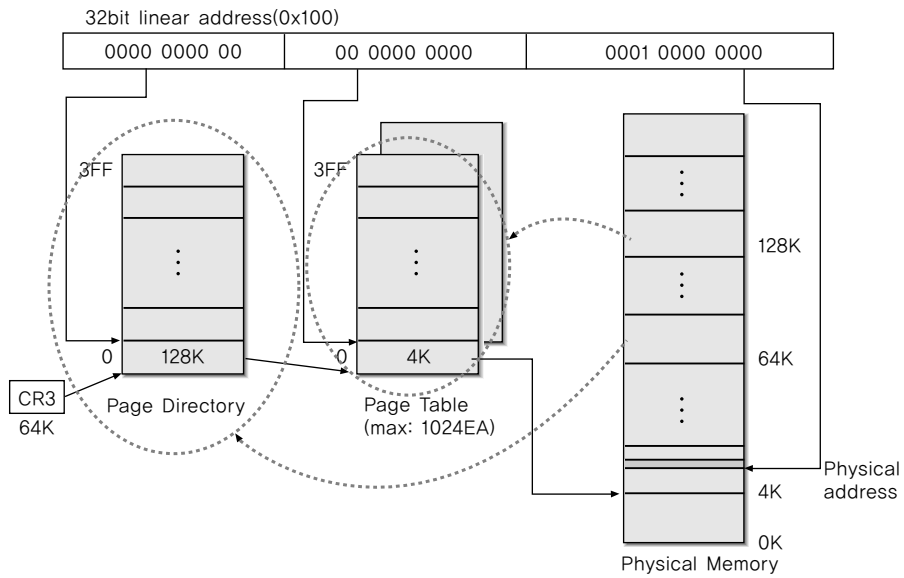
리의 페이지 프레임을 찾는데, 이때 페이지 프레임의 크기가 4KB이므로 범위 내에서 임의의 위치를 찾으려면 12bit가 필요하다. 오프셋은 바로 이를 위한 부분이며, [그림 6-9]에서 오프셋의 크기가 12bit로 구성되어 있음을 알 수 있다.

두 단계 페이징을 통해 접근할 수 있는 메모리의 크기는 페이지 디렉토리의 크기( $1024 \times$  페이지 테이블 개수( $1024 \times$  페이지 프레임 크기(4KB), 즉 4GB가 된다. 이는 앞 절에서 설명한 한 개의 페이지 테이블을 사용하였을 때와 크기가 동일하다.

그러나 이러한 테이블도 물리 메모리에 존재하므로 하드웨어의 도움을 받아 페이징을 하더라도 물리 주소에 접근하기 위해서는 페이지 디렉토리 및 페이지 테이블을 찾기 위해 부가적으로 물리 메모리 공간에 접근해야 하며, 이는 시스템의 성능 저하 요인이 된다. 따라서 MMU 내에 TLB(Translation Lookaside Buffer)라는 캐시를 포함시킨 후 먼저 TLB를 확인하여 캐싱되어 있는 정보가 있으면 주소 변환 과정 없이 바로 사용하고 만약 TLB 내에 없으면 주소 변환을 수행한다. 이때 그 결과는 다시 TLB에 저장해 둬으로써 다음에 이 정보를 활용한다.

2단계 페이징을 통해 물리 주소로 변환되는 과정을 예를 들어 살펴보자. [그림 6-10]과 같이 32bit 선형 주소가 0x100번지라고 가정하면, 이는 0000 0000 0000 0000 0000 0001 0000 0000<sub>2</sub>의 값을 가진다. 이 주소를 세 영역으로 분류하면 상위 10bit는 0000 0000 00이고, 다음 10bit는 00 0000 0000이 되며, 하위 12bit 오프셋은 0001 0000 0000 값을 갖는다. 이제 주소 변환을 위해 먼저 CR3 레지스터의 값을 읽어서 페이지 디렉토리가 시작하는 위치를 알아낸다. 만약 CR3의 값이 64KB라면 페이지 디렉토리가 이 위치에 존재함을 의미한다. 여기서 64K는 주소를 변환하는 방법을 설명하기 위해 페이지 크기로 나눈 임의의 영역임에 주의하자. 당연히 페이지 테이블은 커널 영역에 존재한다. 다음 선형 주소의 상위 10bit가 모두 0이므로 페이지 디렉토리의 0번 엔트리를 참조하여 물리 메모리의 128K에 위치한 페이지 테이블을 찾는다. 즉 페이지 테이블의 시작 주소를 알아내는 것이다. 다음으로 선형 주소의 다음 10bit를 이용하여 페이지 테이블 내의 엔트리를 찾는데, 10bit 모두 0이므로 0번 엔트리를 참조하고 페이지 프레임의 시작 주소인 4K를 알아낸다. 마지막으로 이 페이지 프레임 내의 상대 위치에 해당하는 오프셋 값인 하위 12bit 값을 더하면 원하는 물리 주소를 얻게 된다.



