CS632/SEP564: Embedded Operating Systems (Fall 2008)

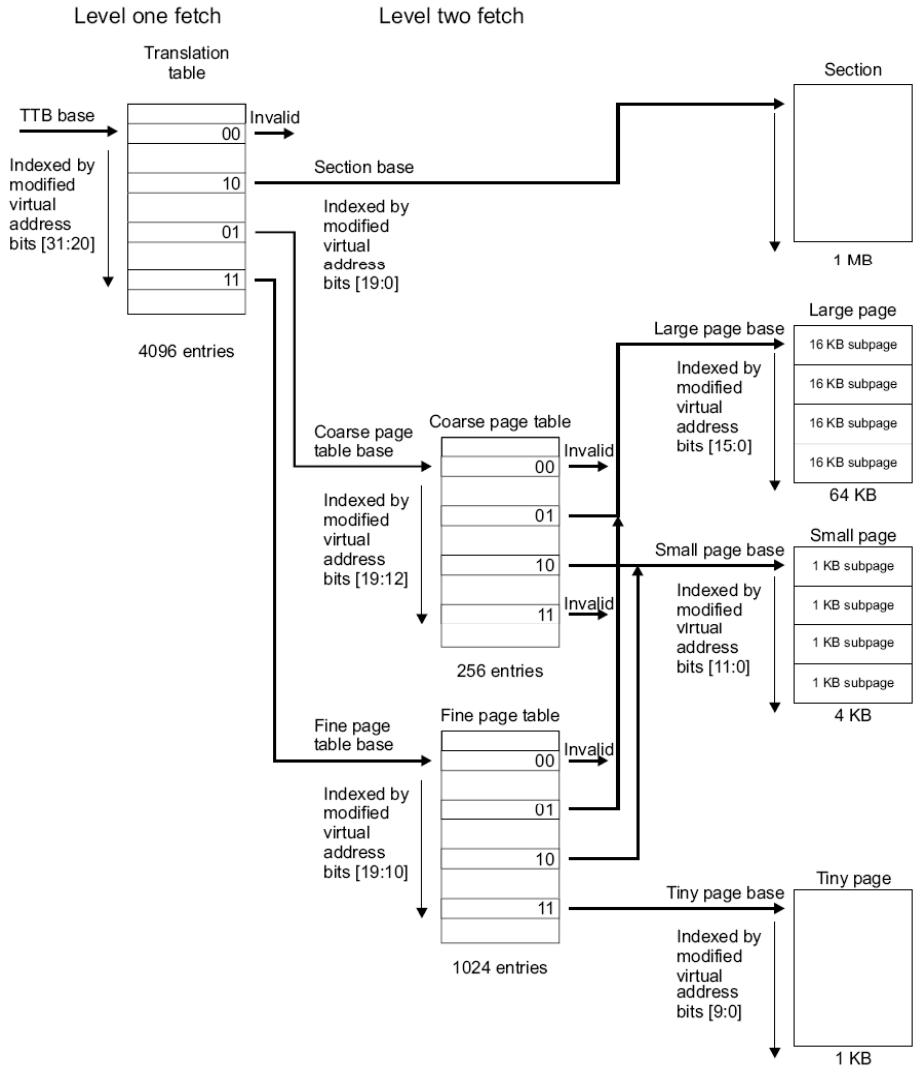
ARM Memory Management Unit (MMU)

KAIST

Overview

- **ARMv4 MMU features**
  - Use paging
    - 32-bit virtual address space to 32-bit physical memory
  - Support multiple page sizes
    - 1MB (sections), 64KB (large pages), 4KB (small pages), 1KB (tiny pages)
    - Access permissions for large pages and small pages can be specified separately for each quarter of the page (subpage).
  - 2-level page tables
  - Hardware page table walks
  - Separate instruction and data TLB (64 entry each)

Address Translation (1)



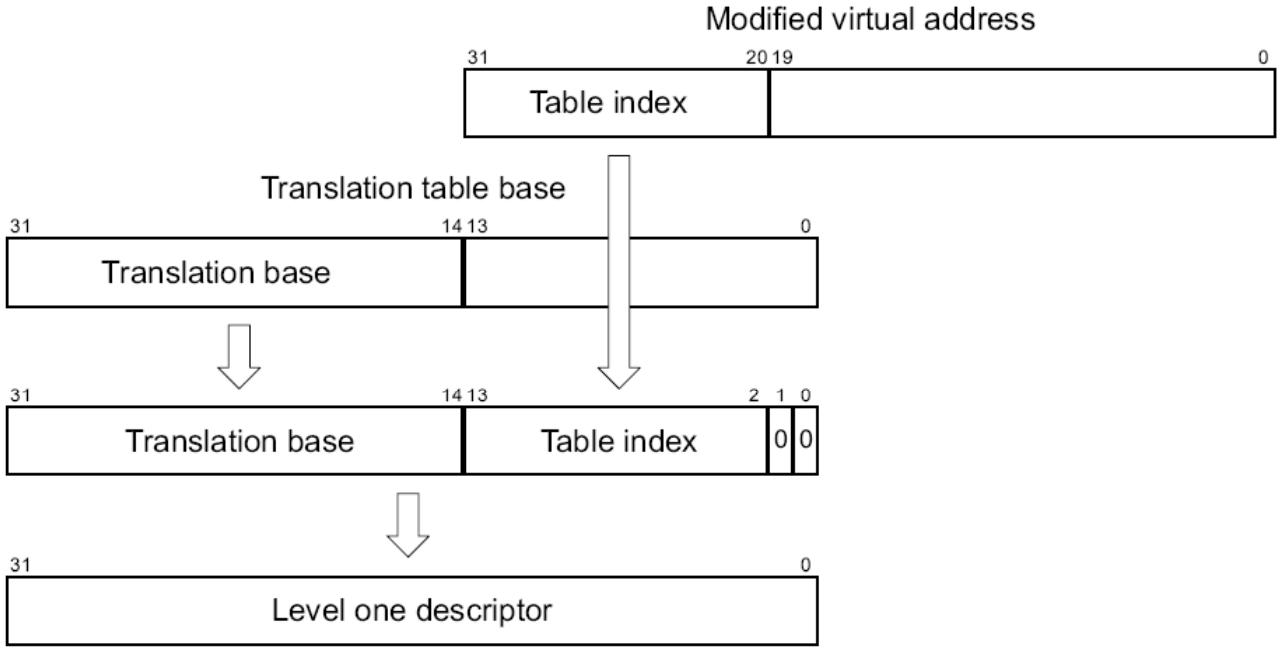
Address Translation (2)

- **Page tables**
  - Two levels
    - L1 master page table (or section page table)
    - L2: coarse or fine page table

Name	Type	Memory consumed by page table (KB)	Page sizes supported (KB)	Number of page table entries
Master/section	Level 1	16	1024	4096
Coarse	Level 2	1	4 or 64	256
Fine	Level 2	4	1, 4, or 64	1024

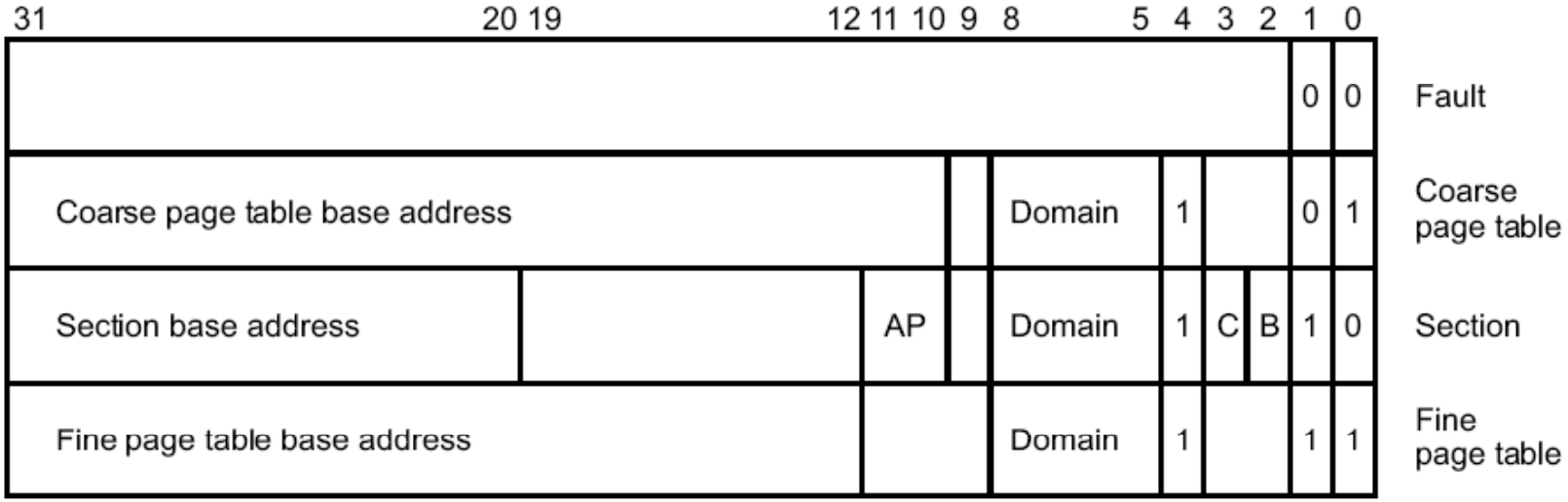
Address Translation (3)

- **Translation table base register**
  - CP15 register 2
  - Hold the physical addr. of the base of the first-level table.
  - The first-level page table must be on a 16KB boundary.



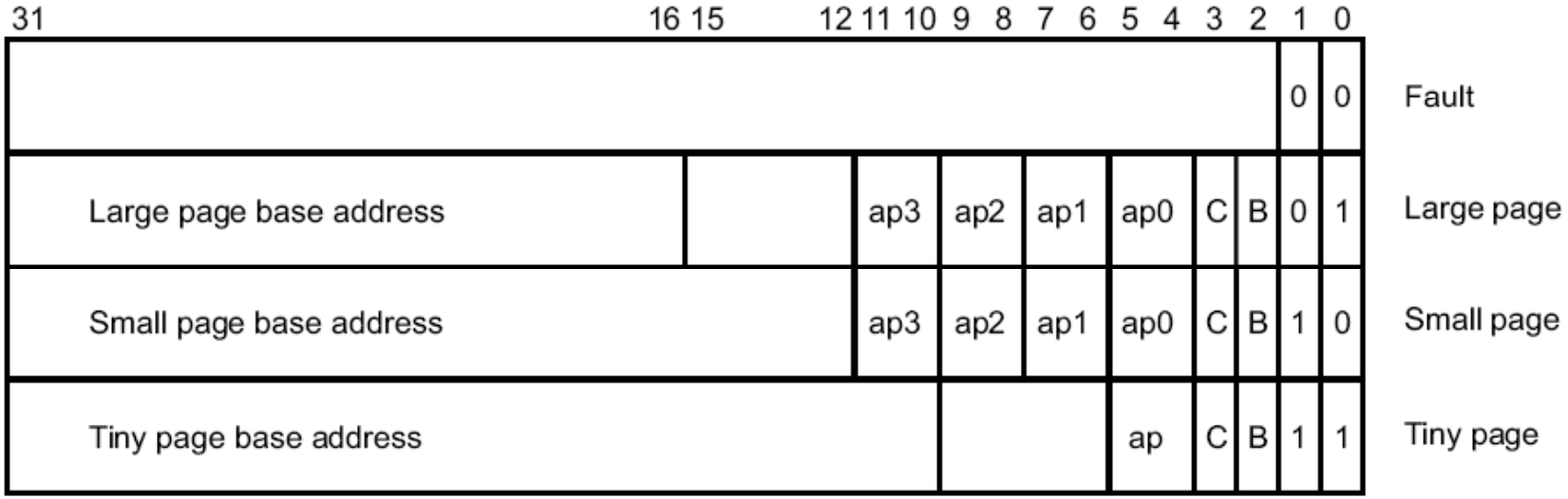
Address Translation (4)

