Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Memory Management Strategies

Last Update: April 24, 2023

Objectives and Outline

Objectives

- Describe ways of organizing memory hardware
- discuss various memory-management techniques, including paging and segmentation
- Description of the Intel x86 architecture, which supports both pure segmentation and segmentation with paging

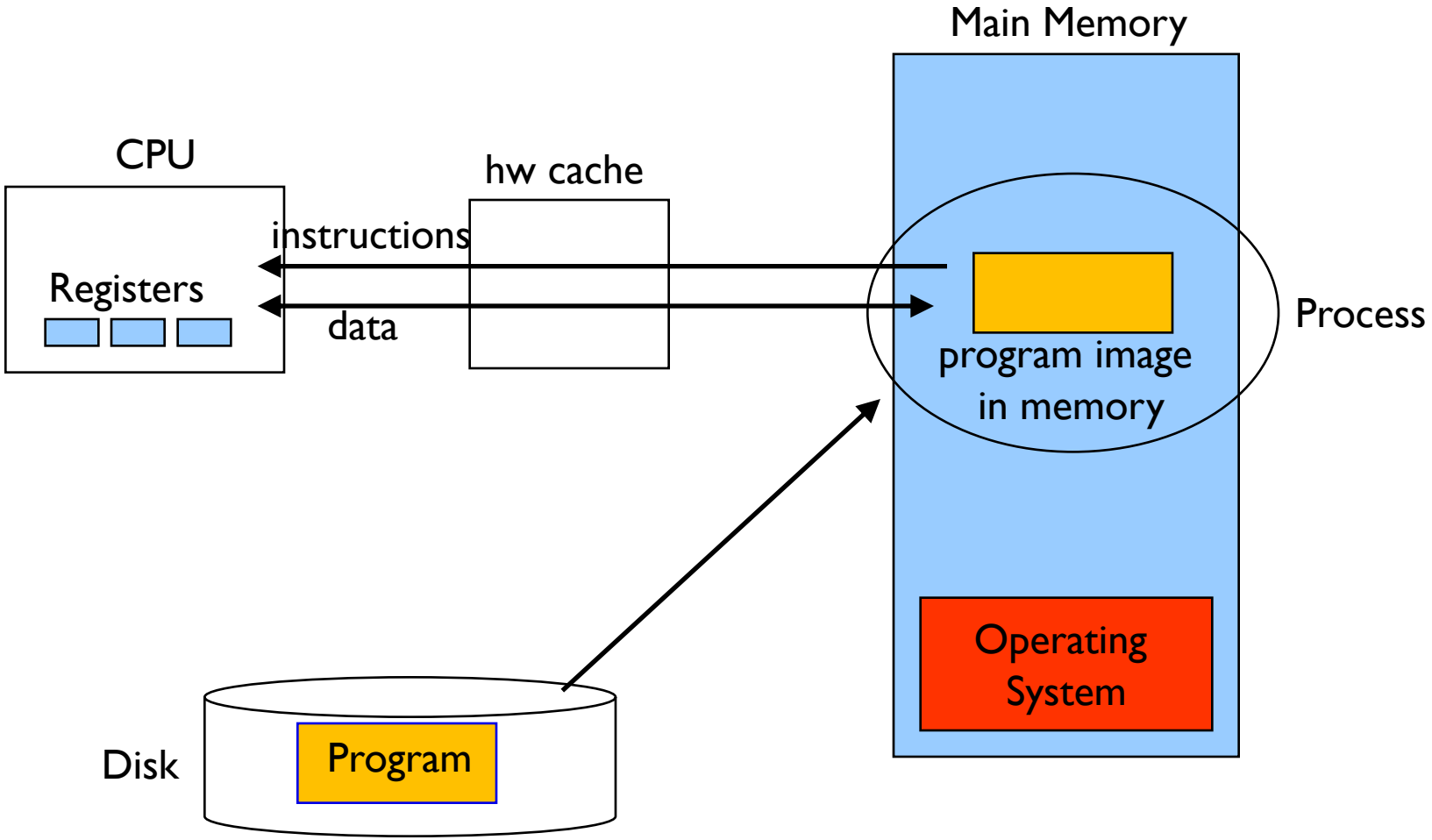
Outline

- Background
  - Address space
  - Logical address space
  - MMU
- Contiguous Memory Allocation
- Paging
  - Structure of the Page Table
- Segmentation

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory (Physical Memory) and registers are only storage CPU can access directly
  - Register access in one CPU clock cycle (or less)
  - Main memory can take many cycles.
  - Cache sits between main memory and CPU registers
- Physical Memory (RAM): a sequence of bytes. Each byte has an address.
- Protection of memory required to ensure correct operation

Background

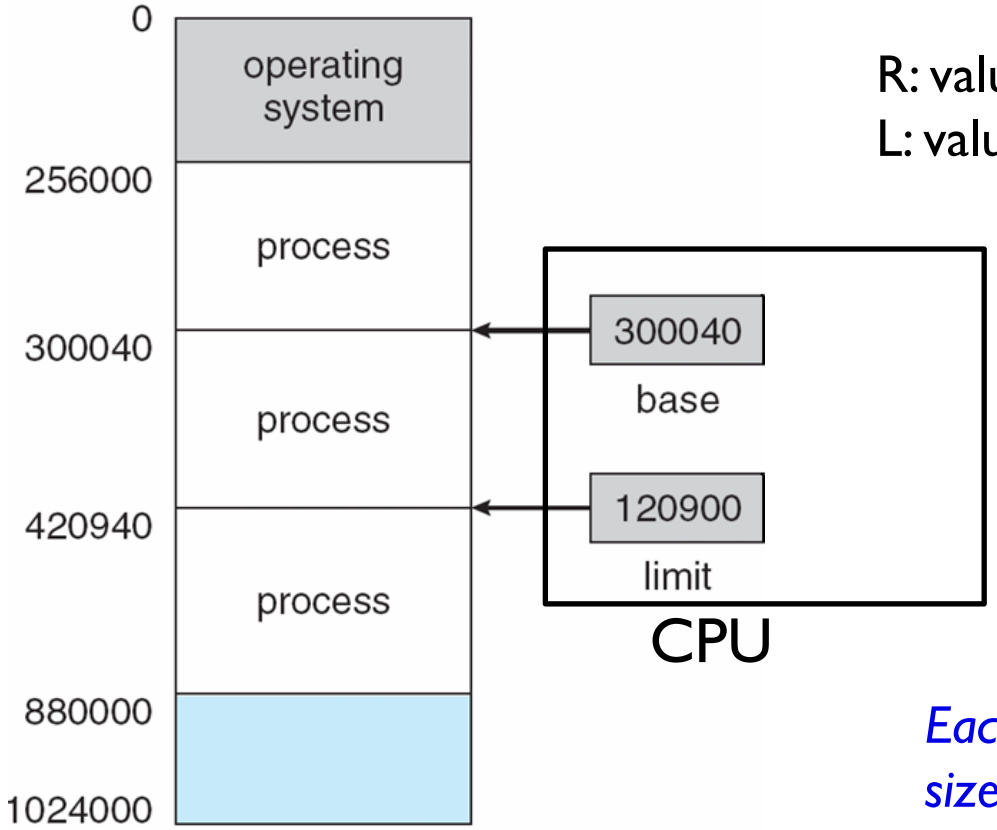

Main Memory
CPU
hw cache
instructions
Registers
data
Process
program image
in memory
Operating
System
Disk
Program

Multiple programs and Protection

- Multiple programs started share main memory.
- Programs should not access each other's memory and OS memory. Protection needed.
  - OS can and should access everything.
  - A program is allowed to access the physical memory allocated to itself only.
- Hardware support needed for protection.
- Having base and limit registers at CPU can be used for a basic protection mechanism.
  - Base register: contains the start physical address (start PA) of the program to run.
  - Limit (bound) register: contains the size of the program.

Protection

A pair of base and limit registers define the physical address space of a process running in the CPU.



R: value in base reg (start PA process)
L: value in limit reg (size of process)

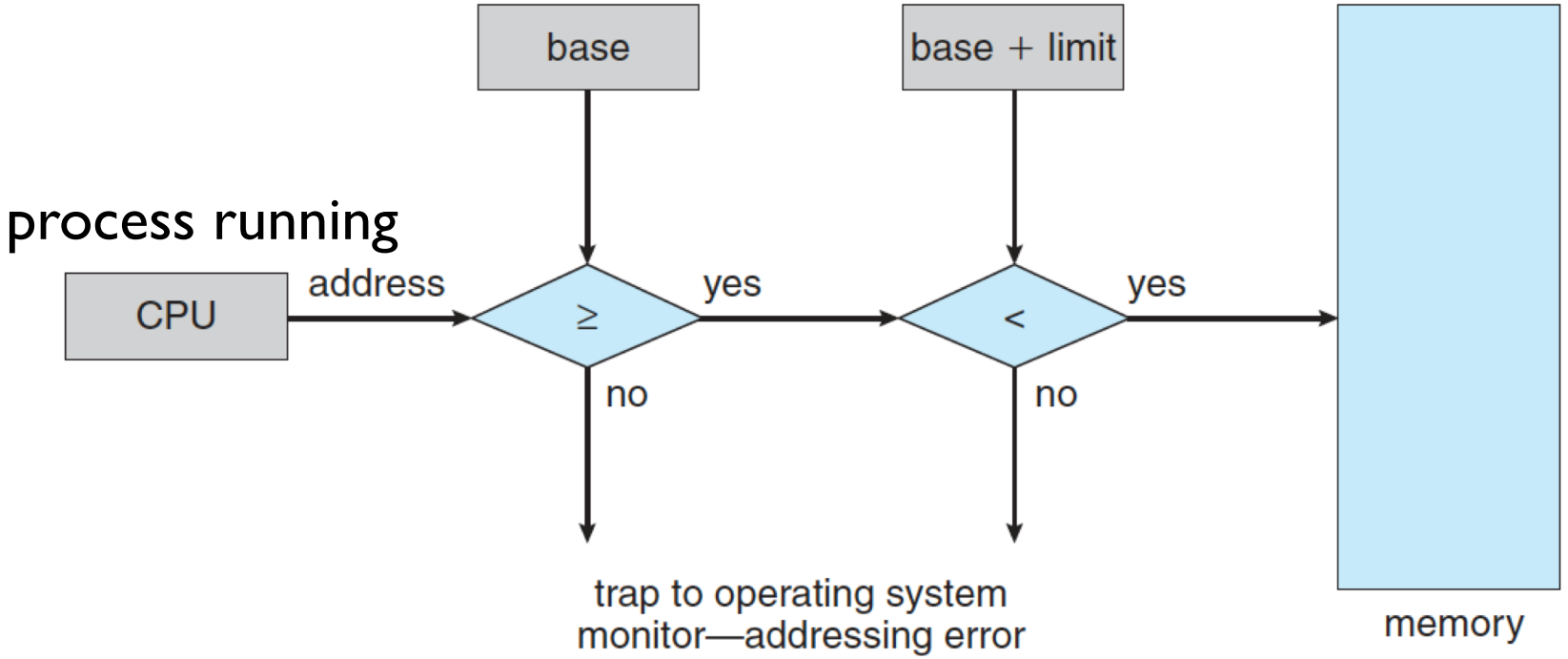
PA range: [R, R+L)

Process can only access that range.
Access to outside the range: memory error.

Each process has its own start PA and size. Can be stored in the PCB of the process.

Protection

Base and limit registers are set by OS before selected process (by scheduler) starts execution in CPU.

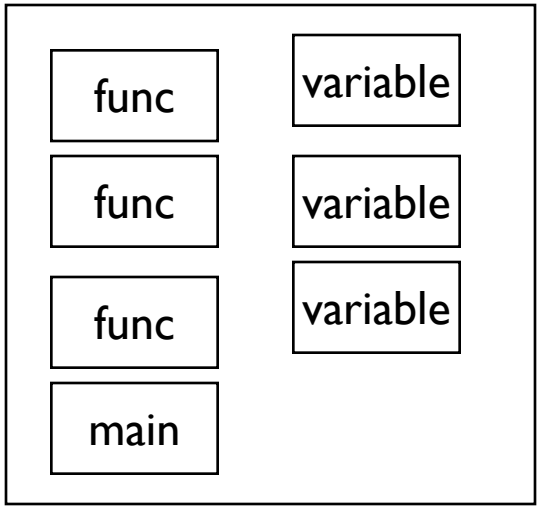


Hardware checks every address.

Access to memory is protected.

Program addresses and memory

- Source code uses symbolic names for program elements (variables, functions, etc.)
- When machine code is generated (via compiler or asssember) numeric memory addresses are used for instructions, data, and functions.
- Those addresses can be
  - physical memory addresses (PA),
  - or, logical memory addresses (LA).

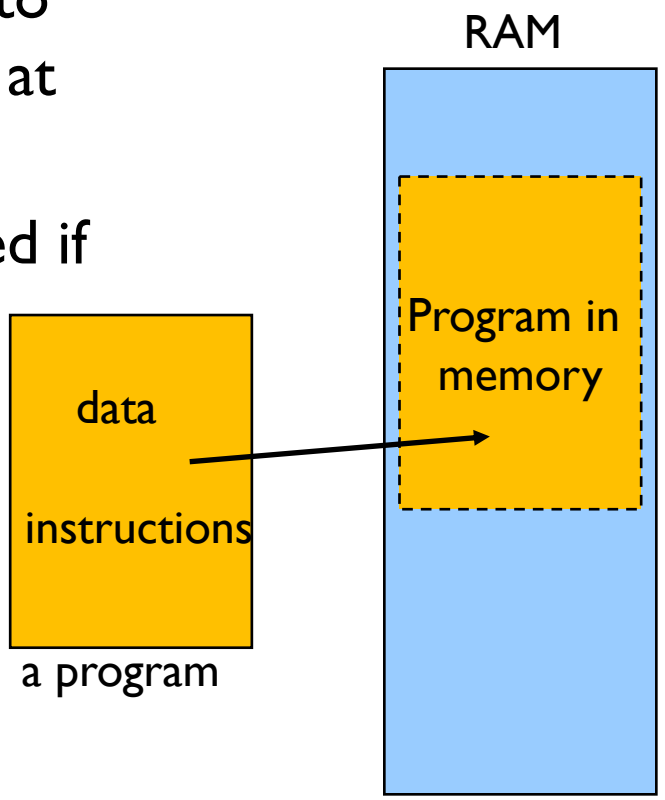


program

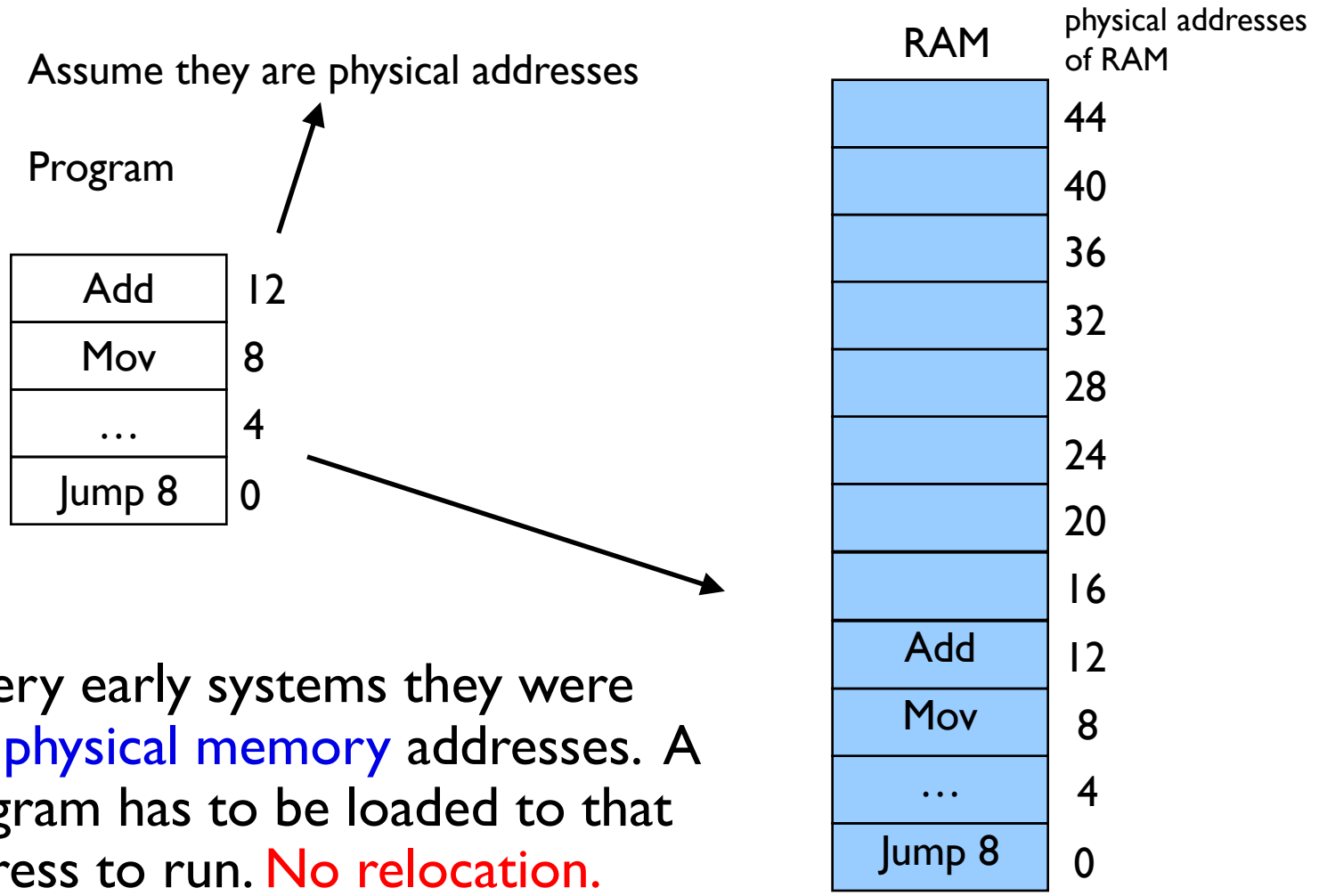
Finally, program elements (instructions, data, etc.) should be loaded to physical locations and will have physical addresses.

Binding of Instructions and Data to Memory

- Address binding of instructions and data to (physical) memory addresses can happen at three different stages.
  - Compile time: absolute code generated if start PA address known.
  - Load time: compiler generates relocatable code. Loader modifies addresses wrt start PA address.
  - Execution time: Binding at run time. Program uses logical addresses *all* the time. Mapping from LA to PA done by HW at runtime. Program can be moved at run time.



Program addresses and memory



In very early systems they were just physical memory addresses. A program has to be loaded to that address to run. No relocation.

Program addresses and memory

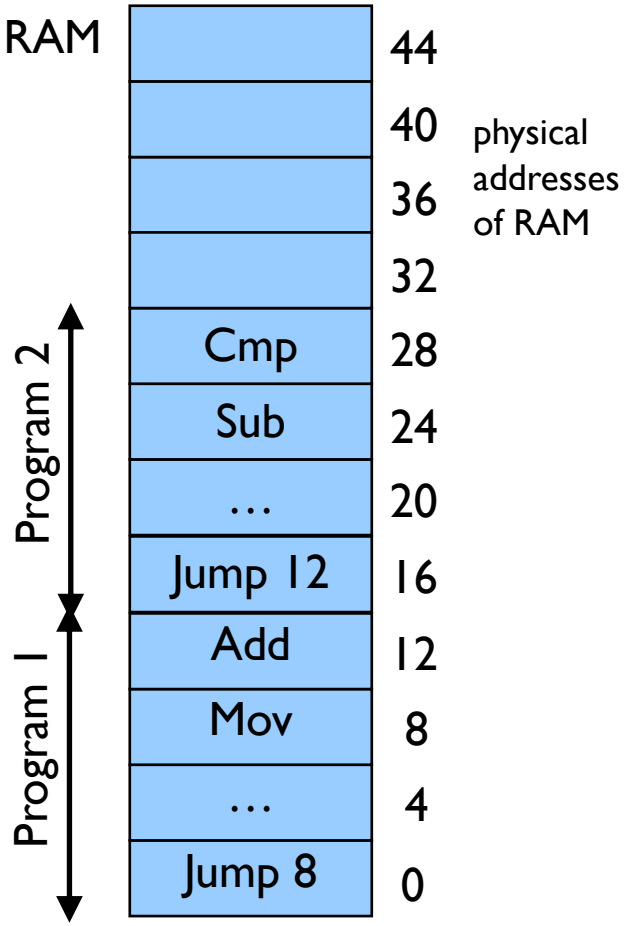
Program 2 will not work correctly.
Since it is not loaded at address 0.

Program 1

Add	12
Mov	8
…	4
Jump 8	0

Program 2

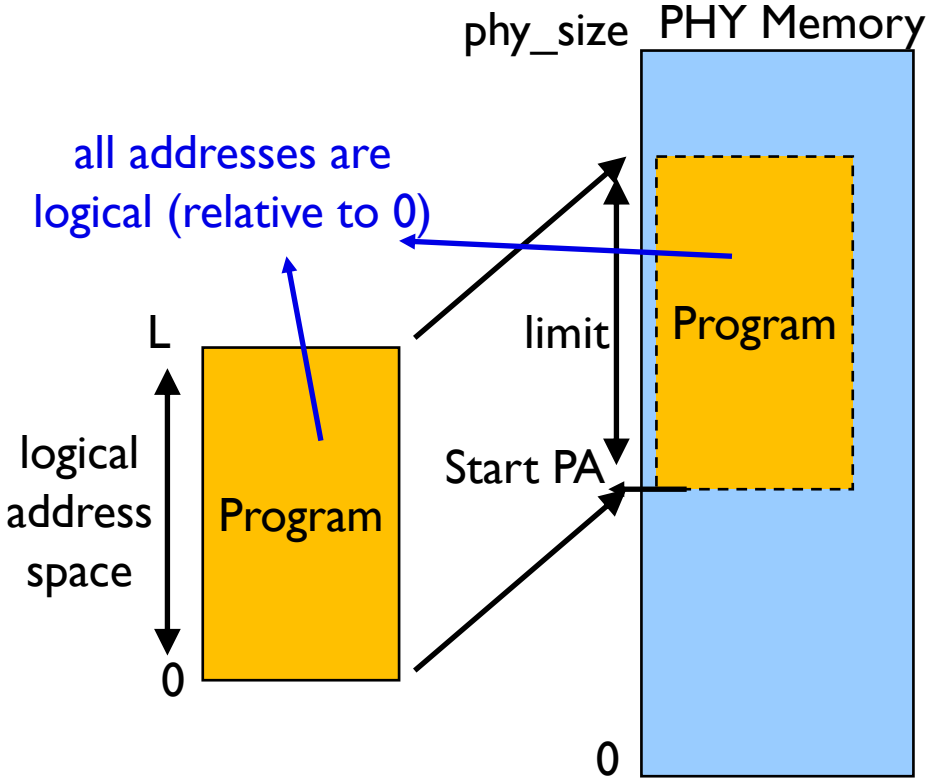
Cmp	12
Sub	8
…	4
Jump 12	0



Use of physical address in the program is not flexible and complicates the job of the programmer/compiler and also complicates the management of memory (OS).

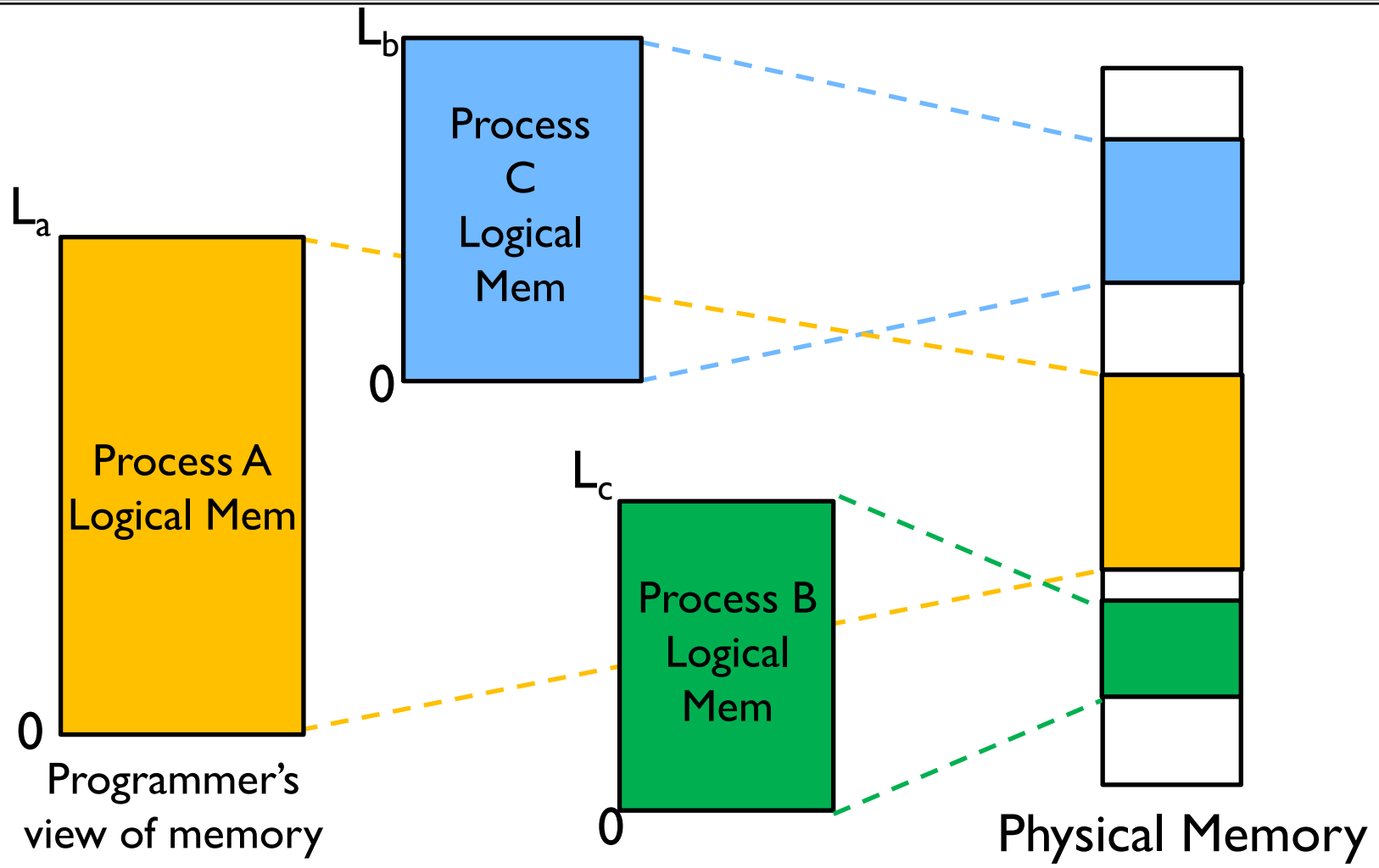
Logical address space concept for a process

- We need logical address space (logical memory) concept, that is different than the physical RAM (main memory) addresses.
- A program uses logical addresses. They start at 0.
- Set of logical addresses used by the program is its logical address space.
  - Logical address space can be, for example, [0, L).
  - L is size of logical memory.

