

2 장. 메모리 주소 지정

- 이 장에서는 주소를 지정하는 기법을 다룬다.
- 리눅스 플랫폼에서 하드웨어에서 제공하는 주소 지정 회로를 활용하여 주소지정을 어떻게 구현하는지를 살펴본다.

■ 80386 이전의 초기 컴퓨터

- 1 MB의 메모리 사용
- 16비트의 오프셋을 사용함에 따라 세그먼트의 크기는 항상 64KB
- 세그먼트는 어떠한 세그먼트 레지스터를 사용하느냐에 따라 코드, 데이터, 스택 세그먼트로 결정.
- 세그먼트에 대해 아무런 보호도 이루어 지지 않음(**real mode**).

■ 이후의 컴퓨터

- 사용될 세그먼트에 대해서 기술(**description**) 이 반드시 필요
- 세그먼트에 각종 보호가 이루어져 낮은 권한을 가진 프로세스는 높은 권한의 세그먼트에 대한 접근이 금지(**protected mode**).
- 오프셋의 크기가 세그먼트의 크기를 결정.

메모리 주소

■ 논리 주소(**logical address**)

- 피연산자의 주소나 명령어 주소를 지정할 때 사용.
- 세그먼트와 실제 주소까지의 거리를 나타내는 오프셋

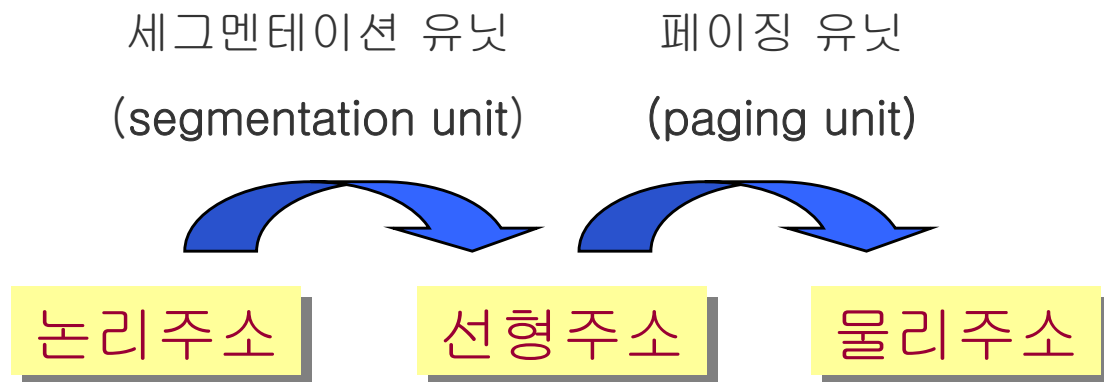
■ 선형 주소(**linear address**)

- 부호 없는 32비트. 선형주소는 보통 16진수로 나타낸다.

■ 물리 주소(**physical address**)

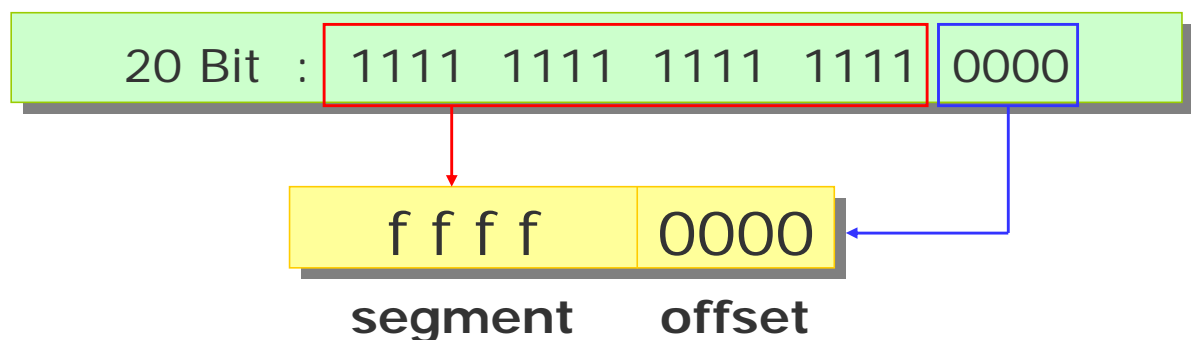
- 메모리 칩에 들어 있는 메모리 셀을 지정하는 데 사용.
- 물리주소는 부호가 없는 32비트 정수로 나타낸다.

논리 주소 변환 과정



논리 주소의 배경

- 초기 컴퓨터의 최대 저장 능력 1MB
 - 1MByte = 1024Byte*1024Byte = 1,048,576Byte(2의 20승)
 - 최초 16비트 컴퓨터가 20비트를 인식할 수 없었기 때문에 4비트를 더 필요하게 됨.