- **L1 page table entries**
  - A 1MB section translation entry
  - A directory entry that points to a fine L2 page table
  - A directory entry that points to a coarse L2 page table
  - A fault entry that generates an abort exception



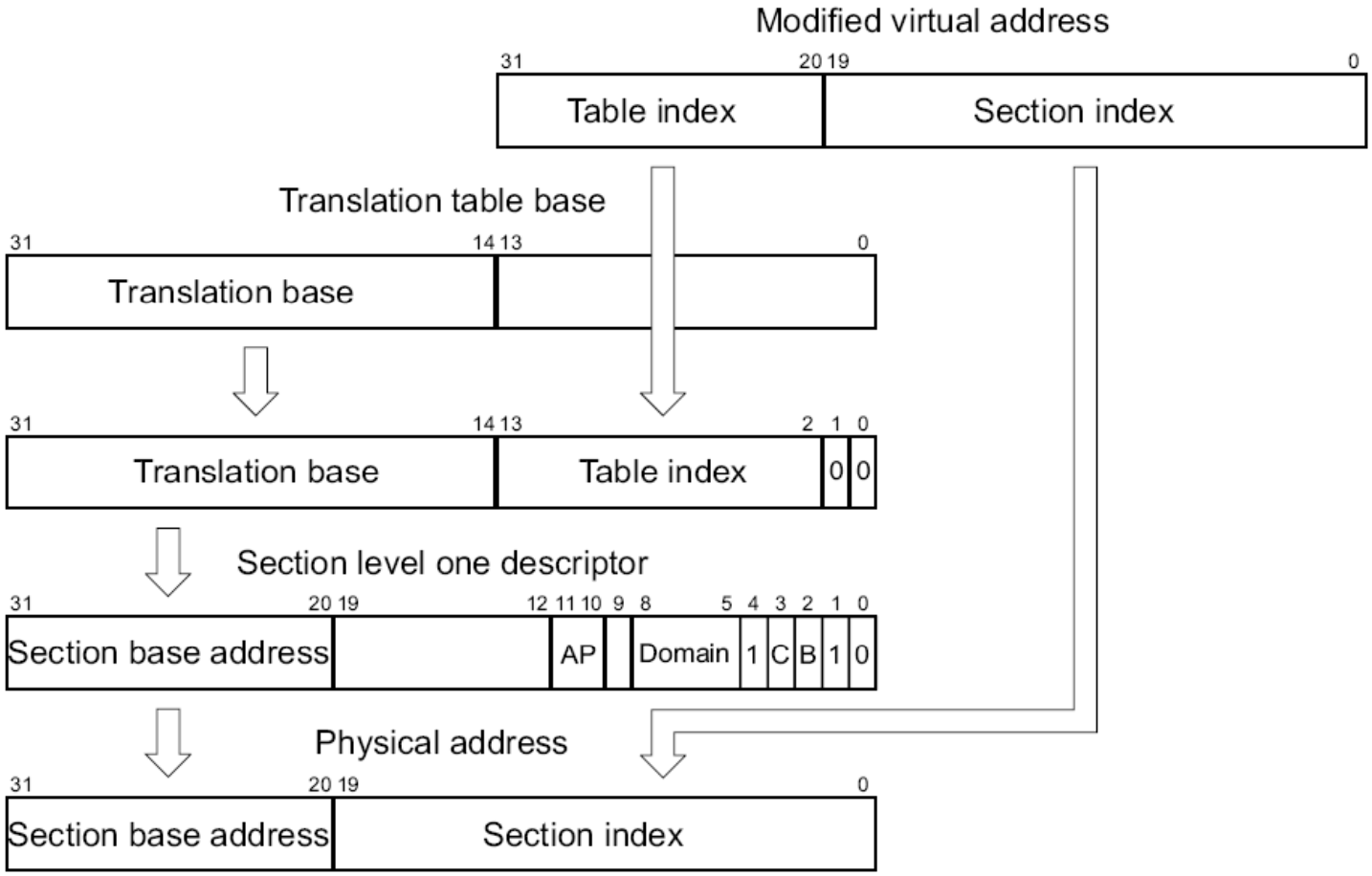
Address Translation (5)

- **L2 page table entries**
  - A large page entry for 64KB page frame
  - A small page entry for 4KB page frame
  - A tiny page entry for 1KB page frame
  - A fault page entry



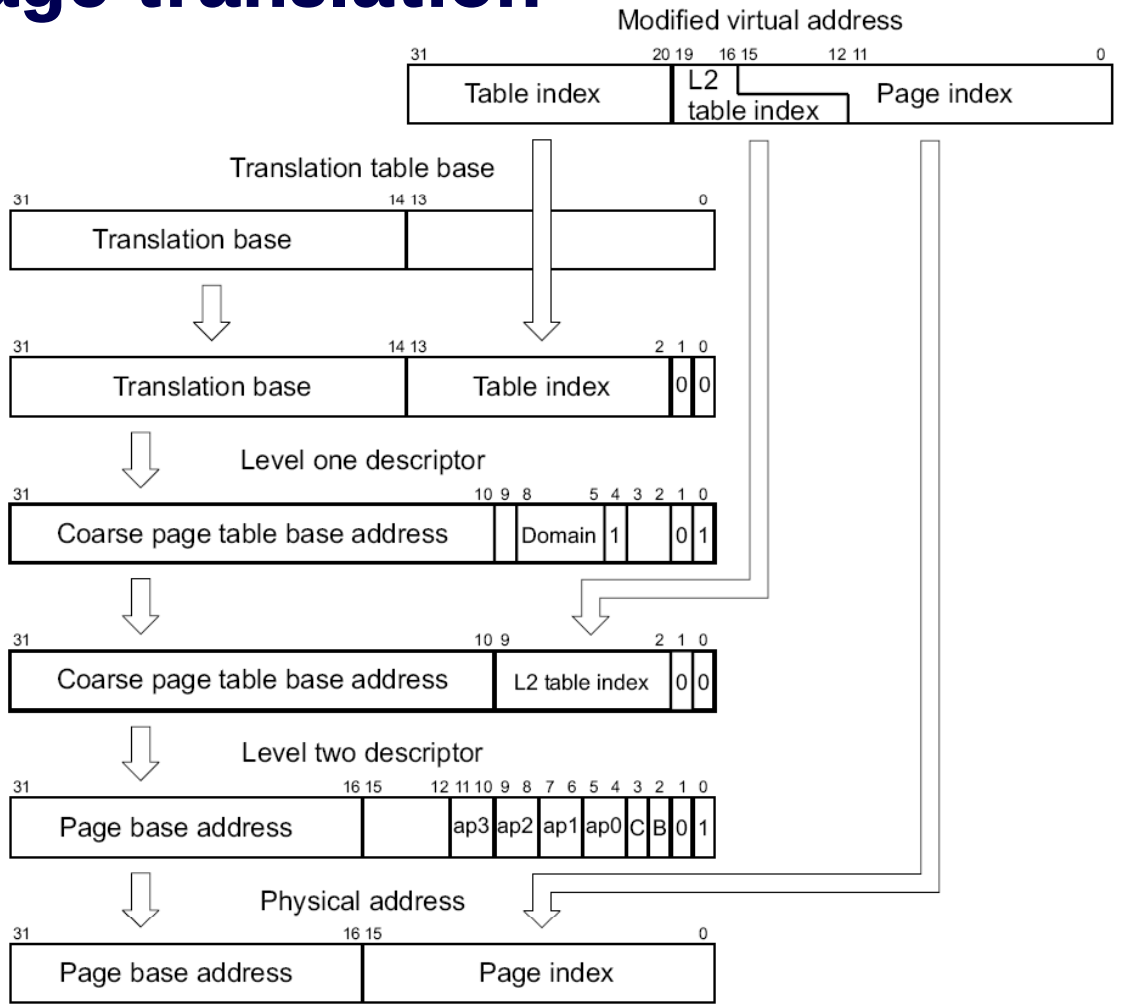
Address Translation (6)

- **Section translation**



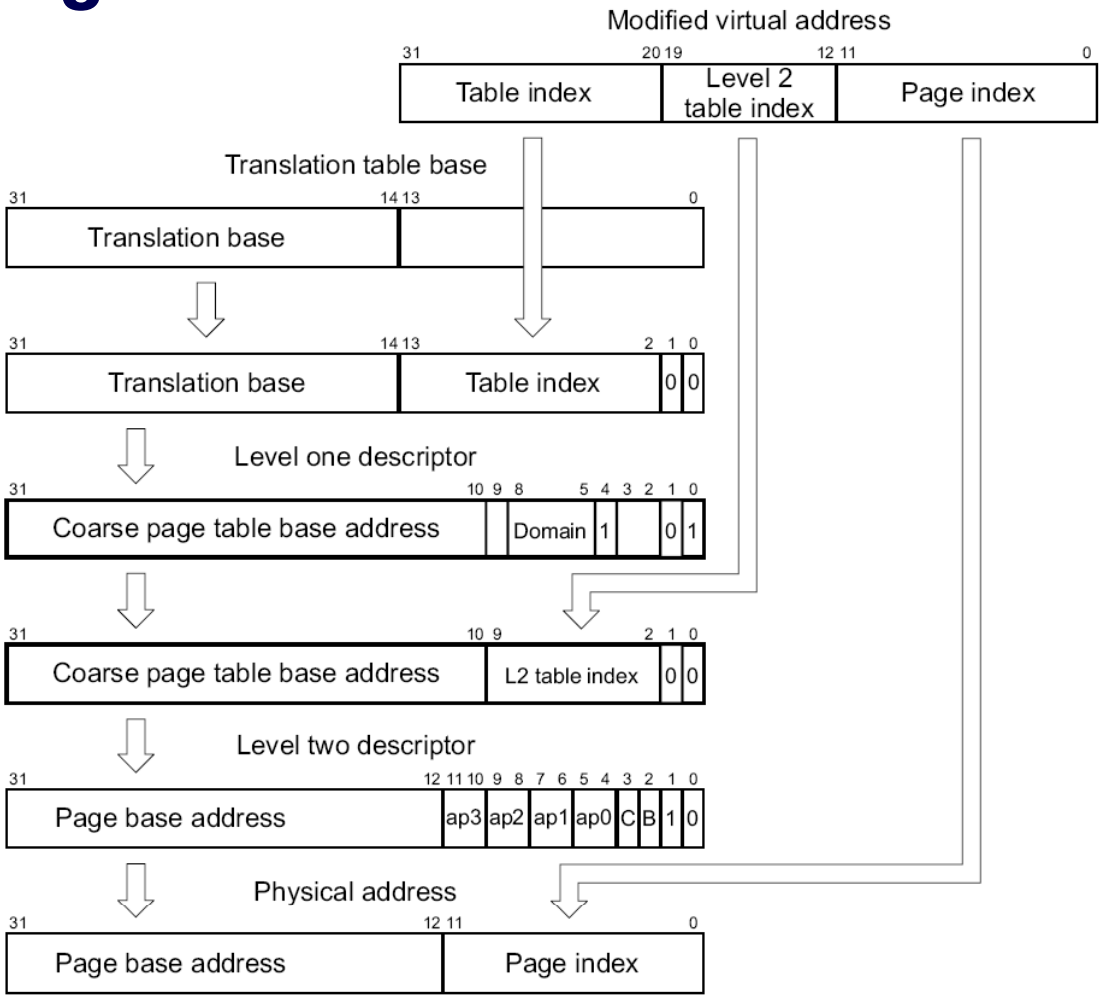
Address Translation (7)