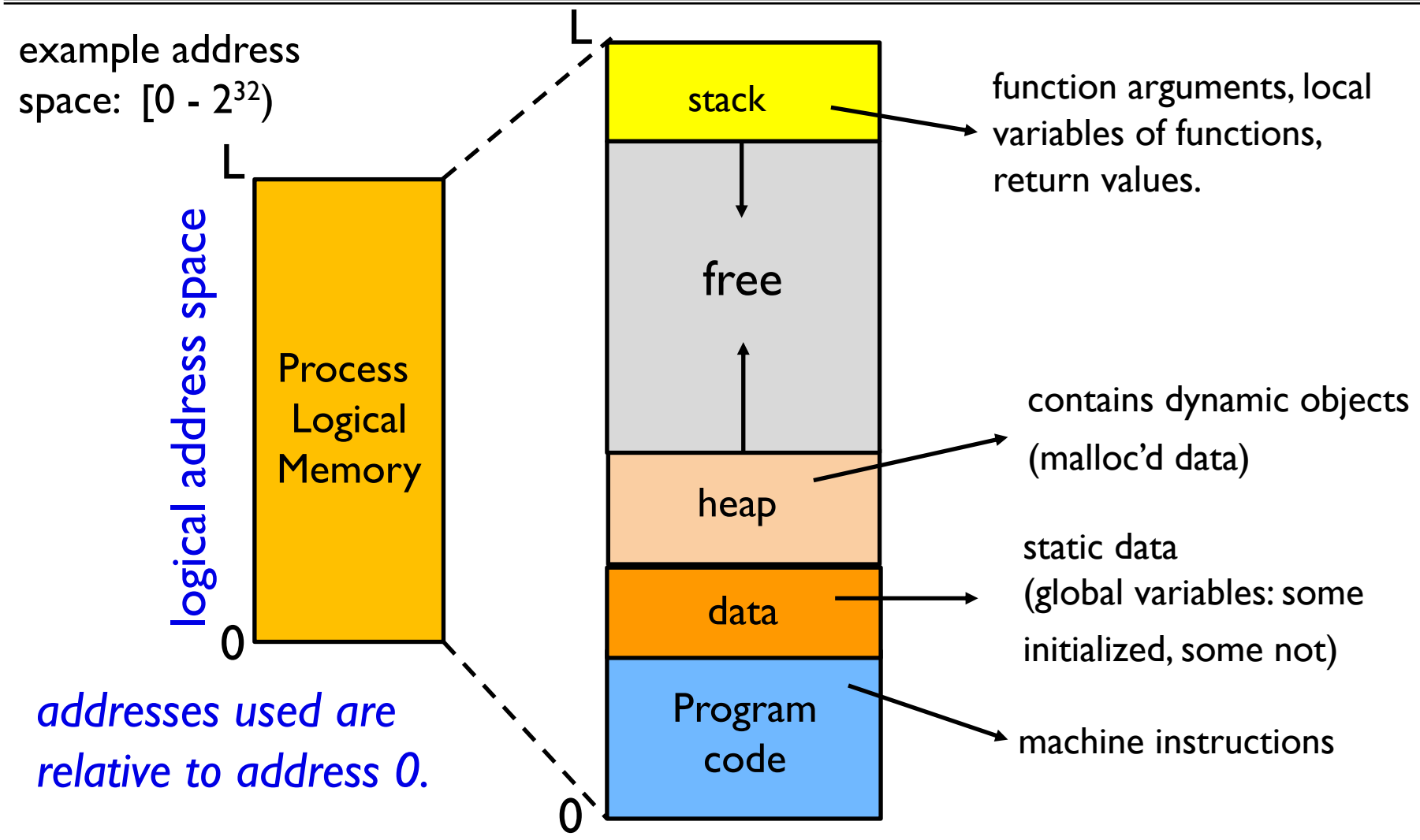


In this way, memory is virtualized. Logical address space has to be mapped somewhere in physical memory when program started.

Logical address space concept



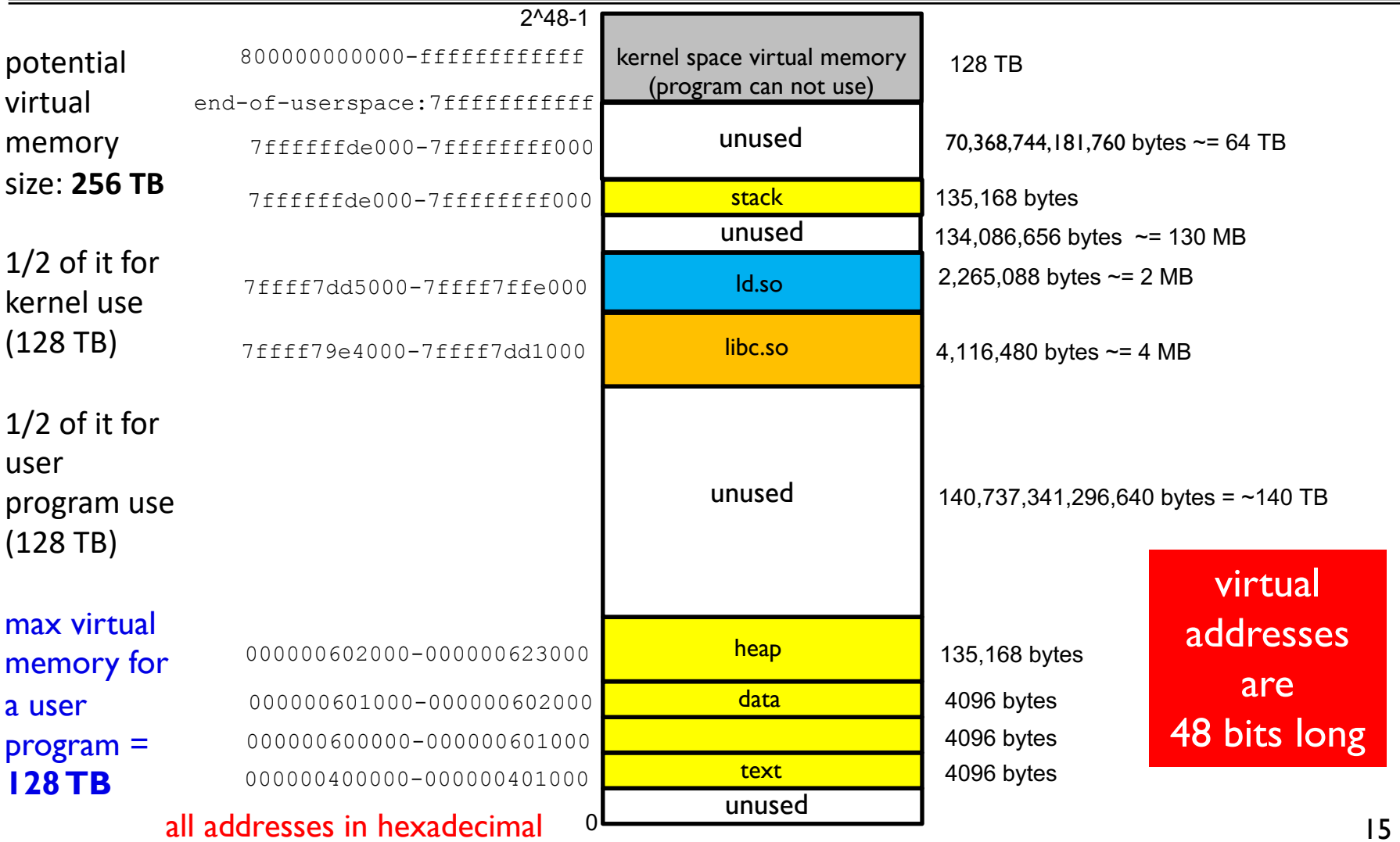
Logical (virtual) memory layout of a process



a real-life example: a 64-bit machine – x86-64 / Linux OS

VM layout for a process (small process)

very large logical address space (potential VM is too big)



VM layout of a process
64-bit machine – x86-64 / Linux OS

output of:

cat /proc/pid/maps

```
00400000-00401000 r-xp 00000000 08:01 50081283                           [code]
00600000-00601000 r--p 00000000 08:01 50081283                           [code-system]
00601000-00602000 rw-p 00001000 08:01 50081283                           [data]
00602000-00623000 rw-p 00000000 00:00 0                                  [heap]
7ffff79e4000-7ffff7bcb000 r-xp 00000000 08:01 67109118                   /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7bcb000-7ffff7dcb000 ---p 001e7000 08:01 67109118                   /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcb000-7ffff7dcf000 r--p 001e7000 08:01 67109118                   /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dcf000-7ffff7dd1000 rw-p 001eb000 08:01 67109118                   /lib/x86_64-linux-gnu/libc-2.27.so
7ffff7dd1000-7ffff7dd5000 rw-p 00000000 00:00 0
7ffff7dd5000-7ffff7dfc000 r-xp 00000000 08:01 67108949                   /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7fdf000-7ffff7fe1000 rw-p 00000000 00:00 0
7ffff7ff7000-7ffff7ffa000 r--p 00000000 00:00 0                          [vvar]
7ffff7ffa000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00027000 08:01 67108949                   /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffd000-7ffff7ffe000 rw-p 00028000 08:01 67108949                   /lib/x86_64-linux-gnu/ld-2.27.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
```

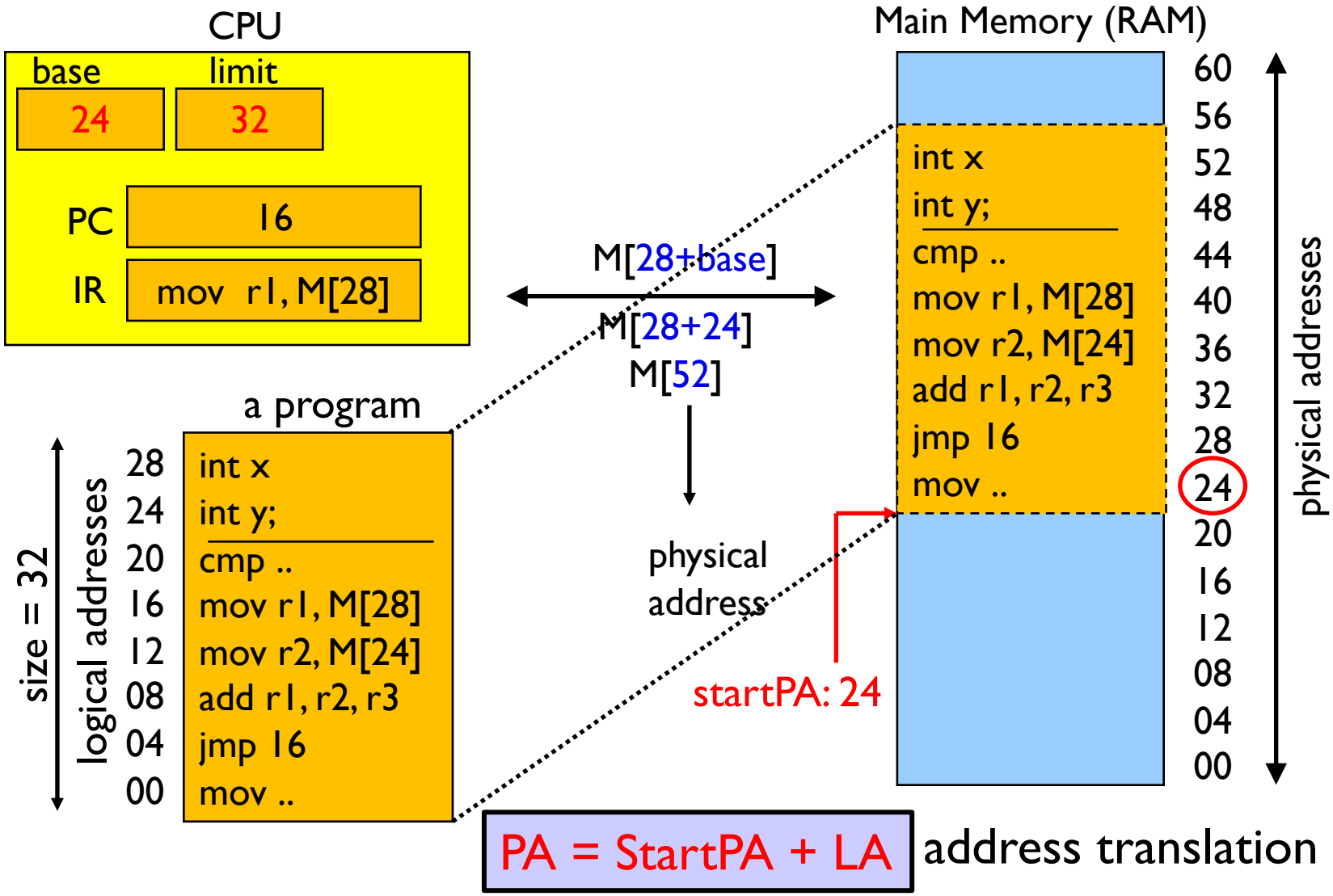
virtual memory regions of a process with id=pid

We can also use pmap command to get VM layout information for a process.

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address (LA) – generated by the CPU; also referred to as virtual address.
  - Physical address (PA) – address seen by the memory unit (RAM)
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes (program in memory has absolute addresses).
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme. Program in memory has logical addresses.

Logical and physical addresses
a small example (stack, etc., not shown)

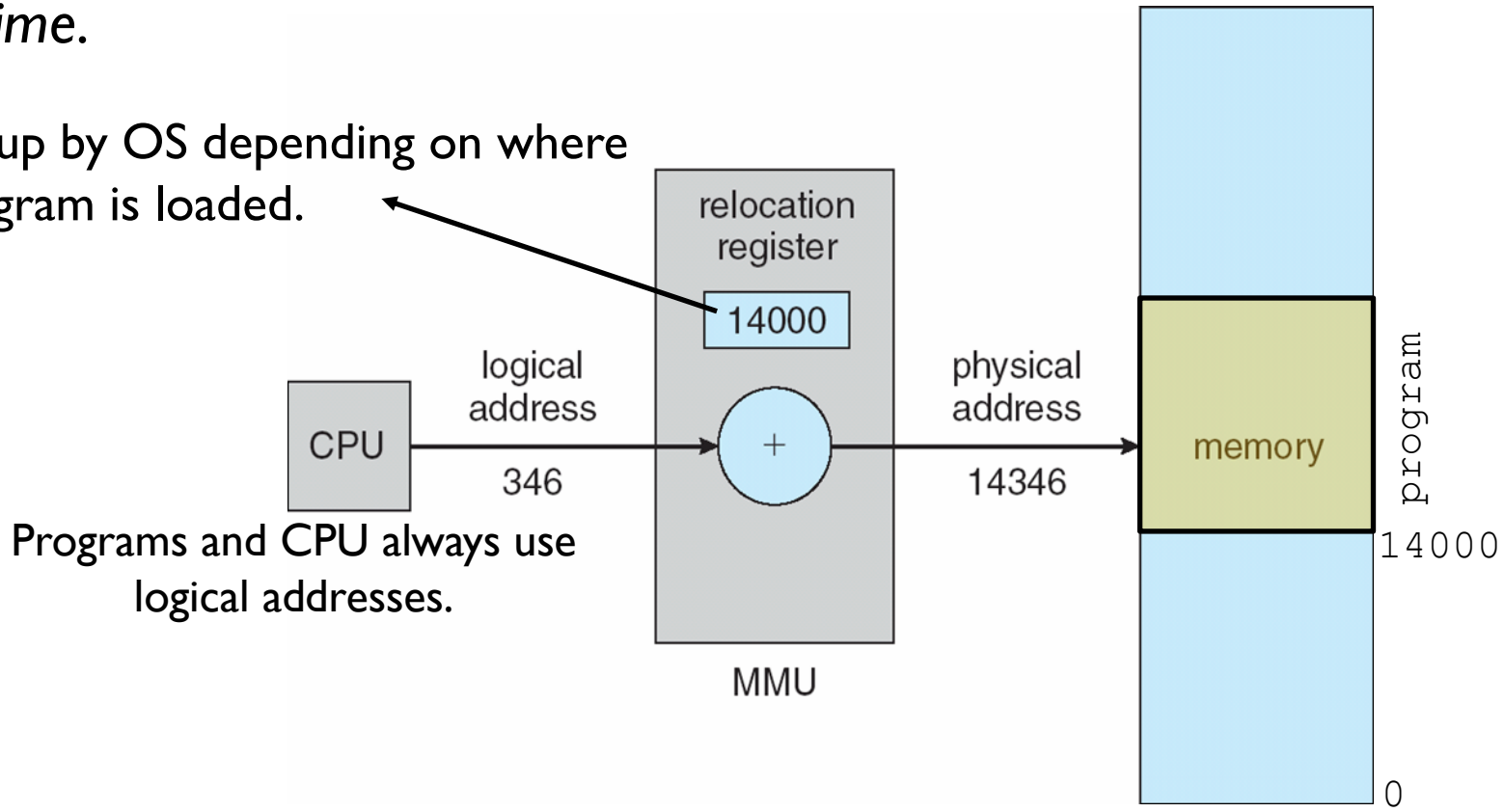


Memory-Management Unit (MMU)

- Hardware device that translates logical (virtual) addresses to physical addresses.
- In MMU, the start PA address of the program is loaded into base register (which is also called relocation register).
- The value in the base register is added to every logical address to obtain physical address.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
  - *Pointer values in a C program are all virtual addresses. Programmer does not see physical addresses.*
- Base (relocation) register together with limit (bound) register is used for both protection and relocation.

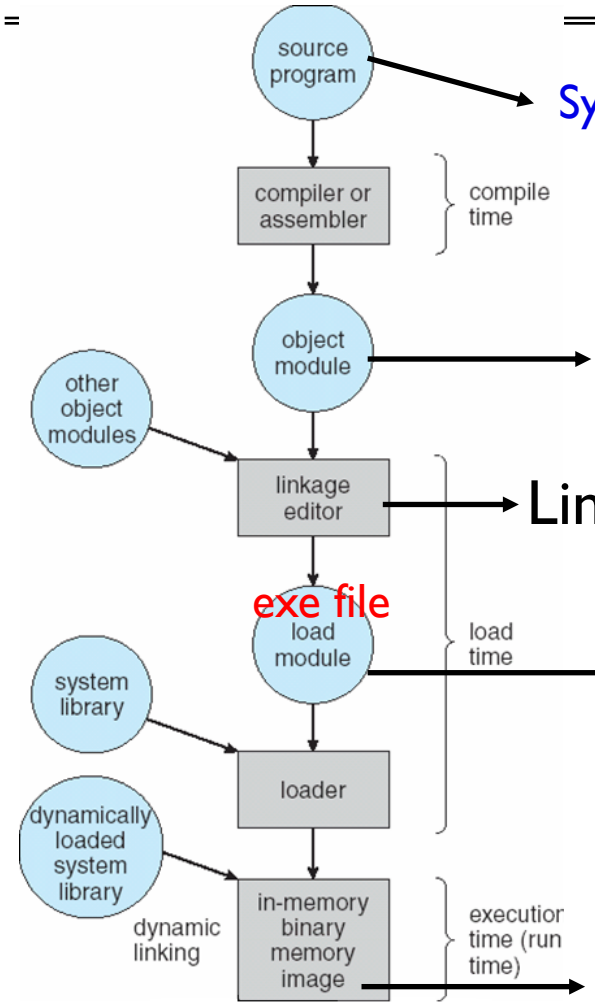
Address translation by HW

HW translates each logical address to a physical address on-the-fly at runtime.



relocation register enables program to be loaded at any location in main memory. Program can be moved as well, if needed.

Multistep Processing of a User Program



Symbolic names for functions and variables

relocatable code. addresses are relative to the module (object file).

Linking

exe file

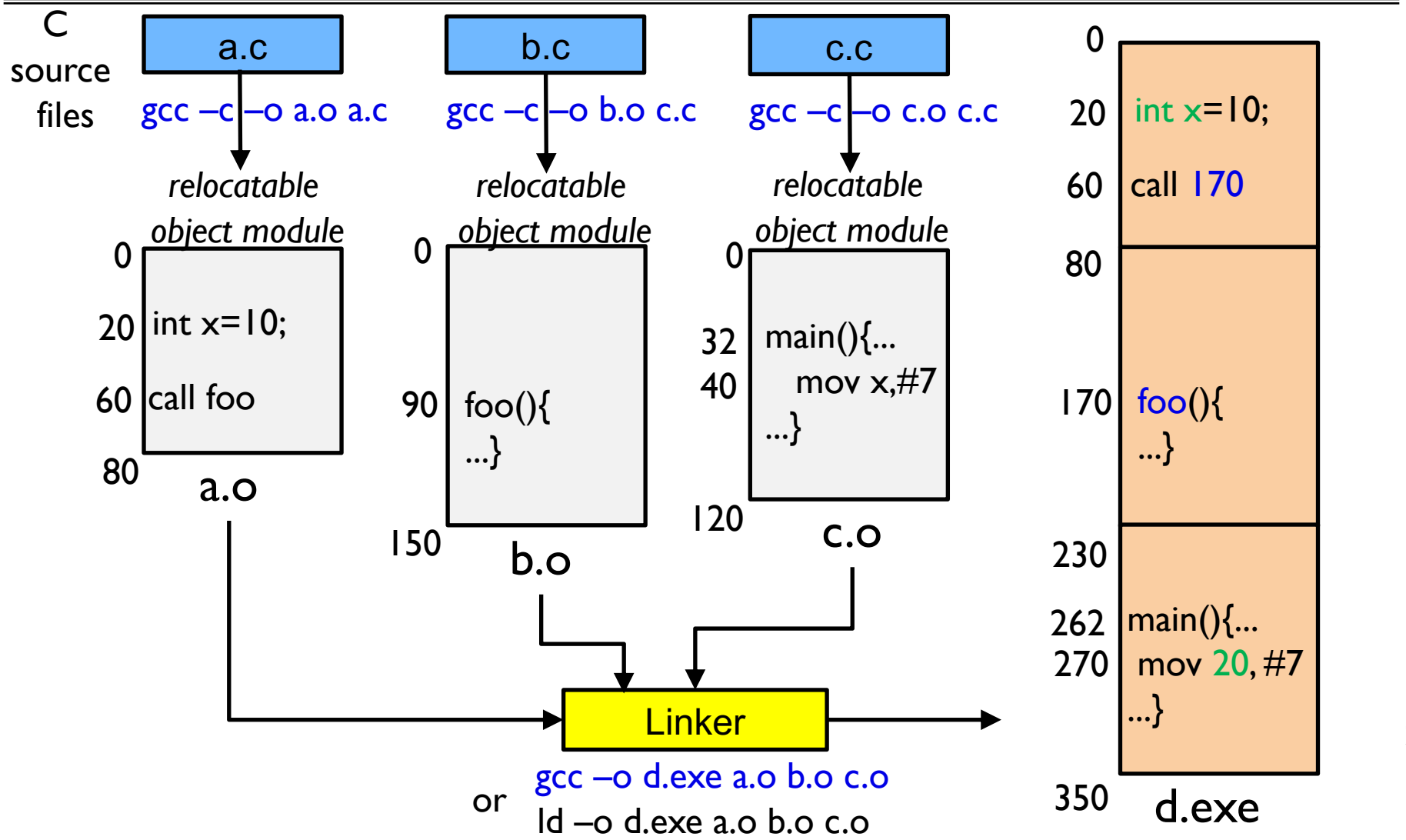
runtime absolute logical addresses (relative to 0)

Runtime logical addresses; translated to PAs at runtime

Addresses for program elements may be represented in different ways during these steps.

a tiny example for linking of multiple modules of a program

(not real code; pseudo-code - to illustrate the idea)



Dynamic Loading

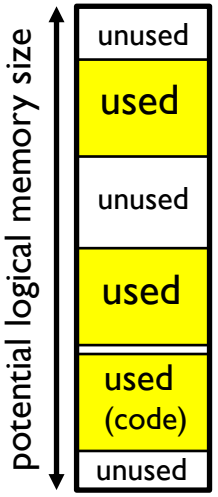
- Routine is not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required,
  - implemented through program design.