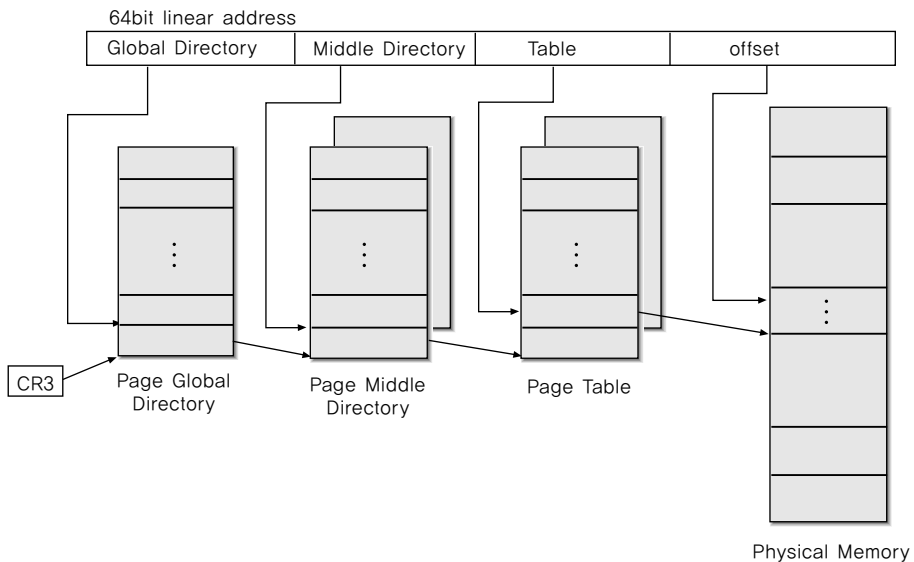


[그림 6-10] 2단계 페이징의 주소 변환 예



## 리눅스의 3단계 페이징 기법

리눅스 커널은 다양한 시스템에 포팅하여 사용할 수 있다. 따라서 리눅스의 페이징은 인텔 i386에만 한정되지 않으며 Alpha CPU와 같이 64bit 주소 공간을 사용하는 아키텍처를 지원하기 위해 앞 절에서 설명한 2단계에 한 단계를 더 추가하여 3단계의 페이징을 이용하여 주소 변환을 수행하며 [그림 6-11]과 같은 과정을 거친다. 참고적으로 리눅스 2.6의 후 반대 버전에서는 4단계 페이징을 지원한다.



[그림 6-11] 리눅스의 3단계 페이징 과정

이렇게 3단계 과정을 거치는 이유는 앞 절에서 2단계의 과정을 거치는 이유와 마찬가지로 동일하다. 즉 64bit의 선형 주소를 사용하는 경우 32bit 선형 주소보다 주소 공간이 훨씬 커지므로 페이지 테이블이 차지하는 용량을 줄이기 위한 것으로 이해하면 된다. 그런데 인텔 i386에서는 2단계의 페이징 단계를 사용하고, 리눅스 커널에서는 3단계를 사용한다면

인텔 i386 환경에서는 커널이 어떻게 처리할까? 개념적으로는 페이지 미들 디렉토리가 존재하지 않는 것으로 여긴다. 그러나 사실 커널은 3단계의 구조를 그대로 유지하면서 2단계의 페이지를 지원하는 방법을 사용하는데, 페이지 미들 디렉토리가 한 개의 엔트리만을 가지는 것으로 정의하고, 페이지 글로벌 디렉토리의 적절한 엔트리로 매핑시킴으로써 기존의 코드를 그대로 사용한다.