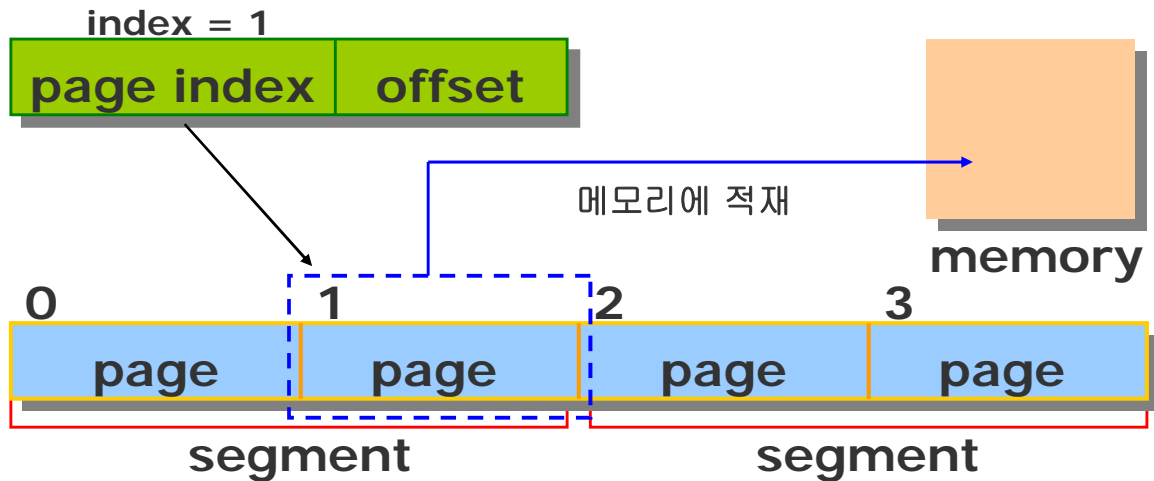


(세그먼트 * 10Hexa) + 오프셋 = 실제메모리 주소

- 4비트의 오프셋이 세그먼트의 크기를 나타냄

선형 주소의 배경

- linear address space를 실제 메모리 크기보다 훨씬 큰 양으로 설정
- 마치 컴퓨터가 linear address space만큼의 크기의 메모리를 가지는 것처럼 시뮬레이션하는 기법

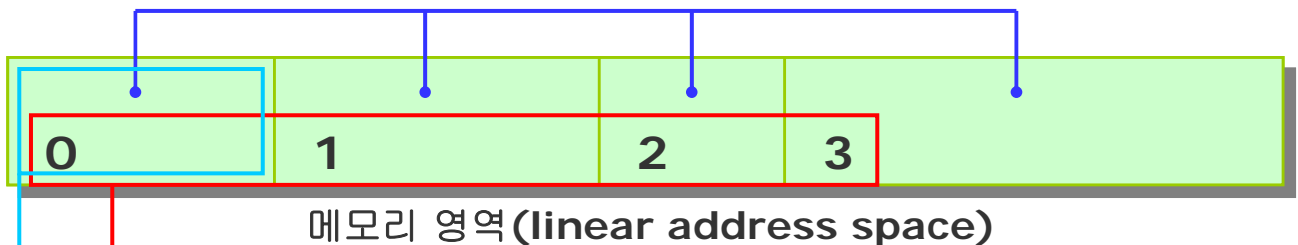


세그멘테이션과 페이징

▪ 세그멘테이션(segmentation)

- 메모리 영역(linear address space)을 세그먼트라고 불리는 프로세스들끼리 서로 침범할 수 없는 (protected) 메모리 영역들로 쪼개서 취급함.

세그먼트(segment)



- 세그먼트들의 인덱스 : 세그먼트 셀렉터
- 세그먼트 셀렉터들을 저장 : 세그먼트 레지스터
- 각 세그먼트들의 정보 저장 : 세그먼트 디스크립터

세그먼트 레지스터

■ 대표적 세그먼트 레지스터

- cs (code segment register)

프로그램 명령어를 포함한 세그먼트를 가리킨다.

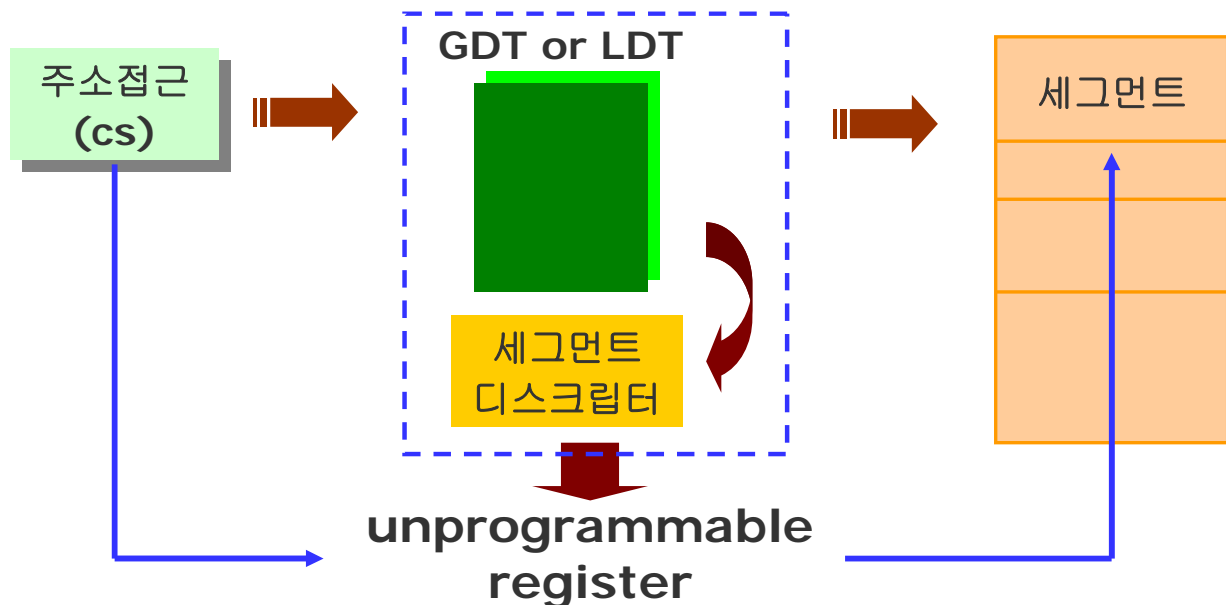
CPU의 현재 권한 수준(CPL, Current Privilege Level)을 나타내는 2비트 값 => 커널 모드(0), 사용자 모드(3)