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for *libraries*.
  - Standard C library is shared library that is dynamically linked, not statically linked.
  - You can link statically if you want. (gcc -static ....)
- Also known as shared libraries.

Basic Memory Allocation Strategies

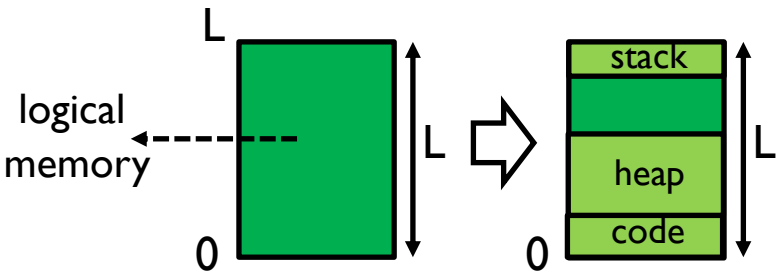
- We will see three basic main memory allocation strategies
  - 1) Contiguous allocation (not much used)
  - 2) Segmentation (used in some system)
  - 3) Paging (heavily used in modern systems)
- In this chapter we will assume that all the *used* logical memory content of a process *must be* in main memory for the process to run (no partial loading of used regions).
  - note that a program may not use all its potential logical memory. a portion of it may be used. For example, logical memory in a 32 bit system can be as big as 4 GB.
- *In the next chapter (virtual memory), we will relax this assumption (partial loading will be possible to run).*

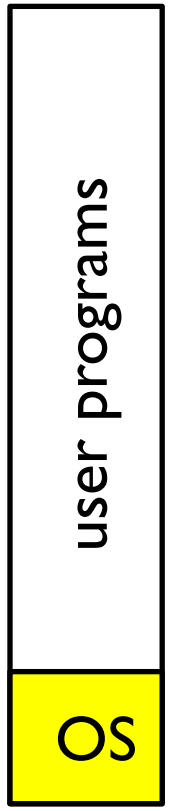


Contiguous Memory Allocation

Contiguous Allocation

- Main memory is partitioned usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory
- OS manages the memory space for user programs.
- Processes are using logical memory. Assume size = L (may need to guess). Logical address range: [0,L).



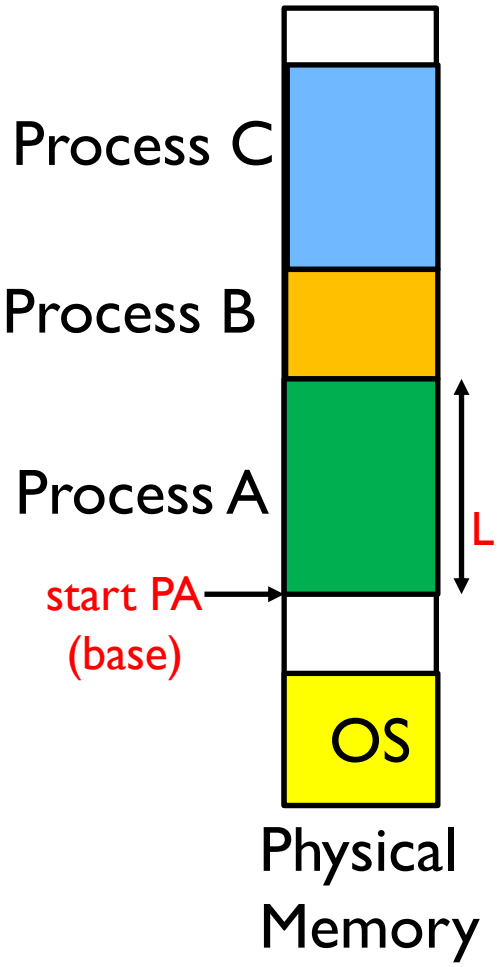


Physical Memory

We assume logical memory of a process fits into physical memory

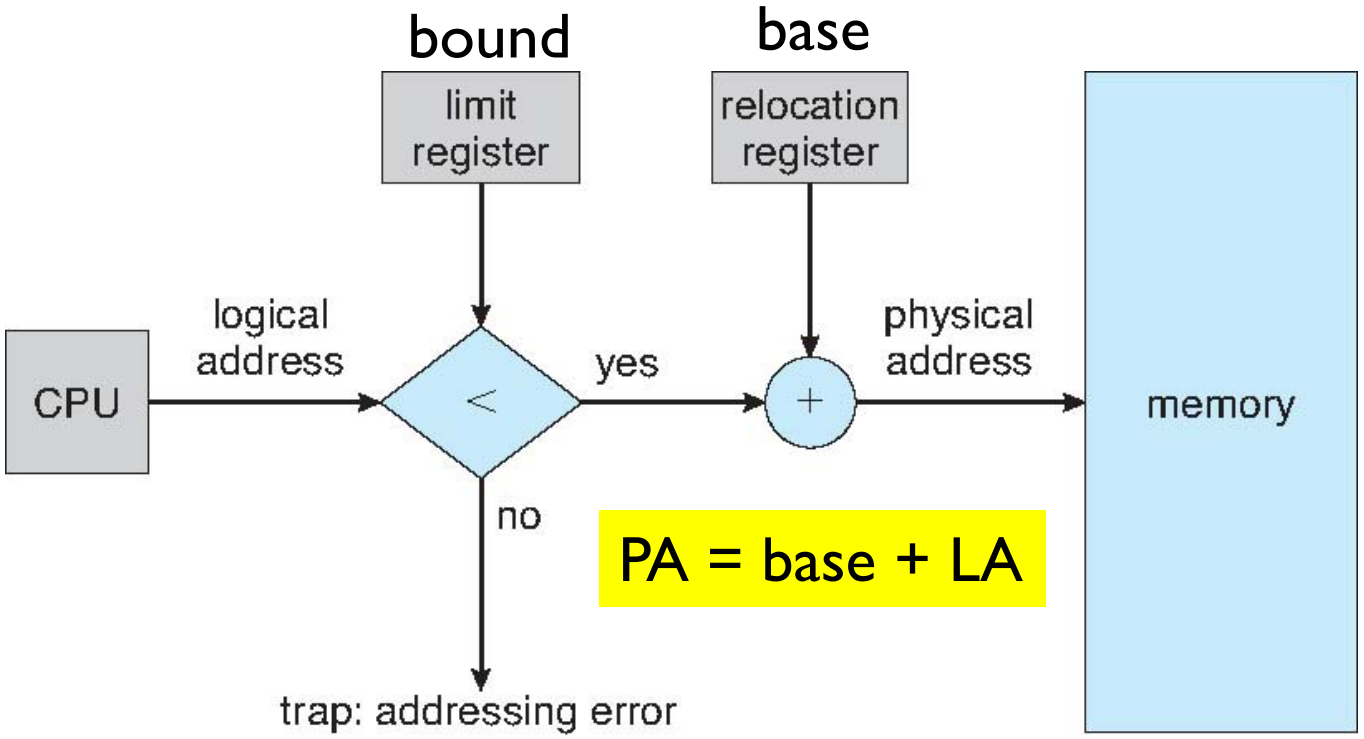
Contiguous Allocation

- A process is allocated a contiguous region of physical memory (of size L) by OS.
- OS loads the program from executable file into allocated space.
- PCB of the process includes the start PA and size (L).
- When process is scheduled, *start PA* and *size* loaded into relocation (base) and limit registers of CPU.
  - relocation (base) register contains smallest PA.
  - limit (bound) register contains range of logical addresses.



HW support for contigous allocation

in terms of relocation and limit registers

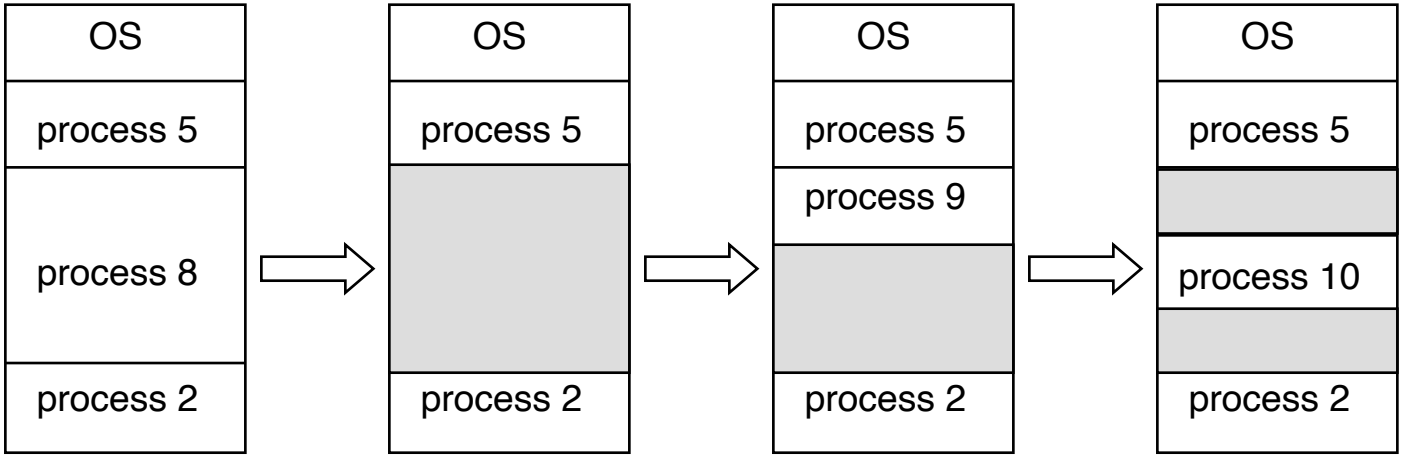


Hardware (MMU) translates logical addresses to physical addresses dynamically at run time.
Also checks the addresses (protection).

In this way, protection and relocation provided.

Contiguous Allocation

- Memory is partitioned to programs, in a contigous manner.
  - Each partition contains a program or is free (hole).
  - Hole – block of available (free) memory; holes of various size are scattered throughout memory after some time.



Contiguous Allocation

- Operating system maintains information about:
  a) allocated partitions (in PCBs)
  b) free partitions (holes)
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- How to satisfy a request of size n from a list of free holes?
  - This is dynamic storage allocation problem.
  - Various strategies exists.

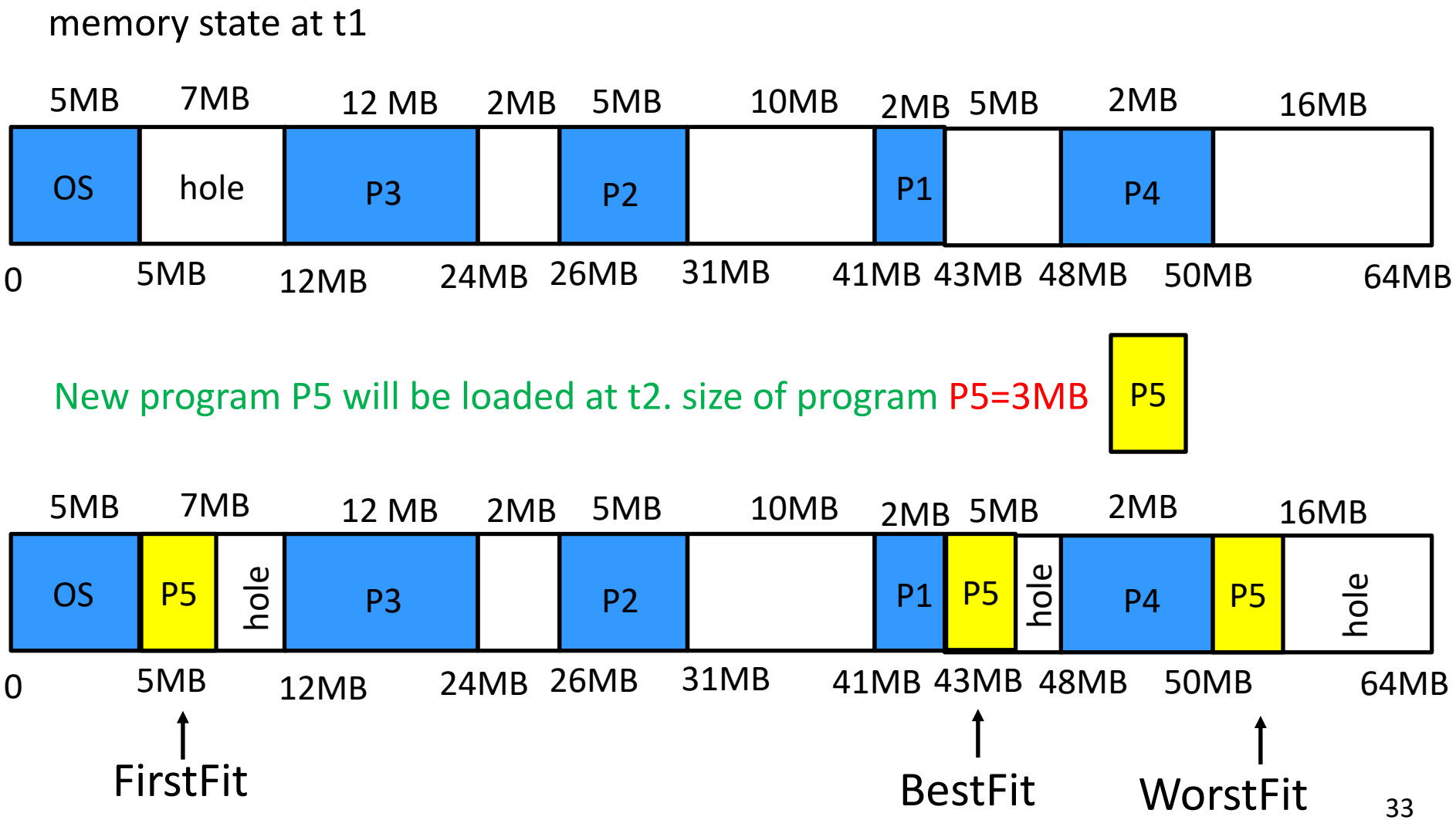
Dynamic Storage-Allocation Problem

Searching and selecting a free block (hole) for request of size n

- First-fit: allocate the *first* hole that is big enough.
- Next-fit: use first-fit, but next time continue from where the previous search stopped.
- Best-fit: allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size.
  - Produces the smallest leftover hole.
- Worst-fit: allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

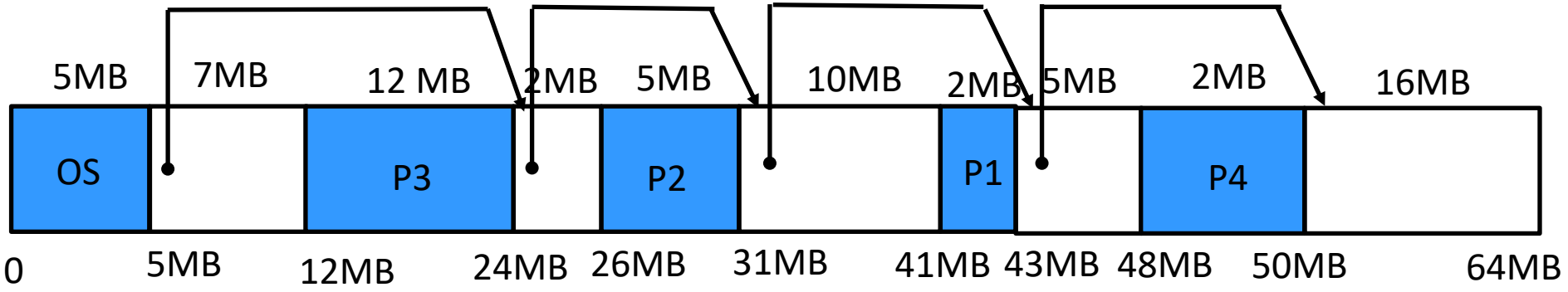
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Small example: where a new program will be loaded with FirstFit, BestFit, WorstFit?



Small example: keeping track of holes.

Use holes (free blocks) themselves. No need for another structure.



each hole will have such a header at the beginning.

```
struct  hole_hdr {
        struct hole_hdr *next_hole; // pointer to the next hole
        struct hole_hdr *prev_hole; // pointer to the prev hole
        int size;                   // size of this hole
};
```

we can have a singly or doubly linked list of holes.

Fragmentation

- Contigous allocation causes external fragmentation.
- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation – allocated memory may be slightly larger than the requested memory; this size difference is memory internal to a partition (allocation), but not being used.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block

Segmentation

Segmentation

- Memory-management scheme that supports user's/programmer's view of memory
- A program is a collection of segments (logical contiguous units)
  - A segment is a logical program unit such as code part, or heap part of the program.
  - Course-grained segments: data segment, code segment, stack segment, heap segment. A few segments.
  - Fine grained segments: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays, … In this case, hundreds of segments may exist in a program.

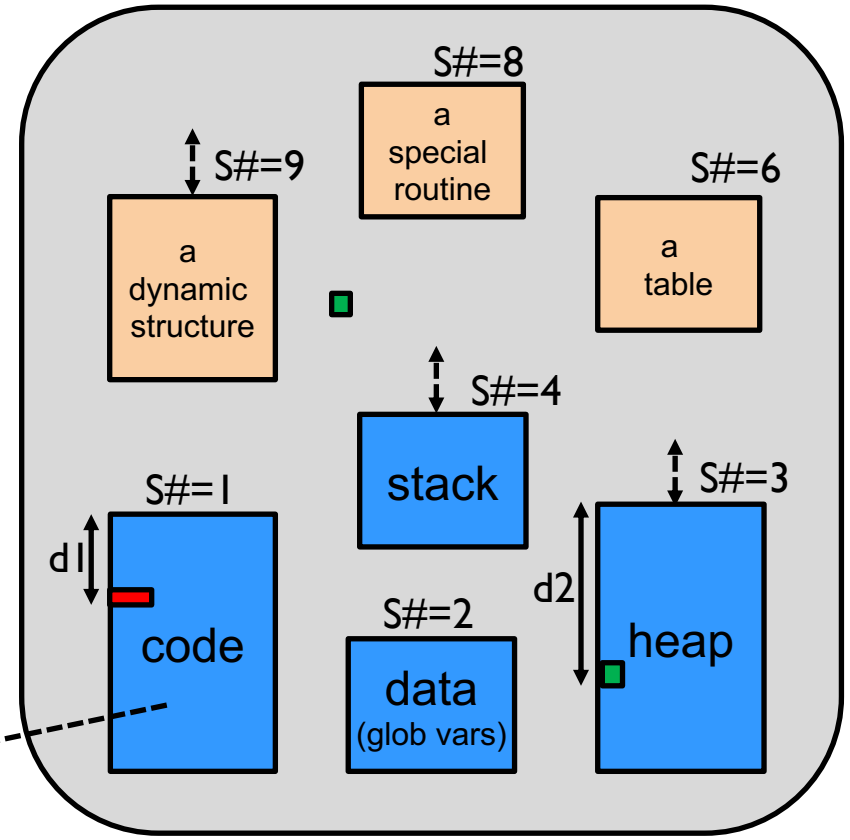
– Segments are of variable size.

User's View of a Program

Which segments will a program have is *decided* by the programmer or compiler.

instruction address consists of:
1) segment number (#1)
2) displacement inside segment (d1)
LA = <1, d1>

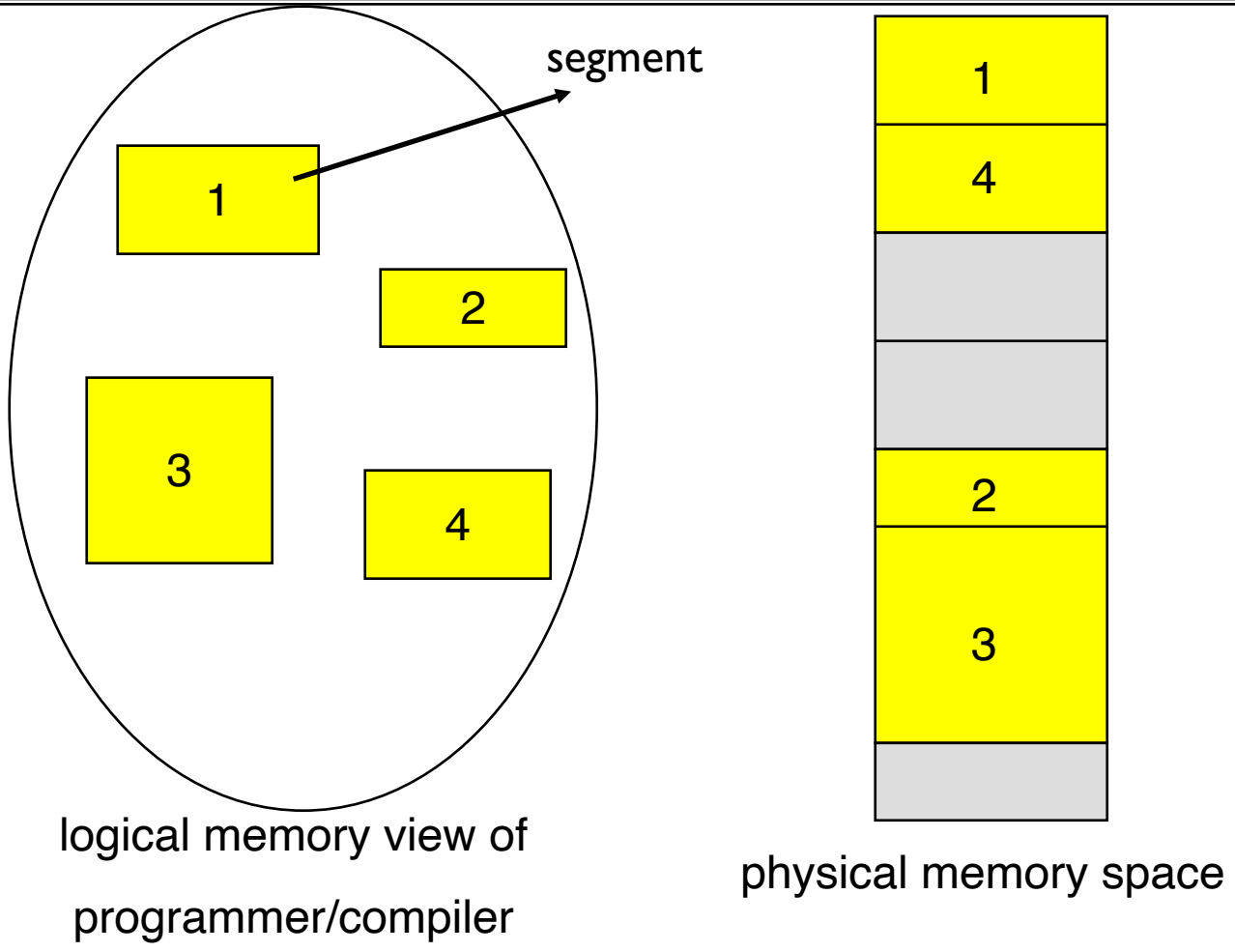
object address consists of:
1) segment number (#3)
2) displacement inside segment (d2)
LA = <3,d2>



logical address space (logical memory)

segments (of variable size). some static. some may be dynamic.

