Memory Management Basics

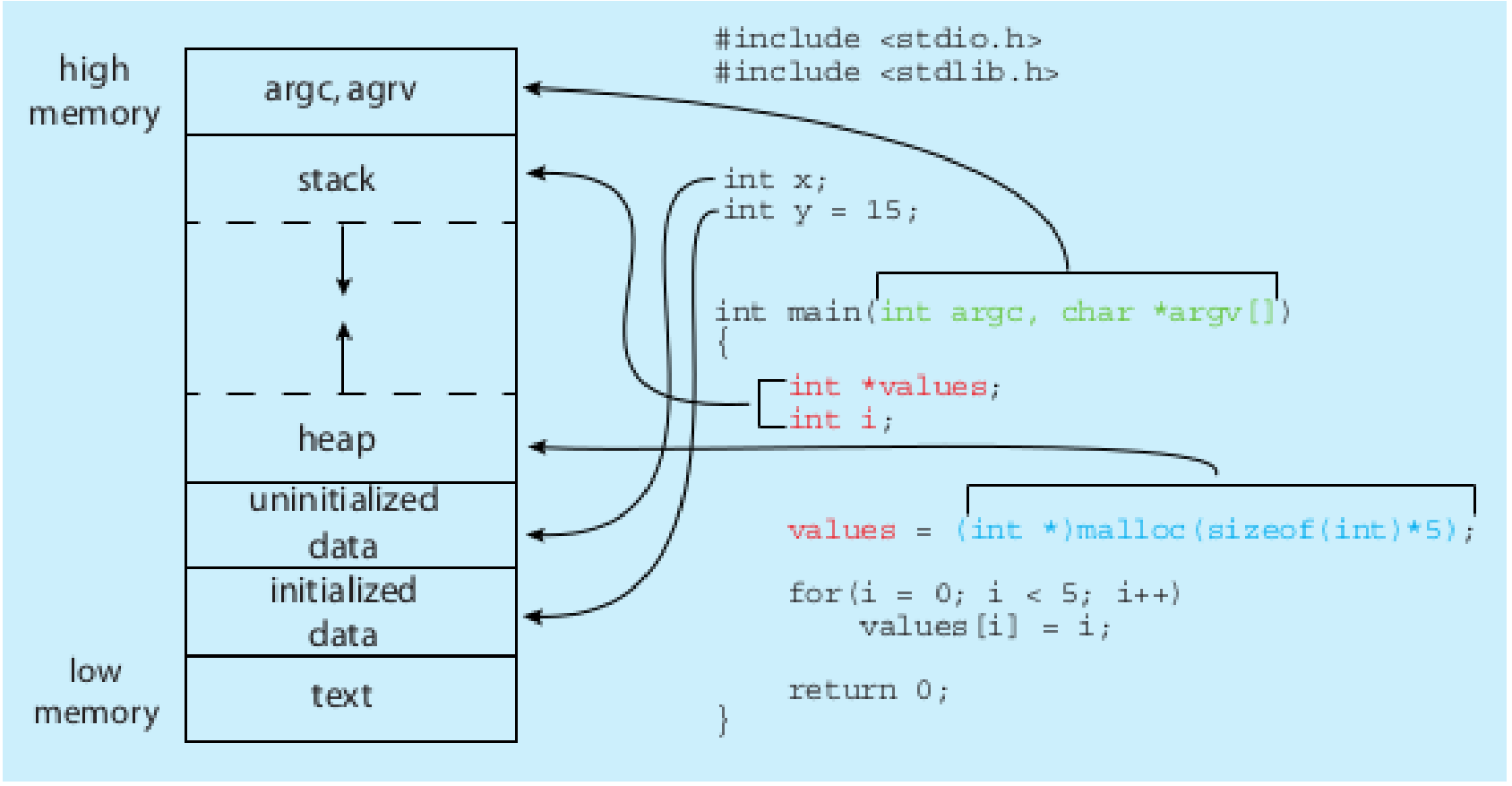
Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
  - Addresses are “issued” by the CPU on address bus
  - One address to fetch instruction from location specified by PC
  - Zero or more addresses depending on instruction
    - e.g. mov $0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program



This “layout” shows
(a) which parts of a C program occupy memory, when the program is running
(b) A typical conceptual layout assumed by many compilers for calculating addresses in the generated machine code.
This does not mean that other layouts are not possible

$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Terminology

- Text
  - The machine code of the program : machine code for functions
- Data
  - Initiazlied global variables
  - Do not get confused with the generic word “data” in English
- BSS
  - Uninitialized global variables
- Heap
  - Region from where malloc() gets memory
- Stack
  - Region from where the local variables and formal paramters are allocated space

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other's code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

The sequence

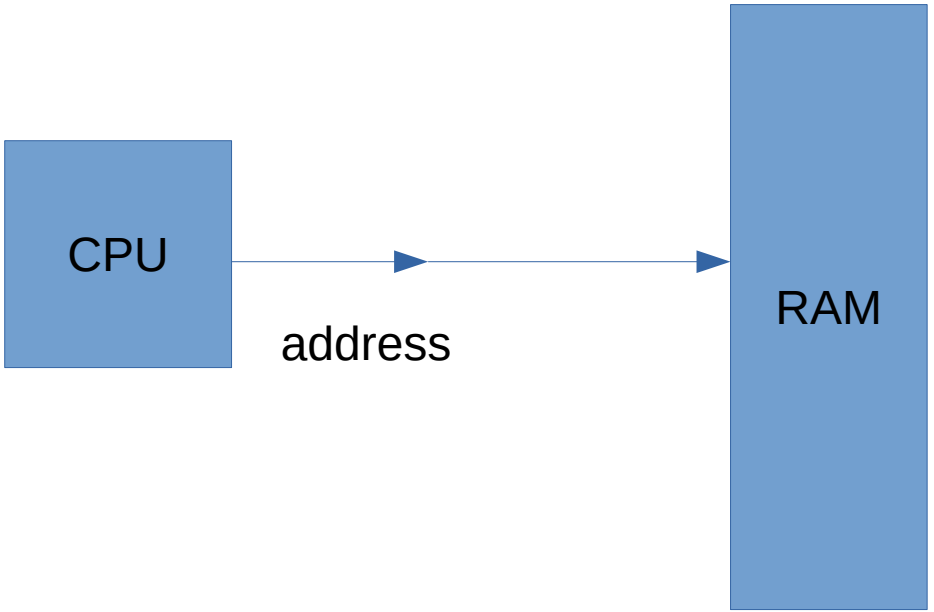
- Do not forget this
  - Machine code is typically generated by compiler
  - This machine code is put in RAM by the Loader (part of kernel) , that is exec() , when it’s requested to run that program
  - The CPU’s PIPELINE will issue addresses on address bus as seen in the executable file
- So question arises
  - How does compiler put in addresses (for code, data, bss, local-variables, etc) in the executable file ?

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

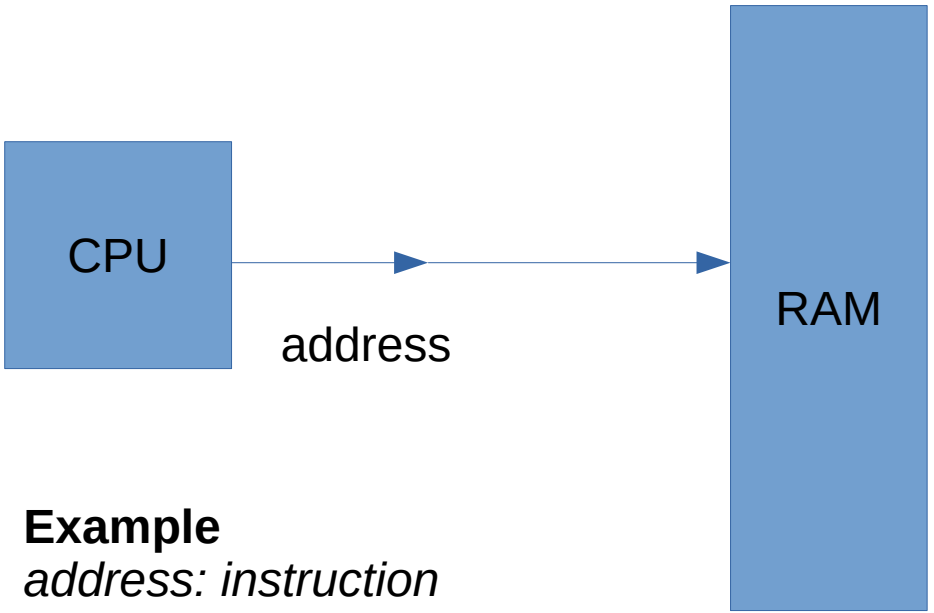
- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case



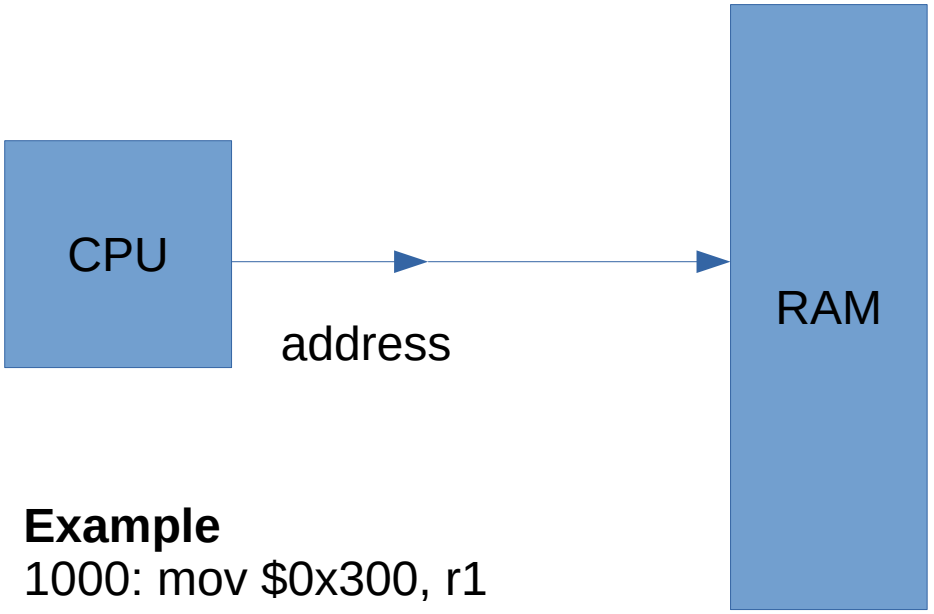
- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
  - How will the compiler know the addresses, at “compile time” ?