리눅스에서 3단계 페이지징에 사용되는 페이지 글로벌 디렉토리, 페이지 미들 디렉토리 및 페이지 테이블은 각각 `pgd_t`, `pmd_t`, `pte_t` 형식을 사용한다. 각 테이블에서 엔트리를 찾을 때 사용하는 함수가 있는데, 페이지 글로벌 디렉토리의 경우에는 `pgd_offset()`, 페이지 미들 디렉토리의 경우에는 `pmd_offset()`, 페이지 테이블의 경우에는 `pte_offset()`을 각각 사용한다.

물리 주소를 얻는 과정을 간단하게 살펴보면 먼저 CR3 레지스터에서 글로벌 디렉토리의 시작 주소를 알아내고, `pgd_offset()` 함수로 글로벌 디렉토리에서 페이지 미들 디렉토리의 위치를 알 수 있는 엔트리의 위치를 알아낸다. 즉 여기서 페이지 미들 디렉토리의 주소를 알아내는 것이다. 다음 `pmd_offset()` 함수를 사용하여 미들 디렉토리에서 페이지 테이블의 위치를 알아내고, 마지막으로 `pte_offset()` 함수를 통해 페이지 테이블에서 물리 페이지의 위치를 알아낸다.

다음은 페이지 테이블을 다루는 데 사용되는 커널 함수 및 매크로다.

- `pgd_offset(mm, address)`  
메모리 디스크립터와 가상 주소를 인자로 받아 페이지 글로벌 디렉토리의 엔트리 주소를 반환한다.
- `pmd_offset(dir, address)`  
페이지 글로벌 디렉토리 엔트리와 가상 주소를 인자로 받아 페이지 미들 디렉토리의 엔트리 주소를 반환한다.
- `pte_offset(dir, address)`  
페이지 미들 디렉토리와 가상 주소를 인자로 받아 페이지 테이블의 엔트리 주소를 반환한다.
- `pgd_none(pgd)`, `pmd_none(pmd)`, `pte_none(x)`  
각 테이블의 엔트리를 인자로 받아 존재 여부를 검사하며 존재하면 0을, 그렇지 않으면 1을 반환한다.

- `pgd_present(pgd)`, `pmd_present(pmd)`, `pte_present(pte)`  
각 테이블의 엔트리를 인자로 받아 해당 엔트리의 Present 플래그를 검사하여 해당 테이블 또는 해당 페이지가 메모리에 존재하면 1을 반환한다.
- `pgd_clear(pgd)`, `pmd_clear(pmd)`, `pte_clear(pte)`  
각 테이블의 엔트리를 인자로 받아 해당 테이블의 엔트리를 지운다.
- `pgd_page(pgd)`  
페이지 글로벌 디렉토리의 엔트리에서 페이지 미들 디렉토리의 가상 주소를 반환한다.
- `pmd_page(pmd)`  
페이지 미들 디렉토리의 엔트리에서 페이지 테이블의 가상 주소를 반환한다.
- `pte_page(x)`  
페이지 테이블 엔트리가 가리키는 페이지 프레임의 페이지 디스크립터의 주소를 반환한다. 페이지 디스크립터는 page 구조체 타입으로서 페이지 프레임의 상태 정보를 저장한다.

**실습하기 [6-2]****태스크에 할당된 가상 메모리 영역의 정보와 파일의 이름 출력하기**

[그림 6-1]의 화면 내용 중 첫 번째 필드와 마지막 필드의 정보를 출력하는 시스템 콜을 작성해보자. 여기서 첫 번째 필드는 태스크가 사용 중인 가상 메모리 영역의 시작 주소와 마지막 주소에 대한 정보이고, 마지막 필드는 전체 경로를 포함한 파일의 이름이다.