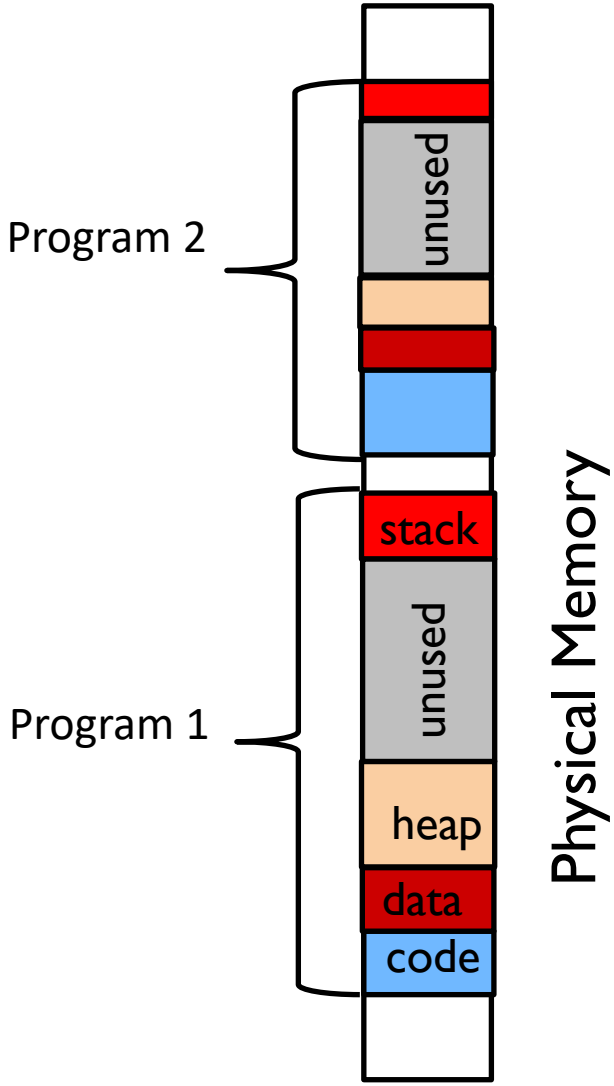
Logical View of Segmentation



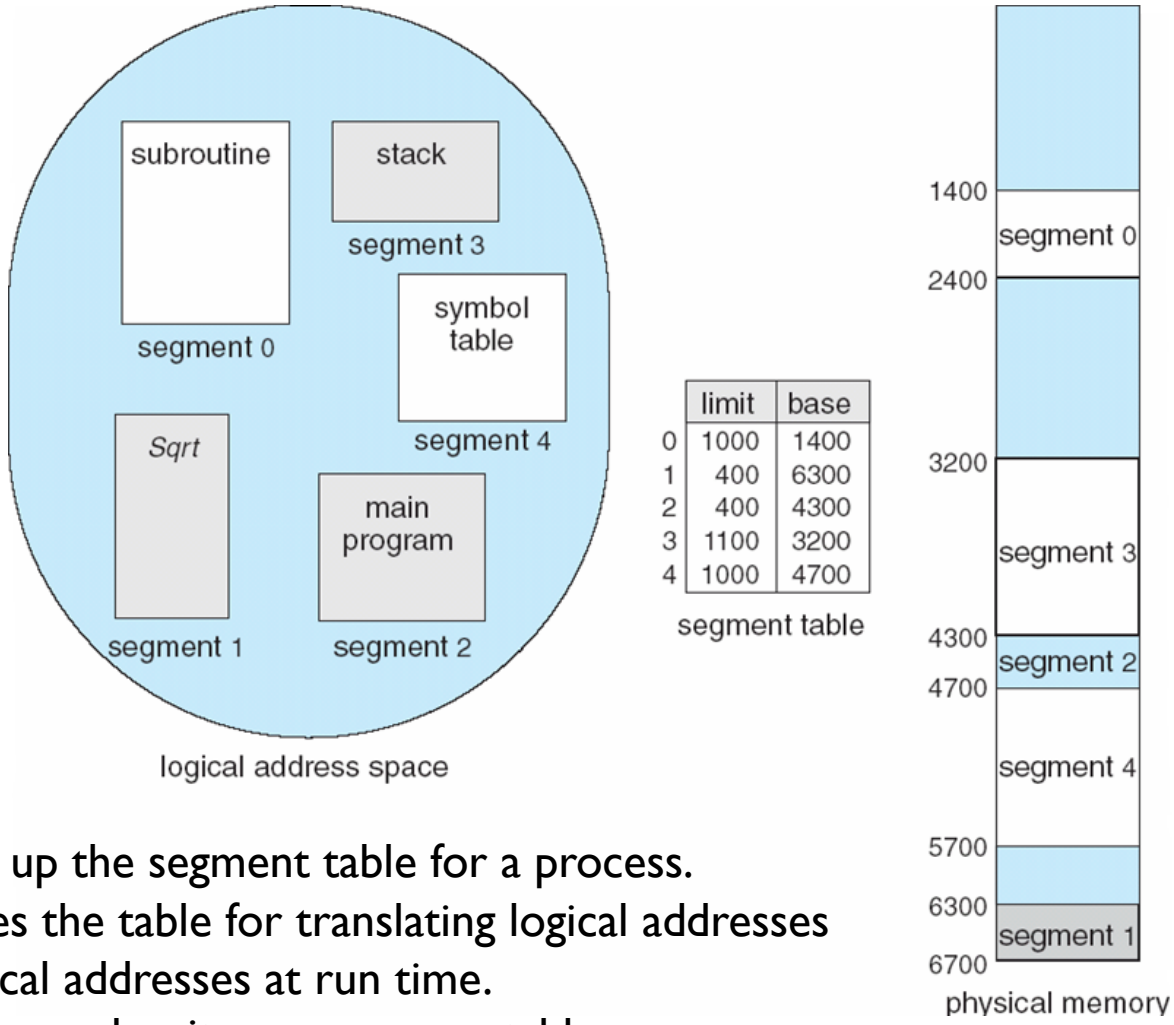
OS needs to allocate contigous memory to each segment.
Segments do not have to be next to each other.

Contiguous Allocation versus Segmentation

- With contiguous allocation, large amount of physical memory may stay unused due sparse logical address space.
- With segmentation, we don't have this problem.
- Additionally, segments are of smaller in size; easier to find an empty hole.



Example of Segmentation



OS sets up the segment table for a process.
HW uses the table for translating logical addresses to physical addresses at run time.
Each process has its own segmen table

Segmentation Architecture

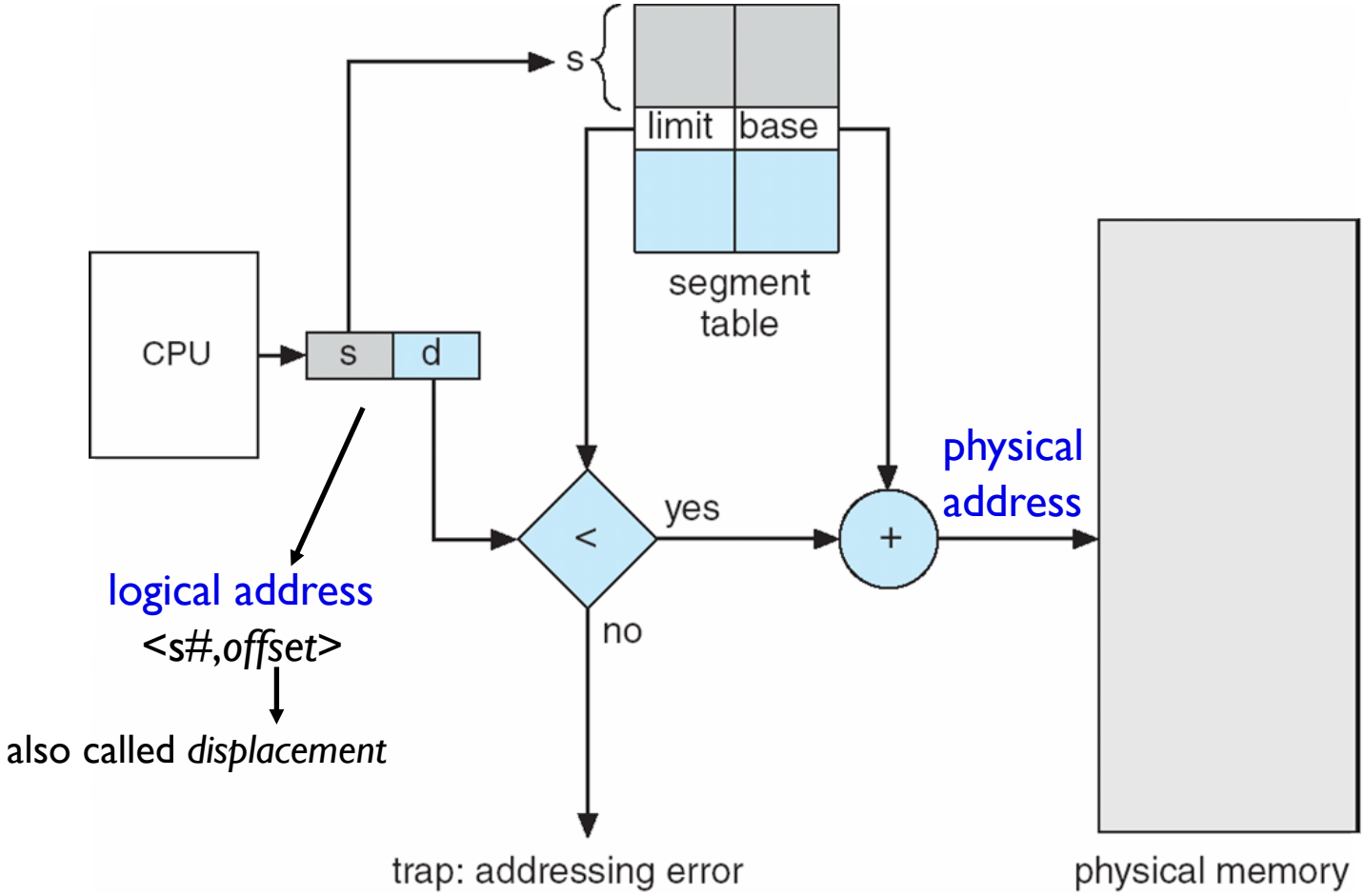
- Logical address is a two tuple (it is 2D):

  <segment-number, offset>

- Segment table: each entry has base and limit register values. Using them, table maps two-dimensional LAs to one-dimensional PAs.
  - base – contains the start PA of segment in memory.
  - limit – specifies the length of the segment.
- Segment-table base register (STBR) points to the segment table's location in memory.
- Segment-table length register (STLR) indicates number of segments used by a program.

  segment number s is legal if s < STLR

Segmentation Hardware address translation



HW maps 2D logical addresses to 1D physical addresses.

Segmentation Architecture

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
  - Code segment: READ-ONLY; sharable; …
  - Data segment: READ-WRITE; not-sharable
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - (similar to contiguous allocation)

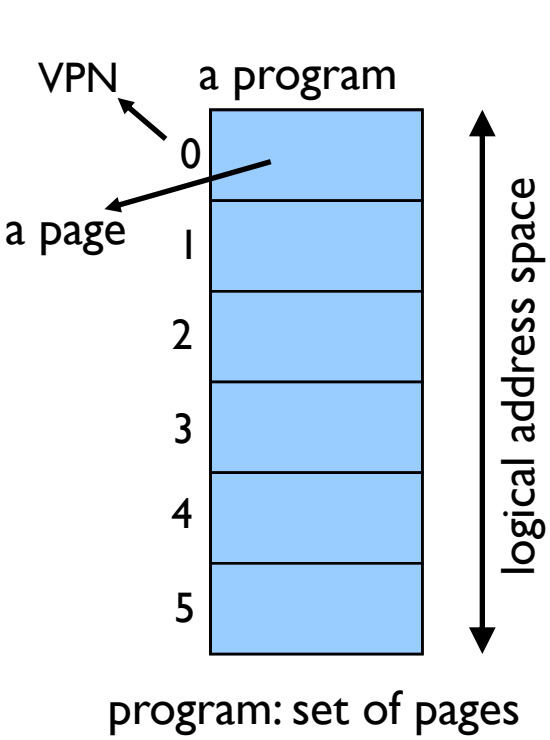
- A

Paging

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, a typical size: 4096)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size **n** pages, need to find n free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

Paging



program: set of pages

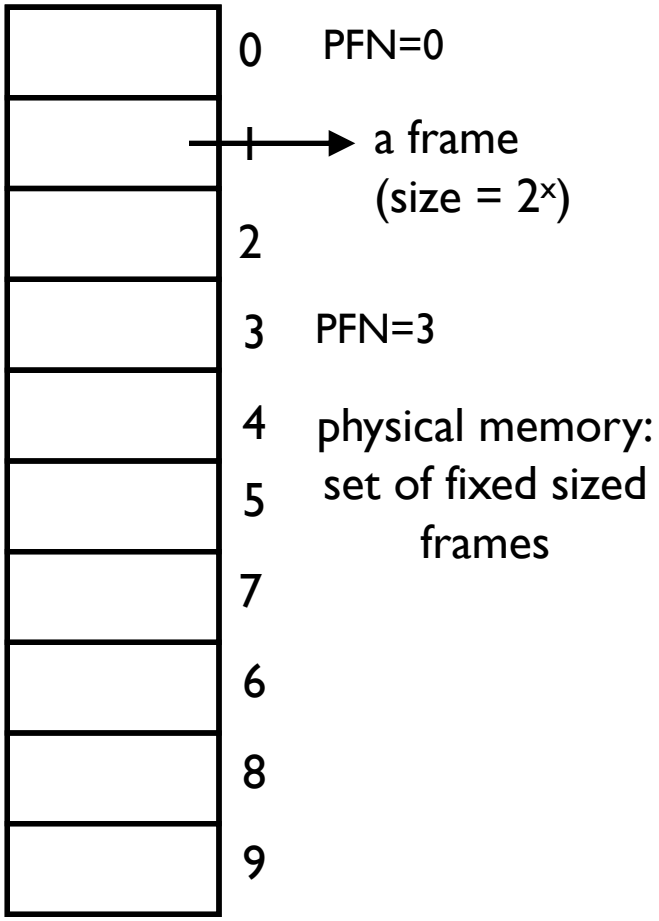
VPN: virtual page number

PFN: physical frame number

Page size = Frame size

typical size:
4 KB

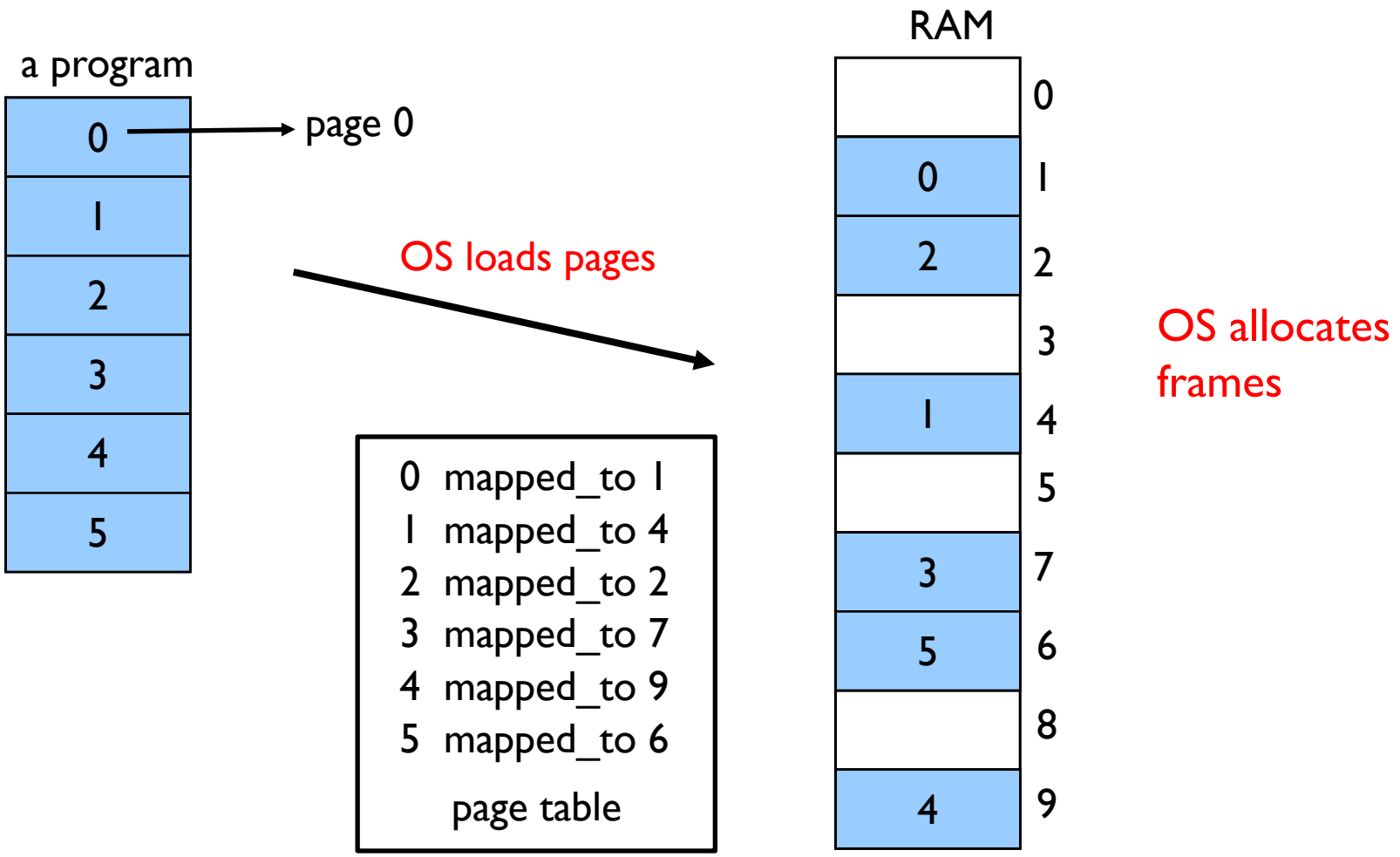
RAM (Physical Memory)



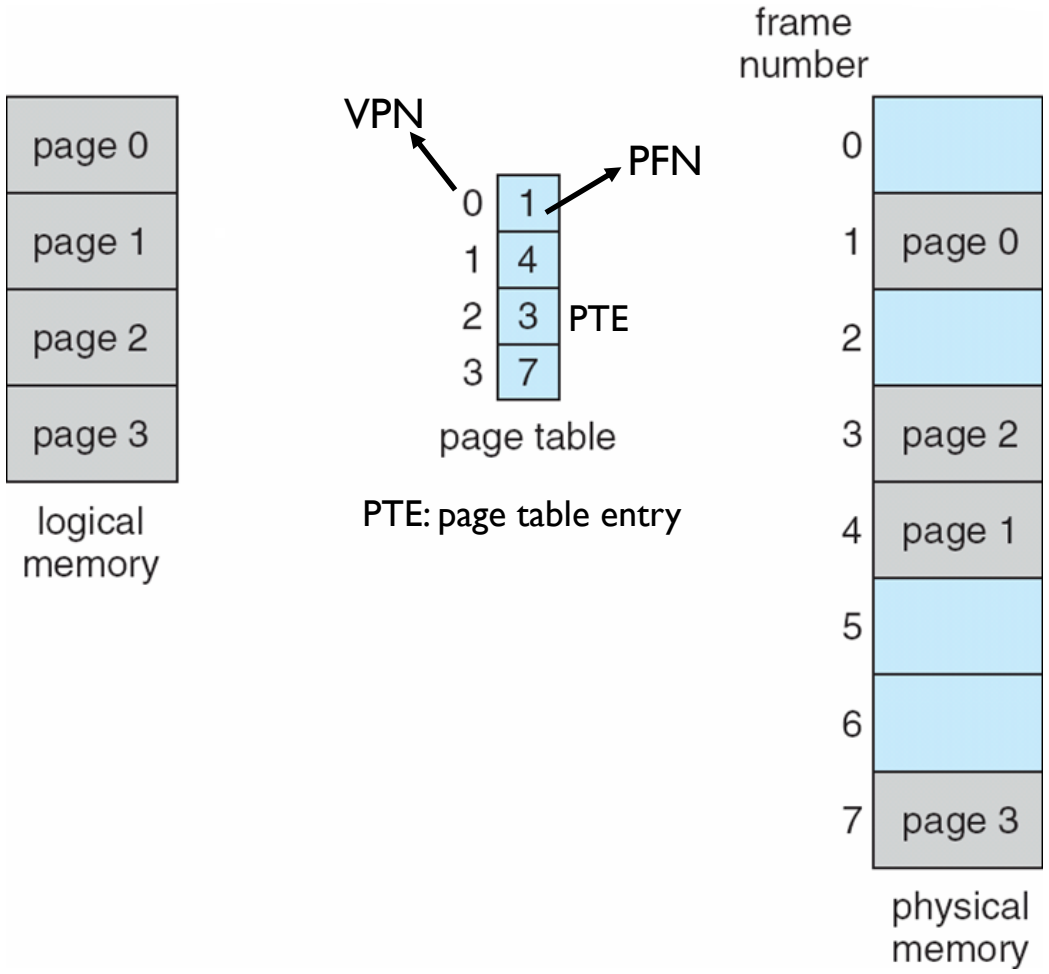
a page contains program content.

a frame is physical memory that can contain/store a page (it is a container).

Paging


a program
0
1
2
3
4
5
page 0
OS loads pages
0 mapped_to 1
1 mapped_to 4
2 mapped_to 2
3 mapped_to 7
4 mapped_to 9
5 mapped_to 6
page table
RAM
0
1
2
3
4
5
6
7
8
9
0
2
1
3
5
4
OS allocates frames
OS sets up the page table

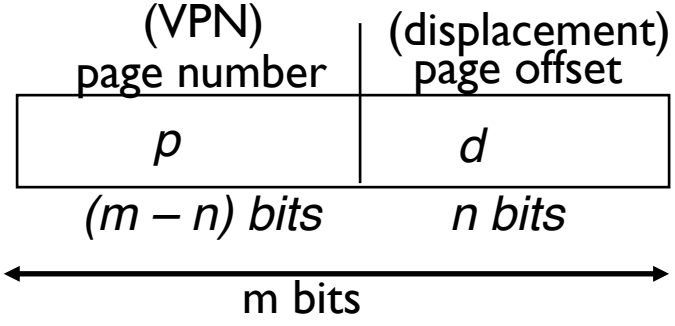
Example


page 0
page 1
page 2
page 3
logical memory
VPN
PFN
0 1
1 4
2 3
3 7
PTE
page table
PTE: page table entry
frame number
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

Address Translation Scheme

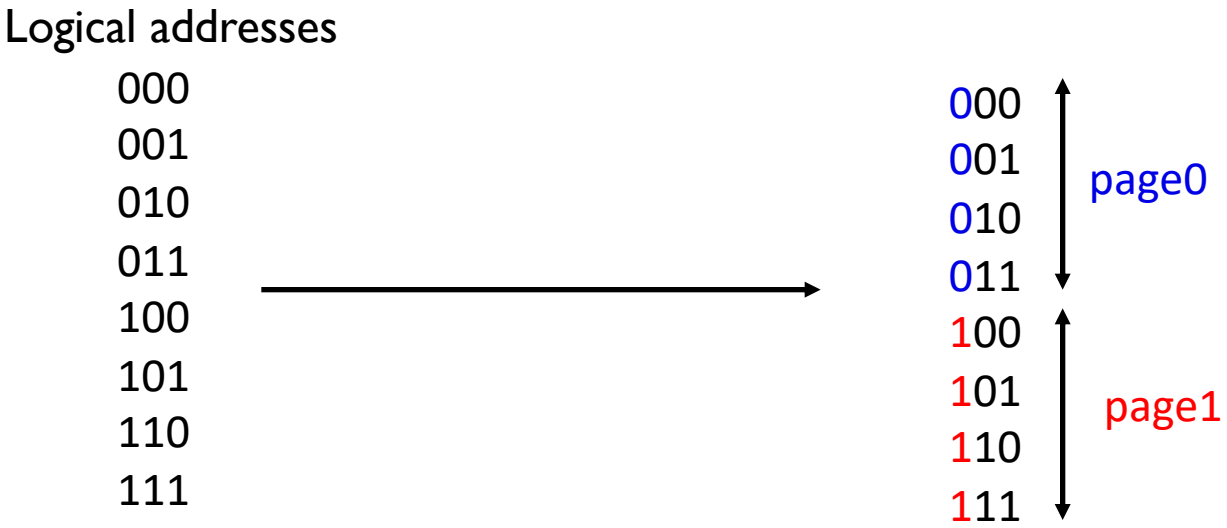
- Assume logical addresses are m bits. Then logical address space is 2^m bytes.
- Assume page size is 2^n bytes.

- Logical Address generated by CPU is divided into:
  - Page number (p) – (also called VPN) used as an index into a page table which contains base address (PFN) of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

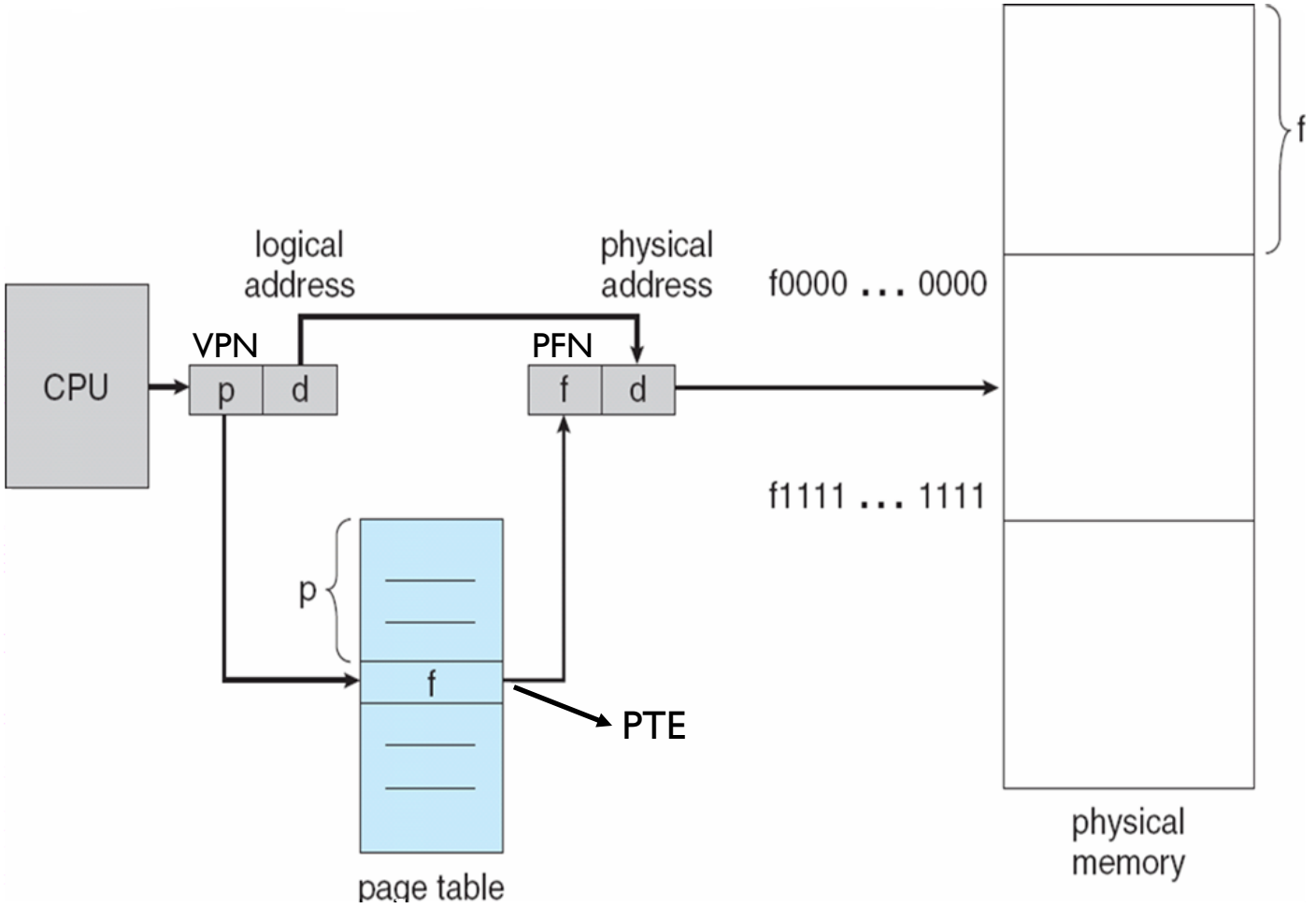


Simple example

Assume m is 3 and n is 2



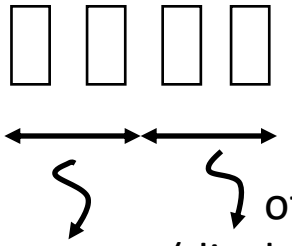
Paging Hardware: address translation

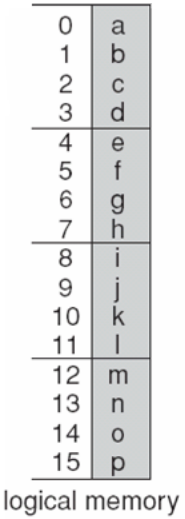


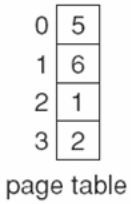
Paging and Address Translation Example

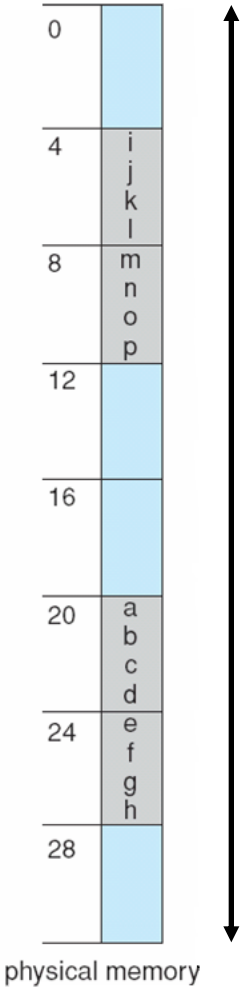
page size = 4 bytes
= 2^2

4 bit logical address









32 byte memory

LA = 5
PA = ?