- **Large page translation**



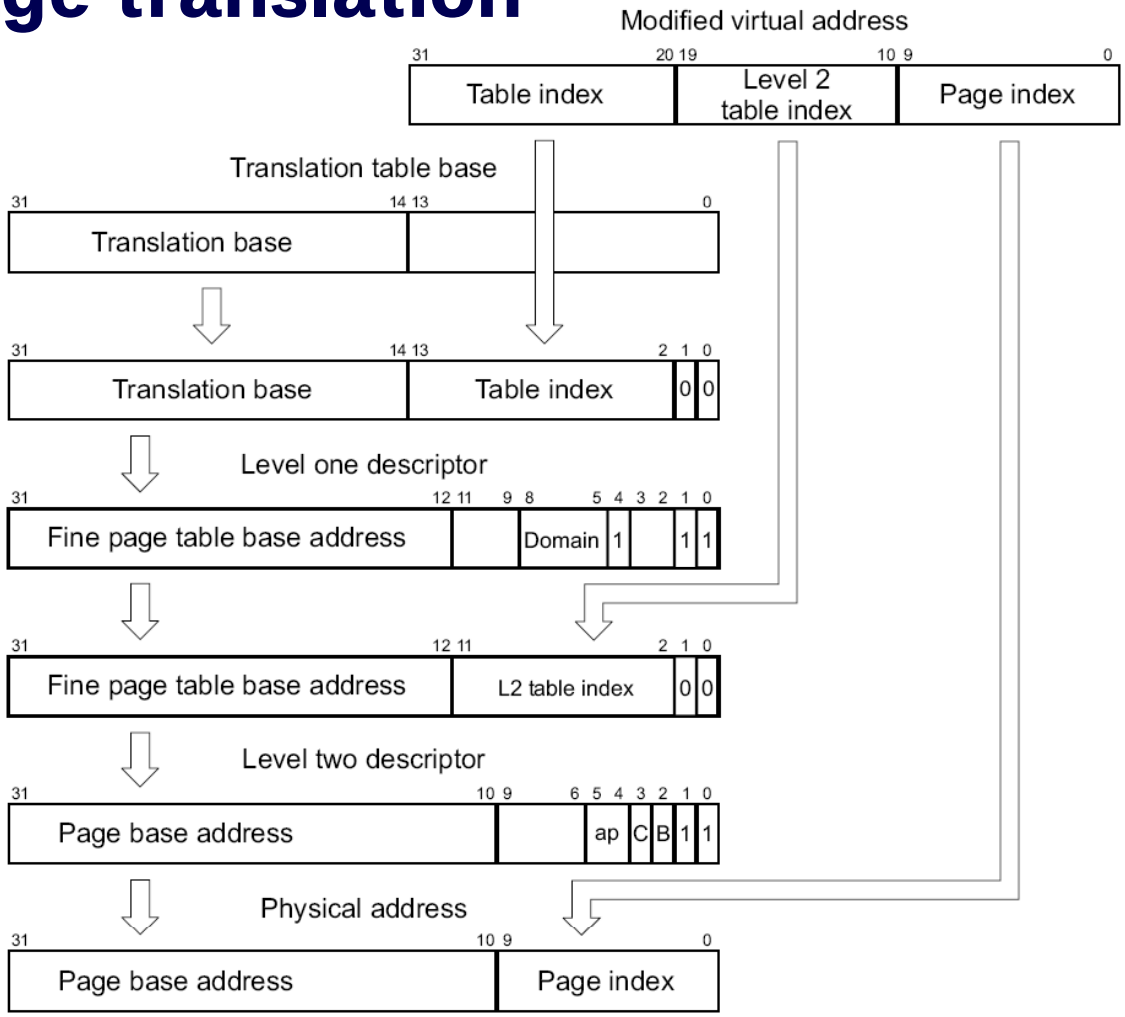
Address Translation (8)

- **Small page translation**



Address Translation (9)

- **Tiny page translation**



Address Translation (10)

- **Selecting a page size**
  - The smaller the page size, the more page frames.
  - The smaller the page size, the less the internal fragmentation.
  - The larger the page size, the more likely the system will load referenced code and data.
  - Large pages are more efficient as the access time to secondary storage increases.
  - As the page size increases, each TLB entry represents more area in memory.
  - Consider the required page table size.

TLB (1)

- **Translation Lookaside Buffer**
  - A special cache of recently used page translations.
    - Unified TLB (ARM720T) or Separate I/D TLB (ARM920T, ARM922T, etc.)
    - Each TLB caches 64 translated entries.
  - Use a round-robin replacement algorithm on a TLB miss.
  - Require page table walk on a TLB miss
    - May search up to two page tables
    - Performed by MMU hardware
  - TLB operations
    - Invalidate all or selected entry
    - Lock translations in the TLB

TLB (2)

- **Subpages**
  - Access permissions can be specified for subpages of small and large pages.
  - If, during a page walk, a small or large page has a non-identical subpage permission, only the subpage being accessed is written into the TLB.
    - If the subpage permission differs in a large page, a 16KB subpage entry is written into TLB.
    - Otherwise, a 64KB entry is put in the TLB.
  - If the page entry then has to be invalidated, all four subpages should be invalidated separately.

CS632/SEP564: Embedded Operating Systems (Fall 2008)

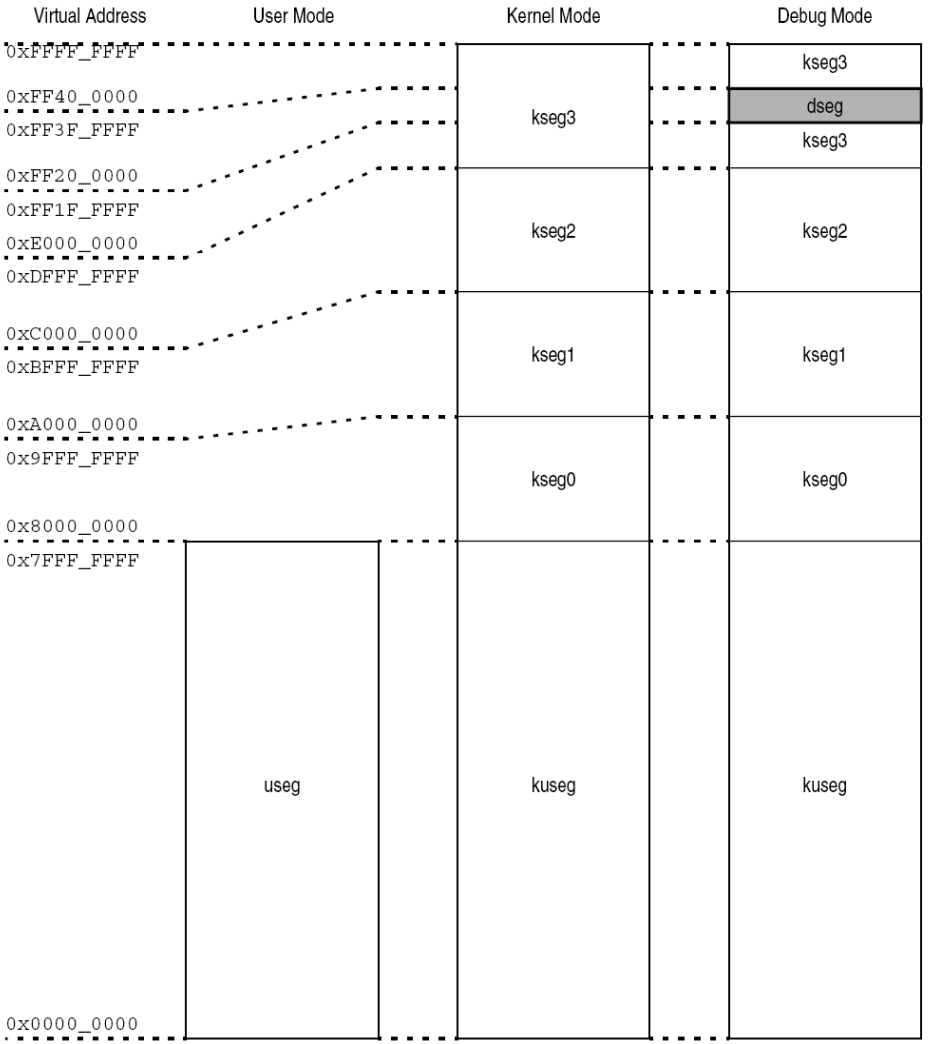
MIPS 4KEc Memory Management Unit (MMU)

KAIST

Virtual Segments (1)

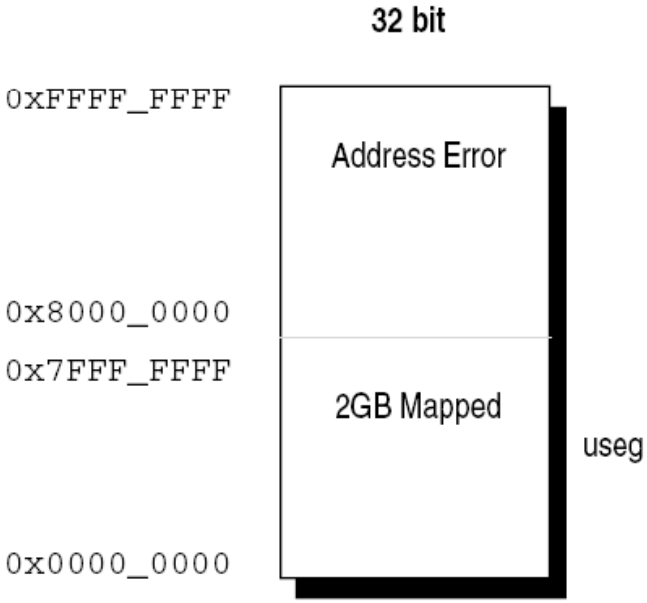
- **Unmapped segments**
  - Does not use TLB
  - A fixed simple translation from virtual to physical address
  - Except for kseg0, always uncached
    - The cacheability of kseg0 is configurable (K0 field in CP0 register)
- **Mapped segments**
  - The translation is handled on a per-page basis
  - The page can be cacheable or not

Virtual Segments (2)



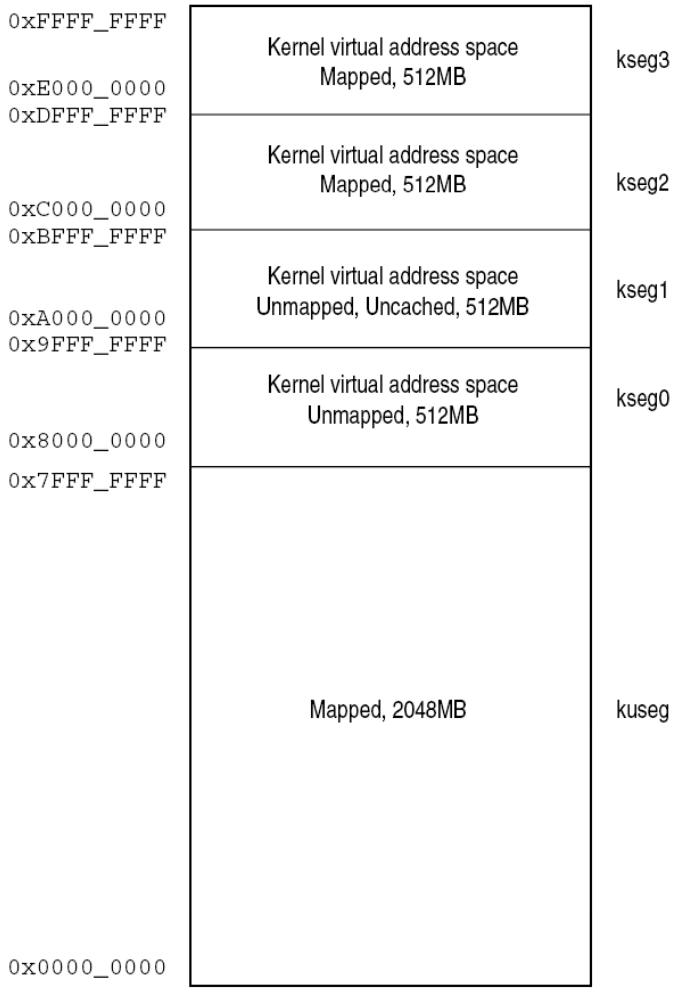
Virtual Segments (3)