※ 문제를 해결하기 위한 조건

- (1) 시스템 콜의 이름은 `getmeminfo`로 하며 태스크의 PID를 인자로 받는다.
- (2) PID 값 1(init 태스크)을 인자로 받아 출력한 예는 다음과 같으며, 가상 주소 영역은 `#cat /proc/PID/maps` 명령으로 나타나는 영역과 동일하다.
- (3) 시스템 콜은 모듈로 작성하여 테스트한다.

[실행 화면 예]

```
00040000 - 0004e000 /sbin/init
0004e000 - 0004f000 /sbin/init
0004f000 - 00052000
40000000 - 40015000 /lib/ld-2.3.2.so
40015000 - 40016000 /lib/ld-2.3.2.so
40016000 - 40017000
40021000 - 40154000 /lib/libc-2.3.2.so
40154000 - 40158000 /lib/libc-2.3.2.so
40158000 - 4015a000
bffff000 - c0000000
```

**1 시스템 콜 처리 함수 작성**

실습의 내용은 커널에서 정보를 출력하도록 되어 있다. 이를 수정하여 응용 프로그램에서 정보를 반환받아 출력해본다. 또는 7장에서 다룰 proc 파일 시스템을 이용하여 출력해보는 것도 한 방법이다.

**[소스 6-4] [실습 6-2]의 시스템 콜 작성**

```

01  asmlinkage int sys_getmeminfo(int pid)
02  {
03      struct task_struct *t;
04      struct mm_struct *mm;
05      struct vm_area_struct *mmap;
06      struct file *file;
07
08      char *filename;
09      char *buf;
10
11      t = find_task_by_pid(pid);
12      mm = t->active_mm;
13      mmap = mm->mmap;
14      buf = (char*)kmalloc( 512, GFP_KERNEL);
15
16      while(mmap) {
17          printk("%08lx - %08lx", mmap->vm_start, mmap->vm_end );
18          file = mmap->vm_file;
19          if(file){
20              filename = d_path(file->f_dentry, file->f_vfsmnt,
21                               buf, 512);
22              printk("%s \n", filename );
23          }
24          else{
25              printk("\n");
26          }
27          mmap = mmap->vm_next;
28      }
29      return 0;
30  }
```

- ① 10행 : 커널 함수 `find_task_by_pid()`를 이용하여 태스크의 `task_struct` 구조체를 얻는다.
- ② 11행~12행 : 태스크의 메모리 정보를 담고 있는 `mm_struct`와 `vm_area_struct` 구조체의 포인터를 얻는다.
- ③ 15행 : 가상 메모리 영역이 존재하는 동안 루프를 반복해서 실행한다.
- ④ 16행 : 각 영역의 시작 주소와 마지막 주소를 출력한다.
- ⑤ 17행~24행 : 만약 영역이 파일에 대한 공간이면 `d_path()` 함수를 사용하여 전체 경로를 포함한 파일의 이름을 얻어 출력한다.
- ⑥ 25행 : 다음 가상 메모리 영역으로 이동한다.

## 2 사용자 응용 프로그램 작성

[소스 6-5] [실습 6-2]의 사용자 응용 프로그램 작성

```

01  /* app_displaymeminfo.c */
02  .....
03  #define __NR_getmeminfo 258
04  _syscall1(int, getmeminfo, int, pid)
05
06  int main(int argc, int *argv[])
07  {
08      int i, pid;
09
10      pid = atoi(argv[1]);
11      i = getmeminfo(pid);
12  }
```

## 3 Makefile 작성과 컴파일 수행

다음과 같이 Makefile을 작성하고 컴파일한다.

[소스 6-6] [실습 6-2]의 Makefile 작성

```

TARGET = getmeminfo
INCLUDE = -isystem /usr/src/linux-2.4.32/include
CFLAGS = -O2 -D__KERNEL__ -DMODULE $(INCLUDE)
//CFLAGS = -O2 $(INCLUDE)
CC = gcc

${TARGET}.o: ${TARGET}.c
```

```
clean:
    rm -rf ${TARGET}.o
```

다음과 같이 응용 프로그램도 컴파일한다.

```
#gcc -o app_displaymeminfo app_displaymeminfo.c
```

#### 4 실행 및 결과 확인

insmod 명령으로 모듈을 적재한 후 응용 프로그램을 실행하여 결과를 확인한다.

```
#insmod getmeminfo.o
#./app_displaymeminfo
```



## 메모리 할당과 해제

커널은 메모리를 할당할 때 버디 시스템 알고리즘과 슬랩 할당자를 사용한다. 버디 시스템 알고리즘은 연속된 빈 페이지 프레임을 그룹으로 관리하여 요청된 페이지 크기에 가장 근접한 연속된 페이지 프레임들을 페이지 단위로 할당한다.

슬랩 할당자는 커널이 같은 타입의 메모리 영역을 반복해서 요청하는 특성에서 착안한 것으로, 미리 객체별로 모아 캐시로 관리하며 객체들이 메모리를 요구할 때 이를 할당한다. 또한 이를 통해 내부 단편화 문제도 해결한다. 현재 시스템에서 관리되고 있는 슬랩 목록은 [그림 6-13]과 같이 `#cat /proc/slabinfo` 명령으로 확인할 수 있다.

```
[root@localhost ~]# cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache      62    70   108    2    2    1
ip_fib_hash     11   112    32    1    1    1
urb_priv        0     0    64    0    0    1
c1ip_arp_cache  0     0   128    0    0    1
ip_mrt_cache    0     0   128    0    0    1
tcp_tw_bucket   11    30   128    1    1    1
tcp_bind_bucket 19   112    32    1    1    1
tcp_open_request 0    30   128    0    1    1
inet_peer_cache 0     58    64    0    1    1
ip_dst_cache    40    45   256    3    3    1
arp_cache       1     30   128    1    1    1
blkdev_requests 3072 3090   128   103   103    1
journal_head    28   154    48    1    2    1
revoke_table    2   250    12    1    1    1
revoke_record   0     0    32    0    0    1
dnotify_cache   12   166    20    1    1    1
file_lock_cache 4     41    92    1    1    1
fasync_cache    2   200    16    1    1    1
uid_cache       8   112    32    1    1    1
skbuff_head_cache 130  135   256    9    9    1
sock            1/5   187  1408   1/2   1/2    4
sigqueue       14    29   132    1    1    1
```

[그림 6-13] 슬랩 목록

메모리를 할당하는 커널 함수들은 다음과 같다. 여기서 각 함수에 사용되는 인자인 `gfp_mask`는 페이지 프레임을 찾기 위해 사용되는 플래그다.

- `alloc_pages(gfp_mask, order)`

$2^{\text{order}}$ 개의 연속된 페이지 프레임을 할당하는 함수다. 반환 값은 첫 번째 페이지 프레임의 페이지 디스크립터 주소다. 페이지 디스크립터는 page 구조체로서, 페이지 프레임의 상태 정보를 가지고 있다.



- `alloc_page(gfp_mask)`  
한 개의 페이지 프레임을 할당하는 매크로로서 `alloc_pages()` 함수를 호출하며, 이때 `order` 값을 0을 사용한다.
- `__get_free_pages(gfp_mask, order)`  
 $2^{\text{order}}$ 개의 연속된 페이지 프레임을 할당하며, 반환 값은 첫 번째 페이지 프레임의 가상 주소다.
- `__get_free_page(gfp_mask)`  
한 개의 페이지 프레임을 할당하며, 반환 값은 첫 번째 페이지 프레임의 가상 주소다.
- `get_zeroed_page(gfp_mask)`  
하나의 페이지 프레임을 할당하며, 이때 페이지 프레임은 0으로 초기화된다.
- `__get_dma_pages(gfp_mask, order)`  
DMA용 페이지 프레임을 할당하는 매크로로서, `__get_free_pages()`를 호출한다. 이때 `gfp_mask`에 `GFP_DMA` 플래그를 포함시킨다.
- `kmalloc(size, flags)`  
`size` 바이트 만큼의 메모리를 할당한다. 요청된 크기가 페이지 프레임보다 큰 경우는 연속된 메모리 영역을 할당한다.
- `vmalloc(size)`  
`size` 바이트 만큼의 불연속적인 물리 메모리를 할당한다.