5 is 0101
PA = 11001

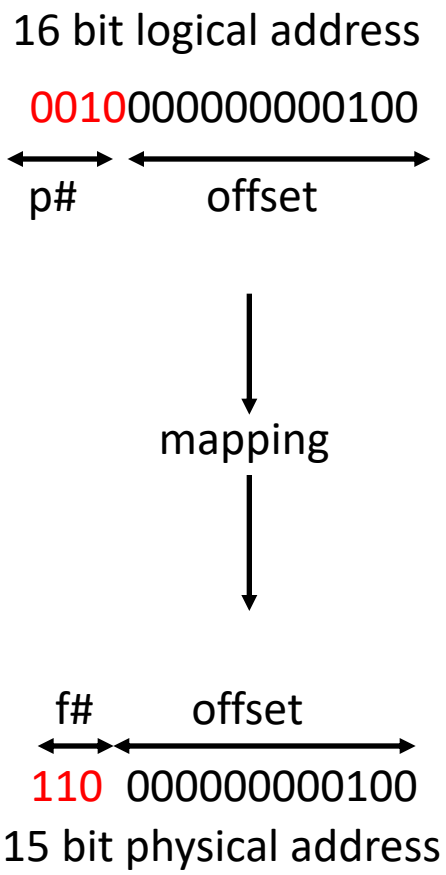
LA = 11
PA = ?

11 is 1011
PA = 00111

LA = 13
PA = ?

13 is 1101
PA = 01001

Address translation example 2



15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1
0	010	1

page table

frame number

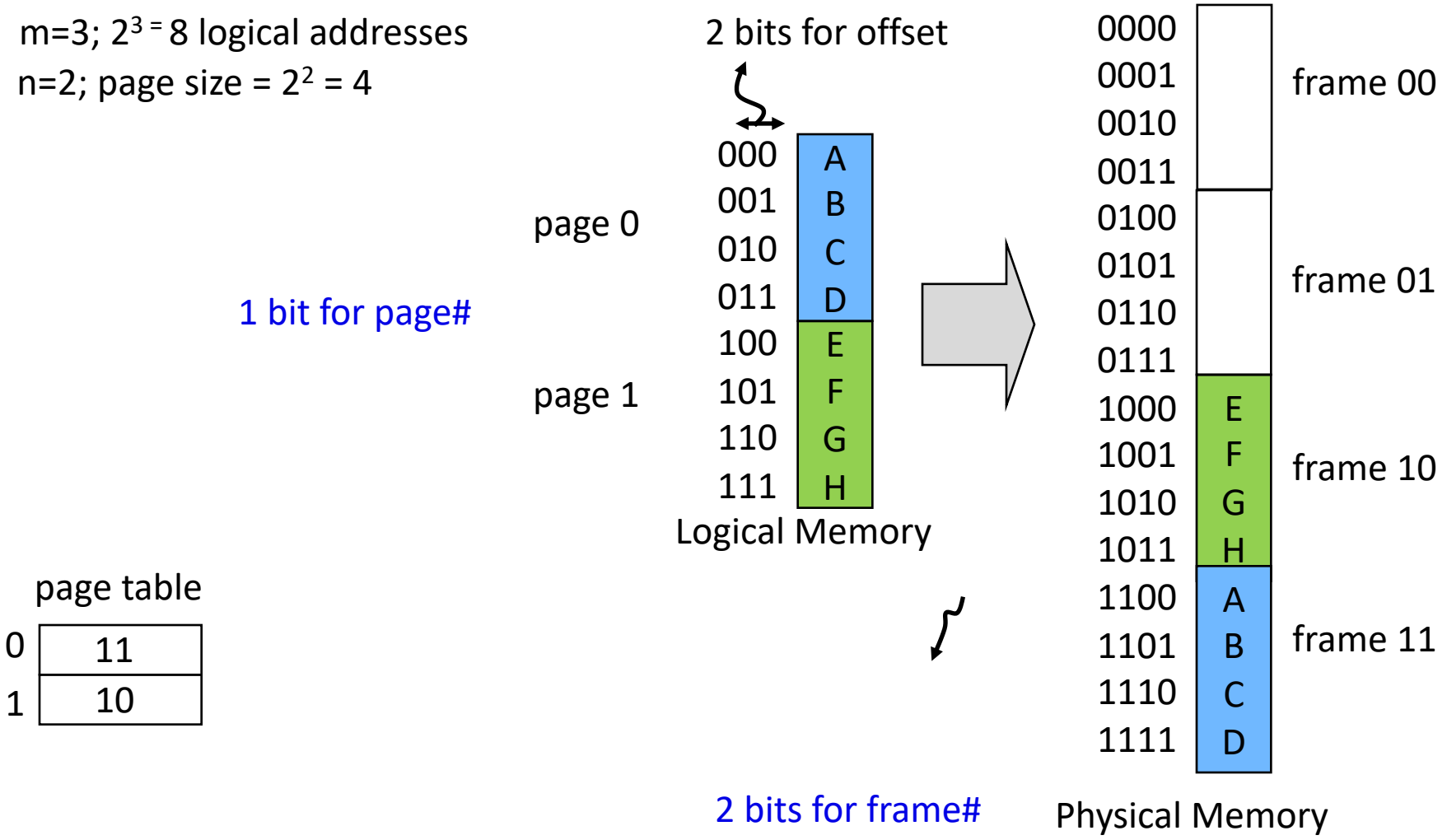
valid/invalid bit

page size = 4096 bytes
(offset is 12 bits)

Address translation example 3

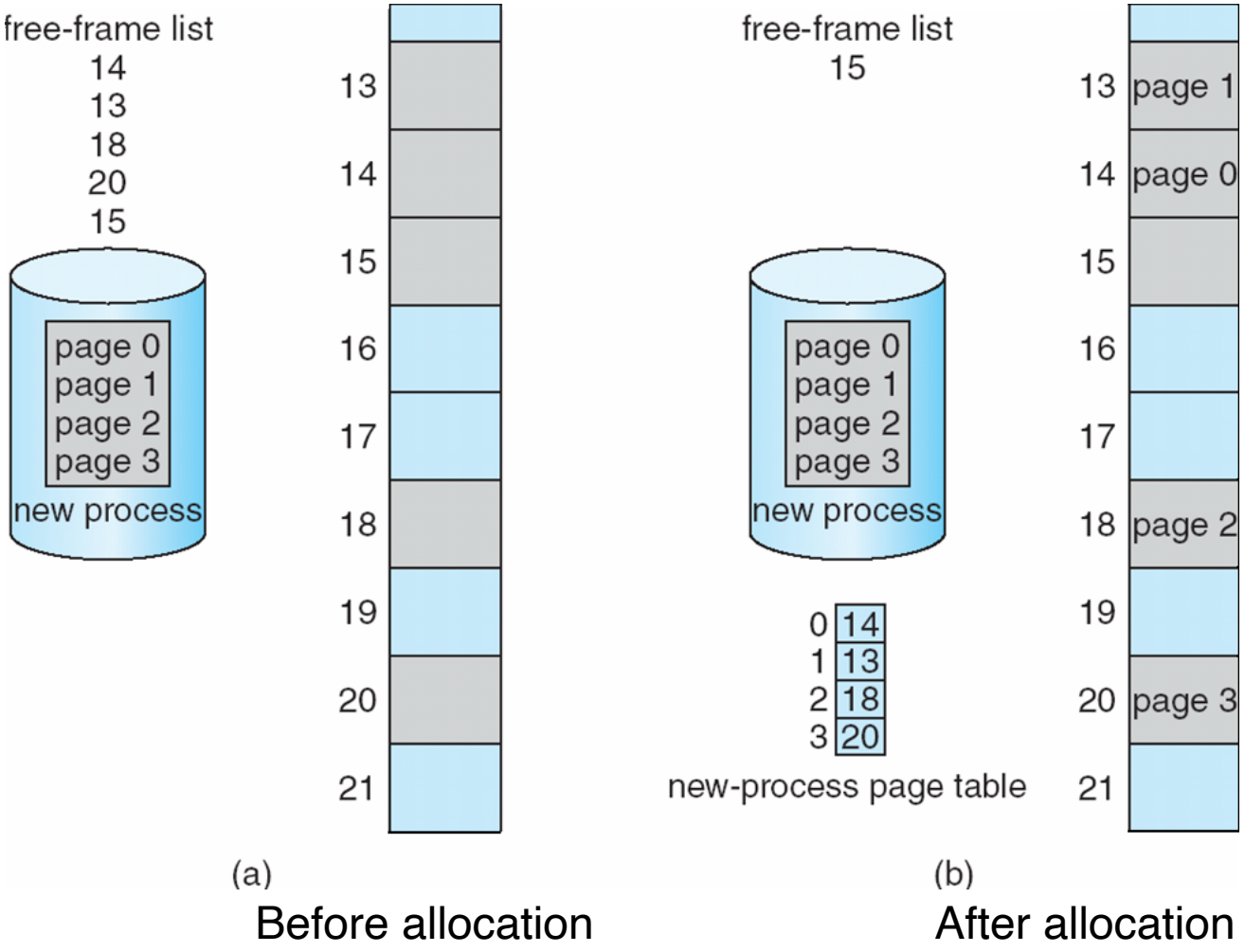
m=3; $2^3 = 8$ logical addresses

n=2; page size = $2^2 = 4$



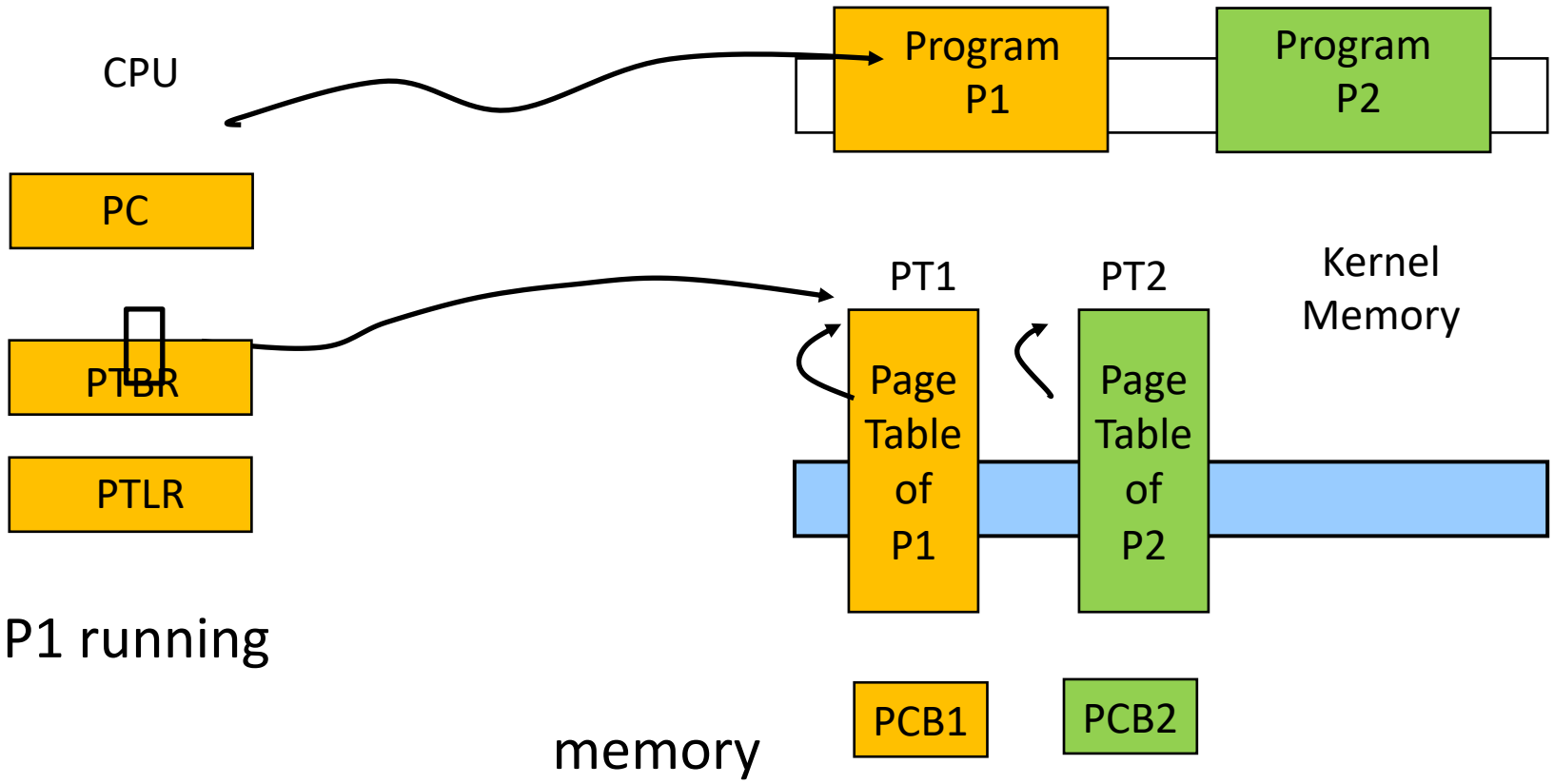
Free Frames

OS also keeps info about the frames in its frame table



Implementation of Page Table

- Page table is kept in main memory.
- Page-table base register (PTBR) points to the page table.
- Page-table length register (PTLR) indicates size of the page table.

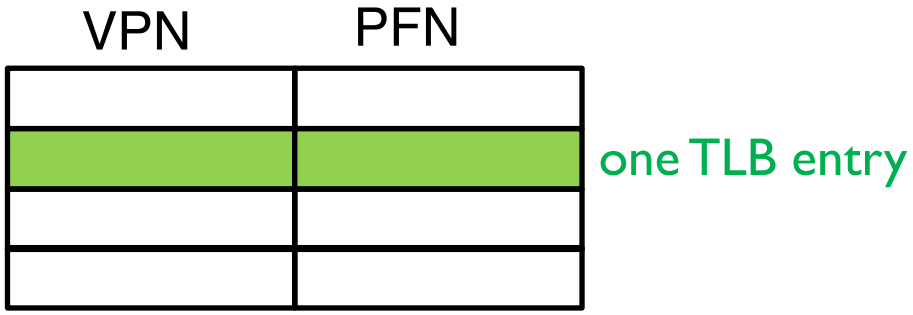


TLB

- With paging, every data/instruction access requires two memory accesses.
    - One for the page table
    - One for the data/instruction.
- Solve it by using of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
- Learned (p#, f#) mappings can be cached in TLB.
  - TLB: a set of entries each entry storing a different mapping.
  - Number of entries limited (~1024, for example) (TLB size)
  - Can be accessed very fast.

TLB Associative Memory

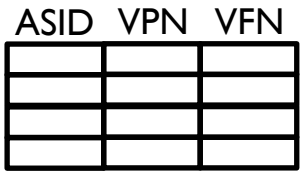
- Associative memory – parallel search



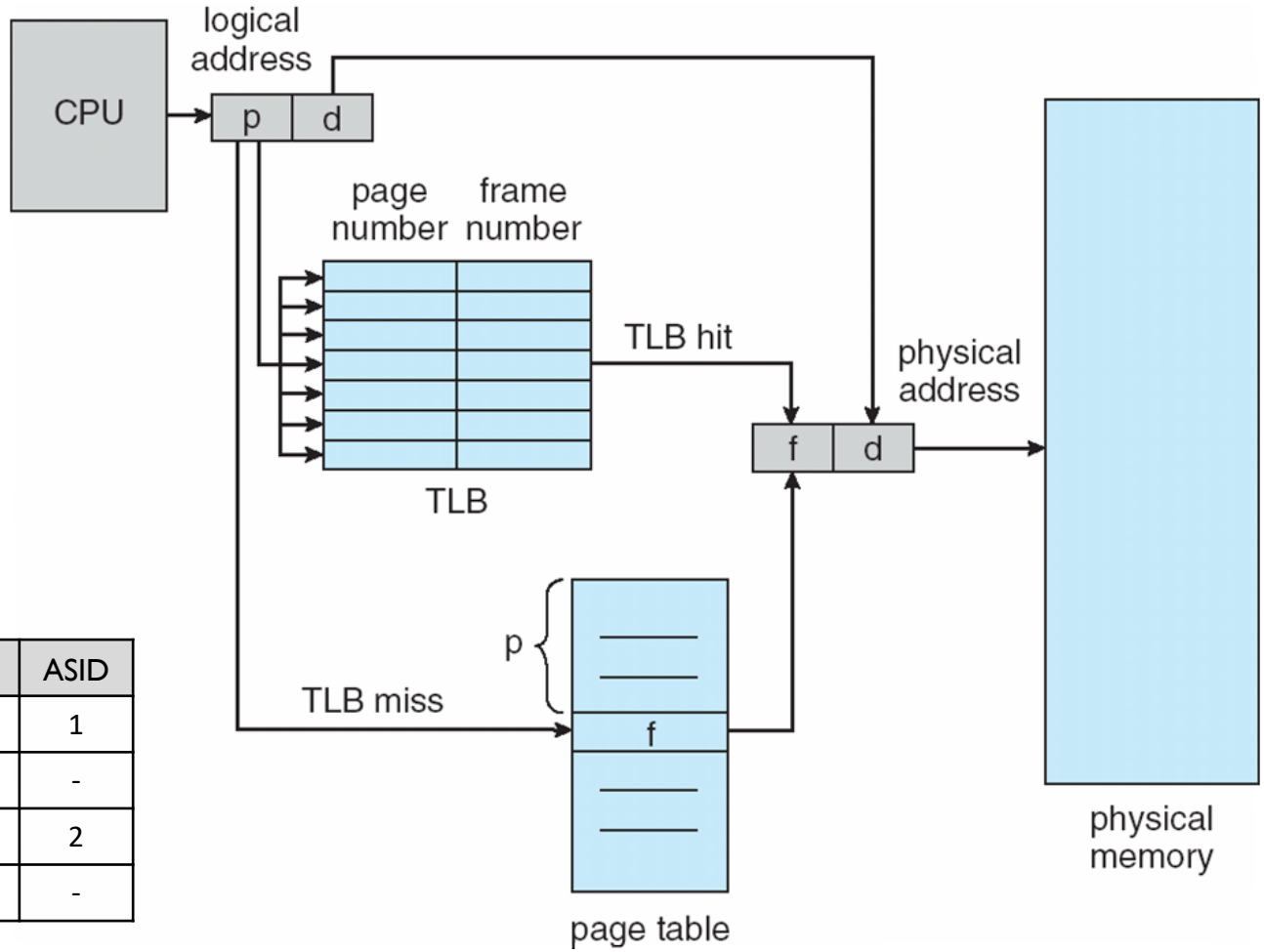
Address translation (p, d)

- If p is in TLB, get frame # out
- Otherwise get frame # from page table in memory

Some TLBs store *address-space identifiers* (ASIDs) in each TLB entry – uniquely identifies *each process* to provide address-space protection for that process.



Paging Hardware With TLB



Example TLB

VPN	PFN	valid	prot	ASID
10	100	1	rw	1
-	-	0	-	-
10	170	1	rw	2
-	-	0	-	-

Effective Memory Access Time

- TLB Lookup = ε time unit
- Assume memory access time is 1 time unit.
- Hit ratio – percentage of times that a page number is found in the TLB; ratio related to the TLB size.
- Hit ratio = α
- Effective (average) Access Time (EAT) to Memory

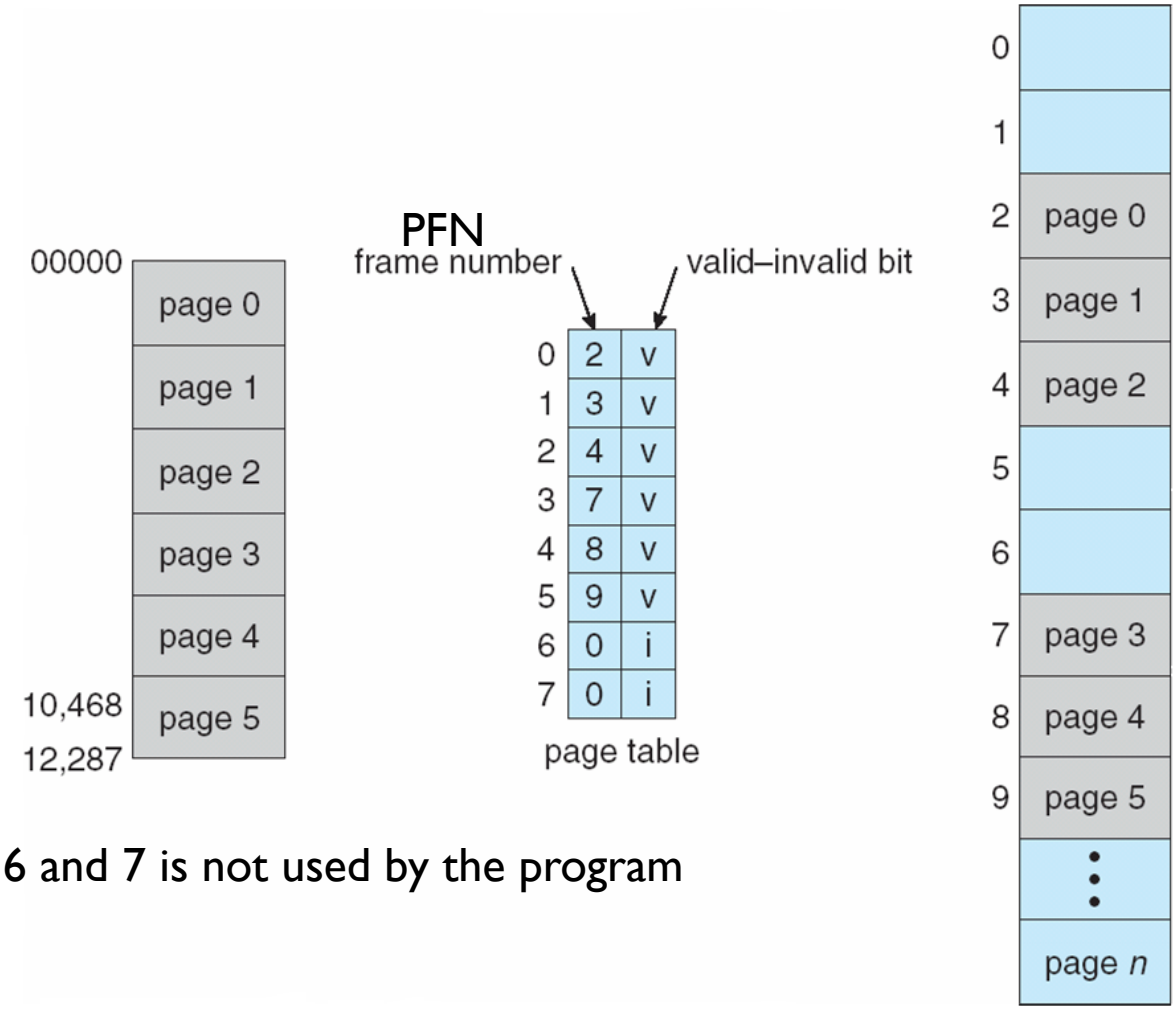
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Memory Protection

- Memory protection implemented by associating a protection bits with each page
  - Read only page
  - Executable page
  - Read-write page
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

Valid (v) or Invalid (i) Bit In A Page Table

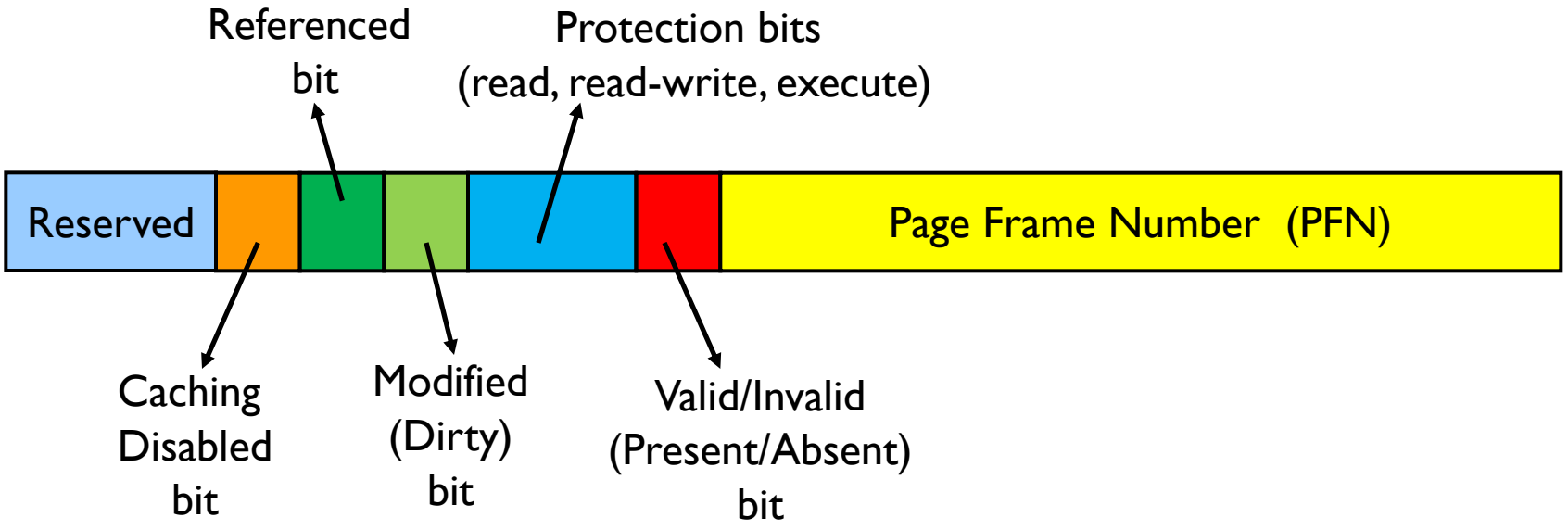


page 6 and 7 is not used by the program

accessing page 6 or 7 generates memory access error (segmentation fault).