- **User mode**
  - Can only access the lower half of the virtual memory map.
  - Any attempt to reference an address with the MSB set causes an address error exception.
  - All references to useg through the TLB
  - Bit settings within the TLB entry for the page determine the cacheability of a reference



Virtual Segments (4)

- **Kernel mode**
  - kseg0: Unmapped
    - PA = VA – 0x80000000
    - The cacheability is configurable by the K0 field in the CP0 register
  - kseg1: Unmapped, Uncached
    - PA = VA – 0xA0000000
    - Memory-mapped I/O devices
  - kseg2 & kseg3: Mapped
    - Mapped through the TLB



Address Translation (1)

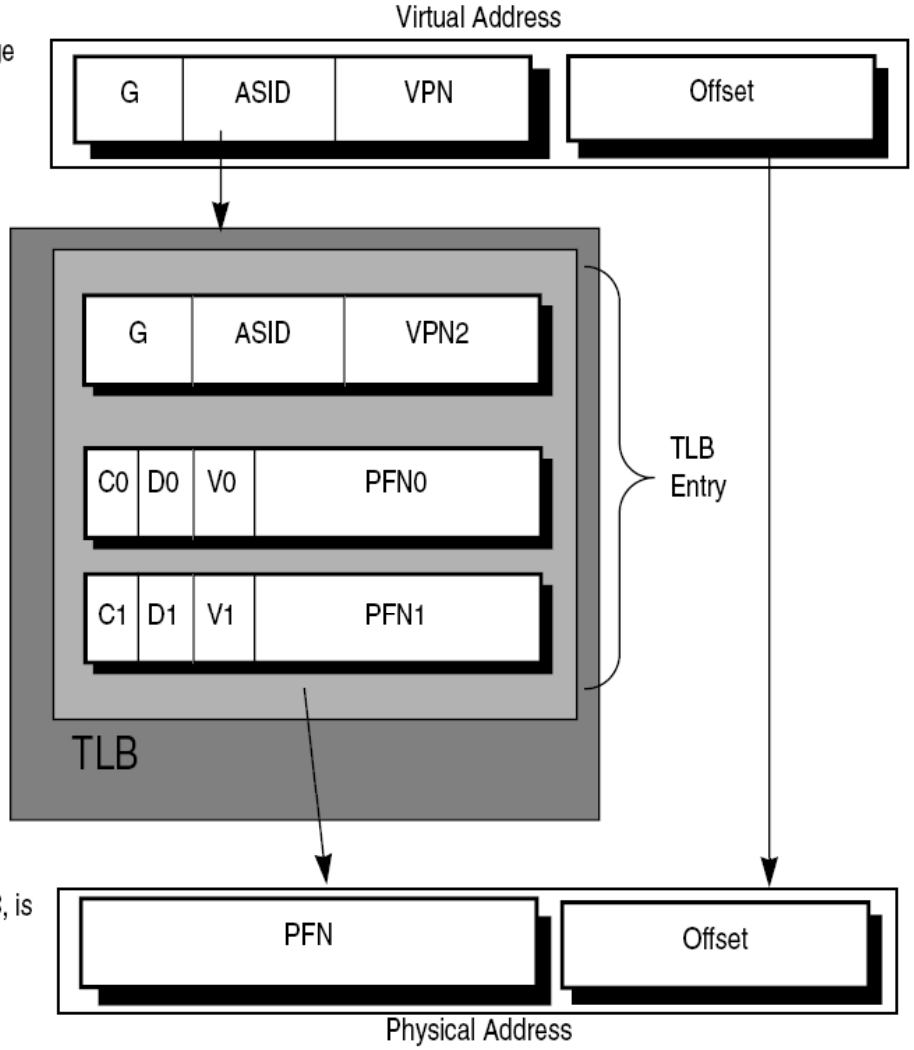
- **Characteristics**
  - Virtual address is extended with ASID
  - Multiple page size support
    - 4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB
    - Optionally 1KB
  - Software-managed TLBs
  - One Joint TLB (JTLB)
    - 16 or 32 dual-entry fully associative
  - Two Micro TLBs
    - 4-entry fully associative Instruction micro TLB (ITLB)
    - 4-entry fully associative Data micro TLB (DTLB)

Address Translation (2)

1. Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in TLB.

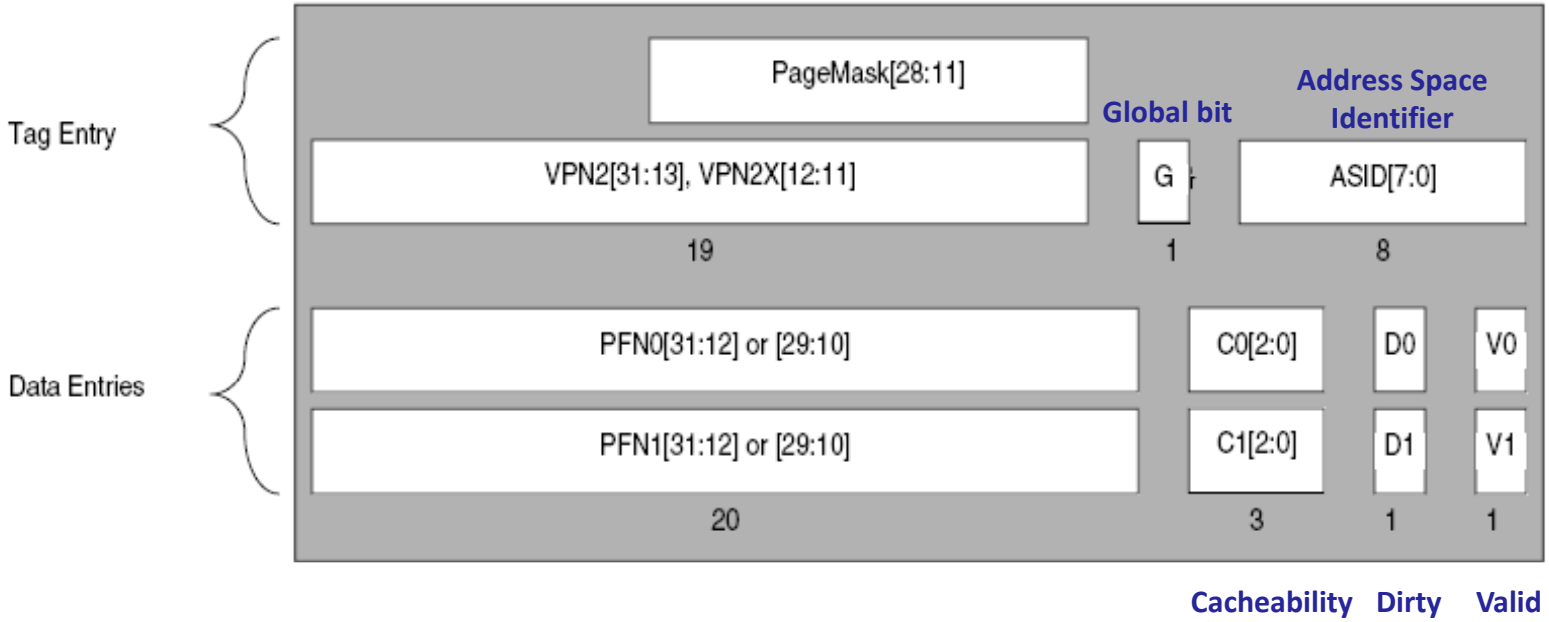
2. If there is a match, the page frame number (PFN0 or PFN1) representing the upper bits of the physical address (PA) is output from the TLB the TLB.

3. The Offset, which does not pass through the TLB, is then concatenated with the PFN.



Address Translation (3)

- **Joint TLB**
  - 16 or 32 dual-entry, fully associative
  - Dual-entry: even and odd entries
  - Page size can vary on a page-pair basis

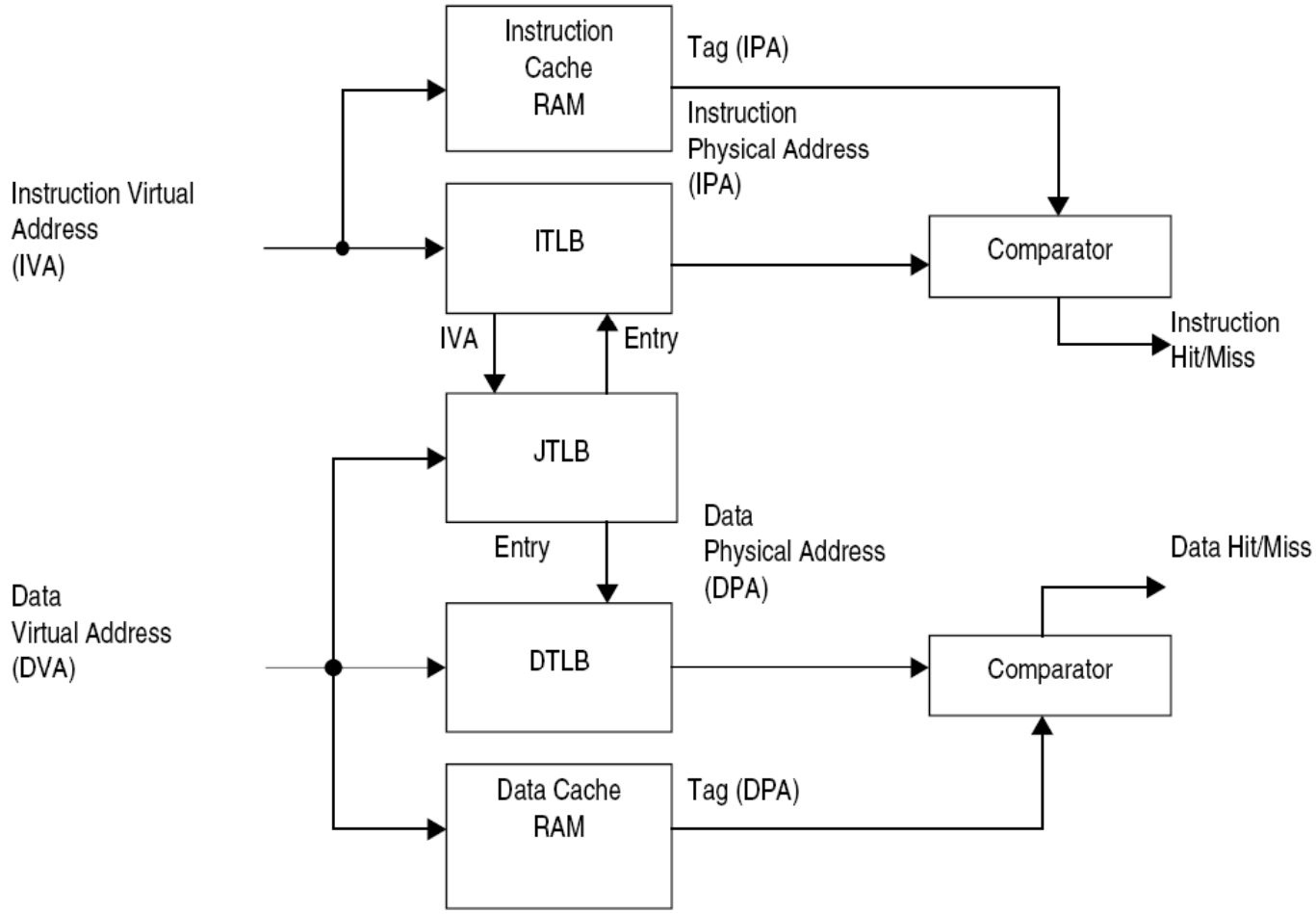


Address Translation (4)