Example
address: instruction

1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

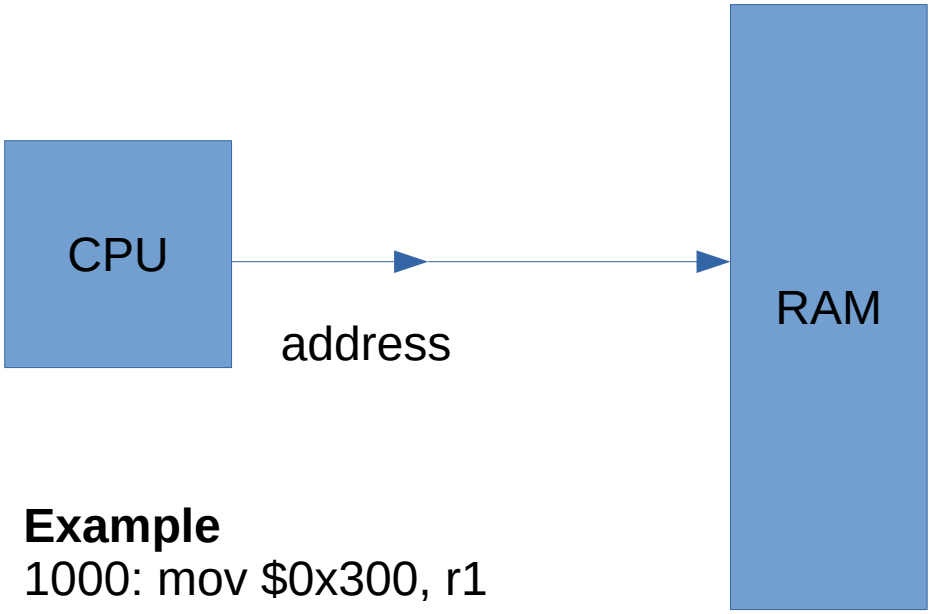
Simplest case



Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file. **Non-relocatable code.**
- Now program can execute properly.

Simplest case



Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

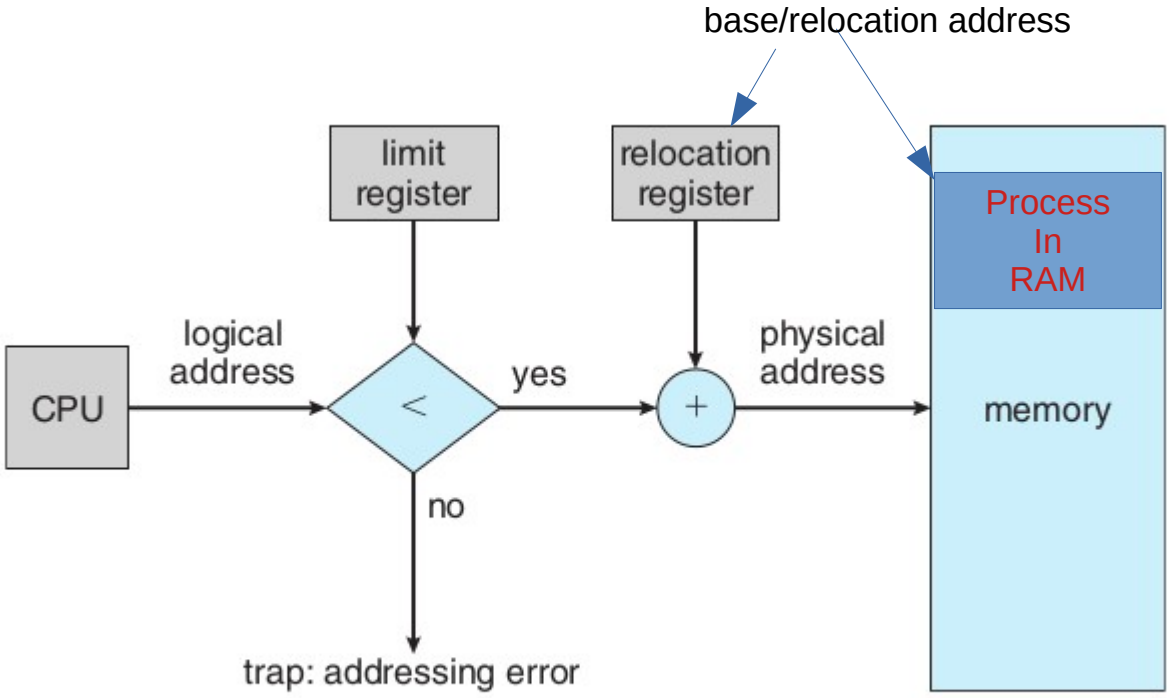


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

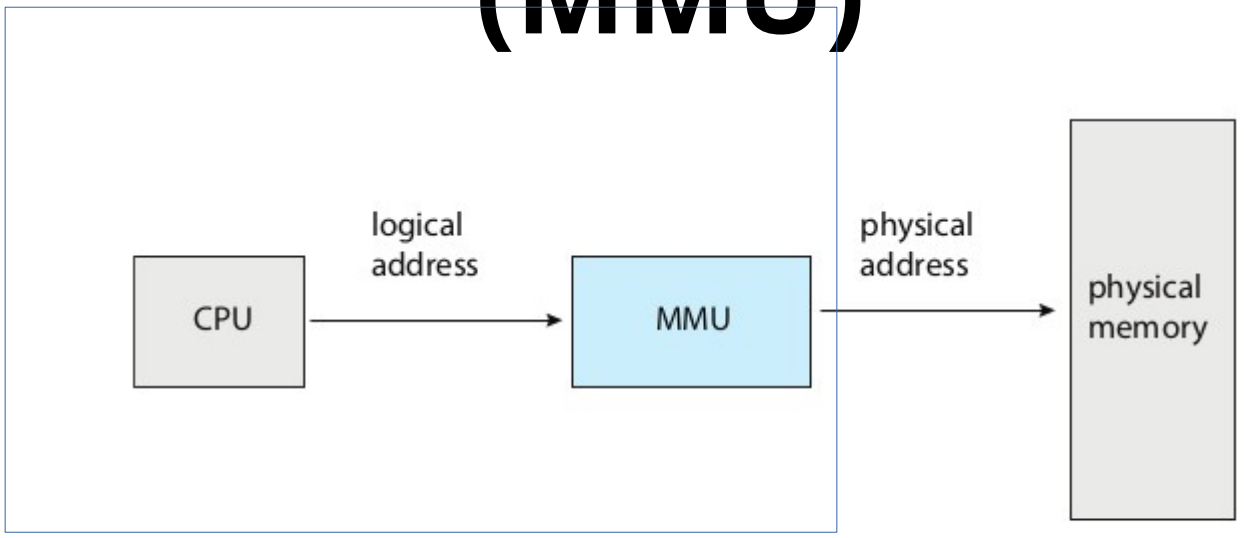
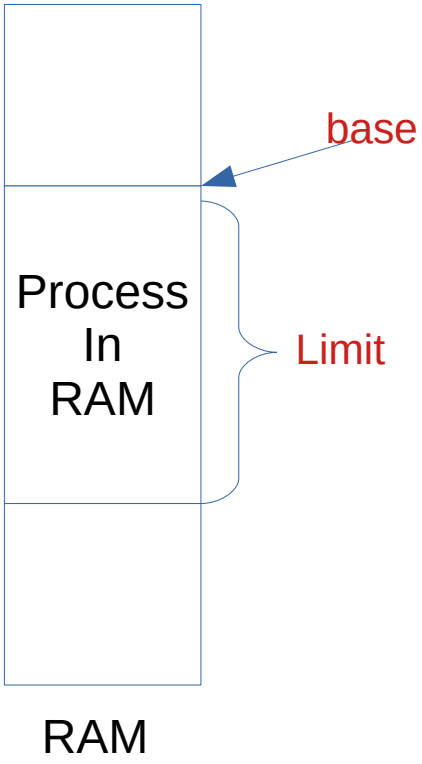
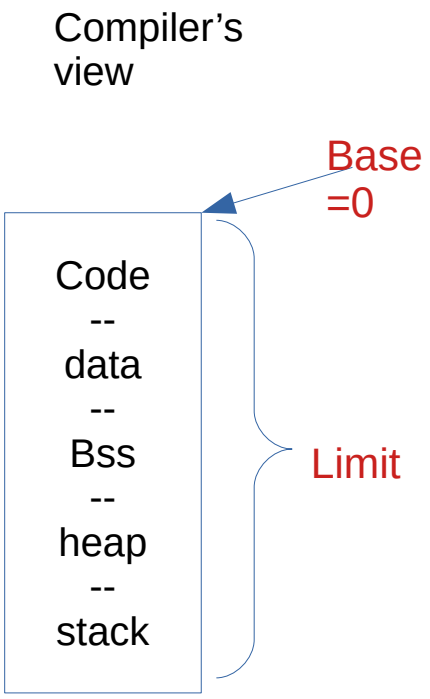


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme





- Compiler's work
  - Assume that the process is one continous chunk in memory, with a size limit
  - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