Page Table Entry (PTE) Structure

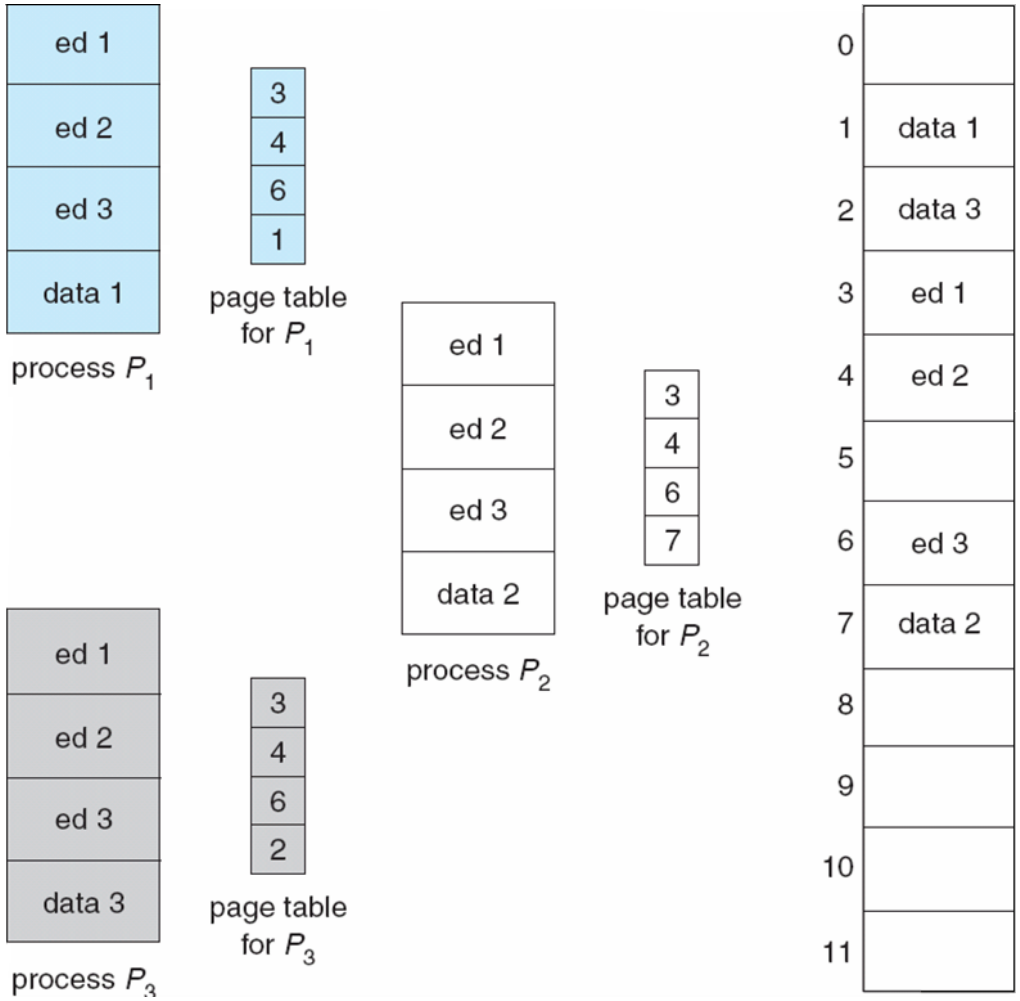
- A typical size of a page table entry can be 32 bits (in a 32-bit system). Depends on the architecture.
- Typically we have the following fields in a page table entry.



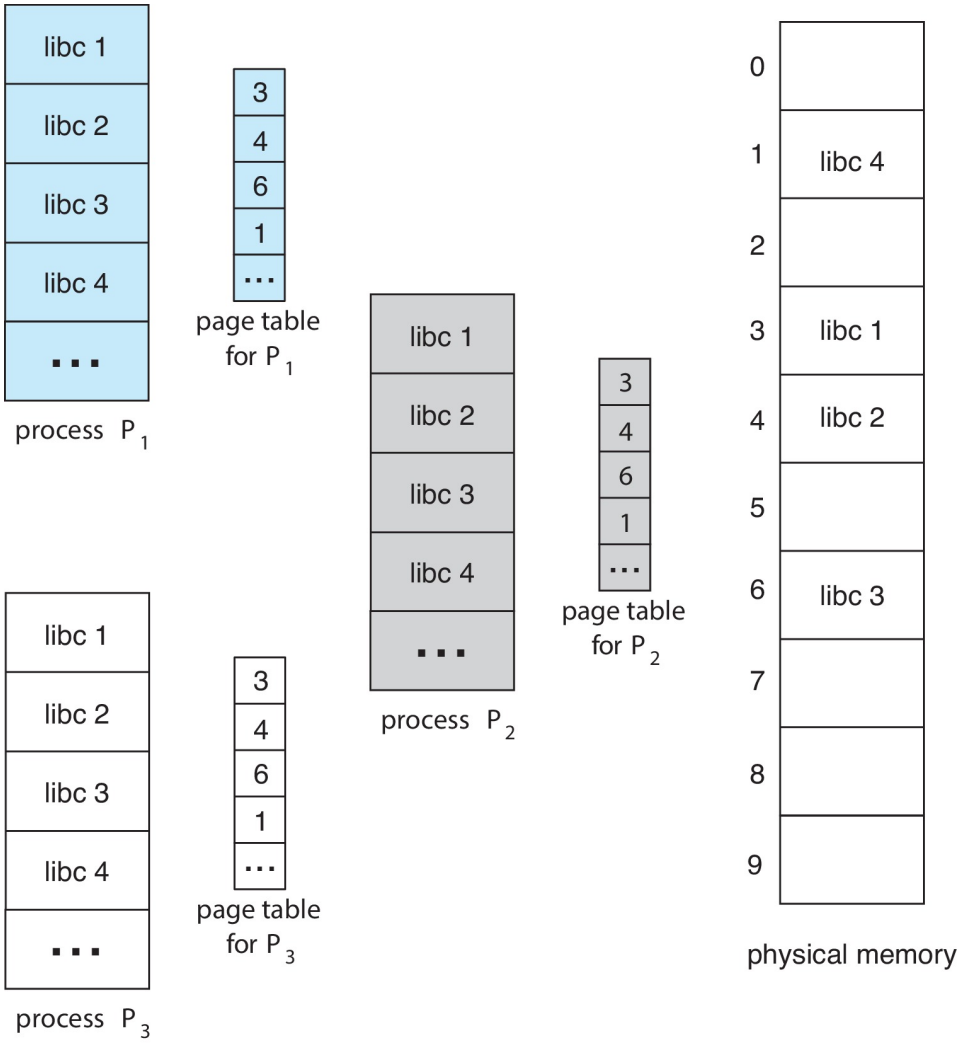
Shared Pages

- A program can be started multiple times (same program executed by multiple process)
  - Code pages can be shared
  - Shared code
    - One copy of read-only (reentrant) code shared among processes (can be used for sharing run-time libraries, etc., as well).
  - Private data
    - Each process keeps a separate copy of the data

Shared Pages Example

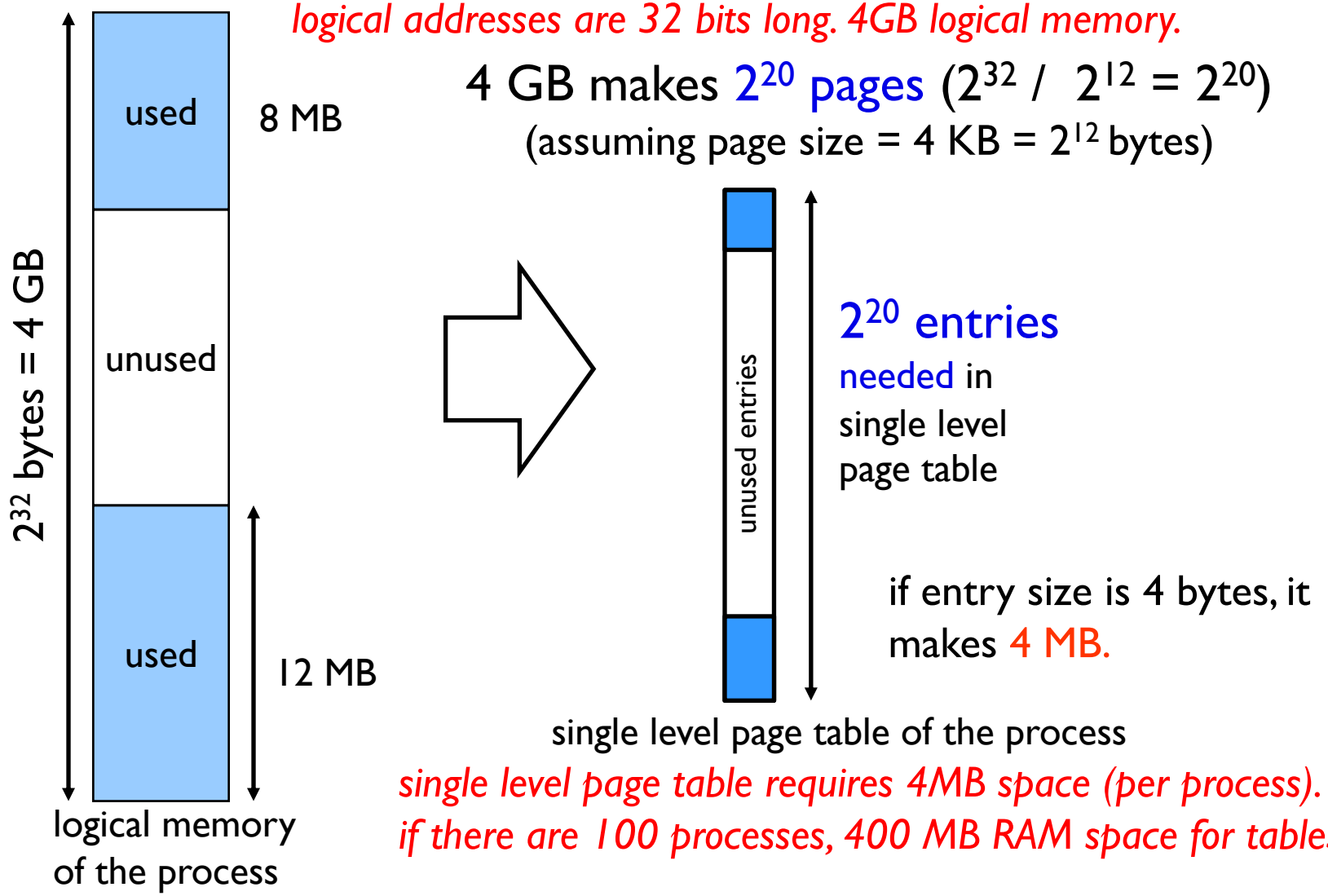


Shared Pages Example: sharing a library: libc



Structure of the Page Table

single level paging may not work for large address spaces. example: 32 bit system.

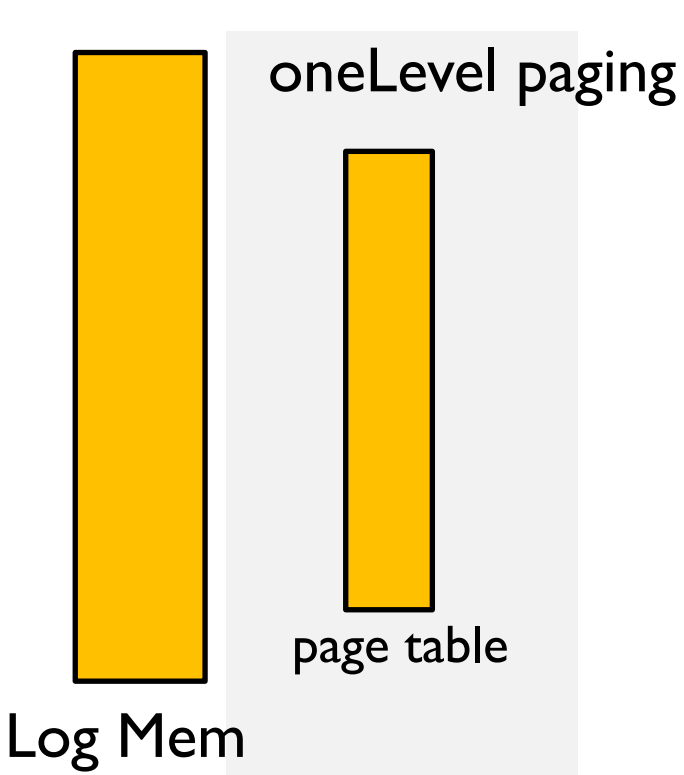


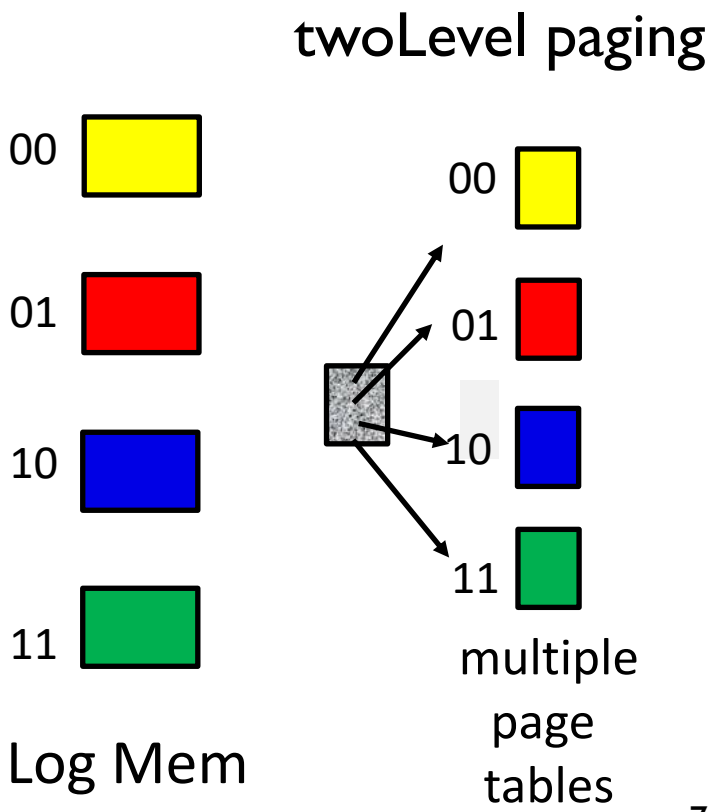
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

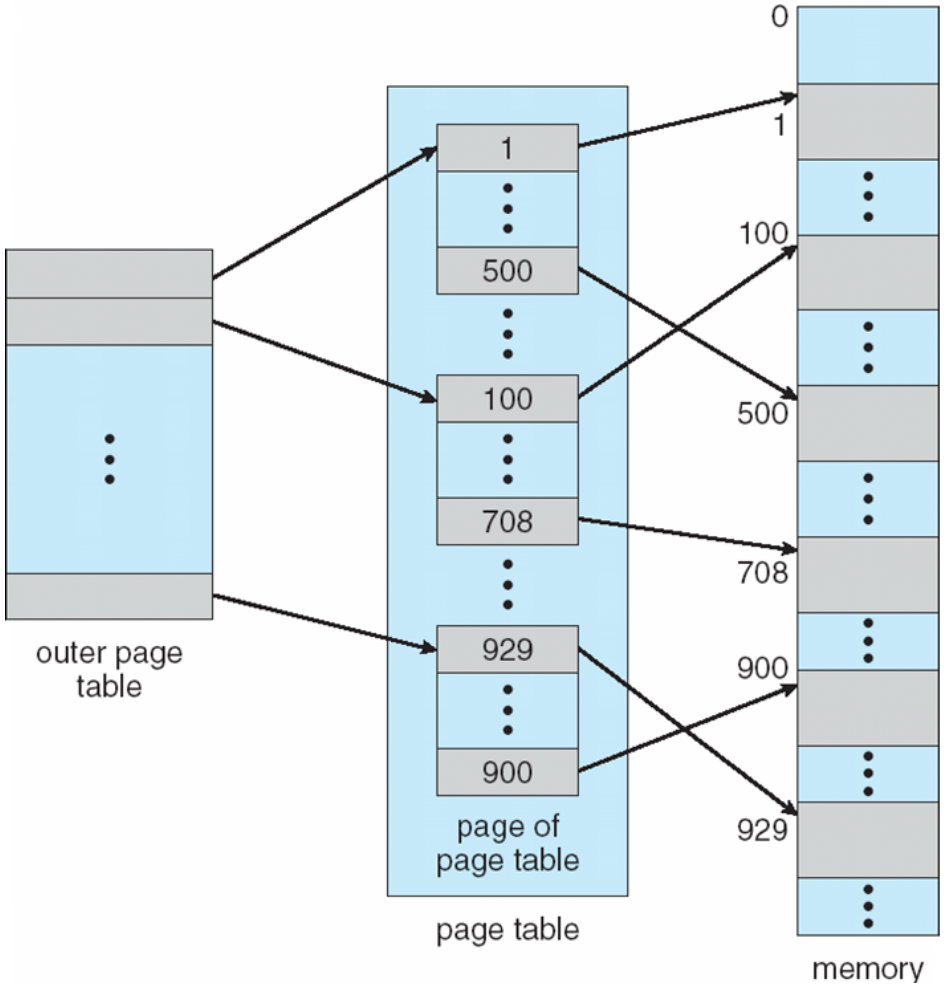
Hierarchical Page Tables

- Break up the logical address space into subspaces. Break up the page table into sub-tables. A sub-table for a sub-space.
- Two-level paging, is the simplest.

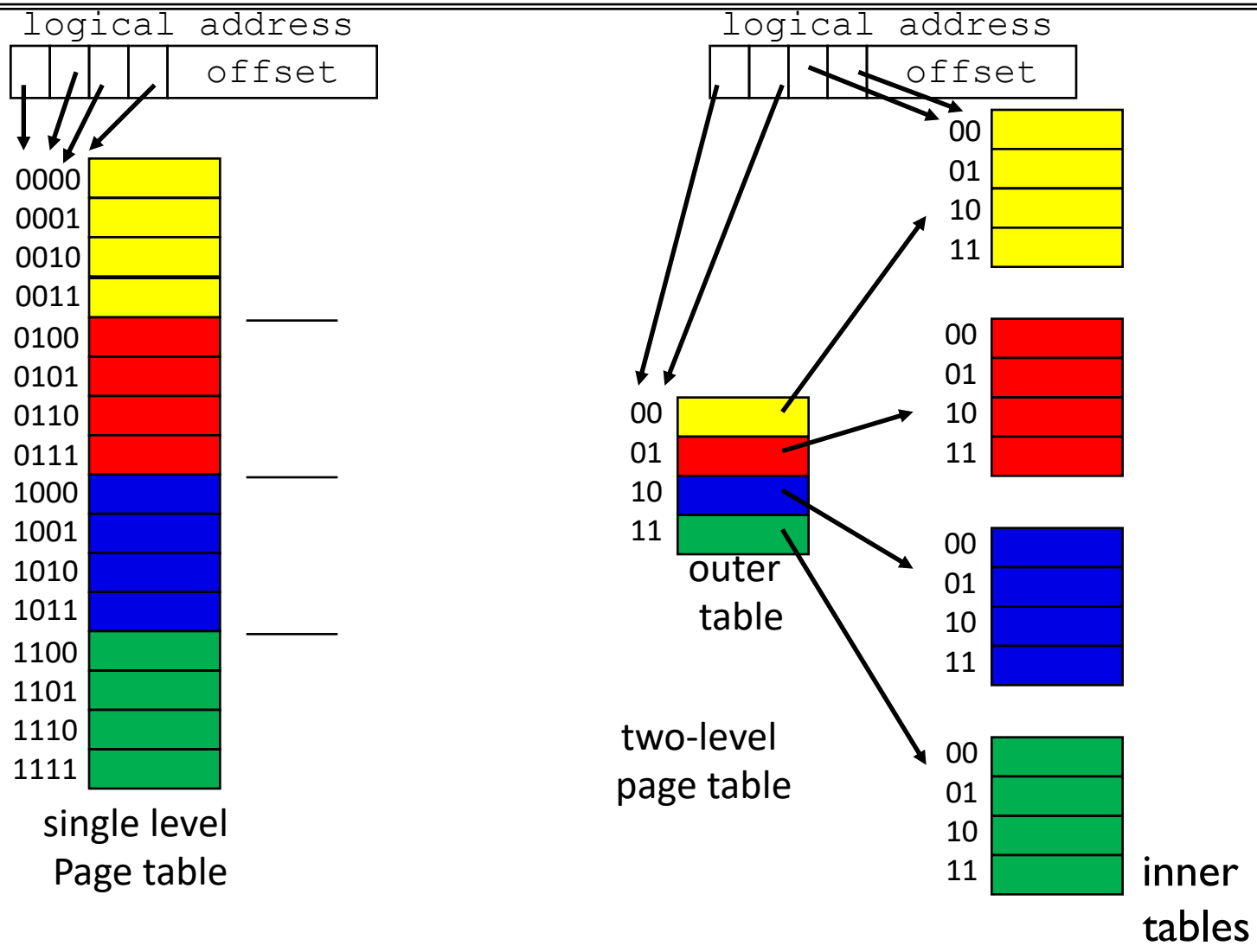




Two-Level Paging Scheme

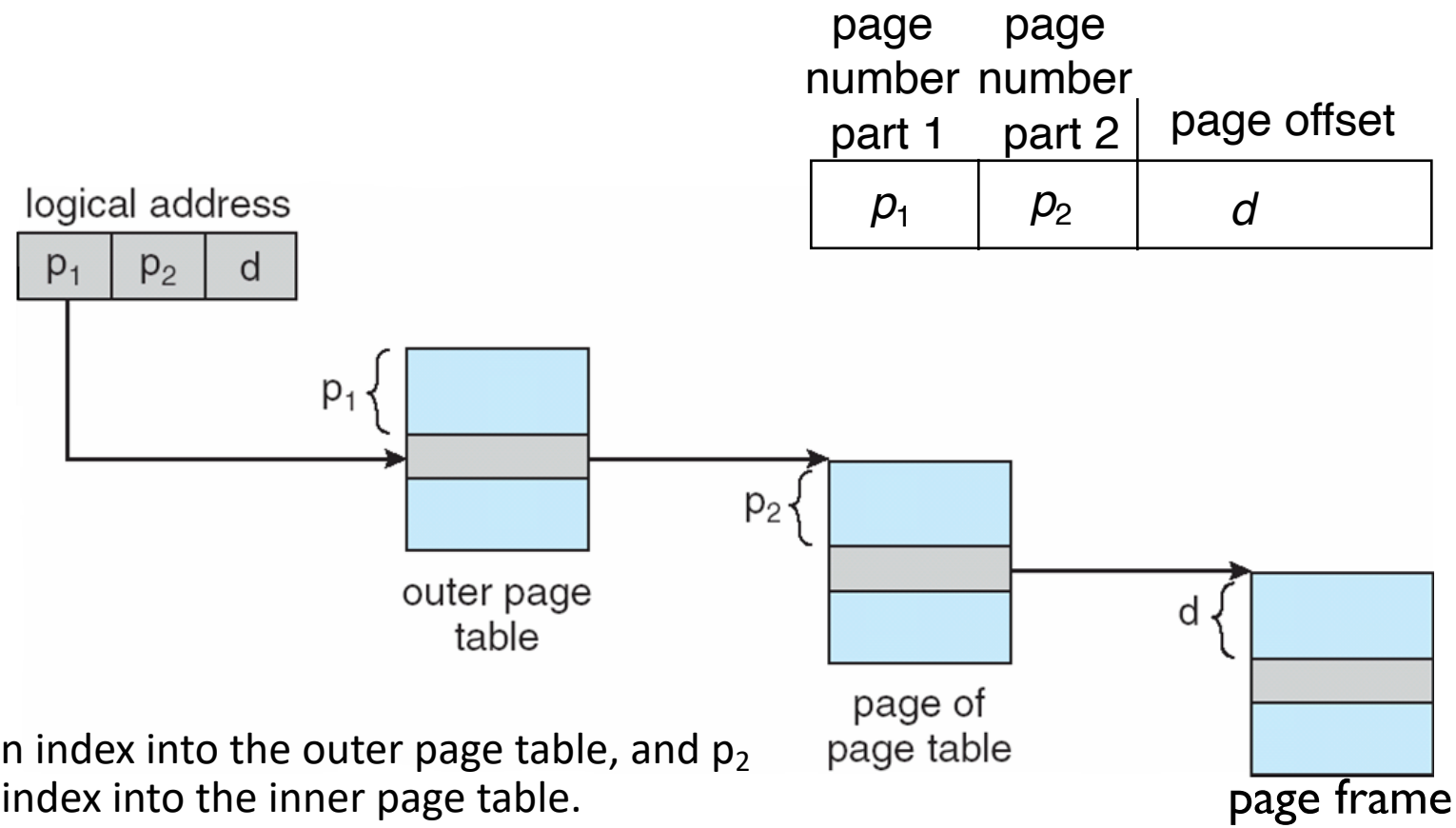


Two-Level Paging Scheme



Address-Translation Scheme

A logical address is divided into 3 parts:



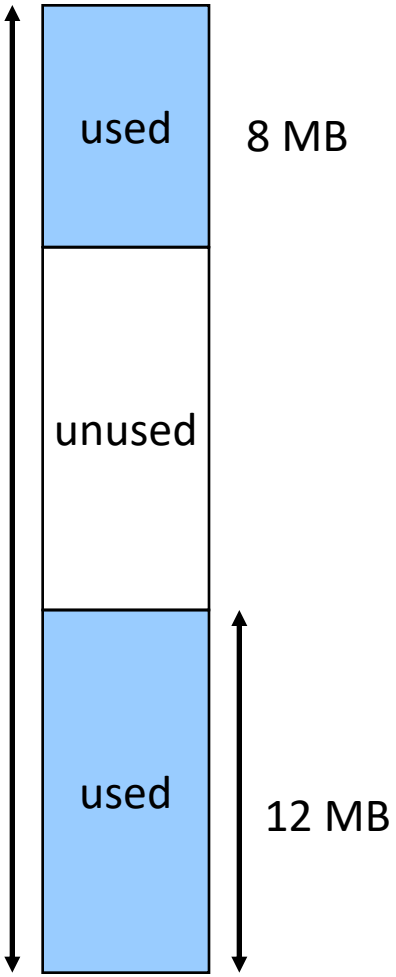
p_1 is an index into the outer page table, and p_2 is the index into the inner page table.