- ss (stack segment register)

현재 프로세서가 수행하고 있는 프로그램이 사용하고 있는 스택이 위치한 메모리의 세그먼트 셀렉터가 저장된다.

- ds (data segment register)

현재 프로세서가 수행하고 있는 프로그램이 사용하는 데이터가 저장되어 있는 메모리의 세그먼트 셀렉터를 저장한다.



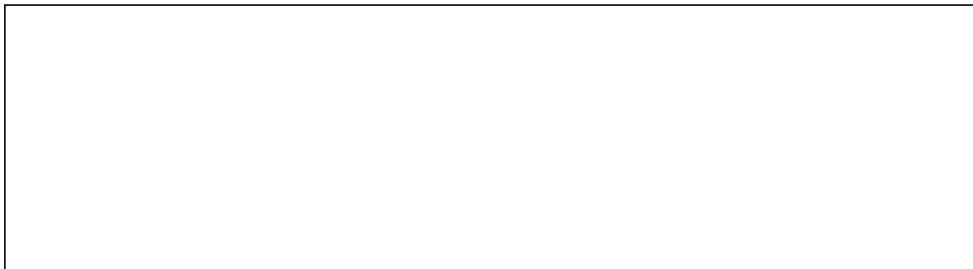
• unprogrammable register

- 세그먼트 레지스터들과 쌍으로 하드웨어 서킷으로 연결
- 세그먼트가 가리키고 있는 다음 실행될 세그먼트 디스크립터를 로딩

리눅스에서 사용하는 세그먼트 목록

- 커널 코드 세그먼트
- 커널 데이터 세그먼트
- 사용자 코드 세그먼트
- 사용자 데이터 세그먼트
- **TSS(Task State Segment)** 세그먼트
 - **task switching**을 하기 위해서 리눅스의 모든 프로세스는 이 세그먼트를 가져야 한다.
 - 프로세스의 수행 환경(**context**)이 저장된다.
 - 오직 **GDT**에만 나타 난다.
- **LDT(Local Descriptor Table)** 세그먼트

하드웨어 페이징

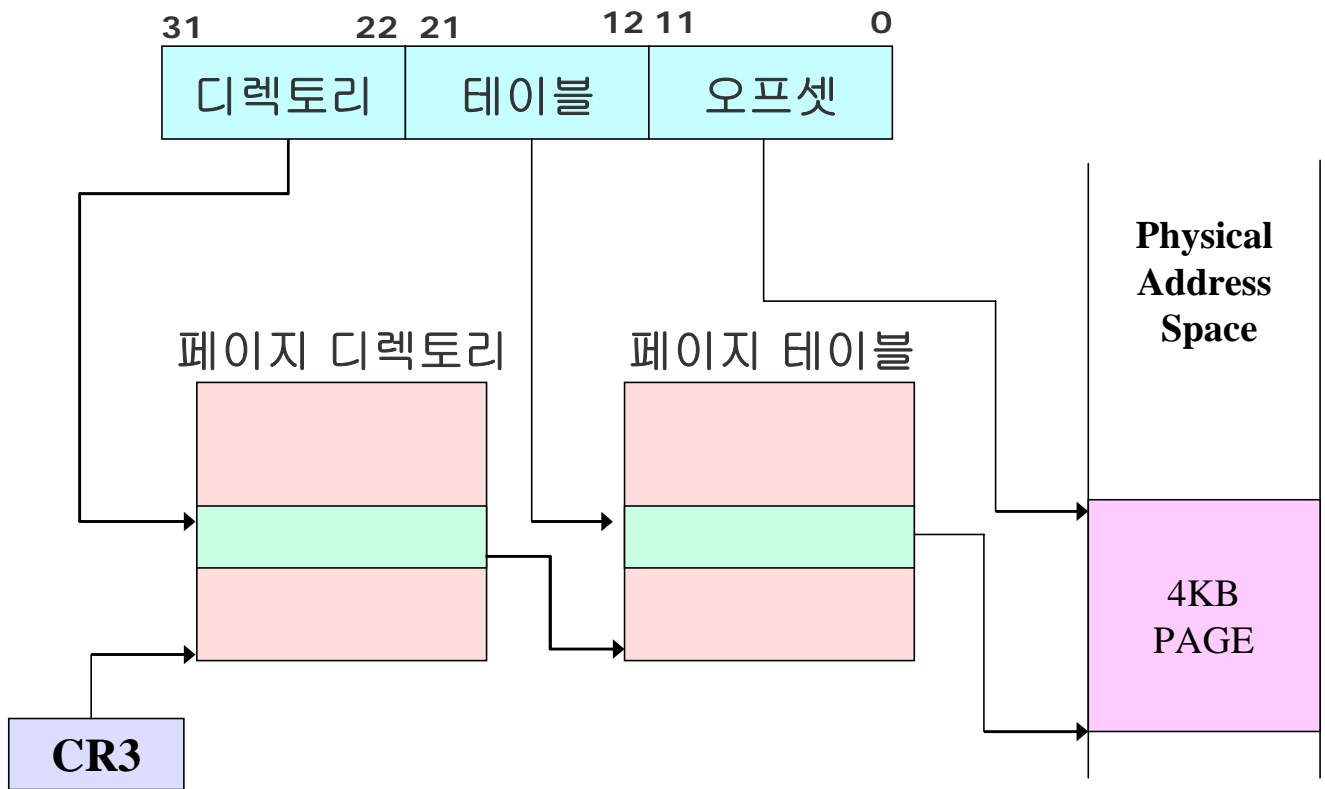


하드웨어 페이징

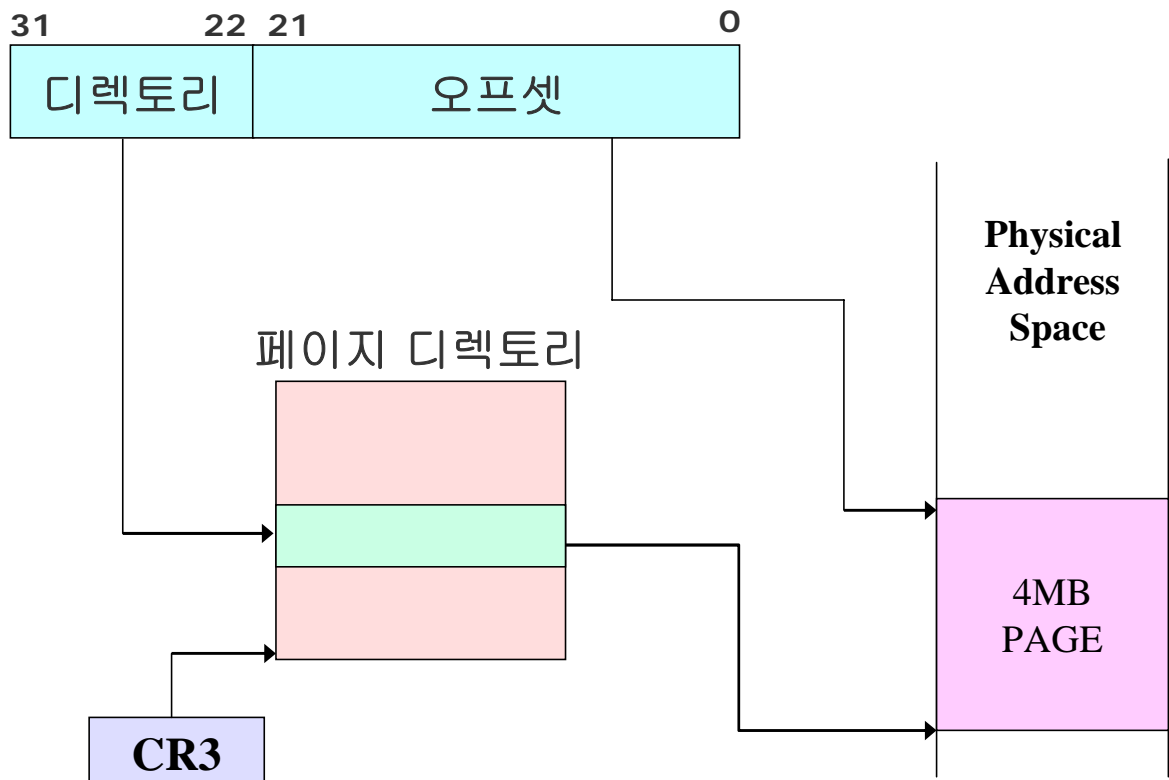
- 페이징 유닛(paging unit)
 - 선형 주소를 물리 주소로 변환
 - 접근 권한 검사
- 페이지
 - 고정크기로 나눈 선형 주소
 - 물리주소와 접근권한 지정 가능
 - 연속된 선형 주소는 연속된 물리 주소로 매핑
- 페이지 프레임
 - 주 메모리의 구성요소(저장영역)
 - 페이지 크기와 일치

- 페이지 테이블
 - 선형주소에서 물리주소로 매핑하는 자료구조
 - 주 메모리에 저장
 - 페이징 유닛 사용전 초기화

정규 페이징



확장 페이징



3단계 페이징

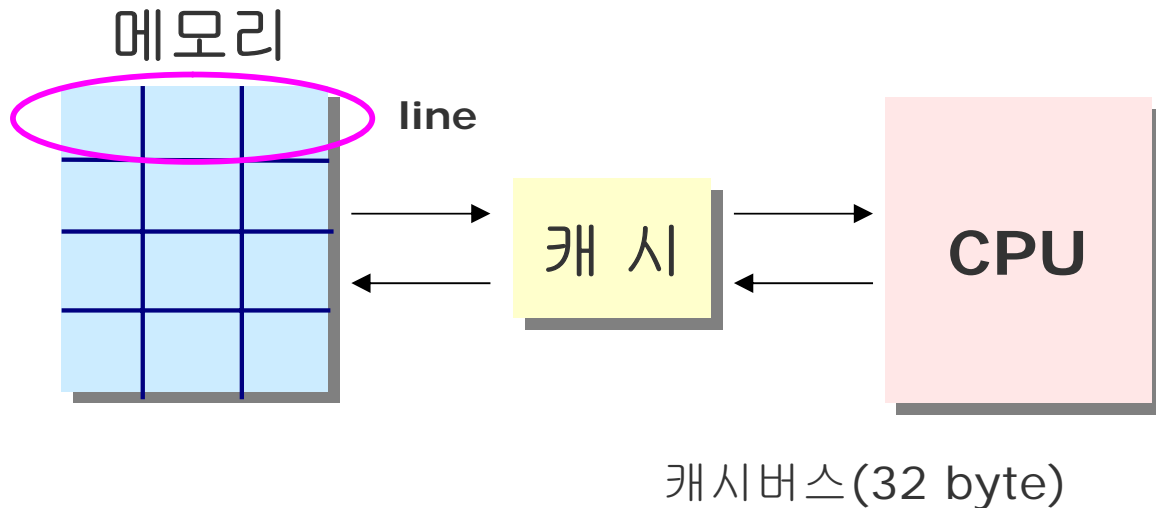
- 알파(Alpha), 울트라스팍(UltraSPARC)
 - 64비트 아키텍처 선택
 - 3단계 페이징 요구
- 컴팩의 알파 마이크로프로세서의 해결책
 - 페이지 프레임의 크기 : 8KB
 - 오프셋 필드 길이 : 13비트
 - 주소중 하위 43비트만 사용
(상위 21비트는 항상 0으로 설정)
 - 3단계 페이지 테이블 사용

하드웨어 보호 정책

- User/Supervisor플래그로 권한 제어
 - CPL(현재권한수준)이 3보다 작은 경우에만 해당 페이지 접근
 - 항상 접근 가능
- 두 가지 권한 수준 사용
 - 읽기, 쓰기

하드웨어 캐시

- 캐쉬는 항상 '라인'의 단위로 **Access** 한다.



■ 캐시 유닛

- 페이징 유닛과 주 메모리 사이에 위치
- 하드웨어 캐시 메모리 : 메모리의 라인들을 실제로 저장
- 캐시 컨트롤러 : 캐시 메모리 각 '라인' 마다의 꼬리표와 캐시라인의 상태를 나타내는 몇 가지 플래그를 포함하는 엔트리를 담고있다.

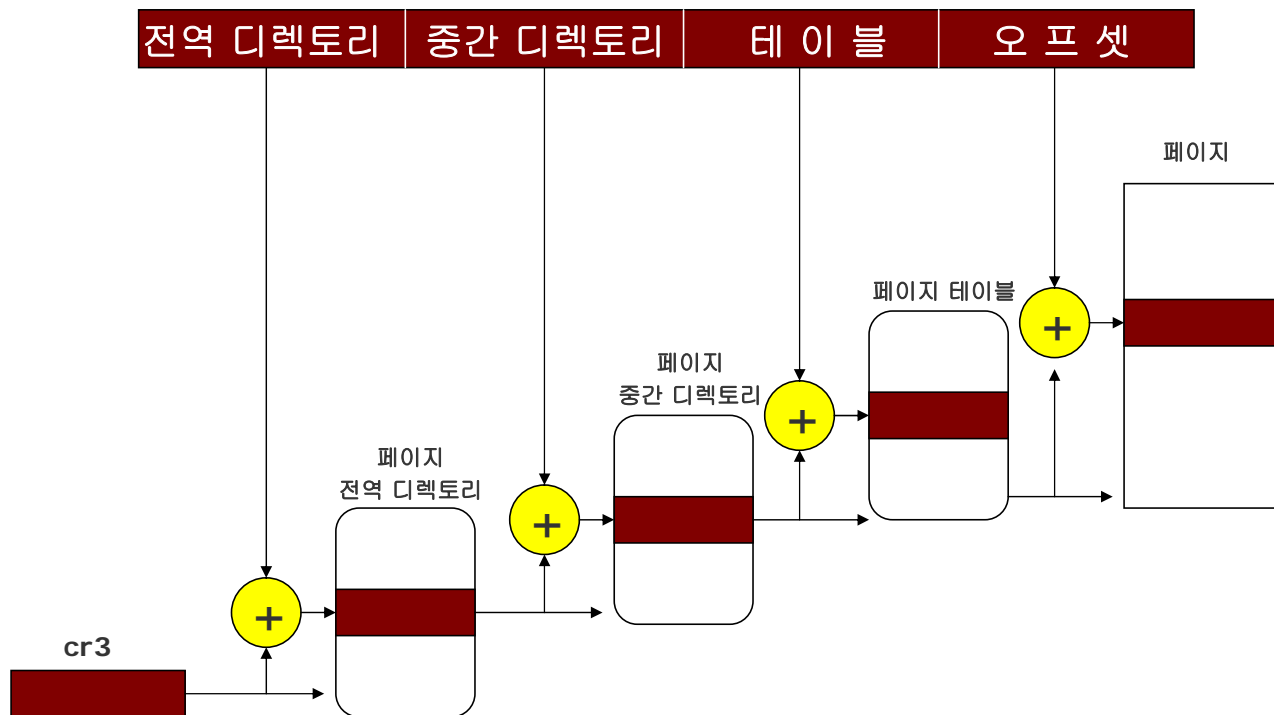
- 캐시의 특징

- 모든 프로세서마다 별도의 하드웨어 캐시가 있으며 각 캐시 동기화 위해 별도의 하드웨어 회로가 필요
- 운영체제가 각 페이지 프레임마다 다른 캐시 관리 정책을 사용(PCD, PWT)

리눅스 페이징



리눅스 페이징



■ 테이블 관리 메크로

■ PAGE_SHIFT

- 오프셋 필드의 비트수 지정

■ PMD_SHIFT

- 페이지 중간 디렉토리가 매핑 할수 있는 영역 크기의 로그 결정

■ PGDIR_SHIFT

- 페이지 전역 디렉토리가 매핑할수 있는 영역의 크기 로그 결정

■ PTRS_PER_PTE, PTRS_PER_PMD, PTRS_PER_PGD

- 페이지 테이블과 중간 디렉토리, 페이지 전역 디렉토리에 들어가는 엔트리 수 계산

페이지 테이블 다루기

- 자료형 변환 매크로
 - `__pte()`, `__pmd()`, `__pgd()`, `__pgprot()`
 - 32비트 unsigned int 자료형 -> 요구한 자료형
 - `pte_val()`, `pmd_val()`, `pgd_val()`, `pgprot_val()`
 - 요구한 자료형 -> 32비트 unsigned int 자료형
- 테이블을 읽고 수정하는 매크로
 - `pmd_bad()`와 `pgd_bad()`
 - 페이지 전역 디렉토리와 중간 디렉토리 검색
 - `pte_bad()`
 - 정의 되어 있지 않음
 - 페이지 테이블 대신 들어 있는 플래그의 현재 값을 알아내는 함수 제공

페이지 테이블 다루기

- 페이지 테이블 엔트리의 플래그의 현재 값을 알아내는 함수
 - `pte_read()`
 - User/Supervisor 플래그 값 반환
 - `pte_write()`
 - present와 read/write가 모두 설정 되어 있으며 1을 반환
 - `pte_exec()`
 - User/Supervisor 플래그 값 반환
 - `pte_dirty`
 - Dirty 플래그 값을 반환
 - `pte_young`
 - Accessed 플래그 값 반환

페이지 테이블 다루기

- 페이지 테이블 엔트리의 플래그 값 설정 함수
 - `pte_wrprotect()`
 - Read/Write 플래그를 0으로 만듦
 - `pte_rdprotect()`와 `pte_exprotect()`
 - User/Supervisor 플래그를 0으로 만듦
 - `pte_mkwwrite()`
 - Read/Write 플래그를 1로 설정
 - `pte_mkread()`와 `pte_mkexec()`
 - User/Supervisor 플래그를 1로 설정
 - `pte_mkdirty()`와 `pte_mkclean()`
 - Dirty 플래그를 각각 1과 0으로 설정해 페이지 수정 여부 표시

페이지 테이블 다루기

- 페이지 테이블 엔트리의 플래그 값 설정 함수
 - `pte_mkyoung()`와 `pte_mkhold()`
 - Accessed 플래그를 각각 0과 1로 설정해 페이지 접근 여부 표시
 - `pte_modify(p,y)`
 - 페이지 테이블 엔트리 `p`내의 모든 권한을 지정한 `y`값으로 설정
 - `set_pte`
 - 지정한 값을 페이지 테이블 엔트리에 기록

페이지 테이블 다루기

- 32비트 페이지 엔트리 구성 함수

- `mk_pte()`
 - 선형 주소와 여러 접근 권한을 조합해 페이지 테이블 엔트리 만듦
- `mk_pte_phys()`
 - 물리 주소와 페이지 접근 권한을 조합하여 페이지 테이블 엔트리 구성
- `pte_page()`와 `pmd_page()`
 - 페이지 테이블 엔트리에서 페이지의 선형주소를 알아냄
- `pgd_offset(p, a)`
 - 메모리 디스크립터 `p`와 선형 주소 `a`를 매개 변수로 만듦
 - 페이지 전역 디렉토리에서 주소 `a`에 해당하는 엔트리 주소 만듦
- `pmd_offset(p, a)`
 - 페이지 전역 디렉토리 `p`와 선형 주소 `a`를 매개 변수로 받음
 - `p`가 참조하는 페이지 중간 디렉토리중에 주소 `a`에 해당 하는 엔트리 주소 반환

페이지 테이블 다루기

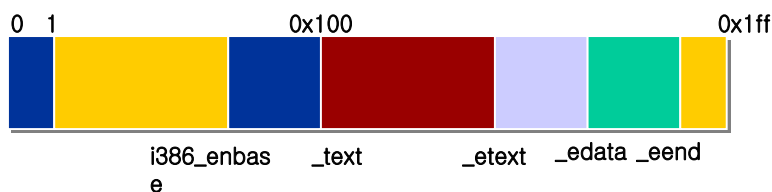
- 그 외의 함수와 매크로






- `pgd_alloc()`
 - `get_pgd_fast()` 함수를 호출하여 새로운 페이지 전역 디렉토리 할당
- `pmd_alloc(p, a)`
 - 3단계 페이징 시스템에서 선형 주소 `a`에 새로운 페이지 중간 디렉토리를 할당할 수 있도록 정의
- `pte_alloc(p, a)`
 - 페이지 중간 디렉토리 엔트리 `p`와 선형주소 `a`를 매개변수로 받아 `a`에 해당하는 페이지 테이블 엔트리 주소 반환
- `pte_free()`, `pte_free_kernel()`, `pgd_free()`
 - 페이지 테이블을 해제 하고 해제된 페이지 프레임을 해당 캐시에 넣음
- `free_one_pmd()`
 - `pte_free()`를 호출 하여 페이지 테이블을 해제

페이지 테이블 다루기

- 그 외의 함수와 매크로
 - `free_one_pgd()`
 - 프로세스의 페이지 전역 디렉토리를 설정
 - `new_page_tables()`
 - 프로세스 주소 공간을 구성하는데 필요한 페이지 전역 디렉토리
모든 페이지 테이블을 할당 받음
 - `clear_page_tables()`
 - `free_one_pgd()` 함수를 반복 호출하여 프로세스의 페이지 테이블
내용을 지움
 - `free_page_tables()`
 - 프로세스의 페이지 전역 디렉토리까지 해제

예약된 페이지 프레임



-  사용할 수 없는 페이지 프레임
-  사용 가능한 페이지 프레임
-  커널 코드
-  초기화된 커널 데이터
-  초기화 되지 않은 커널 데이터

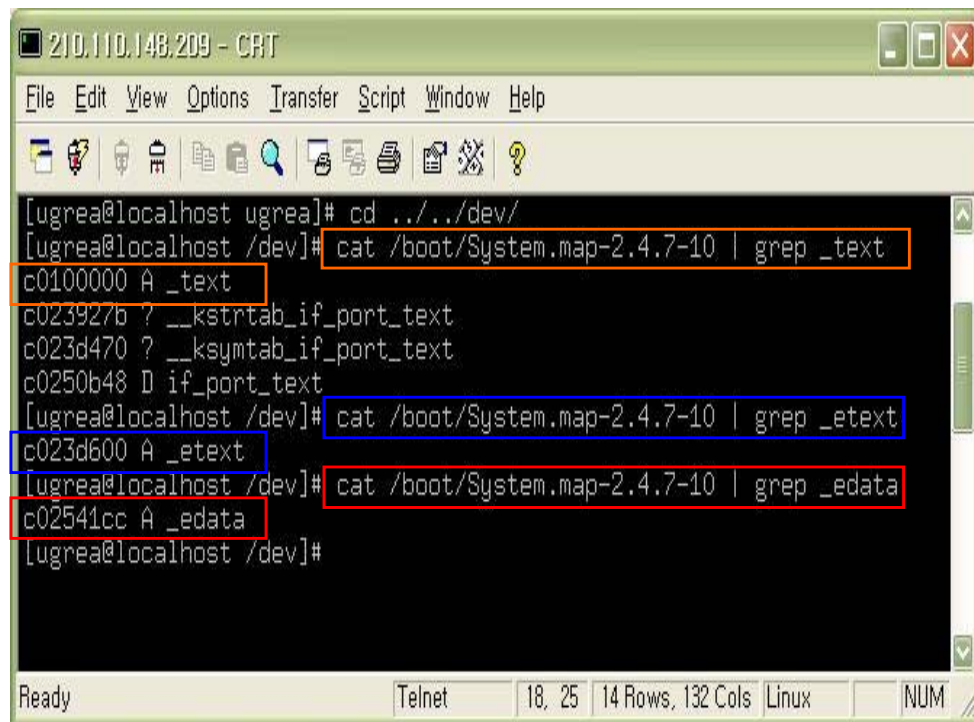
`_text` : 커널코드의 첫 번째 바이트 주소

`_etext` : 커널코드의 끝

`_etext` 다음 : 초기화 된 데이터 시작, `_edata` : 끝

`_edata` 다음 : 초기화 되지 않은 데이터 시작, `_end` : 끝

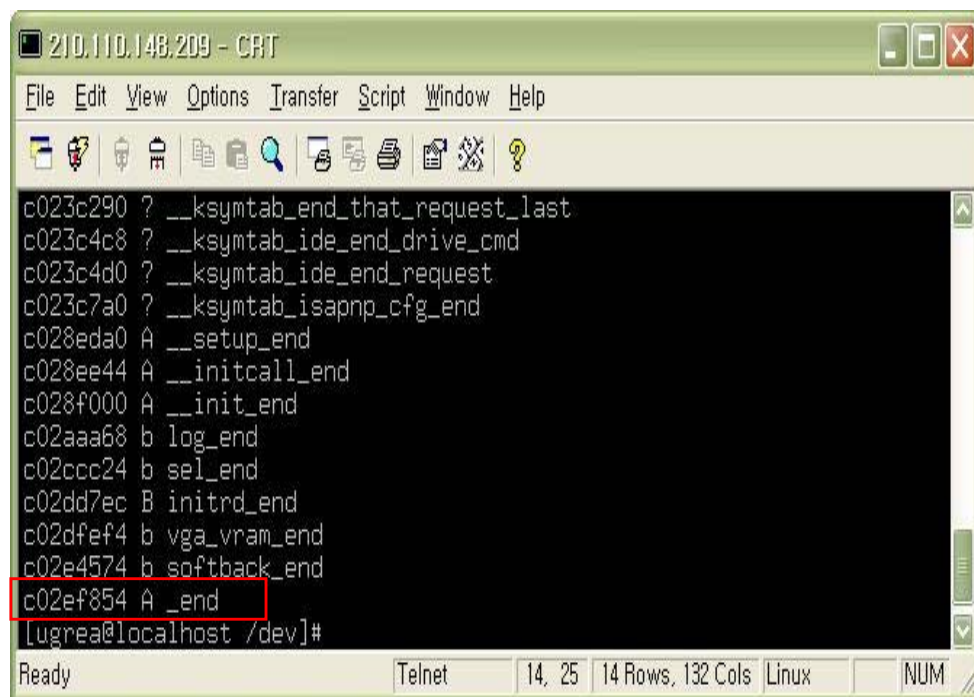
예약된 페이지 프레임



A screenshot of a Telnet window titled "210.110.148.209 - CRT". The window has a menu bar (File, Edit, View, Options, Transfer, Script, Window, Help) and a toolbar. The terminal shows a user "ugrea" at "localhost" in the "/dev" directory. The user runs the command "cat /boot/System.map-2.4.7-10 | grep _text", which outputs "c0100000 A _text". Then, the user runs "cat /boot/System.map-2.4.7-10 | grep _etext", which outputs "c023d600 A _etext". Finally, the user runs "cat /boot/System.map-2.4.7-10 | grep _edata", which outputs "c02541cc A _edata". The status bar at the bottom shows "Ready", "Telnet", "18, 25", "14 Rows, 132 Cols", "Linux", and "NUM".

```
[ugrea@localhost ugrea]# cd ../../dev/
[ugrea@localhost /dev]# cat /boot/System.map-2.4.7-10 | grep _text
c0100000 A _text
c023927b ? __kstrtab_if_port_text
c023d470 ? __ksymtab_if_port_text
c0250b48 D if_port_text
[ugrea@localhost /dev]# cat /boot/System.map-2.4.7-10 | grep _etext
c023d600 A _etext
[ugrea@localhost /dev]# cat /boot/System.map-2.4.7-10 | grep _edata
c02541cc A _edata
[ugrea@localhost /dev]#
```

예약된 페이지 프레임



A screenshot of a Telnet window titled "210.110.148.209 - CRT". The window has a menu bar (File, Edit, View, Options, Transfer, Script, Window, Help) and a toolbar. The terminal shows a user "ugrea" at "localhost" in the "/dev" directory. The terminal displays a list of memory addresses and symbols: "c023c290 ? __ksymtab_end_that_request_last", "c023c4c8 ? __ksymtab_id_end_drive_cmd", "c023c4d0 ? __ksymtab_id_end_request", "c023c7a0 ? __ksymtab_isapnp_cfg_end", "c028eda0 A __setup_end", "c028ee44 A __initcall_end", "c028f000 A __init_end", "c02aaa68 b log_end", "c02ccc24 b sel_end", "c02dd7ec B initrd_end", "c02dfef4 b vga_vram_end", "c02e4574 b softback_end", and "c02ef854 A _end". The status bar at the bottom shows "Ready", "Telnet", "14, 25", "14 Rows, 132 Cols", "Linux", and "NUM".

```
c023c290 ? __ksymtab_end_that_request_last
c023c4c8 ? __ksymtab_id_end_drive_cmd
c023c4d0 ? __ksymtab_id_end_request
c023c7a0 ? __ksymtab_isapnp_cfg_end
c028eda0 A __setup_end
c028ee44 A __initcall_end
c028f000 A __init_end
c02aaa68 b log_end
c02ccc24 b sel_end
c02dd7ec B initrd_end
c02dfef4 b vga_vram_end
c02e4574 b softback_end
c02ef854 A _end
[ugrea@localhost /dev]#
```

프로세스 페이지 테이블

- 프로세스의 선형 주소 공간
 - 0x00000000 ~ PAGE_OFFSET - 1
 - 프로세스가 사용자 모드에 있든 커널 모드에 있든 항상 접근 가능
 - PAGE_OFFSET ~ 0xffffffff
 - 프로세스가 커널 모드에 있을 때만 접근 가능

커널 페이지 테이블

- 초기화
 - 커널을 램에 올리기에 충분한 크기 로 제한된 주소 공간 만듬 (4MB)
 - 커널이 모든 램을 활용해 페이지 테이블을 제대로 만듬
- 최종 커널 페이지 테이블
 - 최종 매핑 : PAGE_OFFSET부터 시작 하는 선형 주소를 0부터 시작 하는 물리적 주소로 변환
 - paging_init()의 매개 변수
 - start_mem : 커널 코드와 데이터 영역 바로 뒤 첫째 바이트의 선형 주소
 - end_mem : 메모리 끝의 선형 주소

리눅스 2.4 예상

- 두 가지 큰 변화

- 현존하는 프로세스와 관련된 모든 TSS 세그먼트 디스크립터를 더 이상 GDT에 저장하지 않음

- 물리 주소 지정과 관련

- ; PAE를 사용해 64GB 크기의 램 지원,
표준 32비트 물리 주소에 4비트 추가

변환 참조 버퍼(TLB)

- 선형 주소 변환속도 증가

- 페이지 테이블에 접근하여 변환된 물리 주소를 TLB 엔트리에 저장

- `invlpg` (Invalidate Translation Look-Aside Buffer Entry (486+ only))

- TLB 엔트리 무효화

- `movl $addr, %eax`

- `invlpg (%eax)`

- TLB 엔트리 갱신

- `movl %cr3, %eax`

- `movl %eax, %cr3`