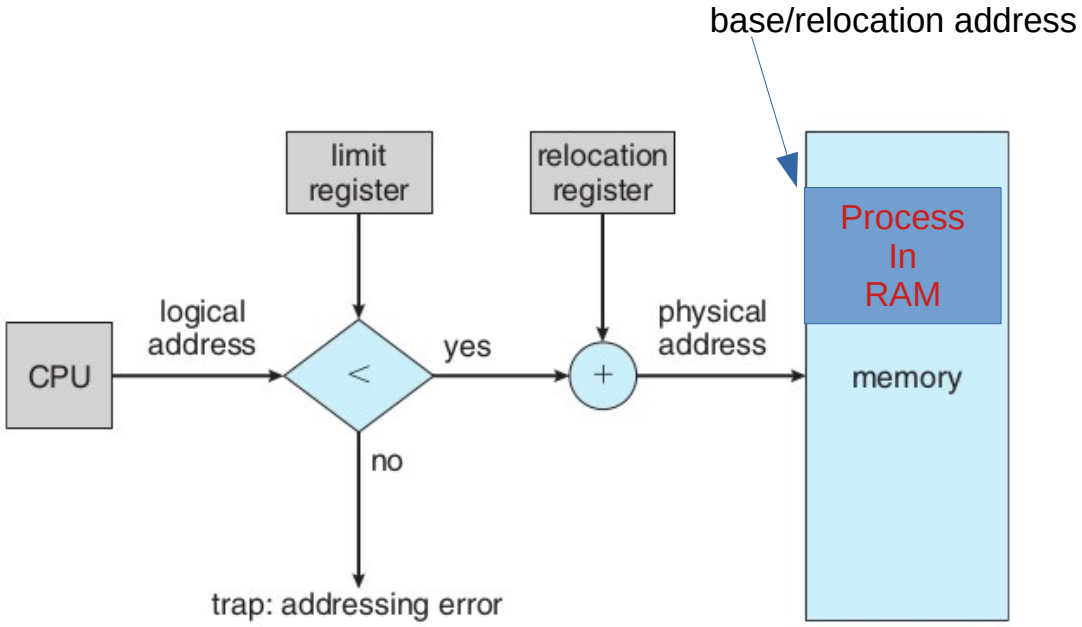


Figure 9.6 Hardware support for relocation and limit registers.

- Loader's job
  - While loading the process in memory – must load as one continous segment
    - Find an empty slot for this!
  - Remember the 'base' and 'limit' values in the struct proc (memory management info in PCB)
    - Setup the limit to be the size of the process as set by compiler in the executable file.
  - Call the scheduler()
- Scheduler()
  - Will replace the base-limit registers with values obtained from PCB before scheduling the process

Base/Relocation + Limit scheme

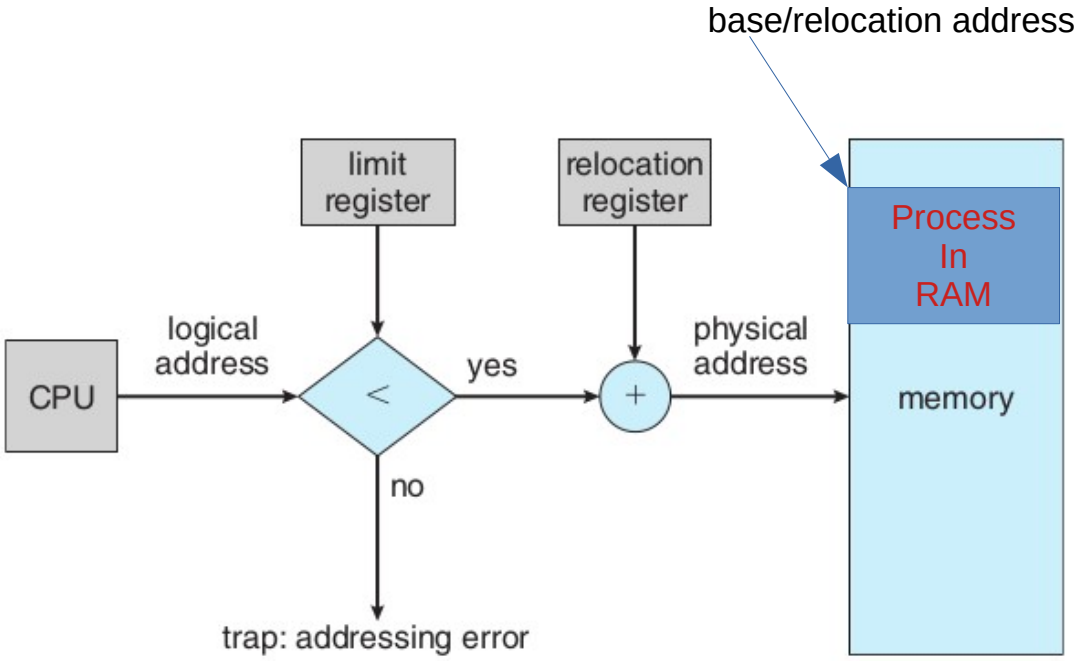
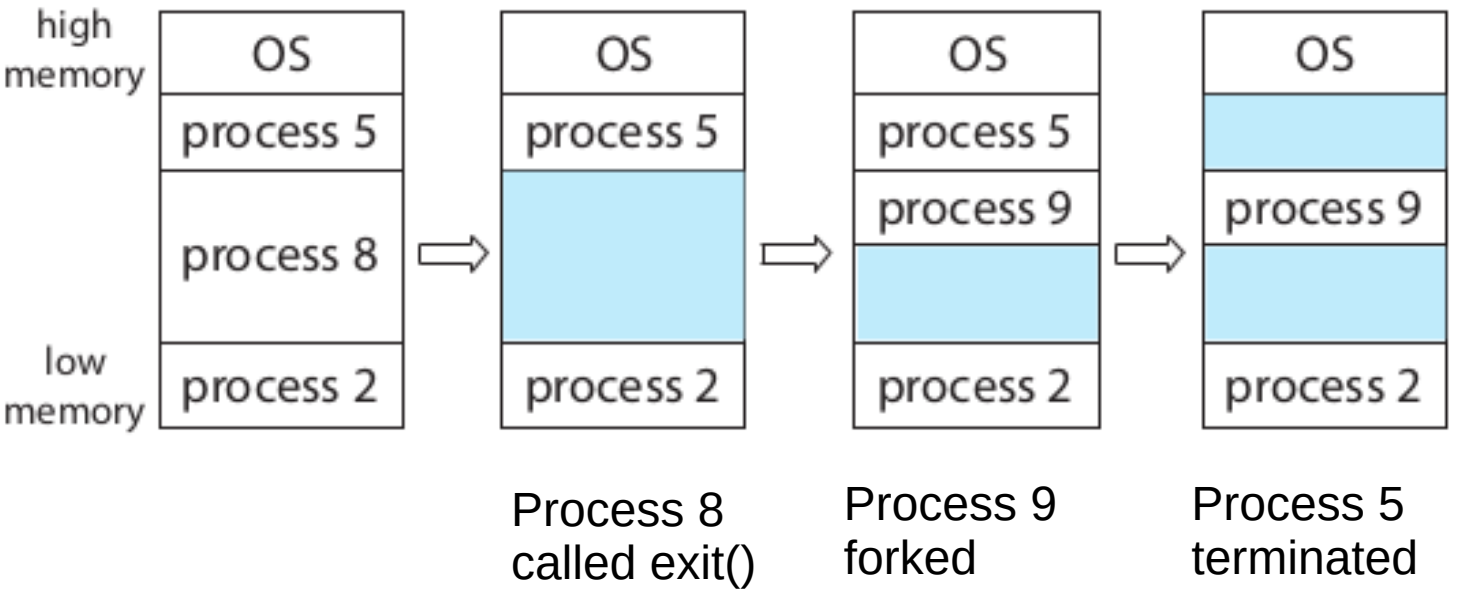


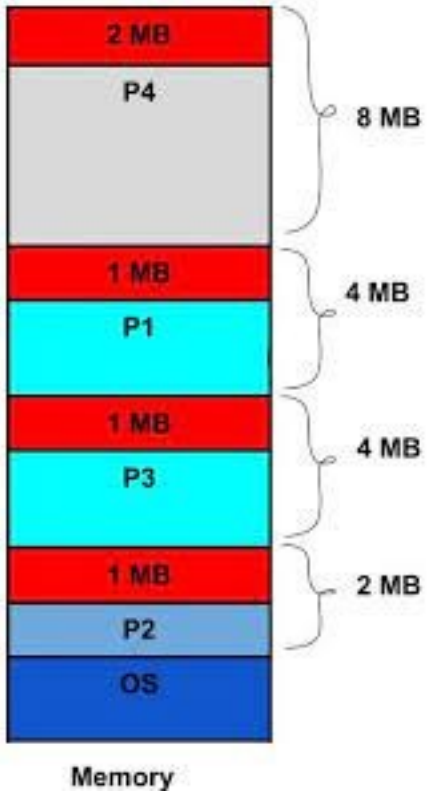
Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - **"Relocatable code" –** the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme



Continuous memory management and external fragmentation problem



Free chunks: 2 MB, 1MB, 1MB, 1MB

Total 5 MB is available!

Can we create a process of size 3MB?

No! - 3 MB not continous!

External Fragmentation

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.

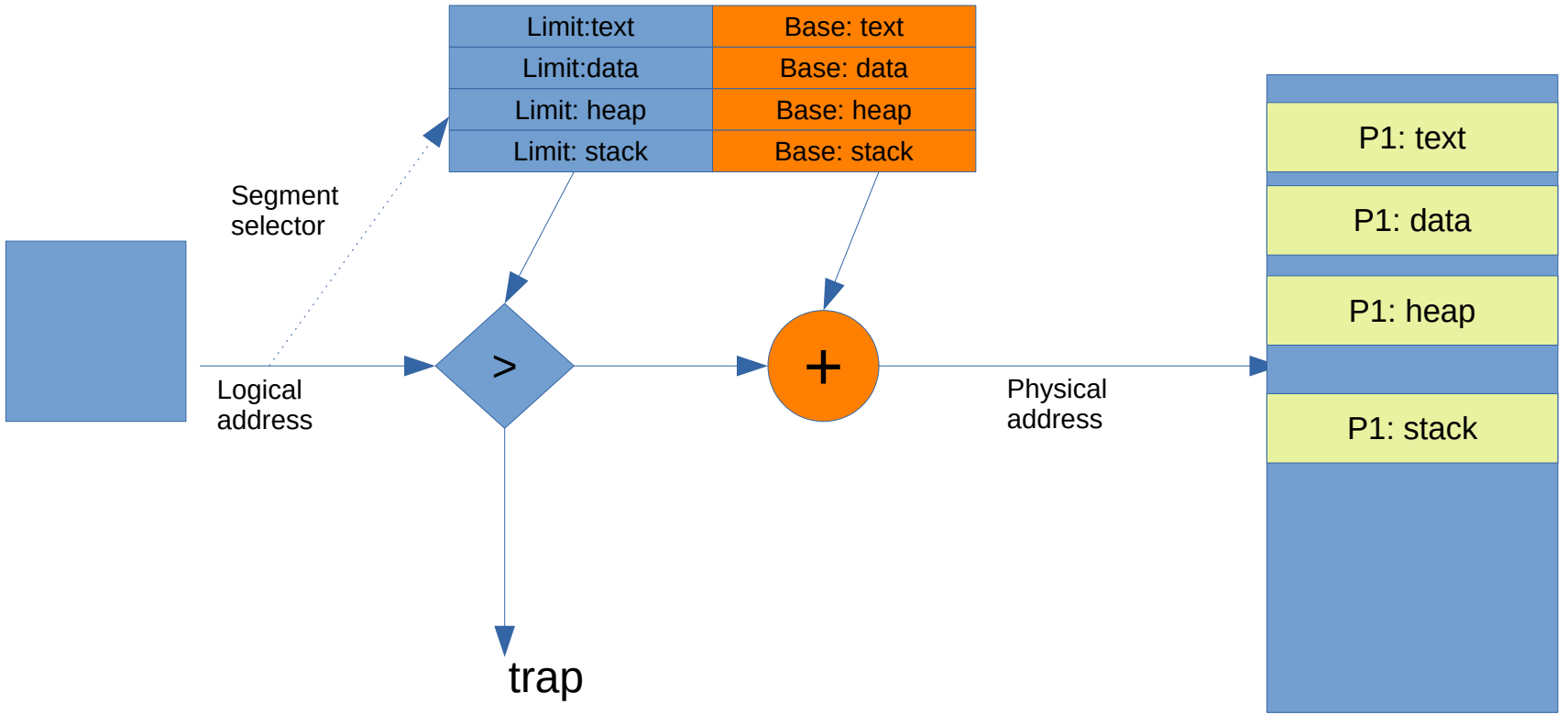
It should be possible to have relocatable code even with “simplest case”

By doing extra work during “loading”.

How?

(Ans: loader replaces all addresses in the code! That is quite a lot of work, and challenging too)

Next scheme:
Multiple base +limit pairs



Specifying “selector”

- Explicitely
  - Mov ES: 300, $30
  - Here selector is Extra Segment Register and Logical address (or offset) is 300
- Implicitely, like in x86
  - Mov 300, $30
  - Here Data Segment register is “implicit” selector

Next scheme: Segmentation Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
  - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS (Loader): will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers

Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Then it's called **Segment Table.**
  - Question: how will the cpu know where it's in memory?
  - One CPU register to point to the location of table in memory . **Segment Table Base Register (STBR)**
    - X86 has two tables. Local Descriptor Table and Global Descriptor Table. Both are segment tables.
    - Accordingly it has LDTR and GDTR.
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
  - Flexibility to have lot more "base+limits" in the array/table in memory

Next scheme:
Multiple base +limit pairs, with further indirection

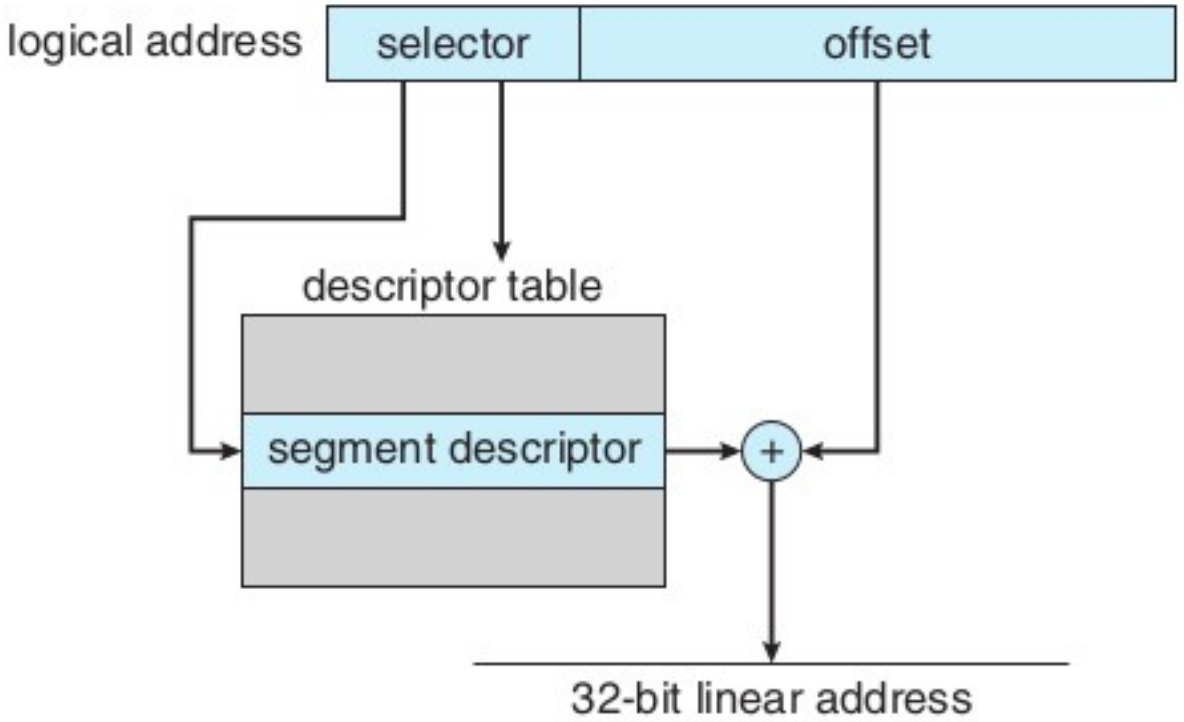


Figure 9.22 IA-32 segmentation.

Note:
”Selector” here is normally a segment register, like CS, DS

The machine code only has offset in it

The “Descriptor table” is a segmentation table

Segmentation and External fragmentation

- Does segmentation also suffer from external fragmentation?
  - Yes!

How many segment tables?

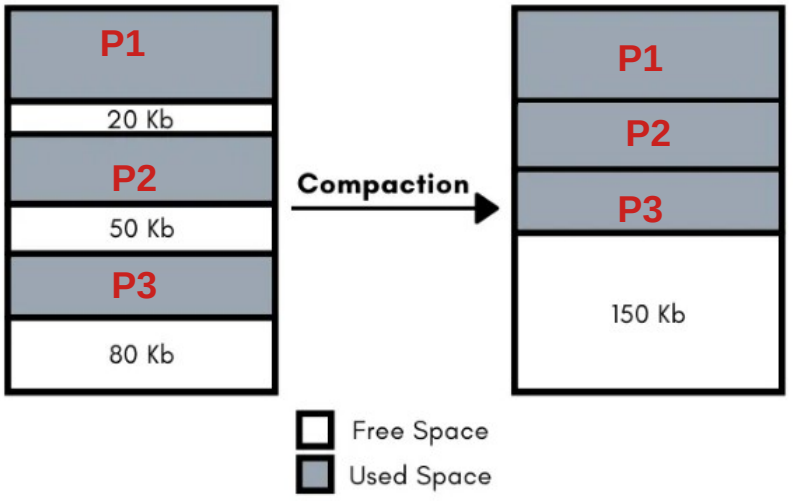
- One per process
- One for the kernel!
  - Remember: kernel code also undergoes translation in MMU!

A point of confusion on “root” user

- “root” user has privileges in accessing files
- When “root” user runs an application, it runs in process-mode, not in kernel-mode!
  - That process can do “more” but , it’s not kernel mode code!
- Kernel is not run by “root” user!
  - Kernel is not run by any user

Solution to external fragmentation

- Compaction !
- OS moves the process chunks in memory to make available continous memory region
  - Then it must update the memory management information in PCB (e.g. base of the process) of each process
- Time consuming
- Possible only if the relocation+limit scheme of MMU is available



Another solution to external fragmentation: Fixed size partitions

- Fixed partition scheme
- Memory is divided by OS into chunks of equal size: e.g., say, 50k
  - If total 1M memory, then 20 such chunks
- Allocate one or more chunks to a process, such that the total size is >= the size of the process
  - E.g.if request is 50k, allocate 1 chunk
  - If request is 40k, still allocate 1 chunk
  - If request is 60k, then allocate 2 chunks
- Leads to internal fragmentation
  - space wasted in the case of 40k or 60k requests above

Solving external fragmentation problem

- Process should not be continous in memory!
  - The trouble is finding a big continous chunk!
- Divide the continuous process image in smaller chunks (let's say 4k each) and locate the chunks anywhere in the physical memory
  - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

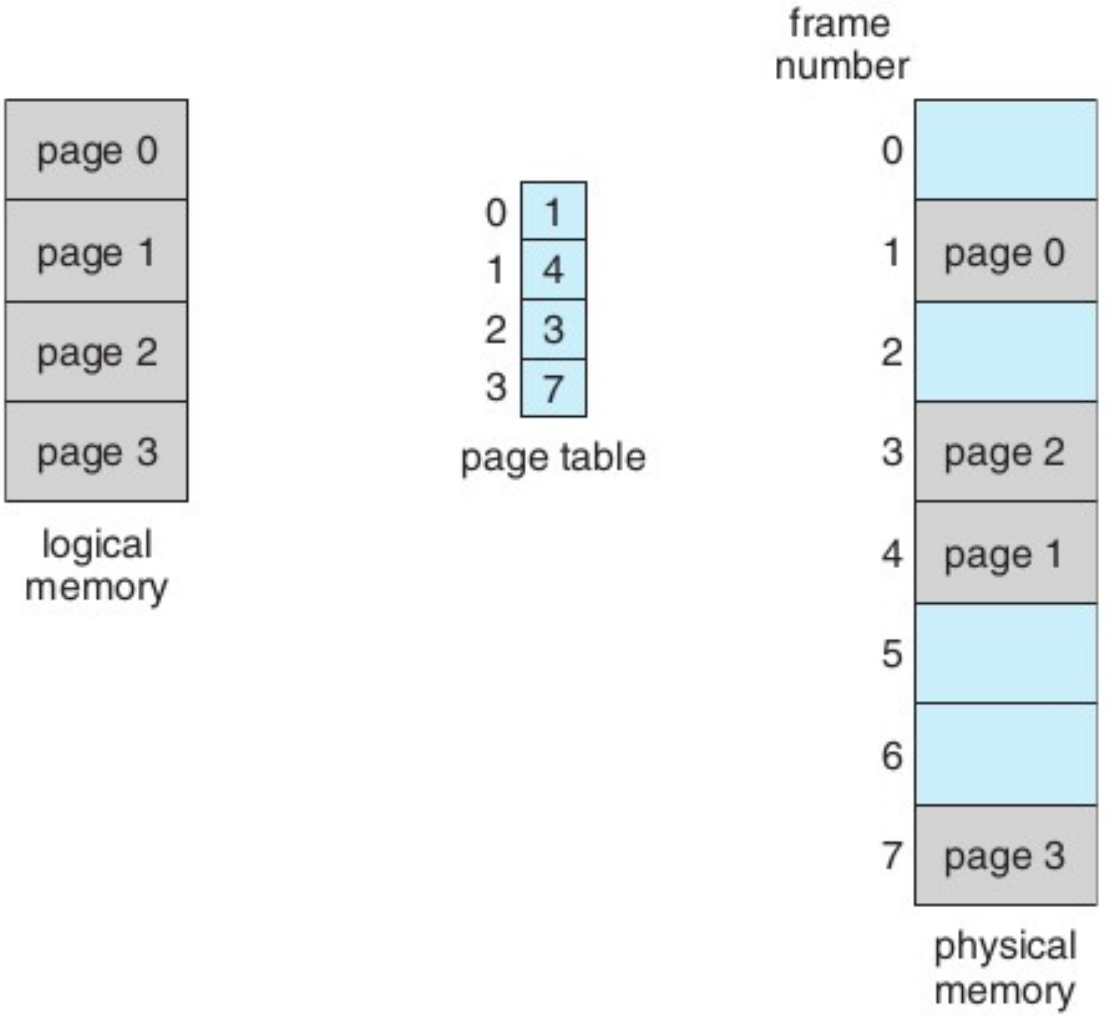


Figure 9.9 Paging model of logical and physical memory.

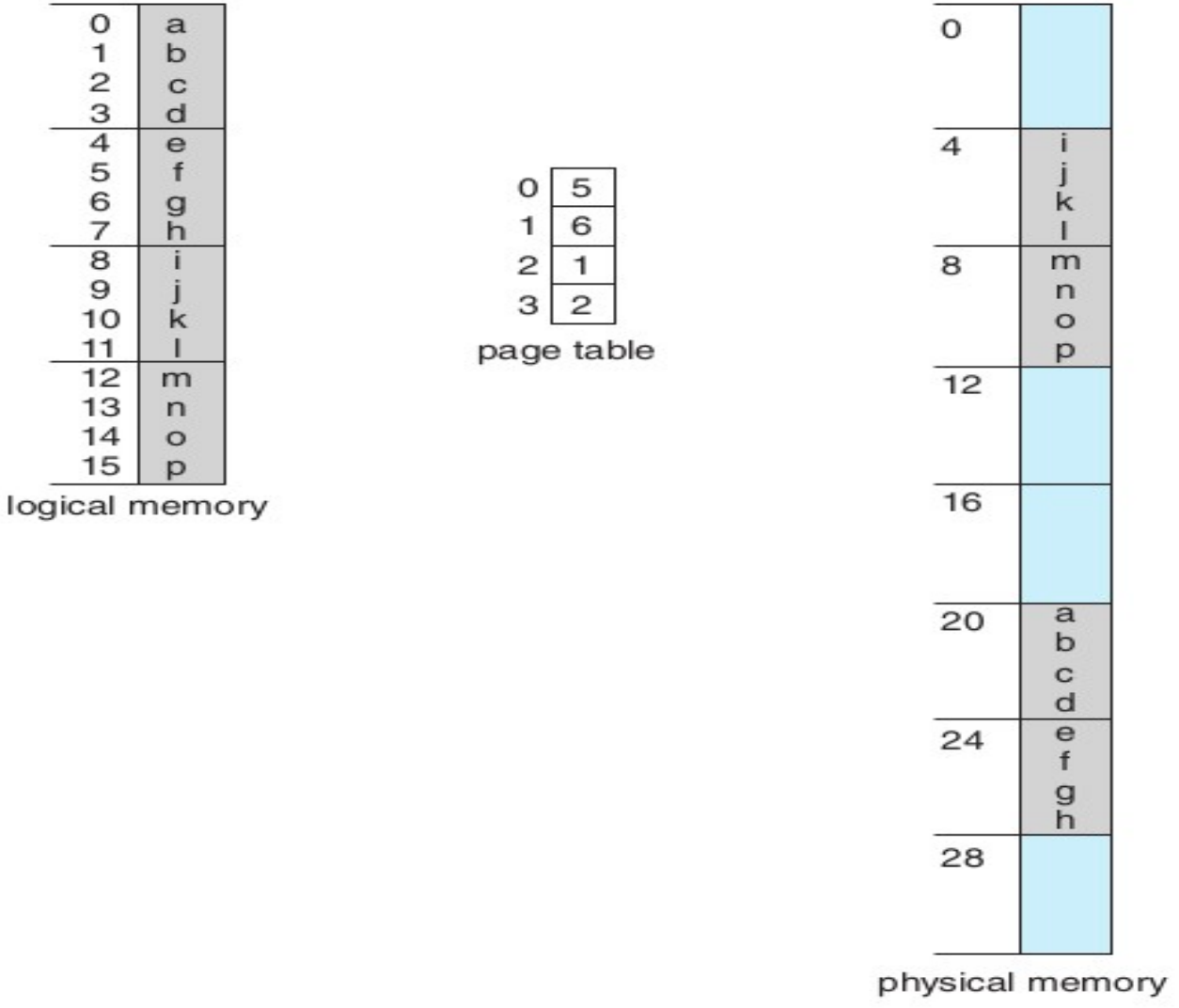


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

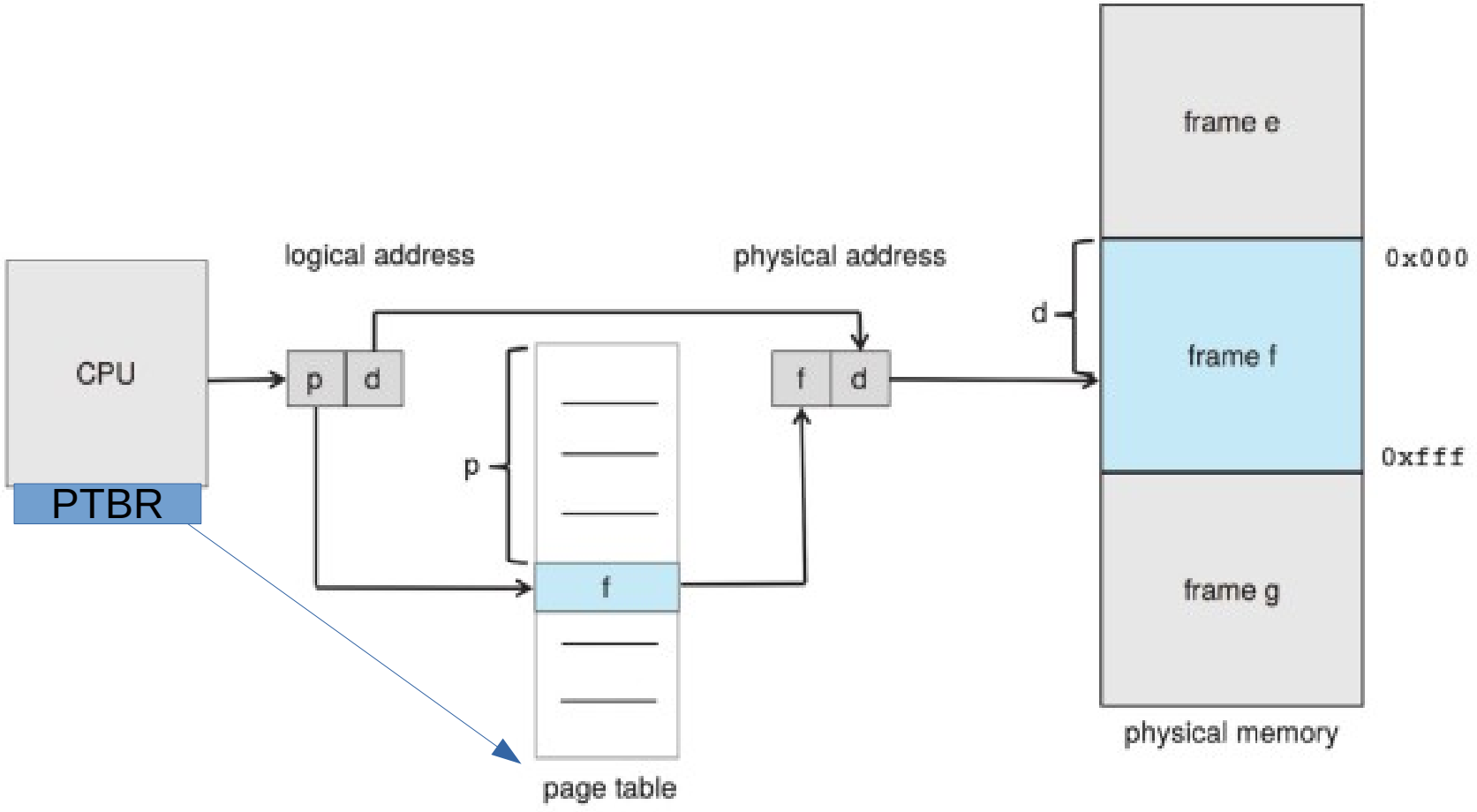


Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A Page Table Base Register (PTBR) inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continous chunk of memory (!) . Generate addresses accordingly . Compiler still treats text,data,bss,.. to be separate chunks, but in multiples of page-sizes.
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

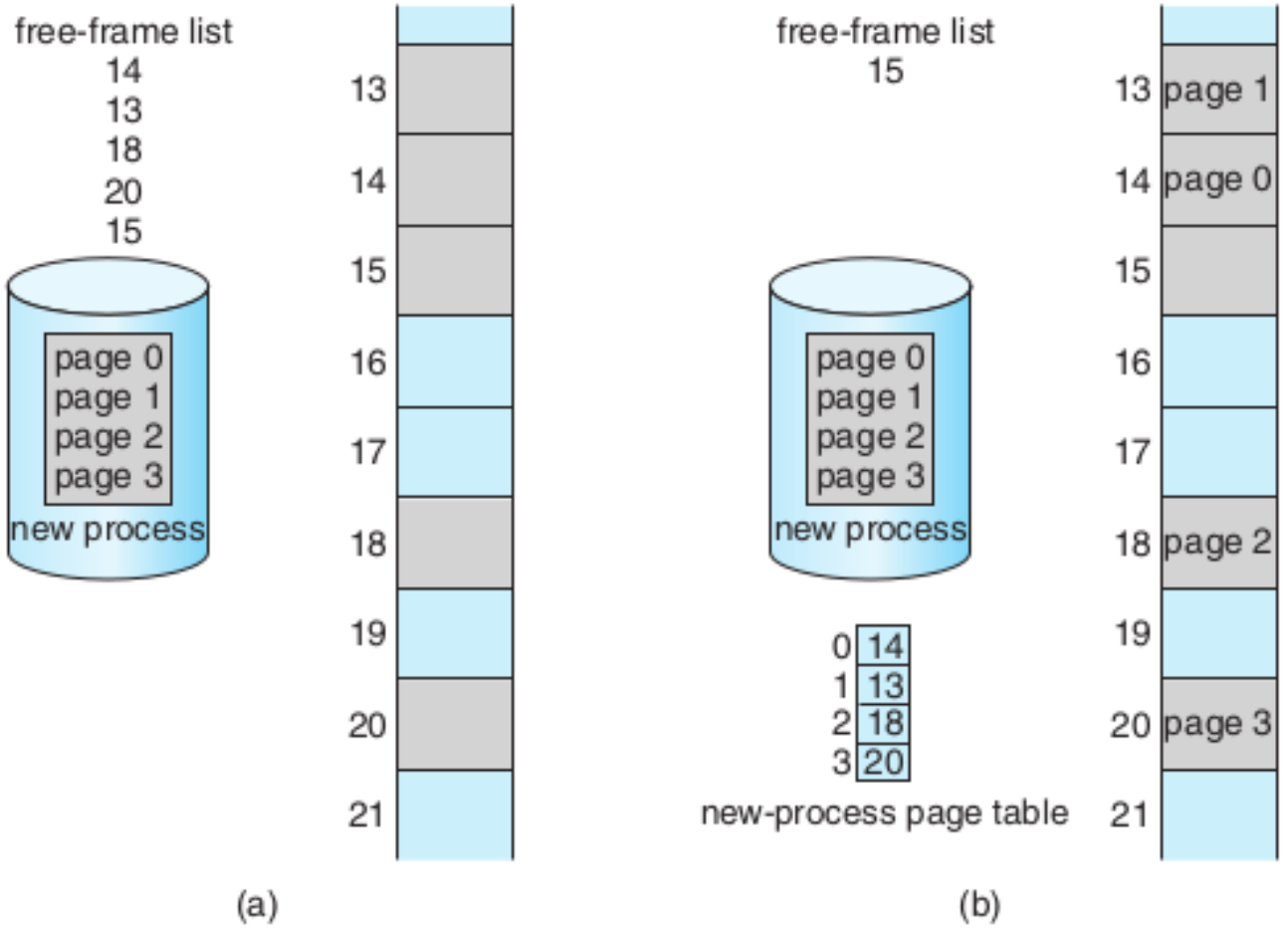
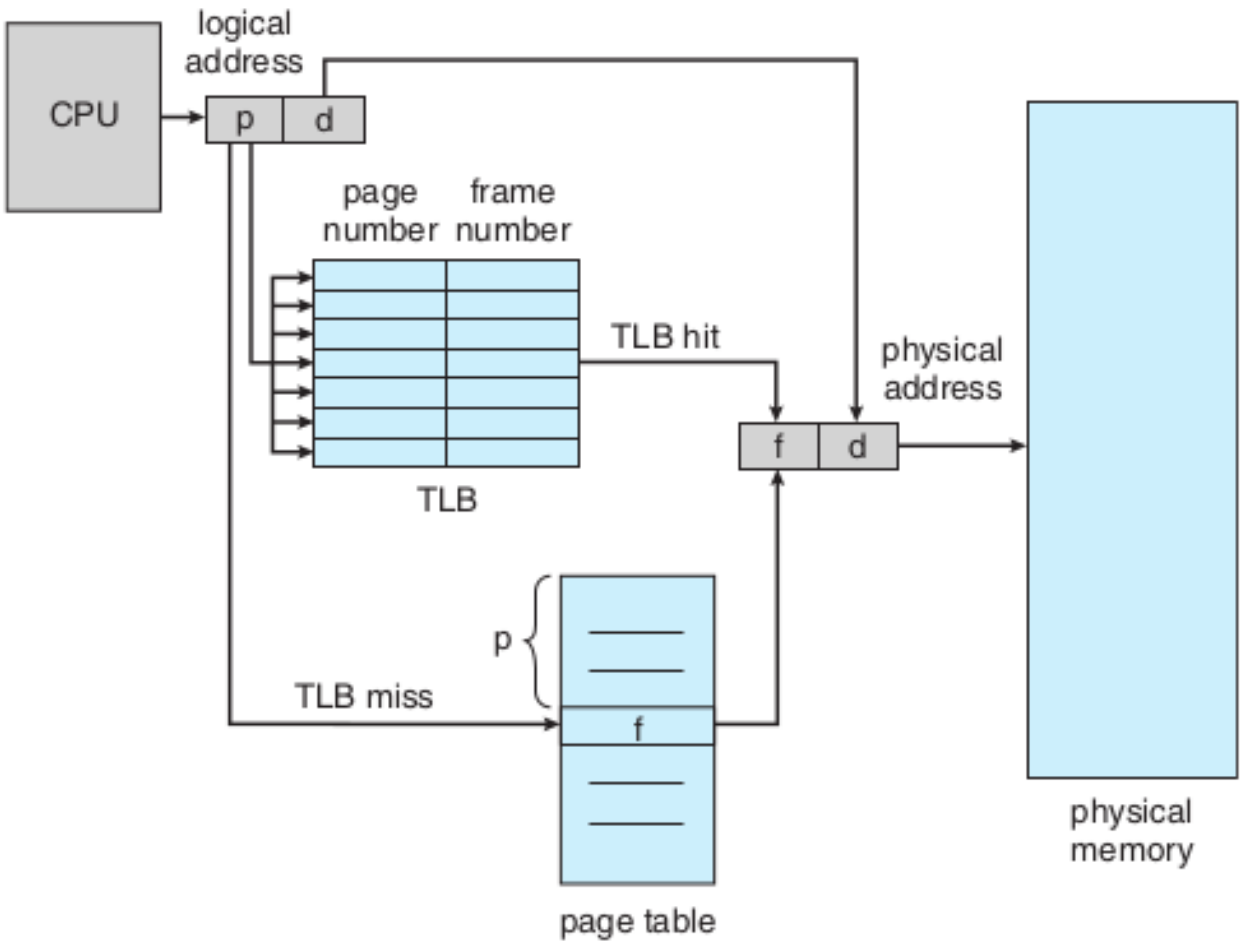


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number



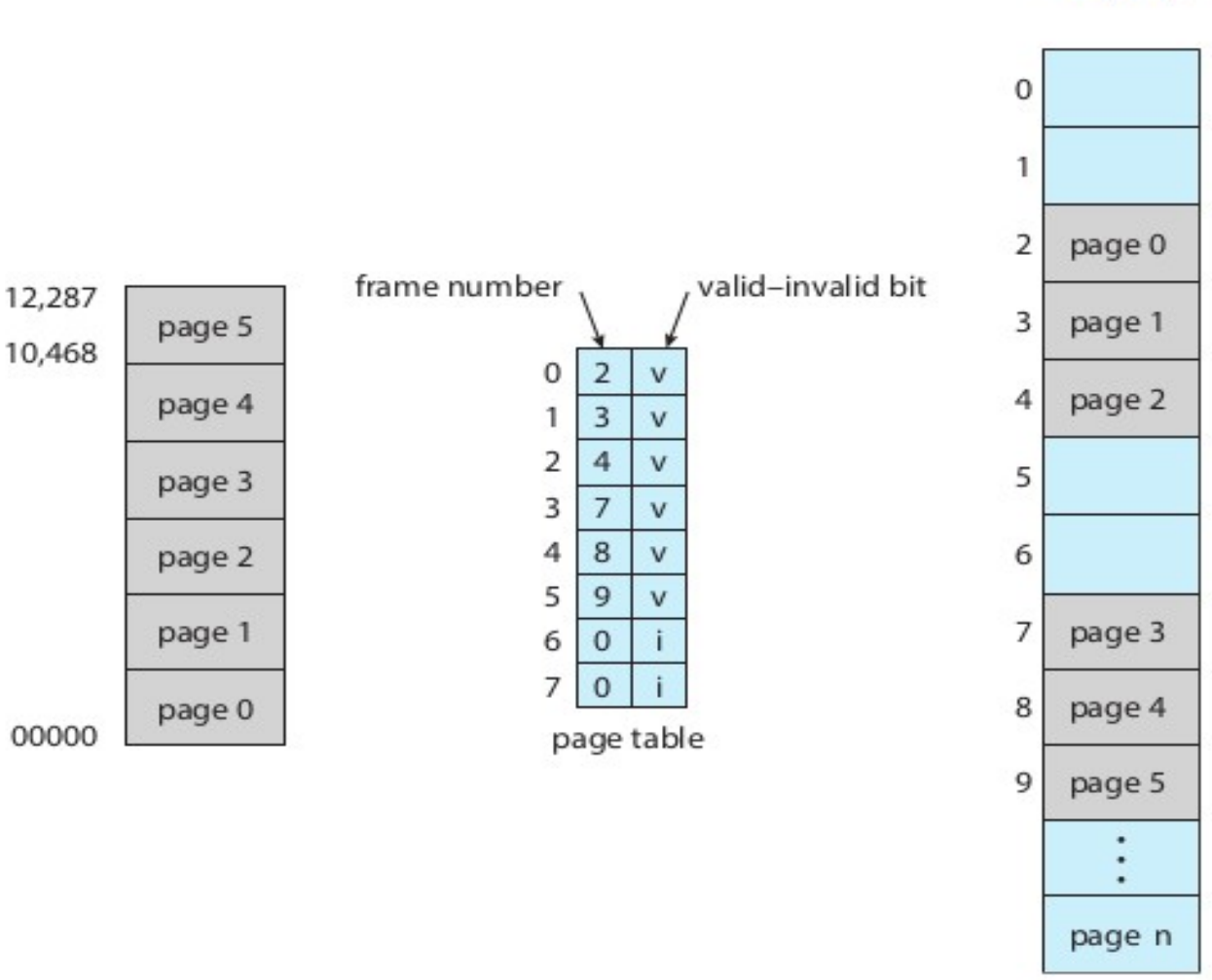


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

Memory protection with paging

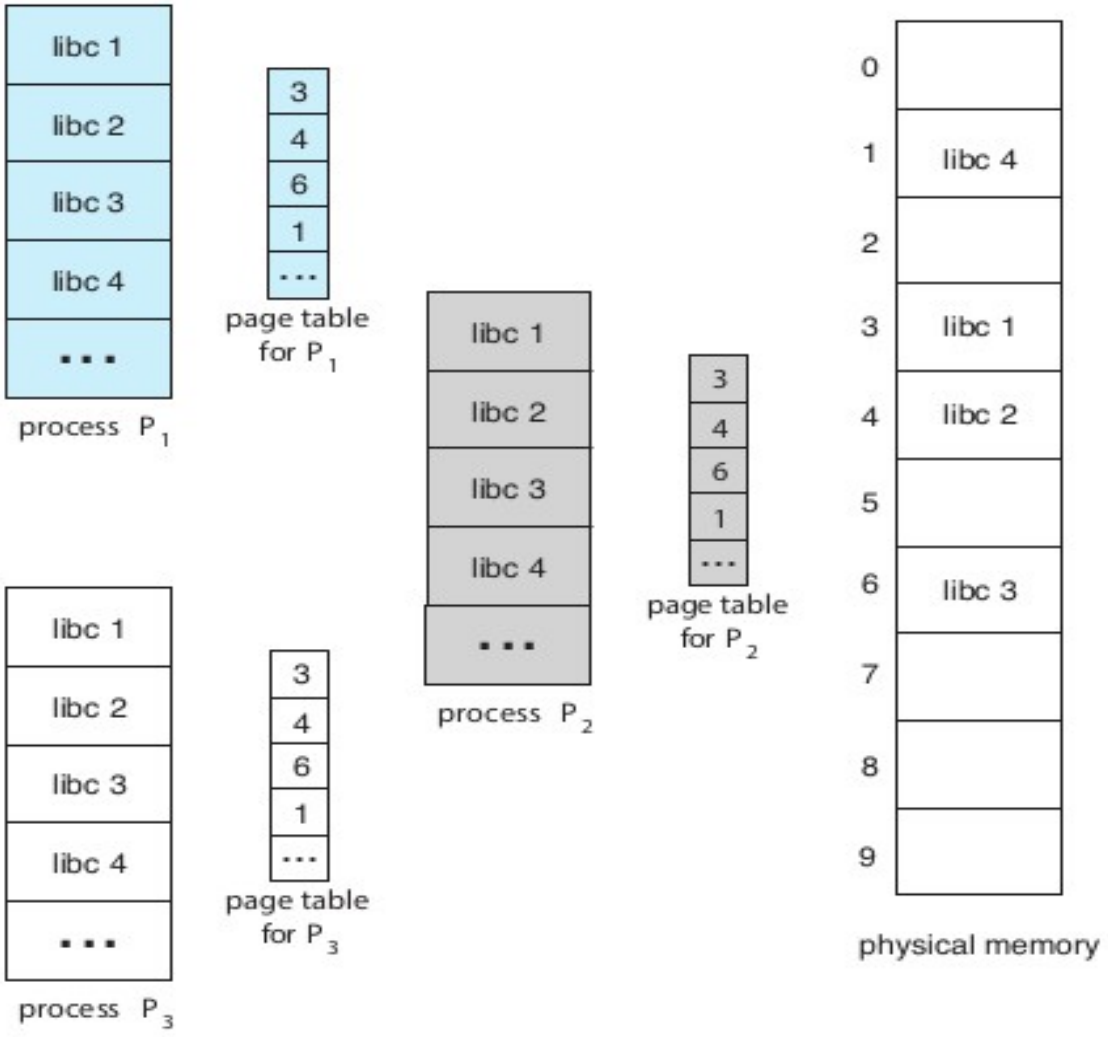


Figure 9.14 Sharing of standard C library in a paging environment.

Shared pages (e.g. library) with paging

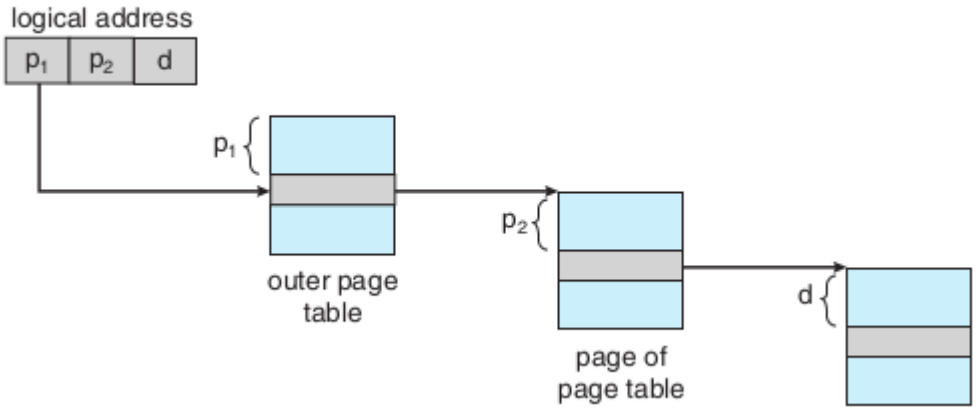
Paging: problem of large PT

- 64 bit address
- Suppose 20 bit offset
  - That means 2^20 = 1 MB pages
  - 44 bit page number: 2^44 that is trillion sized page table!
  - Can’t have that big continuous page table!

Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
  - That means 2^12 = 4 KB pages
  - 20 bit page number: 2^20 that is a million entries
  - Can’t always have that big continuous page table as well, for each process!

Hierarchical paging



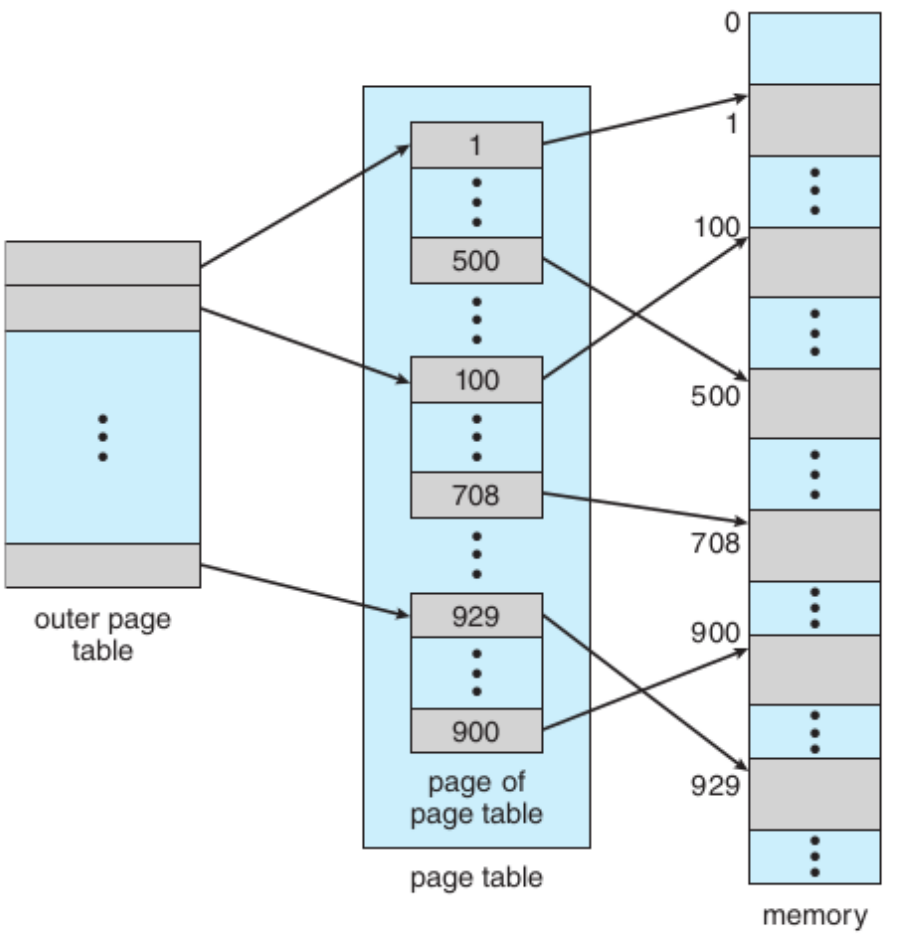
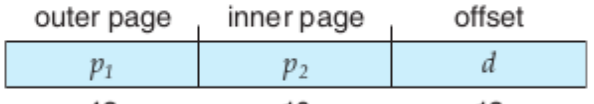


Figure 9.15 A two-level page-table scheme.



X86 memory management

Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

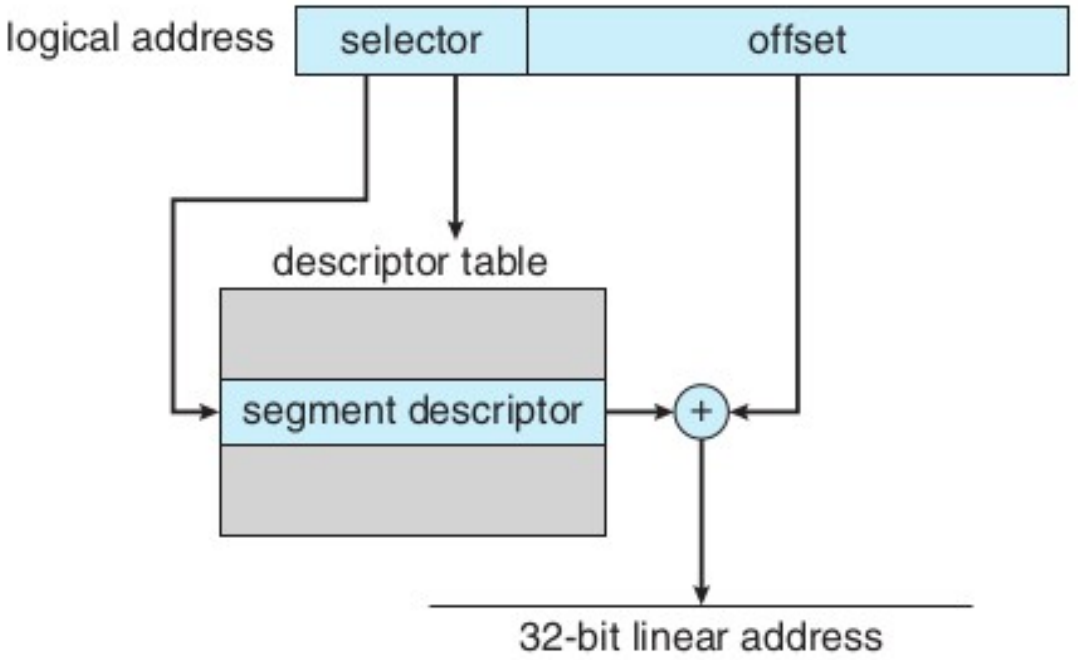


Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table (that is a segmentation table) is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR

Paging in x86

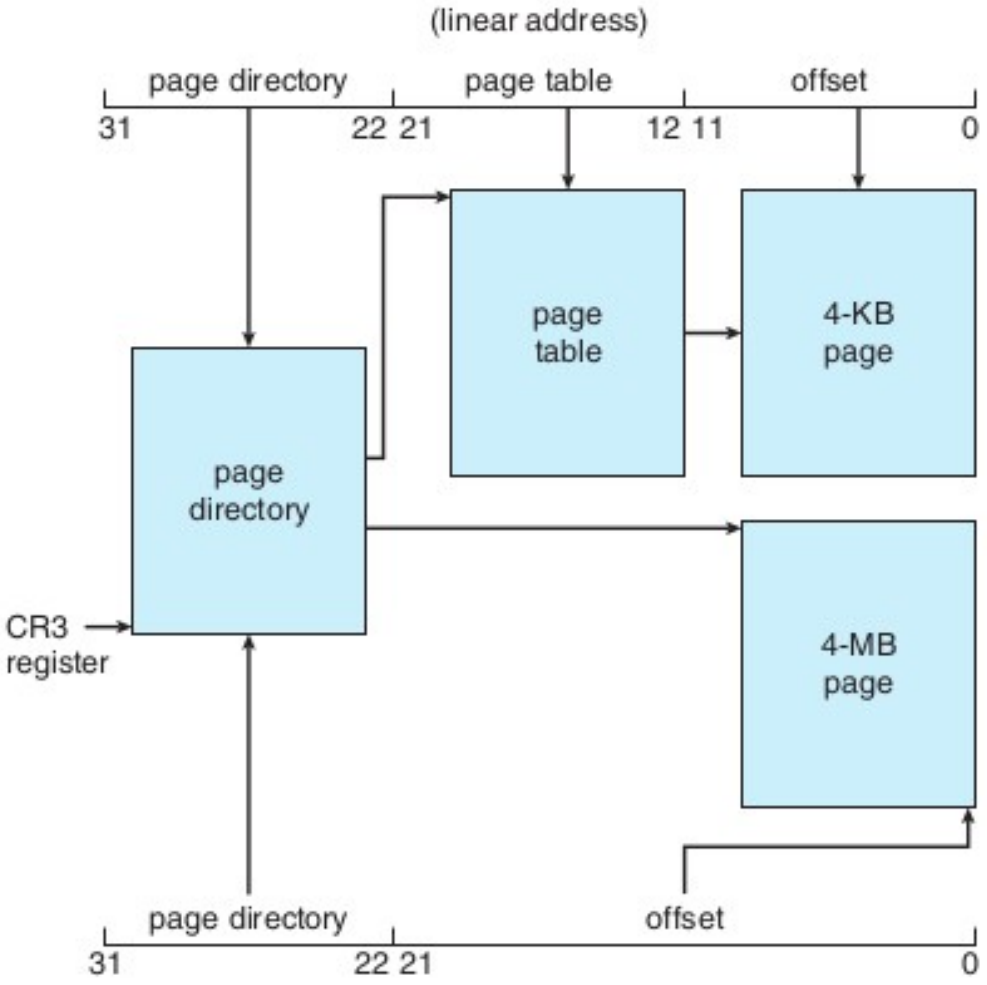
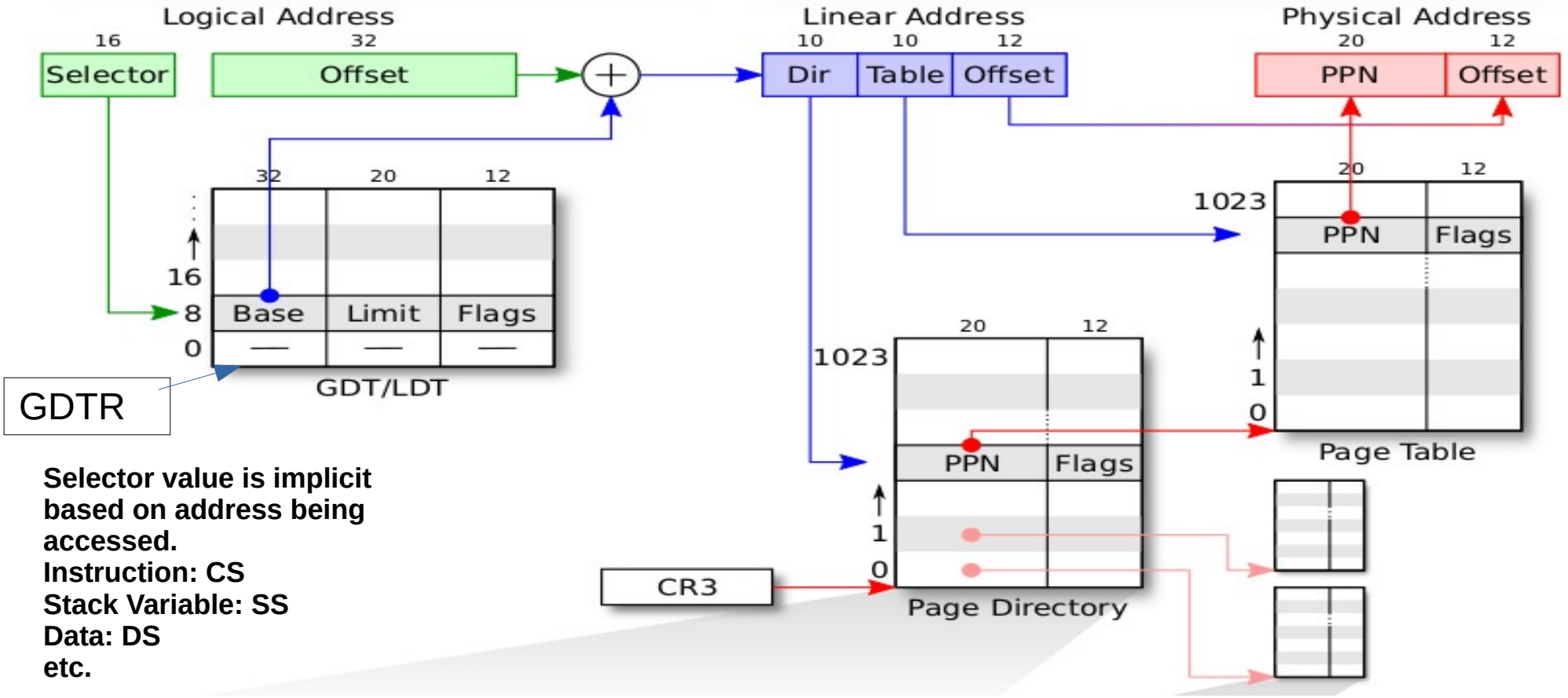


Figure 9.23 Paging in the IA-32 architecture.

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

X86 Segmentation + Paging



Selector value is implicit based on address being accessed.
Instruction: CS
Stack Variable: SS
Data: DS
etc.

X86 Segmentation + Paging

- **Paging is optional, segmentation compulsory**
  - **Setting flags in Control Registers (CR) enables this**
- **Page Table, Page Directory, page - are all size=4k, if 2-level paging is used**
  - **Makes life simpler for the kernel**