Example: two level page table

- Assume:
  - 32 bit virtual (logical addresses) used
  - 2 level paging
  - Address split scheme: [10, 10, 12]
    - 10 bits page number part 1
    - 10 bits for page number part 2
    - 12 bits for offset
  - **Page size** = 2^12 = **4096** bytes **= 4 KB**
  - An outer (top level) table will have 2^10 entries
  - An inner (second level) table will have 2^10 entries

Example: two level page table space requirement

logical address length = 32 bits
pagesize = 4096 bytes
logical address division: 10, 10, 12

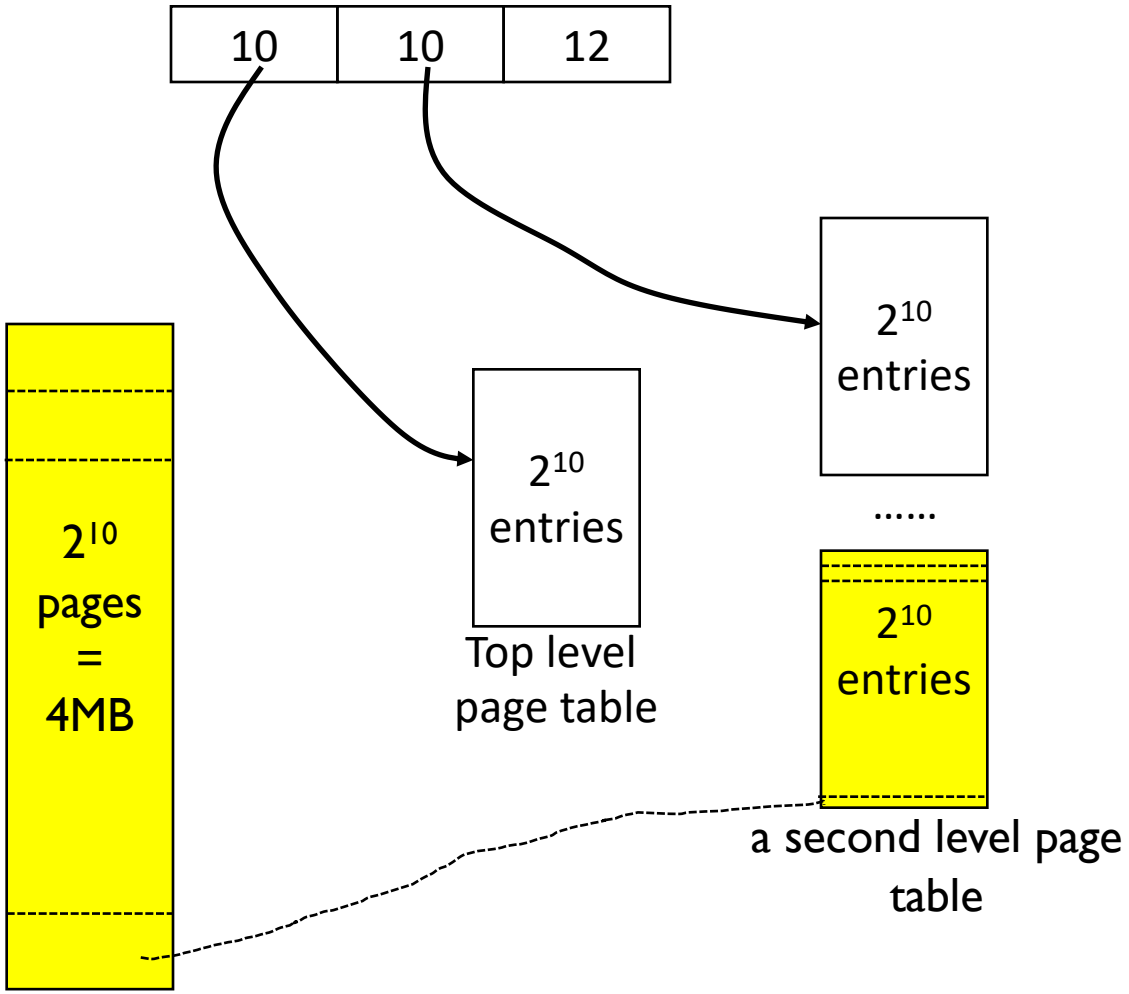


2^{32} bytes = 4 GB

logical address space size
(only partially used)

What is total size of two level page table if entry size is 4 bytes for both top level and inner page tables.

Example: two level page table space requirement

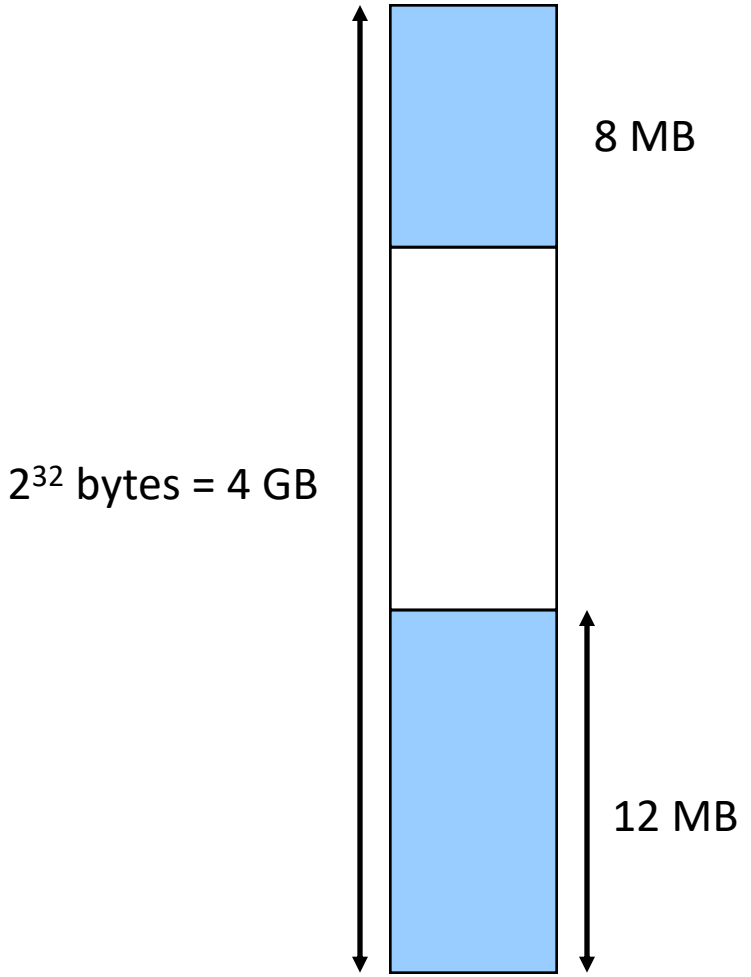


Each entry of a second level page table translates a page# to a frame#;
i.e.. ,each entry maps a page which is 4096 bytes

There are 1024 entries
In a second level page table

Hence, a second level page table can map
$2^{10} * 2^{12} = 2^{22} = 4$ MB
of logical address space

Example: two level page table space requirement

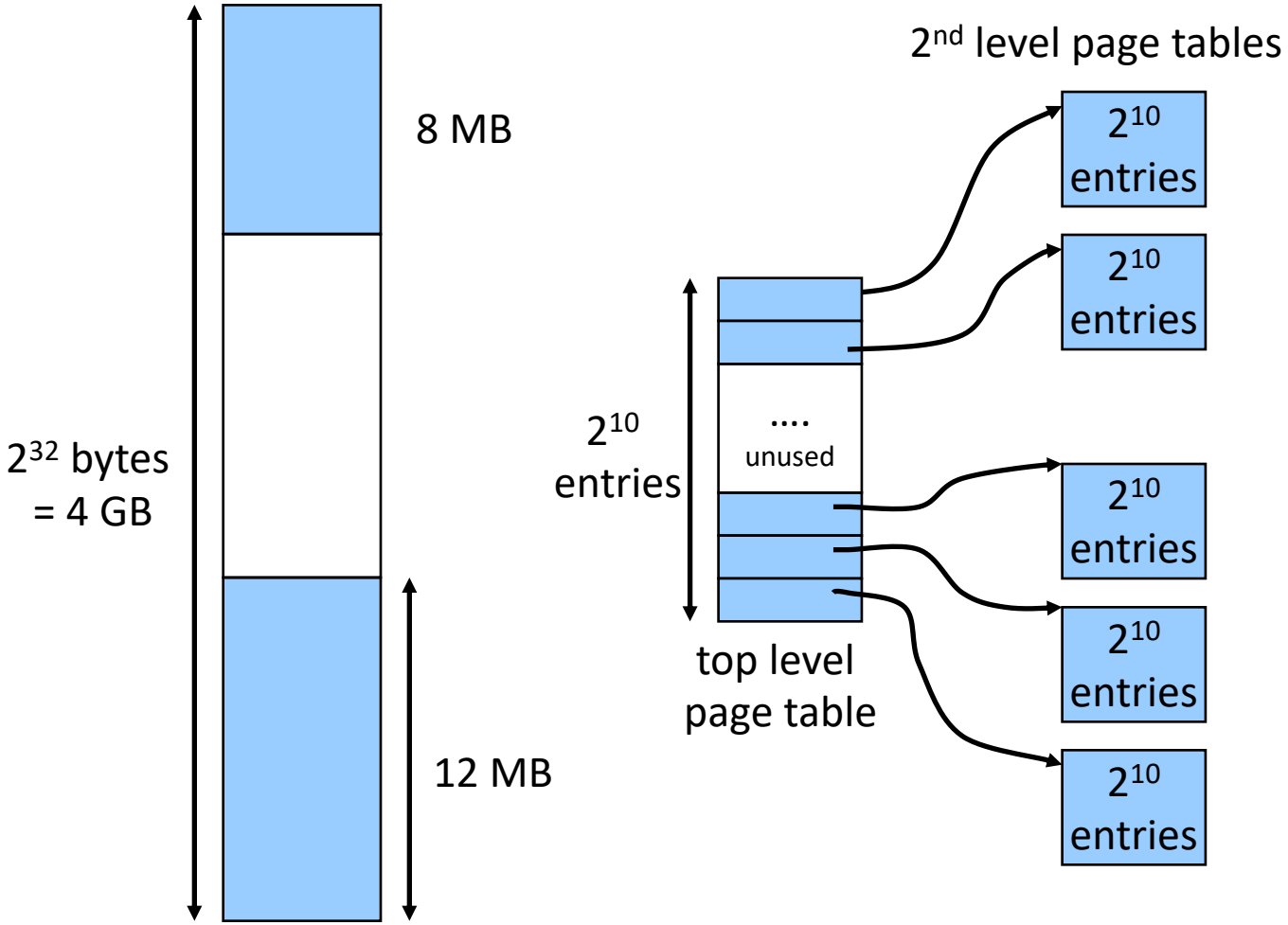


8 MB / 4 MB = 2 second level page tables required to map 8 MB of logical memory

Total = 3 + 2 = 5 second level page tables required.
We also need 1 top level table.

12 MB / 4 MB = 3 second level page tables required to map 12 MB of logical memory

Example: two level page table space requirement



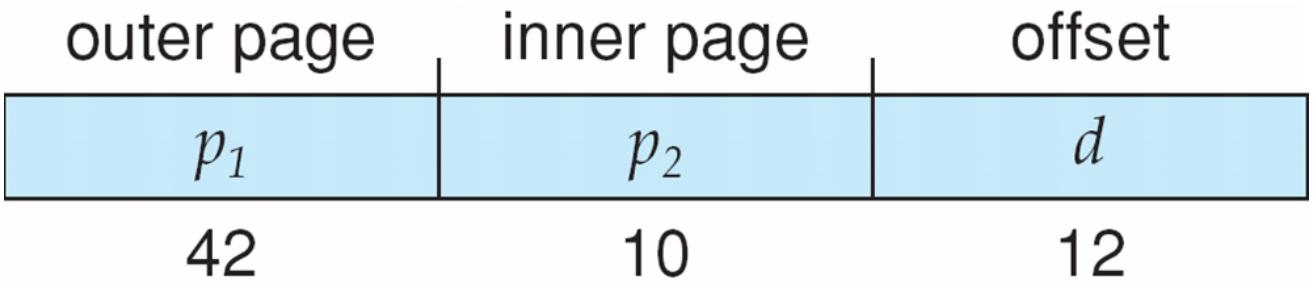
In total 6 tables required. 1+5 = 6. Each table has size=4KB.

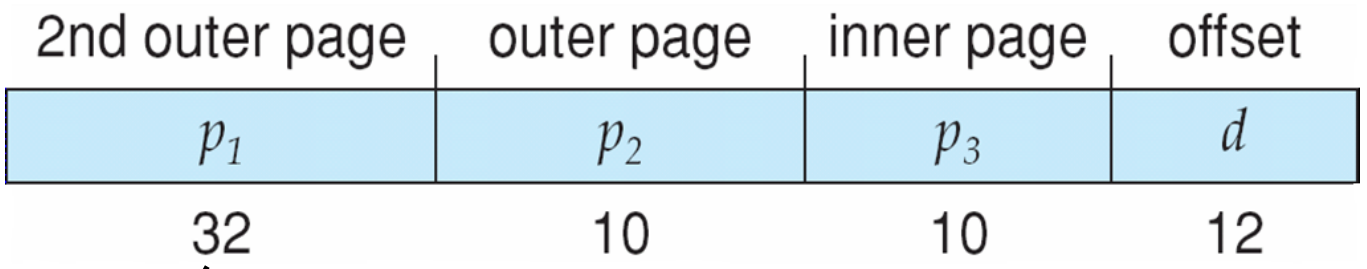
Then we need 6 x 4 KB = 24 KB for page table information

We assume entry size=4 bytes for both top level and second level tables.

Three-level Paging Scheme

64 bit addresses





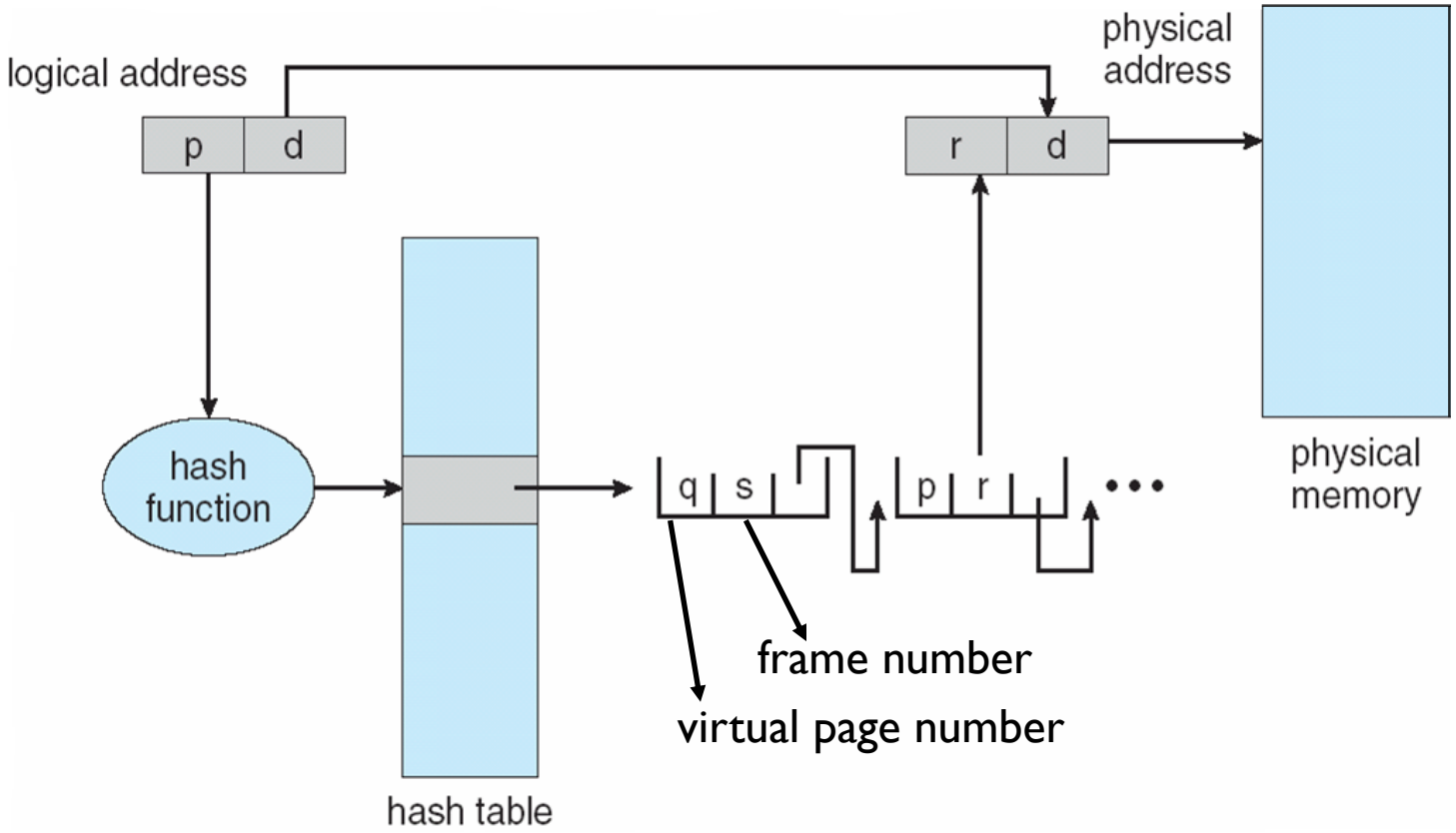
Top level table too big: 2^32 entries

Even 3-level paging is not enough

Hashed Page Tables

- Common in address spaces > 32 bits
  - Large address spaces
- Page table is a hash table
- A virtual page number is **hashed** into a page table entry
- A **page table entry** contains a chain of elements hashing to the same location
  - each element = <a virtual page number, frame number>
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted.

Hashed Page Table



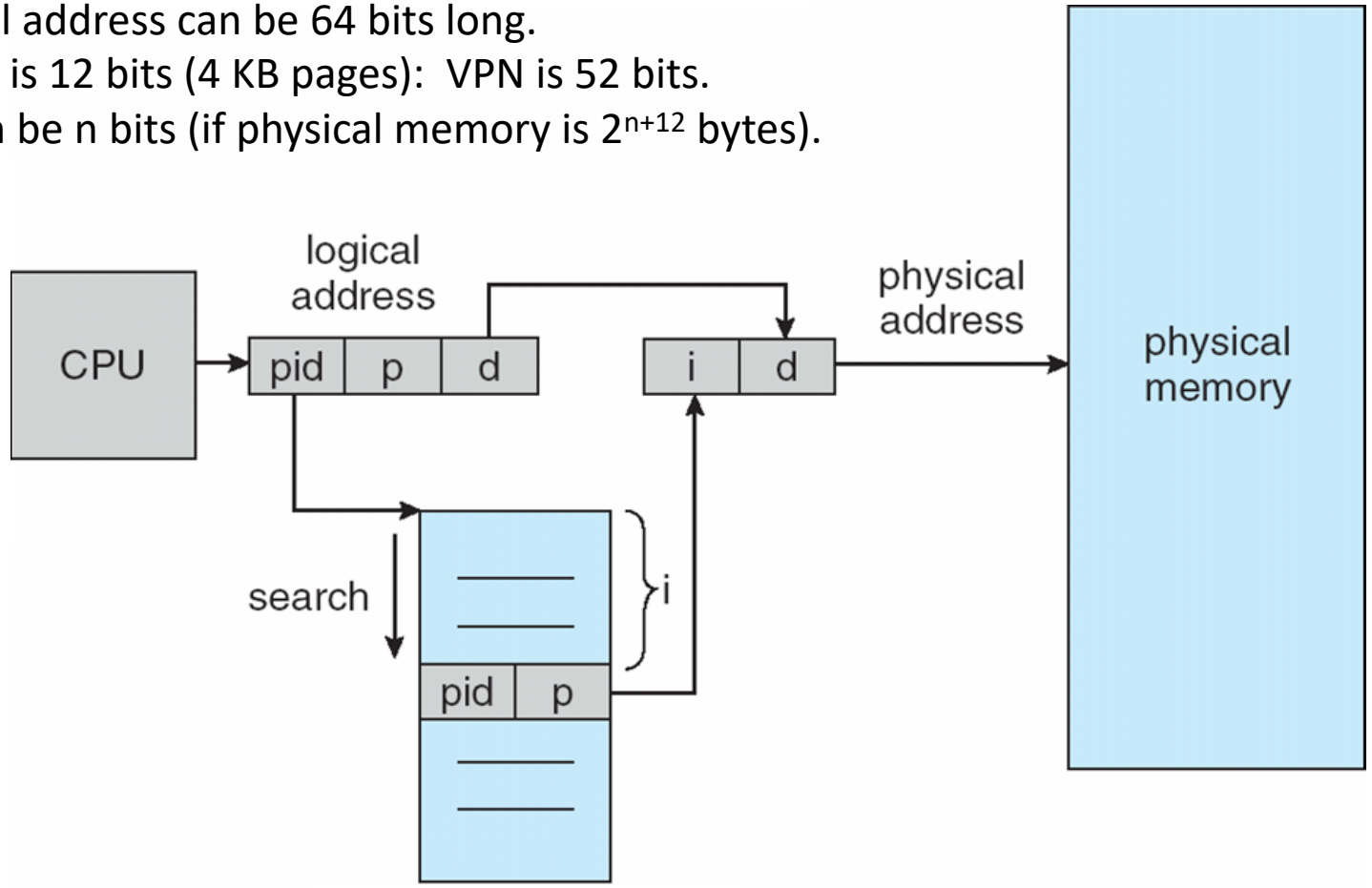
We can limit the hash table size per process, no matter how large the address space is (smaller the hash table, more collisions we have)