- **Instruction TLB (ITLB)**
  - 4-entry, fully-associative
  - Managed by hardware and is transparent to software
  - ITLB miss penalty: at least 2 cycles
    - Lookup the JTLB, copy the entry into ITLB, and re-access ITLB
- **Data TLB (DTLB)**
  - 4-entry, fully-associative
  - Managed by hardware and is transparent to software
  - DTLB miss penalty: 1 cycle
    - DTLB and JTLB are accessed in parallel

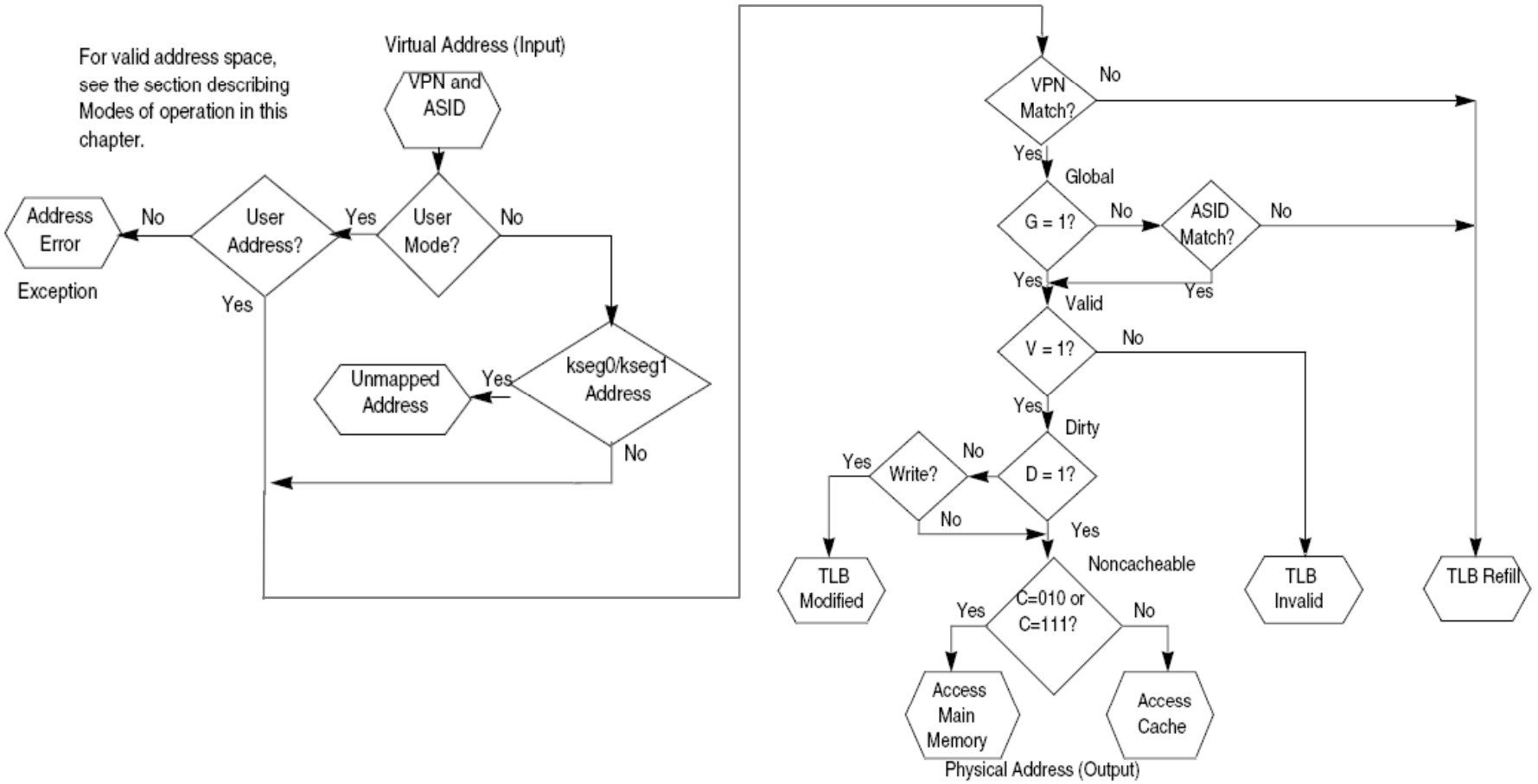
Address Translation (5)

- **MIPS 4KEc**



Address Translation (6)

- **TLB access flow**



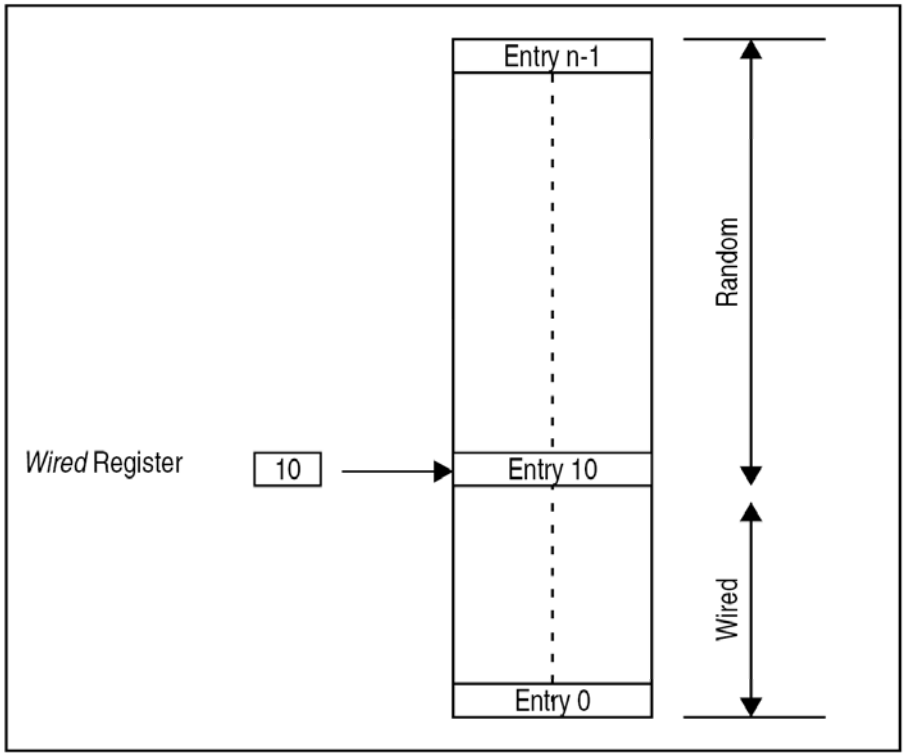
TLB Management (1)

- **TLB instructions**

Instruction	Description
TLBP	**TLB Probe** Probe TLB for matching entry
TLBR	**Read Indexed TLB Entry** Read an entry pointed to by the Index register
TLBWI	**Write Indexed TLB Entry** Write an TLB entry indexed by the Index register
TLBWR	**Write Random TLB Entry** Write a TLB entry indexed by the Random register

TLB Management (2)

- **Replacement algorithm**
  - Random replacement algorithm
  - A programmable number of mappings can be locked into the TLB
  - Wired entries are fixed, non-replaceable entries that are not overwritten by a TLBWR instruction.
  - Wired entries can be overwritten by a TLBWI instruction.



CS632/SEP564: Embedded Operating Systems (Fall 2008)

ARM Memory Protection Unit (MPU)

KAIST

MPU

- **Overview**
  - ARM cores with an MPU:
    - ARM740T, ARM940T, ARM946E-S, ARM1026EJ-S
  - MPU uses regions to manage system protection
    - A region is a set of attributes associated with an area of memory.
      - » Starting address, length, access rights, cache and write buffer policies, etc.
    - These attributes are held in several CP15 registers.
    - Each region is identified by a number.
    - MPU compares the region's access permission attributes with the current processor mode.
    - If a memory access violation occurs, generate an abort signal.

Protected Regions (1)

- **Regions**
  - The number of regions are predefined.
  - Each region is referenced by an identifying number between zero and seven.
    - In ARM940T, each region number has a pair of regions, one data region and one instruction region.

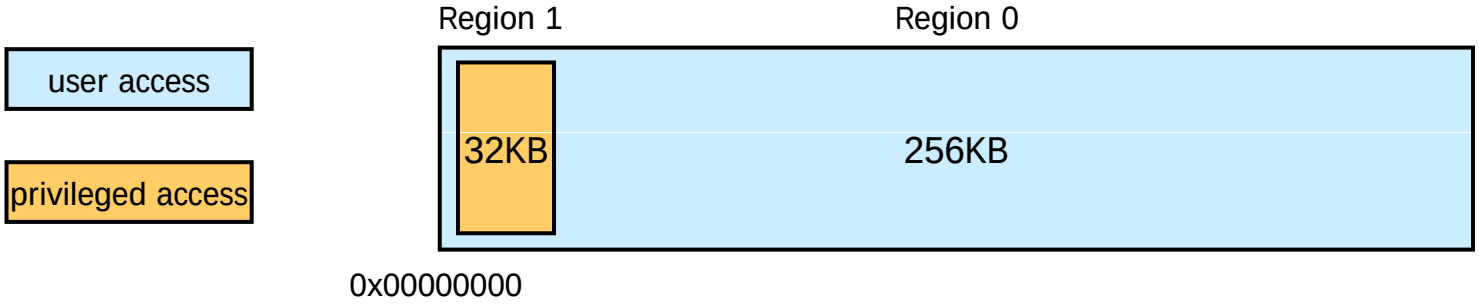
ARM core	Number of regions	Separate instruction and data regions	Separate configuration of instruction and data regions
ARM740T	8	No	No
ARM940T	16	Yes	Yes
ARM946E-S	8	No	Yes
ARM1026EJ-S	8	No	Yes

Protected Regions (2)

- **Rules for regions**
  - Regions can overlap other regions.
  - Regions are assigned a priority number that is independent of the privilege assigned to the region.
  - When regions overlap, the attributes of the region with the highest priority number take precedence over the other regions.
  - A region's starting address must be a multiple of its size.
  - A region's size can be any power of two between 4KB and 4GB.
  - Accessing an area of main memory outside of a defined region results in an abort.

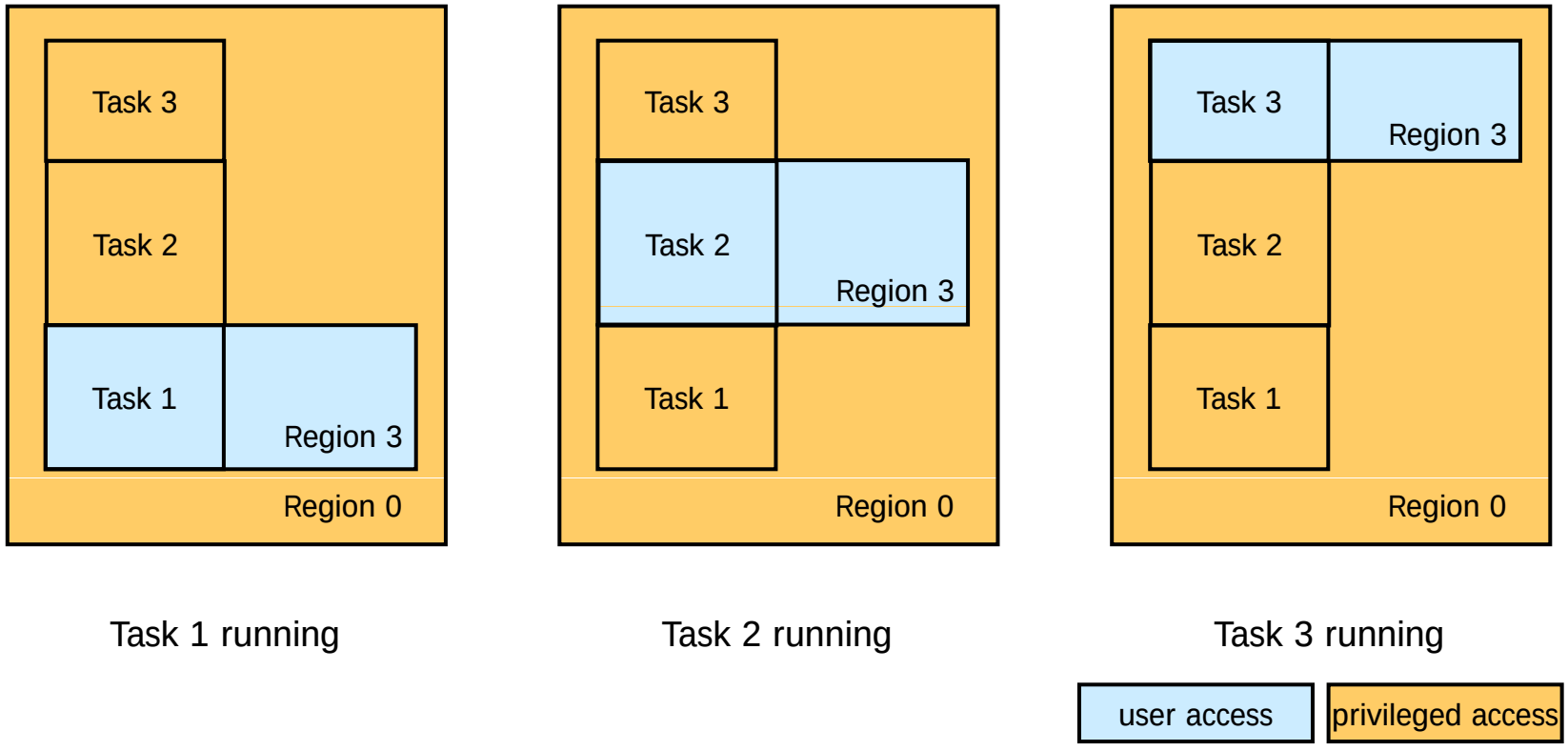
Protected Regions (3)

- **Overlapping regions**
  - A small embedded system which has 256KB of available memory starting at address 0x00000000.
  - Need to protect a privileged system area from user mode reads and writes.
    - The privileged area code, data, and stacks fit in a 32KB region starting, with the vector table, at 0x00000000.



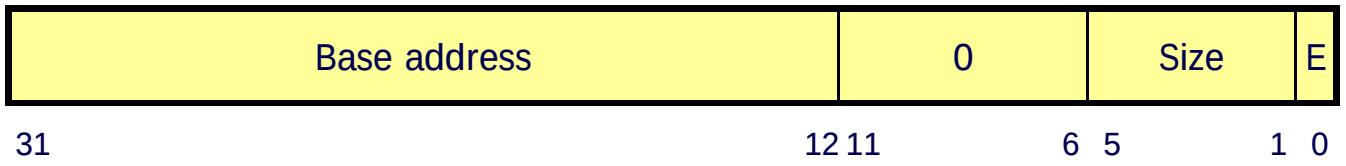
Protected Regions (4)

- **Background regions**
  - A low-priority region used to assign the same attributes to a large memory area.



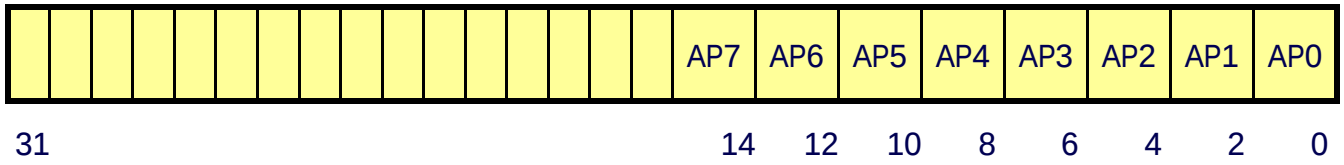
Setting MPU (1)

- **System control**
  - CP15:c1:c0 register, bit 0
    - 0: MPU disabled
    - 1: MPU enabled
- **Region size and location**
  - CP15:c6:c0 ~ CP15:c6:c7 (for each region)
    - Base address should be a multiple of the size
    - Size is encoded in 5 bits
      - » 4KB = 01011, 8KB = 01100, ..., 4GB = 11111
    - E: region enable (1) or disable (0)



Setting MPU (2)

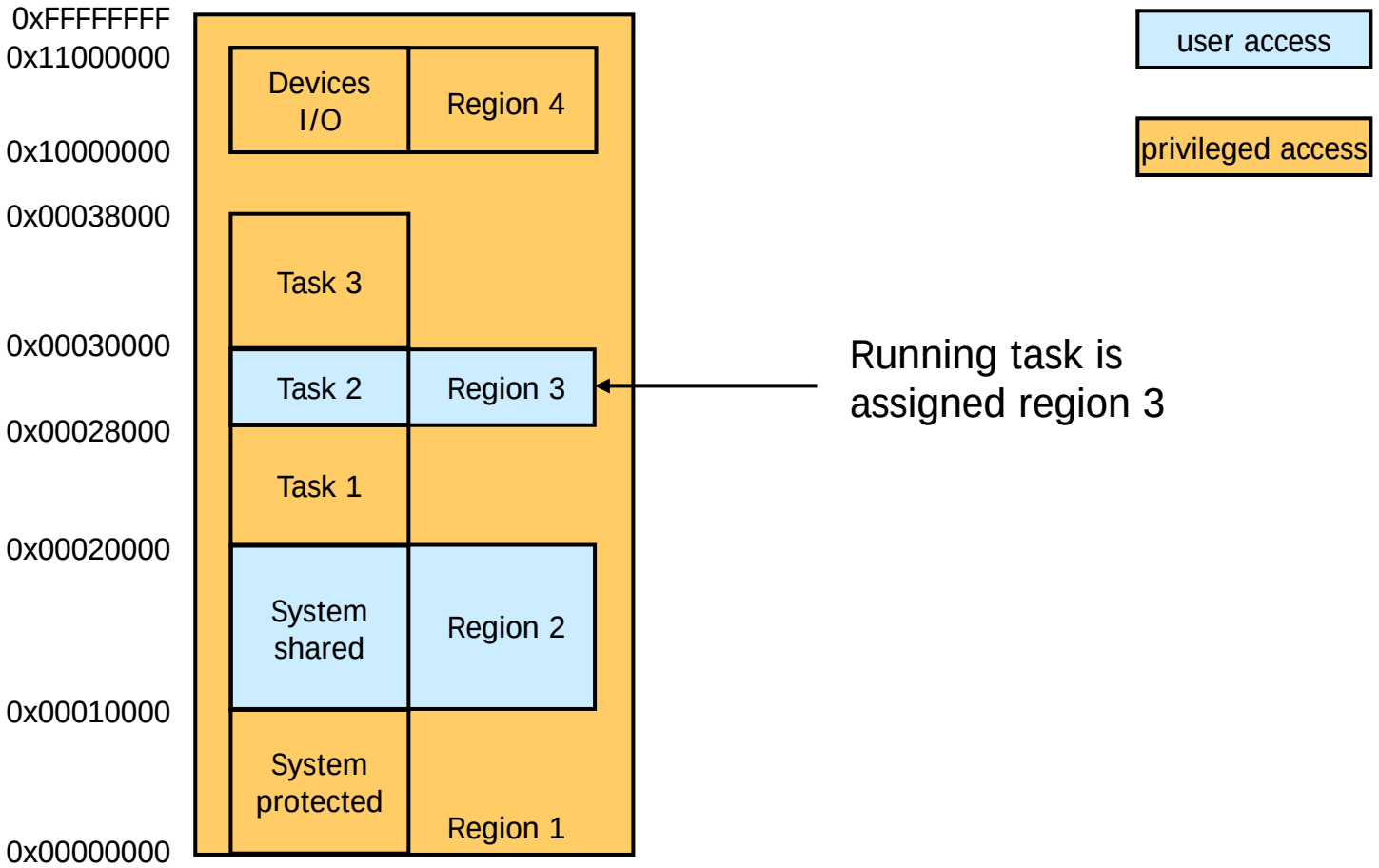
- **Region access permissions**
  - CP15:c5:c0 register: AP for standard instruction region
  - CP15:c5:c1 register: AP for standard data region



  - Access permissions encoding (Supervisor/User)
    - 00: No access/No access
    - 01: Read Write/No access
    - 10: Read Write/Read only
    - 11: Read Write/Read Write
  - Extended access permissions can be specified for newer ARM cores (ARM946E-S and ARM1026EJ-S)

Example (1)

- **Memory map**



Example (2)

- **Regions**

#	Region	Base	Size	Access permission			
				Instruction		Data	
				System	User	System	User
1	Protected system	0x00000000	4GB	Read only	None	Read Write	None
2	Shared system	0x00010000	64KB	Read only	Read only	Read Write	Read Write
3	User task 1	0x00020000	32KB	Read only	Read only	Read Write	Read Write
3	User task 2	0x00028000	32KB	Read only	Read only	Read Write	Read Write
3	User task 3	0x00030000	32KB	Read only	Read only	Read Write	Read Write
4	Protected memory-mapped peripheral devices	0x10000000	1MB	Read only	None	Read Write	None

CS632/SEP564: Embedded Operating Systems (Fall 2008)

Executable and Linking Format (ELF)

KAIST

Introduction (1)

- **Background**
  - The a.out format served the Unix community well for over 10 years.
  - However, to better support advanced system features, a.out has been replaced by the ELF file format.
    - Cross-compilation, dynamic linking, initializer/finalizer, ...
  - Adopted in many operating systems
    - Unix-like: Linux, Solaris, IRIX, *BSDs, HP-UX, ...
    - Non-Unix: OpenVMS for Itanium, BeOS Rev. 4+ for x86, PSP/PS2/PS3, Wii, Sony Ericsson/Siemens/Motorola phones, ...
    - (cf.) PE (Portable Executable) format:
      - » Used in Microsoft Windows, a modified version of the Unix COFF (Common Object File Format)

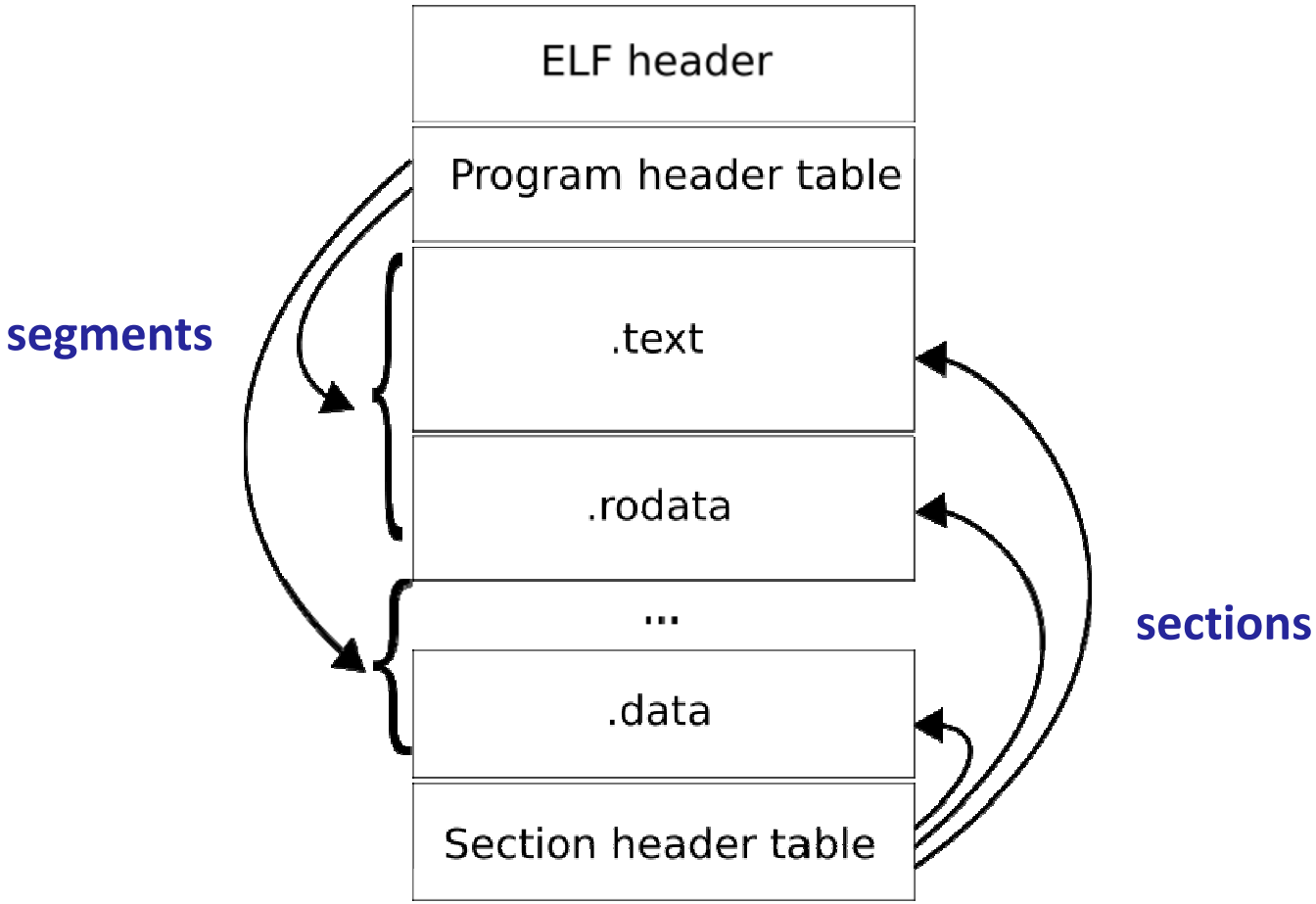
Introduction (2)

- **ELF Characteristics**
  - The standard binary file format for Unix-like systems
  - Very flexible and extensible
  - Processor- or architecture-independent
  - Common file formats for
    - **Executable file**: holds a program suitable for execution
    - **Relocatable file**: holds code and data suitable for linking with other object files to create executable or shared object
    - **Shared object**: suitable for static or dynamic linking
    - **Core dump file**: holds the recorded state of the working memory of a process at a specific time, generally when the program has terminated abnormally
  - Supports position-independent code (PIC)

ELF Structure (1)

- **Section header table**
  - Compilers, assemblers, and linkers treat the file as a set of logical sections described by a section header table.
  - Every section has an entry in the table.
  - Each entry gives information such as the section name, the section size, etc.
- **Program header table**
  - The system loader treats the file as a set of segments described by a program header table.
  - Tells the system how to create a process image.
  - Executable files must have program header table.

ELF Structure (2)



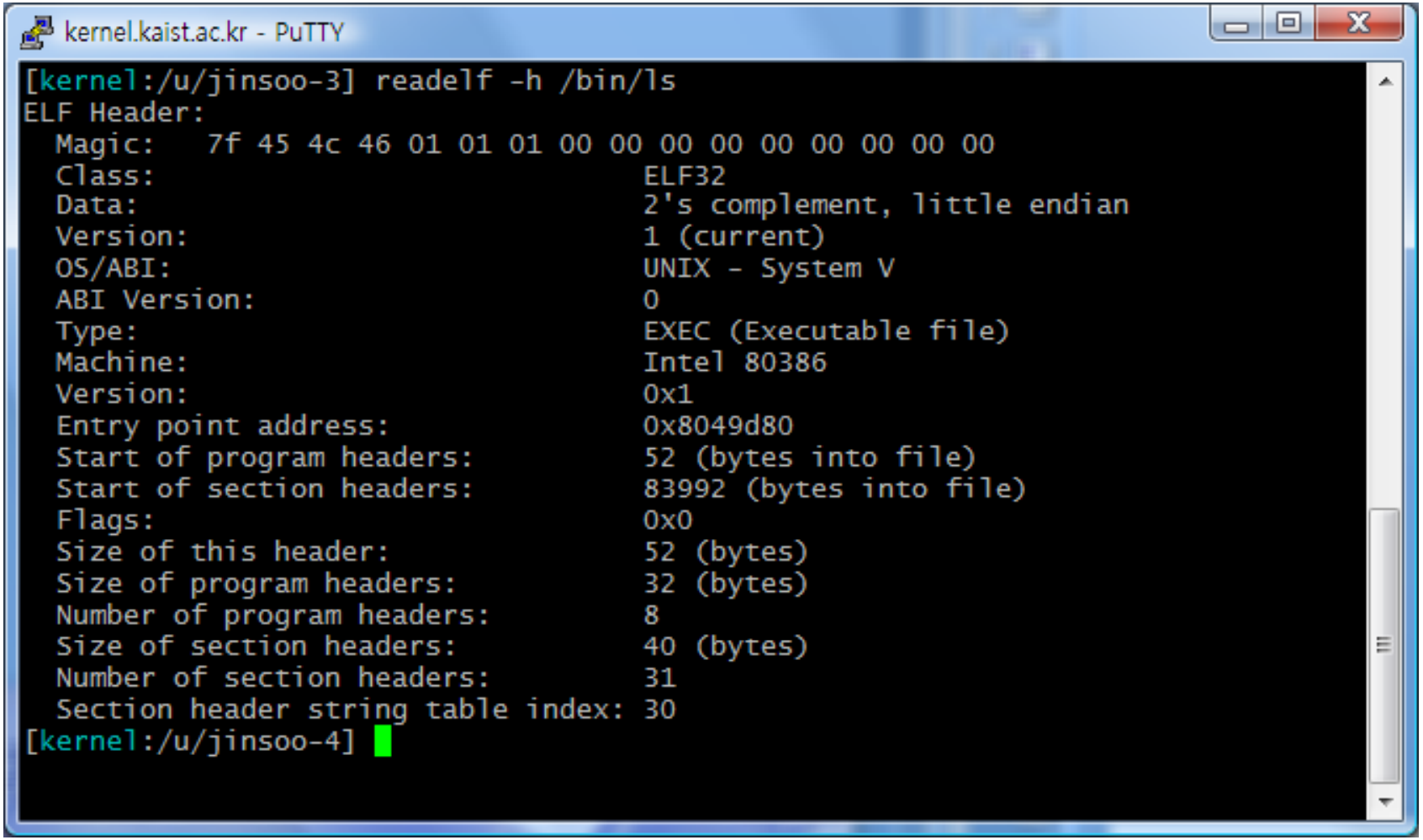
ELF Header (1)

- **ELF header**
  - The ELF header is always at offset zero of the file.
  - The program header table and the section header table's offset in the file are defined in the ELF header.
  - The header is decodable even one machines with a different byte order from the file's target architecture
    - After reading class and byteorder fields, the rest fields can be decoded.
    - The ELF format can support two different address sizes
      - » 32 bits
      - » 64 bits

ELF Header (2)

char magic[4] = “₩177ELF”;	magic number
char class;	address size, 1 = 32-bit, 2 = 64-bit
char byteorder;	1 = little-endian, 2 = big-endian
char hversion;	header version, always 1
char pad[9];	
short filetype;	1 = relocatable, 2 = executable, 3 = shared object, 4 = core image
short archtype;	2 = SPARC, 3 = x86, 4 = 68K, etc.
int fversion;	file version, always 1
int entry;	entry point if executable
int phdrpos;	file position of program header or zero
int shdrpos;	file position of section header or zero
int flags;	architecture-specific flags, usually zero
short hdrsize;	size of this ELF header
short phdrent;	size of an entry in program header
short phdrcnt;	number of entries in program header or zero
short shdrent;	size of an entry in section header
short shdrcnt;	number of entries in section header or zero
short strsec;	section number that contains section name strings

ELF Header (3)


kernel.kaist.ac.kr - PuTTY
[kernel:/u/jinsoo-3] readelf -h /bin/ls
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8049d80
Start of program headers: 52 (bytes into file)
Start of section headers: 83992 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 8
Size of section headers: 40 (bytes)
Number of section headers: 31
Section header string table index: 30
[kernel:/u/jinsoo-4]

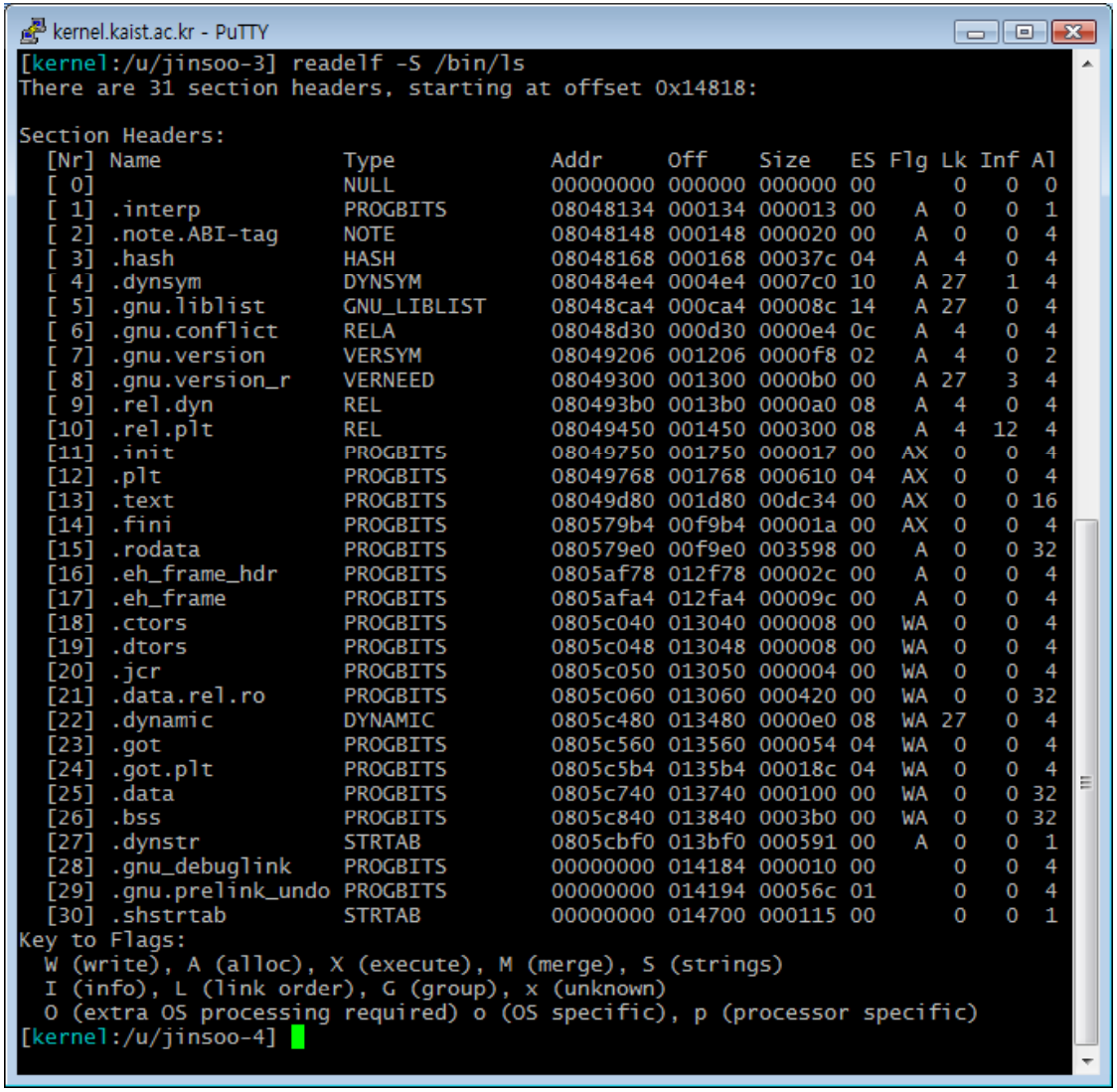
Section Header (1)

int sh_name;	name, index into the string table
int sh_type	section type
int sh_flags;	flag bits **WRITE:** contains data writable during process execution. **ALLOC**: occupies memory during process execution **EXECINSTR:** contains executable machine instructions.
int sh_addr;	base memory address if loadable, or zero
int sh_offset;	file position of beginning of section
int sh_size	size in bytes
int sh_link;	section number with related info or zero
int sh_info;	more section-specific info
int sh_align;	alignment granularity if section is moved
int sh_entsize;	size of entries if section is an array

Section Header (2)

- **Section types**
  - **NULL**: This value marks the section header as inactive.
  - **PROGBITS**: This holds program contents including code, data, and debugger information.
  - **NOBITS**: Like PROGBITS. However, it occupies no space.
  - **SYMTAB** and **DSYMTAB**: These hold symbol table.
  - **STRTAB**: This is a string table.
  - **REL** and **RELA**: These hold relocation information.
  - **DYNAMIC** and **HASH**: These hold information related to dynamic linking.
  - **NOTE**: Information that marks the file in some way.

Section Header (3)



Sections (1)

- **.text**
  - This section holds executable instructions of a program.
- **.data**
  - This section holds initialized data that contributes to the program's image.
- **.rodata**
  - This section holds read-only data.

Sections (2)

- **.bss**
  - This section holds uninitialized data that contributed to the program's image.
  - By definition, the system will initialize the data with zero when the program begins to run.
- **.rel.text, .rel.data, and .rel.rodata**
  - These contain the relocation information for the corresponding text or data sections.
- **.symtab**
  - This section holds a symbol table.

Sections (3)

- **.strtab**
  - This section holds strings.
- **.init**
  - This section holds executable instructions that contribute to the process initialization code.
- **.fini**
  - This section holds executable instructions that contribute to the process termination code.

Sections (4)

- **.interp**
  - This section holds the pathname of a program interpreter.
  - If this section is present, rather than running the program directly, the system runs the interpreter and passes it the ELF file as an argument.
  - In practice, this is used to run the run-time dynamic linker to load the program and to link in any required shared libraries.

Sections (5)

- **.debug**
  - This section holds symbolic debugging information.
- **.line**
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code.
- **.comment**
  - This section may store extra information.

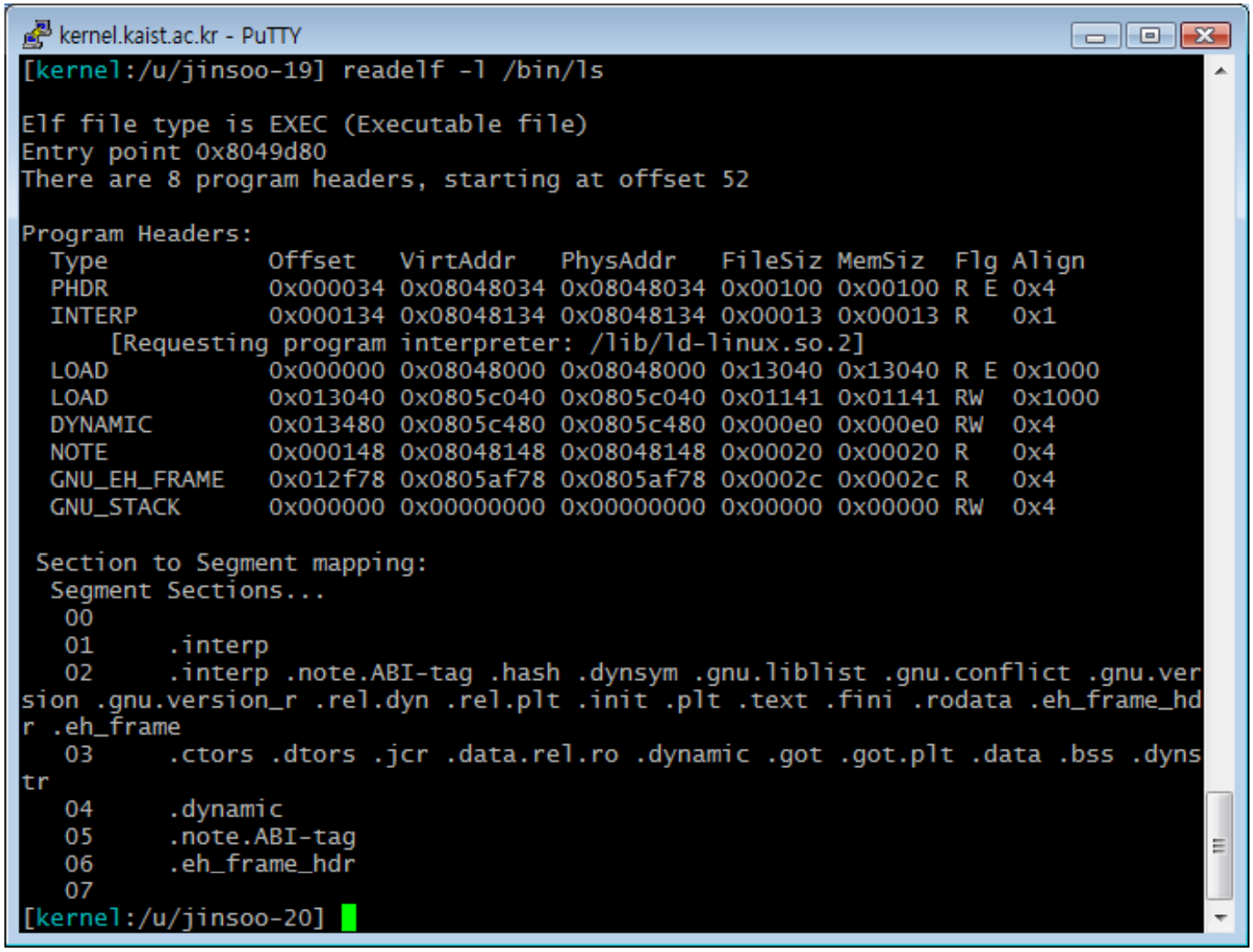
Sections (6)

- **.got**
  - This section holds the global offset table.
- **.plt**
  - This section holds the procedure linkage table.
- **.note**
  - This section contains some extra information.

Program Header (1)

int p_type;	0 = PT_NULL, 1 = PT_LOAD, 2 = PT_DYNAMIC, 3 = PT_INTERP, 4 = PT_NOTE, 5 = PT_SHLIB, etc.
int p_offset;	file offset of segment
int p_vaddr;	virtual address to map segment
int p_paddr;	physical address (on systems for which physical addressing is relevant; usually not used.)
int p_filesz;	size of segment in file, may be zero.
int p_memsz;	size of segment in memory (bigger if contains bss), may be zero.
int p_flags;	read, write, execute bits
int p_align;	required alignment, invariably hardware page size

Program Header (2)



Program Loading (1)

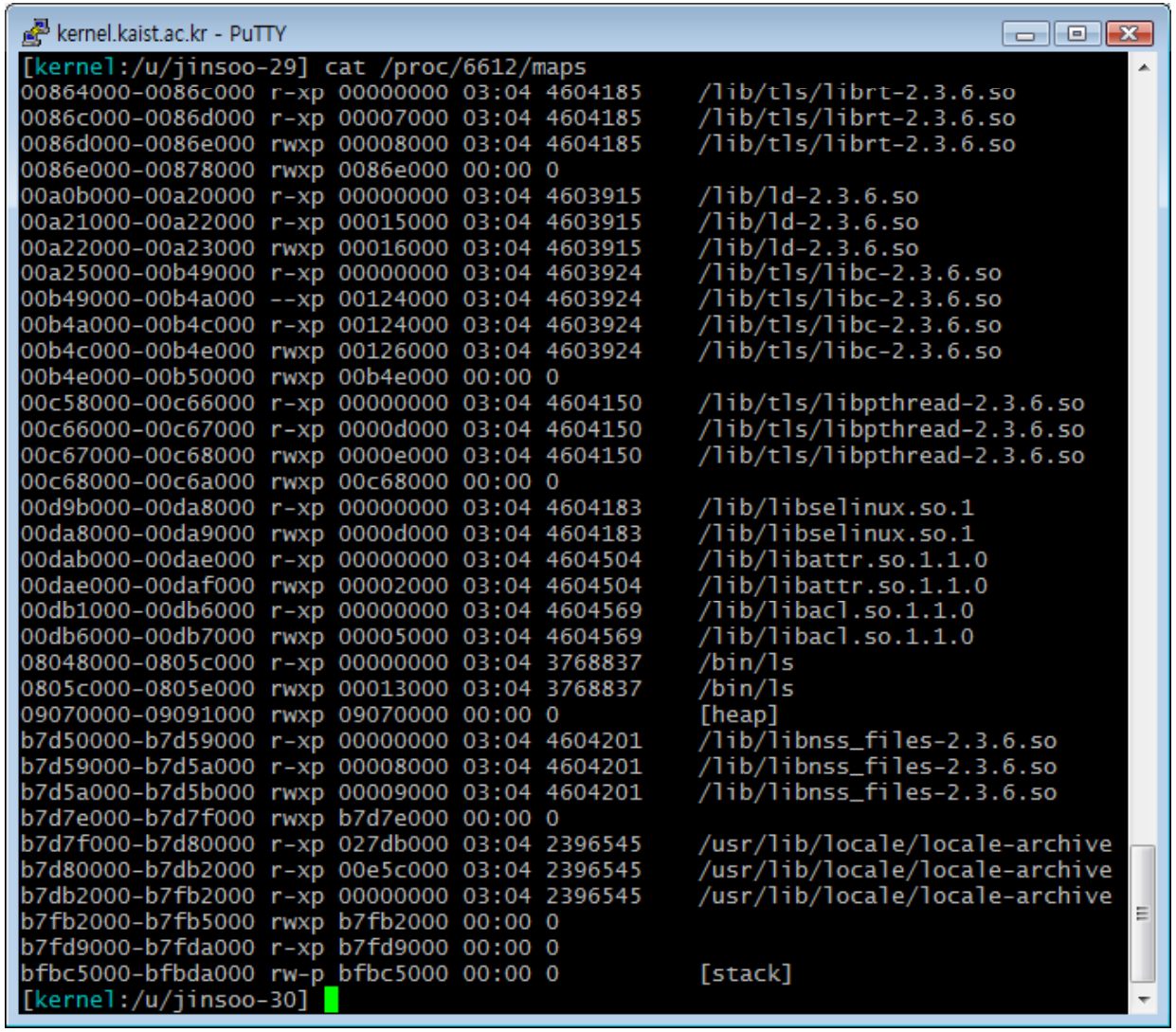
- **Loading**
  - Process of preparing the executable for running.
  - PT_LOAD sections are mapped in memory to their vaddrs.
  - The kernel sets up the stack (cmdline pointer vector, environ vector, cmdline strings, environ strings, etc.)

- **Static executables**
  - PT_INTERP segment is absent.
  - Control transferred to the entry point.

Program Loading (2)

- **Dynamic executables**
  - PT_INTERP segment is present; contains the interpreter name.
  - The program interpreter is mapped.
  - Control is transferred to the interpreter's entry point.
  - The interpreter (dynamic linker) resolves library dependencies, maps them as necessary and starts the program.

Program Loading (3)



Program Loading (4)