메모리를 반환하는 함수들은 다음과 같다.

- `__free_pages(page, order)`  
페이지 디스크립터를 인자로 받아  $2^{\text{order}}$ 개의 페이지 프레임을 반환한다.
- `free_pages(addr, order)`  
가상 주소를 인자로 받아  $2^{\text{order}}$ 개의 페이지 프레임을 반환한다.
- `__free_page(page)`  
페이지 디스크립터를 인자로 받아 한 개의 페이지 프레임을 반환한다.
- `free_page(addr)`  
가상 주소를 인자로 받아 한 개의 페이지 프레임을 반환한다.
- `kfree(objp)`  
`kmalloc()`으로 할당된 메모리 영역을 반환한다.
- `vfree(addr)`  
`vmalloc()`으로 할당된 메모리 영역을 반환한다.

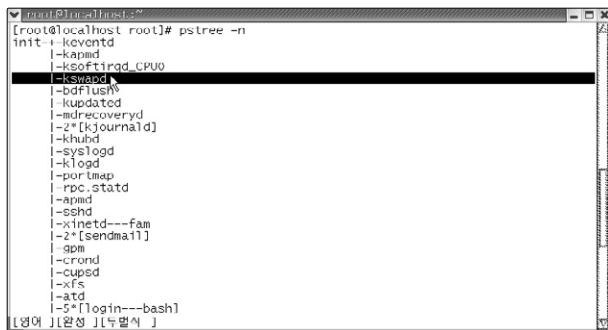
## 페이지 교체

태스크들에게 페이지 프레임을 계속해서 할당하다보면 물리 메모리가 부족해진다. 따라서 태스크가 요구하는 페이지를 원활히 할당할 수 있도록 물리 메모리의 페이지 프레임 가운데 일부를 제거하여 빈 페이지 프레임을 확보하는 방법이 필요하다. 이를 페이지 교체라고 하며, 이때 어떤 페이지를 제거할지를 결정하는 것을 페이지 교체 정책이라고 한다. 페이지 교체 정책은 아주 중요한 부분이다.

알려진 페이지 교체 알고리즘으로는 FIFO, OPT(Optimal Page replacement), LRU (Least Recently Used), LFU(Least Frequently Used), MFU(Most Frequently Used) 등이 있다. 그러나 FIFO 방식은 성능상의 문제로 잘 사용되지 않는다. OPT는 앞으로 제일 오랫동안 참조되지 않을 페이지를 교체하는 알고리즘으로서 가장 최적이지만, 태스크가 앞으로 참조할 메모리의 위치를 알 수 없으므로 실제로 구현할 수가 없다. LFU와 MFU는 참조 횟수가 가장 작거나 큰 페이지를 교체하는 방법이지만, 비용과 성능상의 문제로 잘 쓰이지 않는다. 따라서 페이지 교체 알고리즘으로 LRU 방식을 많이 선호한다.

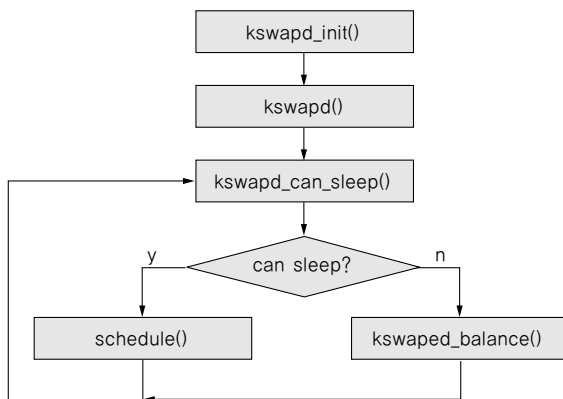
리눅스도 LRU 기법을 사용하는데, 기본적으로 가장 많은 물리 메모리를 차지하고 있는 태스크를 선택하여 그 태스크의 페이지들 중에서 교체할 페이지를 선택할 때 이 정책을 사용하여 결정한다. LRU의 기본 정책은 가장 오랫동안 참조되지 않은 페이지를 교체하는 방식이다. LRU 기법은 상당히 효율적으로 동작하는 것으로 알려져 있다. 그러나 LRU 기법을 그대로 구현하기에는 오버헤드 문제나 코드가 복잡해지는 문제가 있다. 즉 이 알고리즘을 구현하려면 각 페이지마다 참조 시간이나 참조 횟수를 기록하고 유지해야 한다. 따라서 리눅스에서는 완전한 LRU 기법을 사용하는 대신 이와 비슷한 방법을 사용한다. 이러한 방법을 LRU 근사 페이지 교체(LRU Approximation Page Replacement) 기법이라고 하며, 페이지를 참조할 때나 내용이 변경될 때 설정되는 Accessed 플래그와 Dirty 플래그를 이용하여 스왑 아웃시킬 후보를 결정한다.

리눅스에는 [그림 6-14]와 같이 kswapd 데몬이 돌고 있다. 이 데몬은 효율적인 메모리 관리를 위해 불필요하다고 판단되는 페이지들을 주기적으로 스왑 아웃시킴으로써 메모리 내에 빈 프레임을 충분히 유지하도록 한다.



[그림 6-14] kswapd 데몬

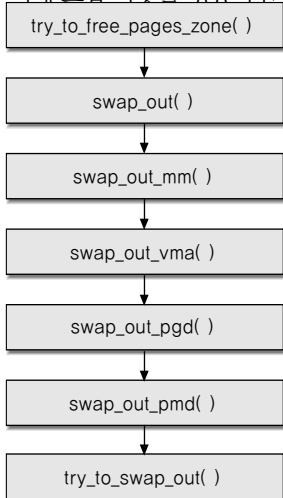
kswapd 데몬은 [그림 6-15]와 같이 커널 초기화 과정에서 호출되는 `kswapd_init` 함수에 의해 생성되며, 이 데몬이 깨어날 때마다 `kswapd_can_sleep()` 함수를 호출하여 자신이 슬립할 수 있는 상태인지를 판별하고 슬립 가능한 상태면 스케줄링을 수행한다. 슬립이 가능하다는 것은 아직 페이지 프레임을 회수할 필요가 없다는 것을 의미한다. 그렇지 않으면, 즉 페이지 프레임을 회수해야 되는 경우라면 `kswapd_balance()` 함수를 호출하여 회수를 시작한다.



[그림 6-15] kswapd 데몬의 수행 흐름

kswapd\_balance( ) 함수에서 시작하여 kswapd\_balance\_pgdat( ) 함수를 거쳐 try\_to\_free\_pages\_zone( ) 함수에서 호출되는 shrink\_caches( ) 함수에서는 회수할 페이지 프레임 수만큼 페이지 프레임을 확보한다. 이때 비활성 리스트에서 회수할 페이지 프레임을 찾는다. 비활성 리스트는 오랫동안 참조되지 않은 페이지들의 리스트를 의미하며, 반대로 활성 리스트는 최근에 참조된 페이지들의 리스트를 의미한다. 이 두 리스트를 LRU 리스트라고 부르며, 이중 연결 리스트로 관리되고 각각의 헤드는 active\_list와 inactive\_list에 있다. 이 두 리스트를 사용하여 페이지를 교체할 때 활성 리스트의 페이지는 교체 대상에서 제외되고, 비활성 리스트 내의 페이지를 대상으로 교체함으로써 효율성을 높인다.

페이지를 스왑 아웃시키는 함수는 swap\_out( )이며, try\_to\_free\_pages\_zone( ) 함수에서 호출되고 [그림 6-16]의 흐름과 같이 여러 함수들을 거쳐 최종적으로 try\_to\_swap\_out( ) 함수를 호출하여 페이지 테이블 엔트리에서 해당 페이지의 스왑 아웃을 시도한다. 이때 스왑 아웃을 성공하면 1을 실패하면 0 값을 반환한다.



[그림 6-16] 스왑 아웃의 함수 흐름도

[그림 6-16]의 함수를 간단히 살펴보면 swap\_out 함수는 태스크에서 스왑 아웃할 페이지를 찾기 위해 태스크의 mm\_struct 구조체를 통해 관리되고 있는 가상 주소 영역을 따라가면서(swap\_out\_vma( )) 페이지 글로벌 디렉토리와 페이지 미들 디렉토리를 차례로 검사하여(swap\_out\_pgd( ), swap\_out\_pmd( )) 페이지 테이블을 찾고, 마지막으로 try\_to\_swap\_out( ) 함수를 호출하여 해당 페이지에 대해 스왑 아웃을 시도한다.



## 요약

- 1 가상 메모리(Virtual Memory)는 물리 메모리(Physical Memory)의 한계를 극복하기 위한 한 방법이다. 여기서 물리 메모리는 시스템에 장착된 실제 메모리인 반면 가상 메모리는 실제로 존재하지는 않지만 마치 아주 큰 메모리가 존재하는 것과 같은 효과를 준다.
- 2 가상 메모리는 각 프로세스마다 4GB씩 할당된다. 3GB는 사용자 주소 공간으로 할당하며, 나머지 1GB는 커널이 차지한다. 그러나 이는 개념적으로 제공하는 공간이며, 실제로 4GB의 공간을 제공하는 것은 아니다.
- 3 가상 주소를 물리 주소로 바꾸는 것을 주소 변환이라고 하며, 이를 위해 페이징 기법을 사용한다.
- 4 각 프로세스마다 하나의 페이지 테이블을 사용하는 경우, 페이지 테이블의 엔트리 개수가 많아지면서 불필요하게 메모리 공간을 많이 차지하므로 계층적인 페이지 테이블을 사용하여 이를 해결한다.
- 5 태스크가 사용하는 메모리에 대한 정보를 유지하기 위한 자료구조는 task\_struct 구조체에 struct mm\_struct \*mm으로 선언되어 있으며, 이를 메모리 디스크립터라고 한다.
- 6 커널은 버디 시스템 알고리즘과 슬랩 할당자를 사용하여 메모리를 할당한다.
- 7 페이지 교체는 물리 메모리의 페이지 프레임 가운데 일부를 제거하여 빈 페이지 프레임을 확보한 후 태스크가 요구하는 페이지를 적재하는 것을 의미한다. 이때 어떤 페이지를 제거할지를 결정하는 것을 페이지 교체 정책이라고 한다.



## 연습문제

- 1 #cat /proc/PID/status 명령으로 다음과 같은 태스크 정보와 메모리 정보를 확인할 수 있다. 이 파일의 내용 가운데 Name, State, VmSize, VmLck, VmRSS, VmData, Vmstk, VmExe, VmLib 항목을 돌려받는 시스템 콜을 작성하시오.

```

root@localhost:~
[root@localhost root]# cat /proc/1/status
Name:   init
State:  S (sleeping)
Tgid:   1
Pid:    1
PPid:   0
TracerPid: 0
Uid:    0           0           0
Gid:    0           0           0
FDSize: 32
Groups:
VmSize: 1388 kB
VmLck:  0 kB
VmRSS:  476 kB
VmData: 24 kB
VmStk:  4 kB
VmExe:  24 kB
VmLib: 1312 kB
SigPnd: 0000000000000000
SigBtk: 0000000000000000
SigTgn: ffffffff7f0d8fc
SigCgt: 00000000280b2603
CapInh: 0000000000000000
CapPrm: 00000000ffffffff
[영어 ][완성 ][두벌식 ]

```

- 2 태스크의 pid를 인자로 받아 각 가상 메모리 영역의 페이지 가운데 실제 물리 메모리에 적재되어 있는 페이지의 개수를 출력하는 시스템 콜 처리 함수를 작성하시오.

[실행 화면 예]

```

08048000 - 0804e000 Page Frame Count 6
0804e000 - 0804f000 Page Frame Count 1
0804f000 - 08052000 Page Frame Count 2
40000000 - 40015000 Page Frame Count 19
40015000 - 40016000 Page Frame Count 1
40016000 - 40017000 Page Frame Count 1
40021000 - 40154000 Page Frame Count 89
40154000 - 40158000 Page Frame Count 4
40158000 - 4015a000 Page Frame Count 2
bffff000 - c0000000 Page Frame Count 1

```