Inverted Page Table

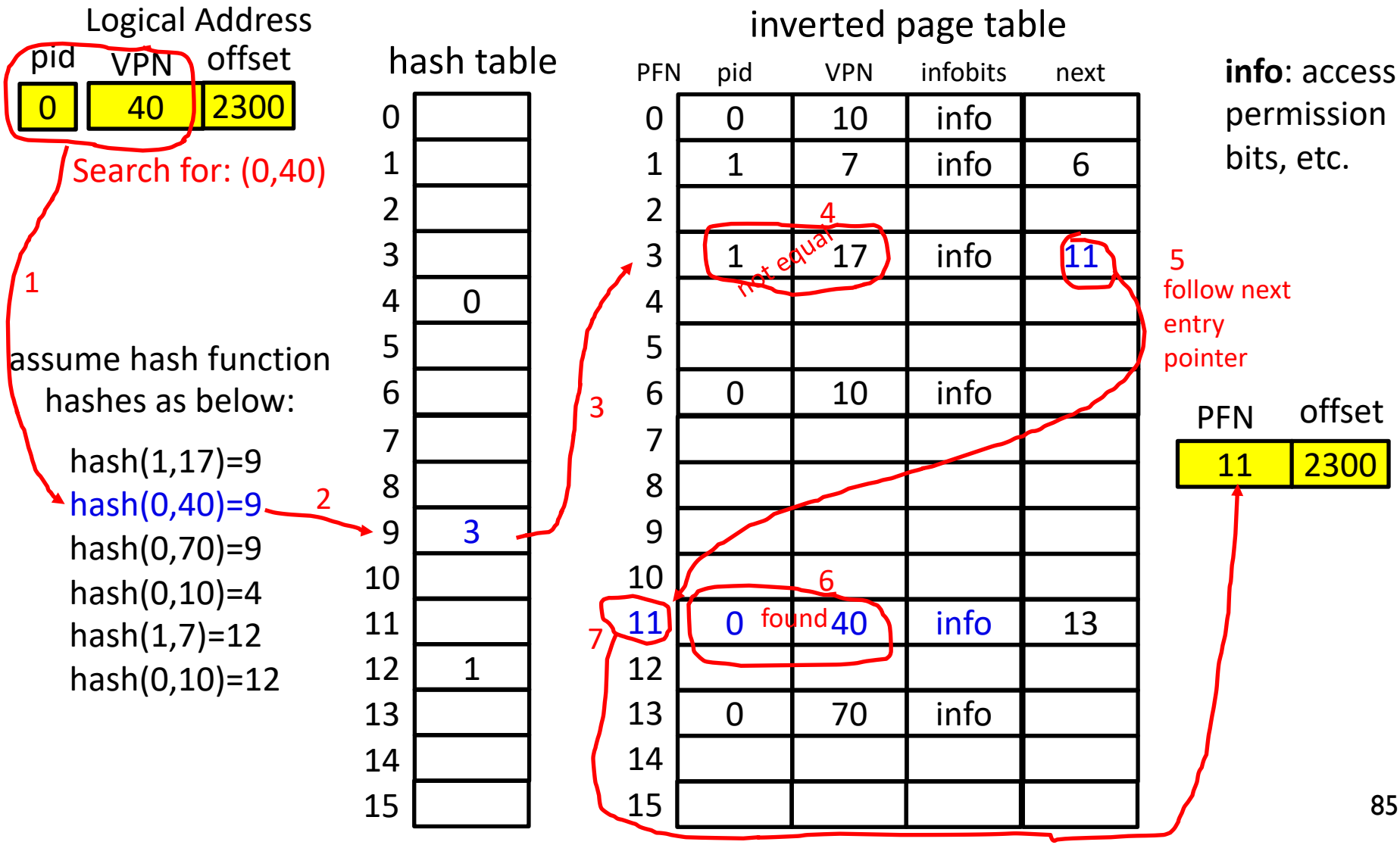
- One entry for each real page frame of physical memory
- Entry consists of the page number of the virtual page stored in that real memory location (frame), with information about the process that owns that page.
  - Entry content: <pid, virtual page number>
- Decreases memory needed to store page table
- All processes share the same inverted table
  - One inverted table in the system.
- Searching is problematic (search the p# in the table).
  - Given a p#, we need to find f# (therefore search needed)
- Search time can be reduced by using a hash table as well.

Inverted Page Table Architecture

A logical address can be 64 bits long.
If offset is 12 bits (4 KB pages): VPN is 52 bits.
PFN can be n bits (if physical memory is 2^{n+12} bytes).



Inverted Table Example with a hash table in front



Example from a real system.
Intel x86 architecture
support for memory management

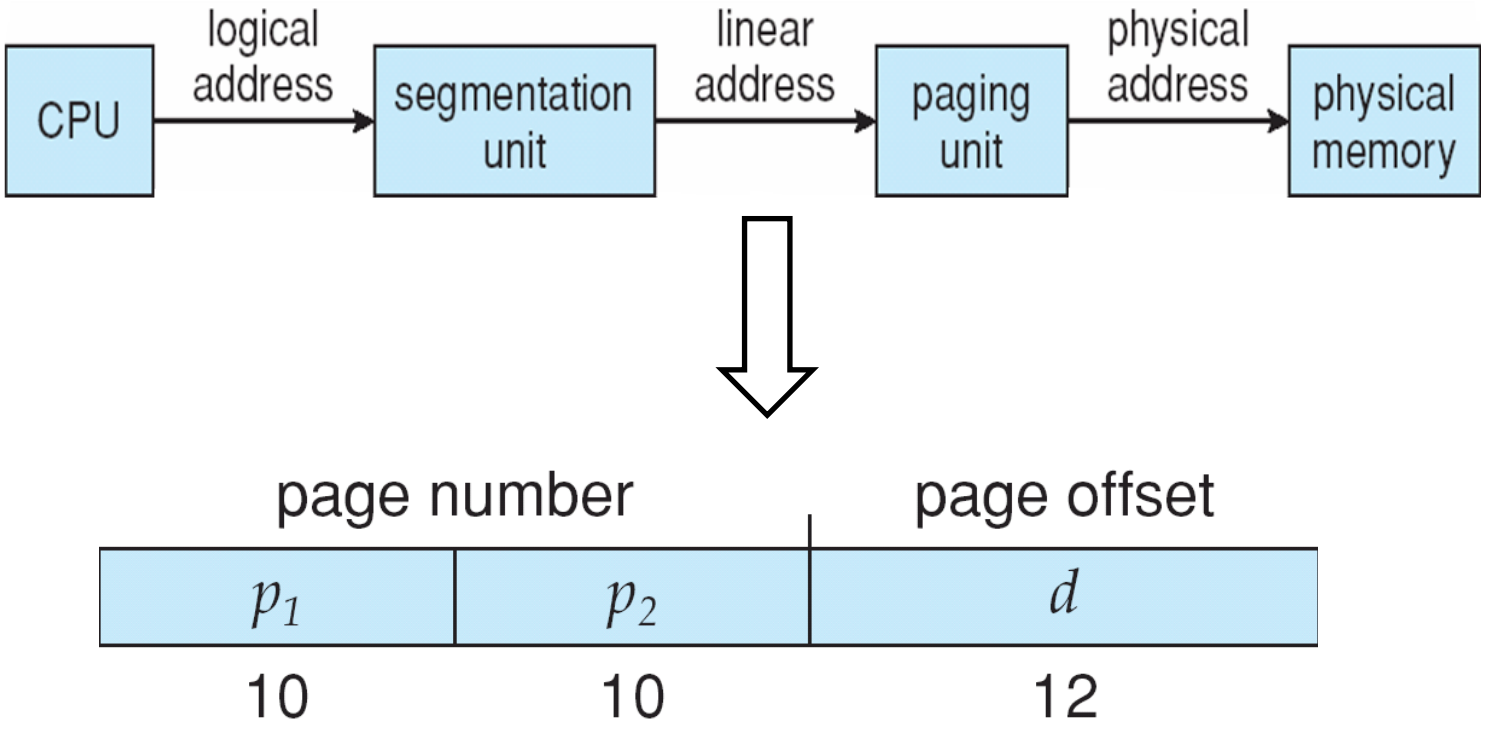
Example: The Intel 32 bit (IA32) architecture (x86 architecture)

- Supports both segmentation and segmentation with paging
- CPU generates logical address (<segment#, offset> pair)

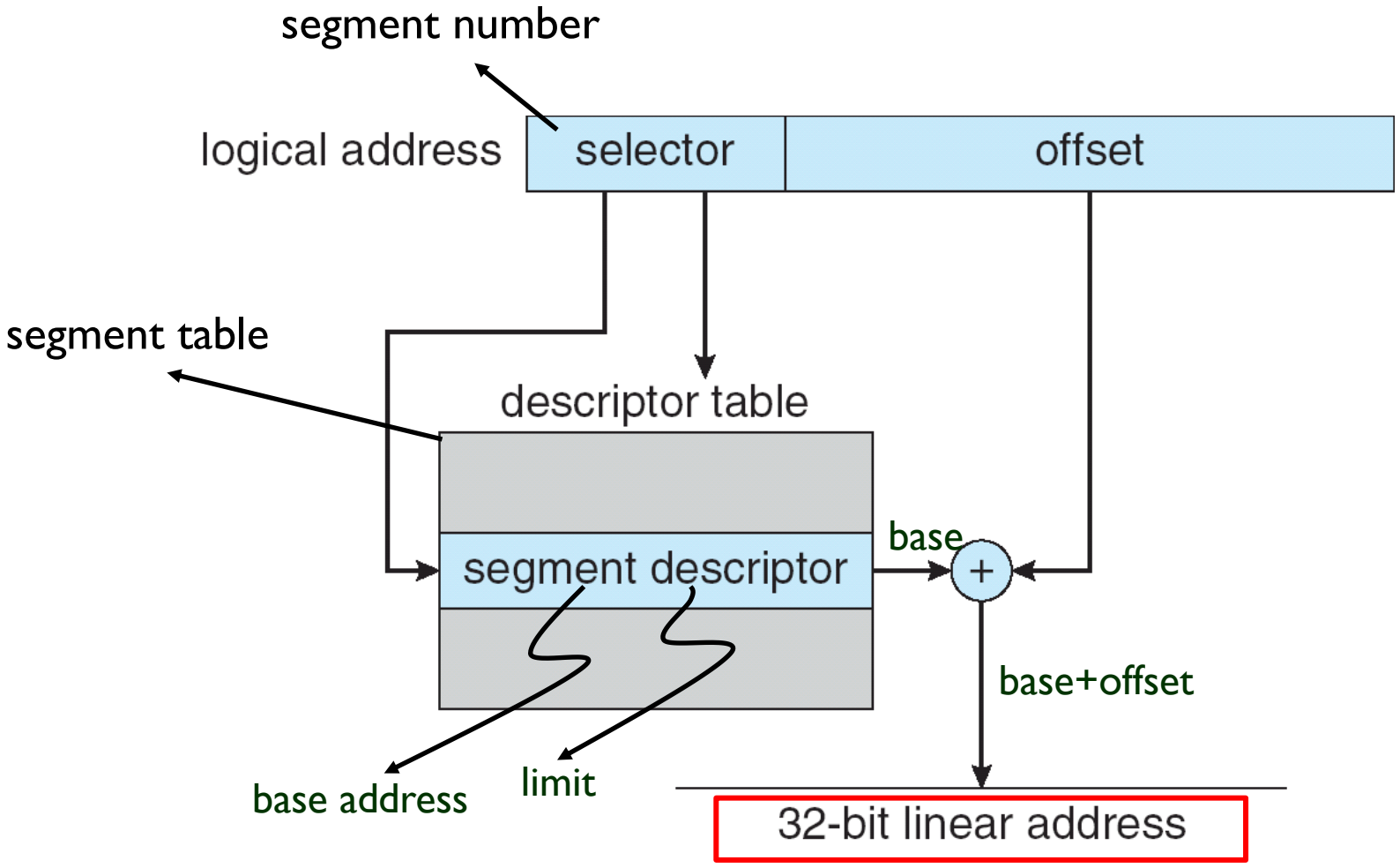
  (this is a two-dimensional address)
  - Given to segmentation unit
    - Which produces linear addresses

      (this is a one dimensional address)
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging unit forms equivalent of MMU

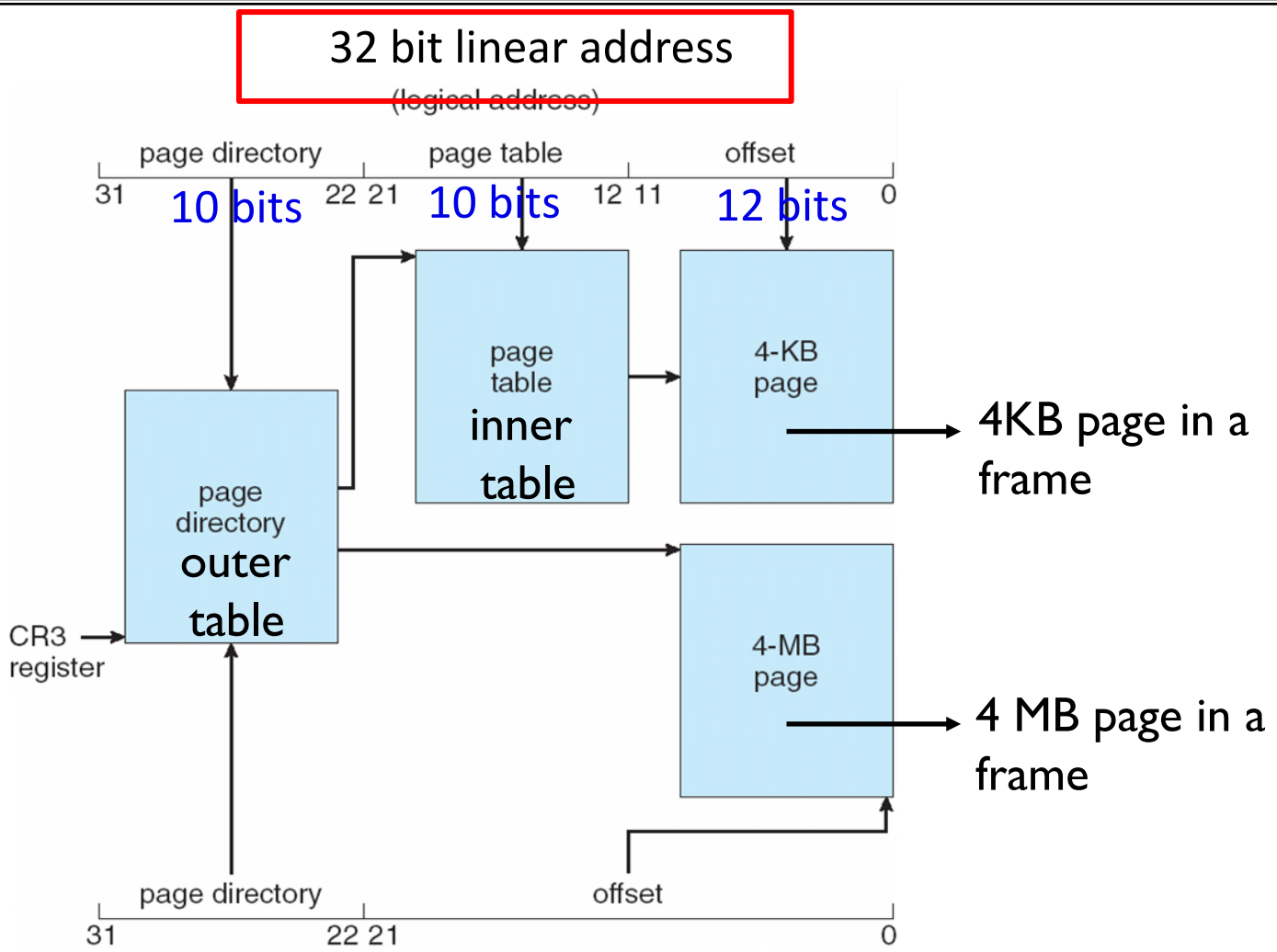
Logical to Physical Address Translation in Intel IA32



Intel IA32 Segmentation



IA32 Paging Architecture



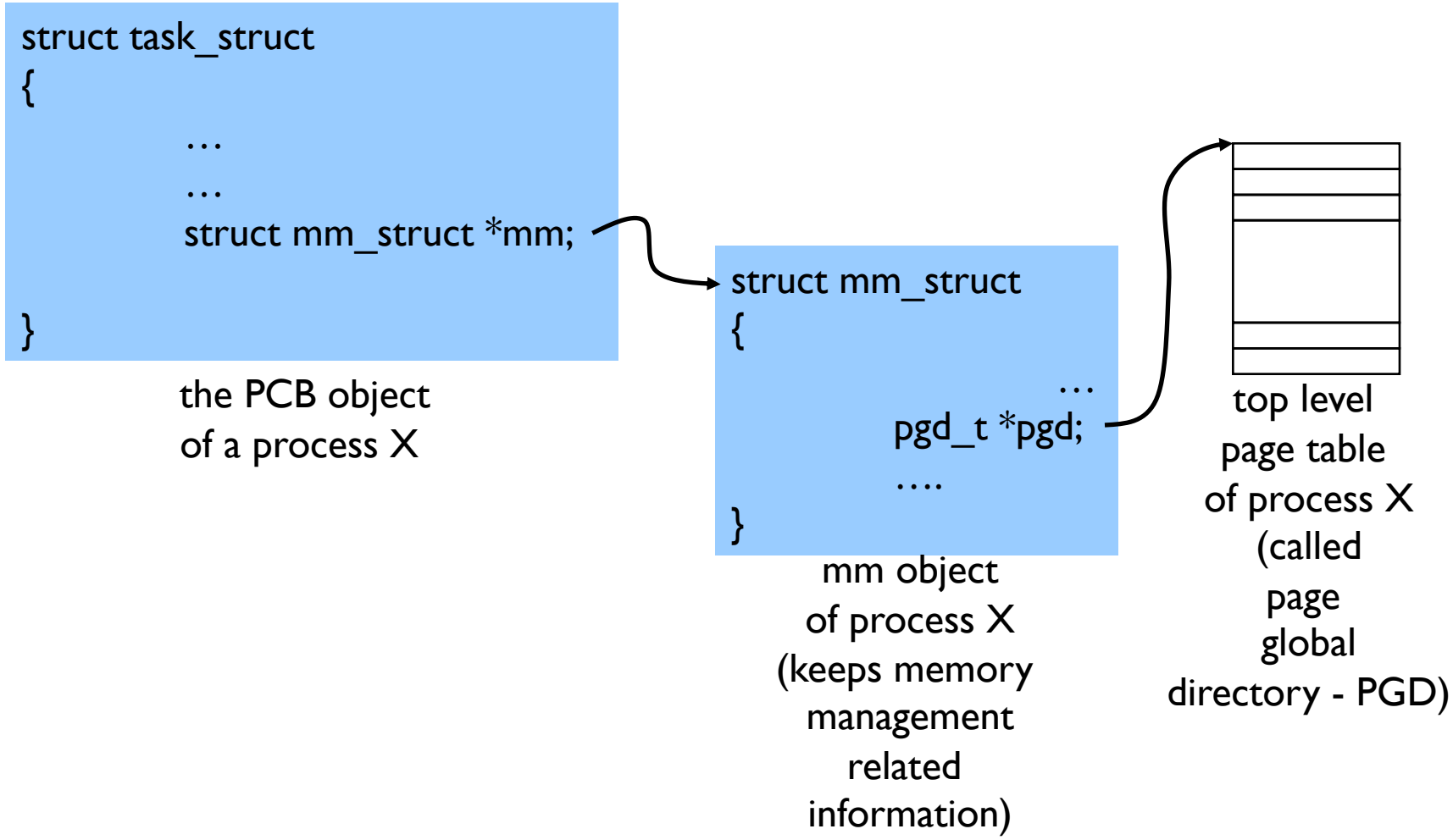
IA32 Paging Architecture

- IA32 architecture allows a page size of either 4 KB or 4 MB.
- For 4 KB pages, two-level paging scheme is used in a 32 bit machine
  - Address division: <10, 10, 12> bits
- For 4 MB pages, we can skip the inner page tables (secondary page tables). A top level page table entry will point directly to a 4 MB page.

Linux on IA32: Segmentation

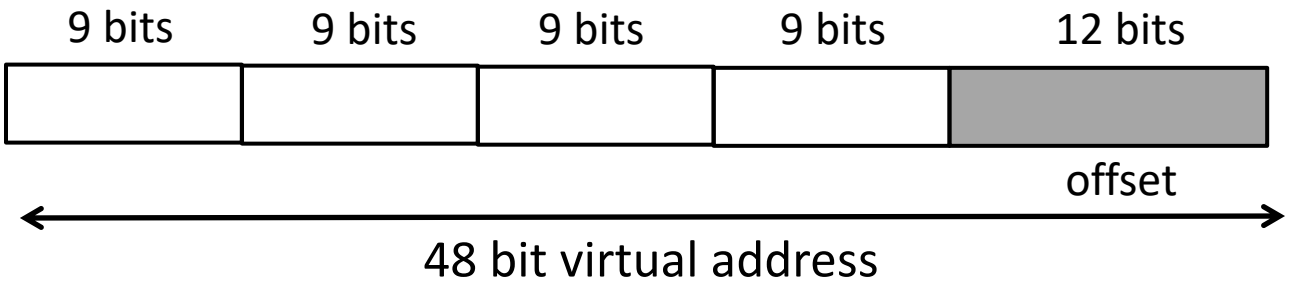
- Linux is designed to run on different hardware platforms:
  - x86,
  - Arm,
  - Motorola,
  - Sparc,
  - MIPS,
  - …
- Therefore it does not rely on segmentation and makes minimal use of segmentation in Intel IA32 architecture.

Linux Paging: some kernel structures (32 bit)

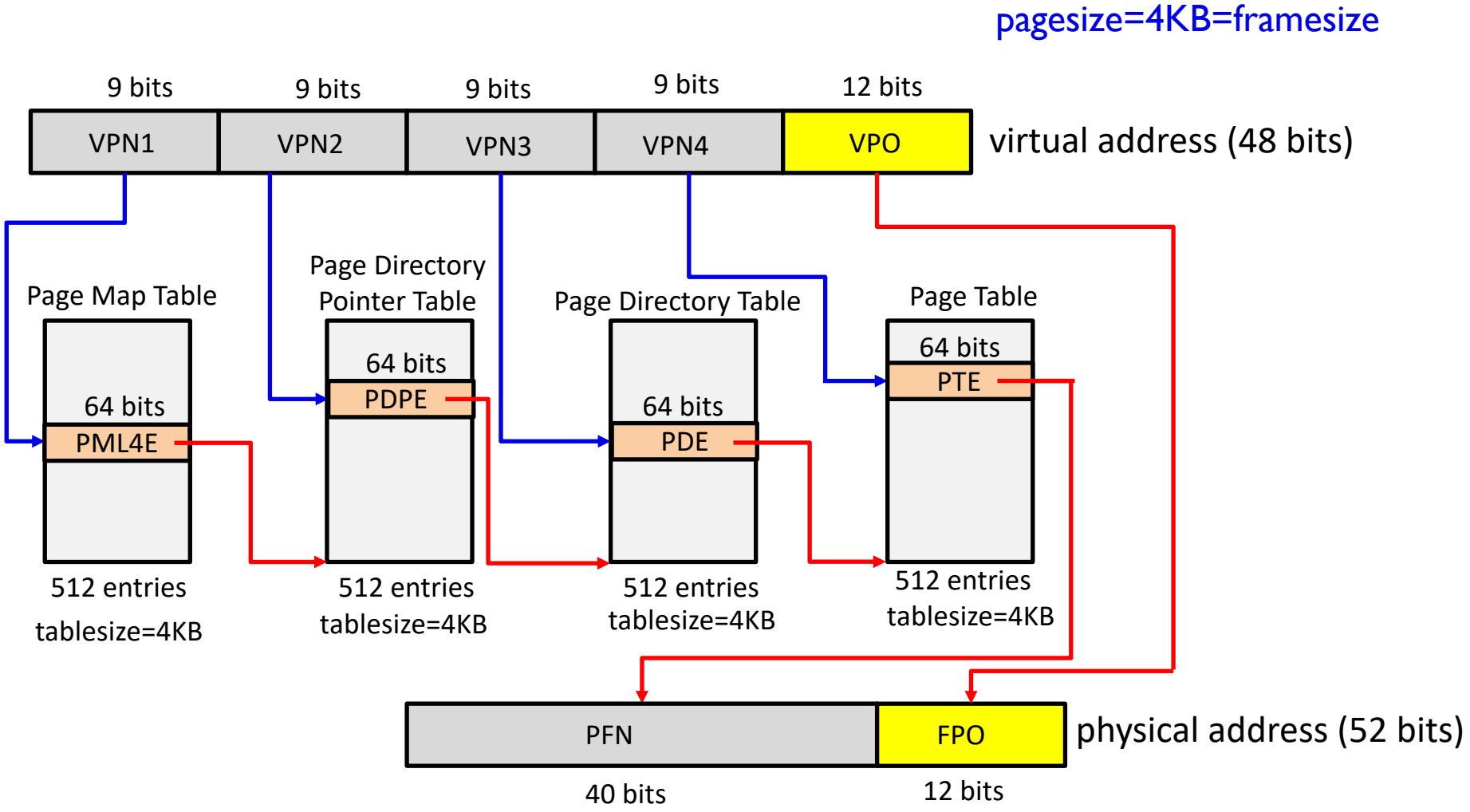


4 level paging in x86-64 architecture

- **Intel 64 bit** architecture (x86-64) uses four level paging. Linux supports x86-64.
- Virtual addresses are 48 bits long. Address division scheme is: [9,9,9,9,12]



x86-64 - translating virtual addresses to physical addresses



References

- Operating System Concepts, Silberschatz et al, Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum.
- Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al.
- Intel Architecture Manuals.

Additional Study Material

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping