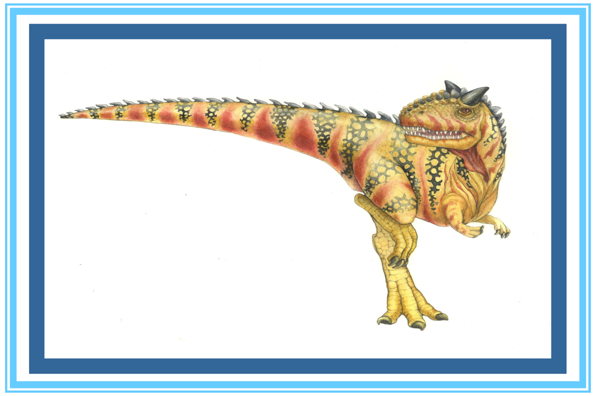
Chapter 8: Memory-Management Strategies

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- Important keywords
  - Logical (=Virtual) Memory, Paging (TLB)
  - We use logical address and virtual address interchangeably in this text.

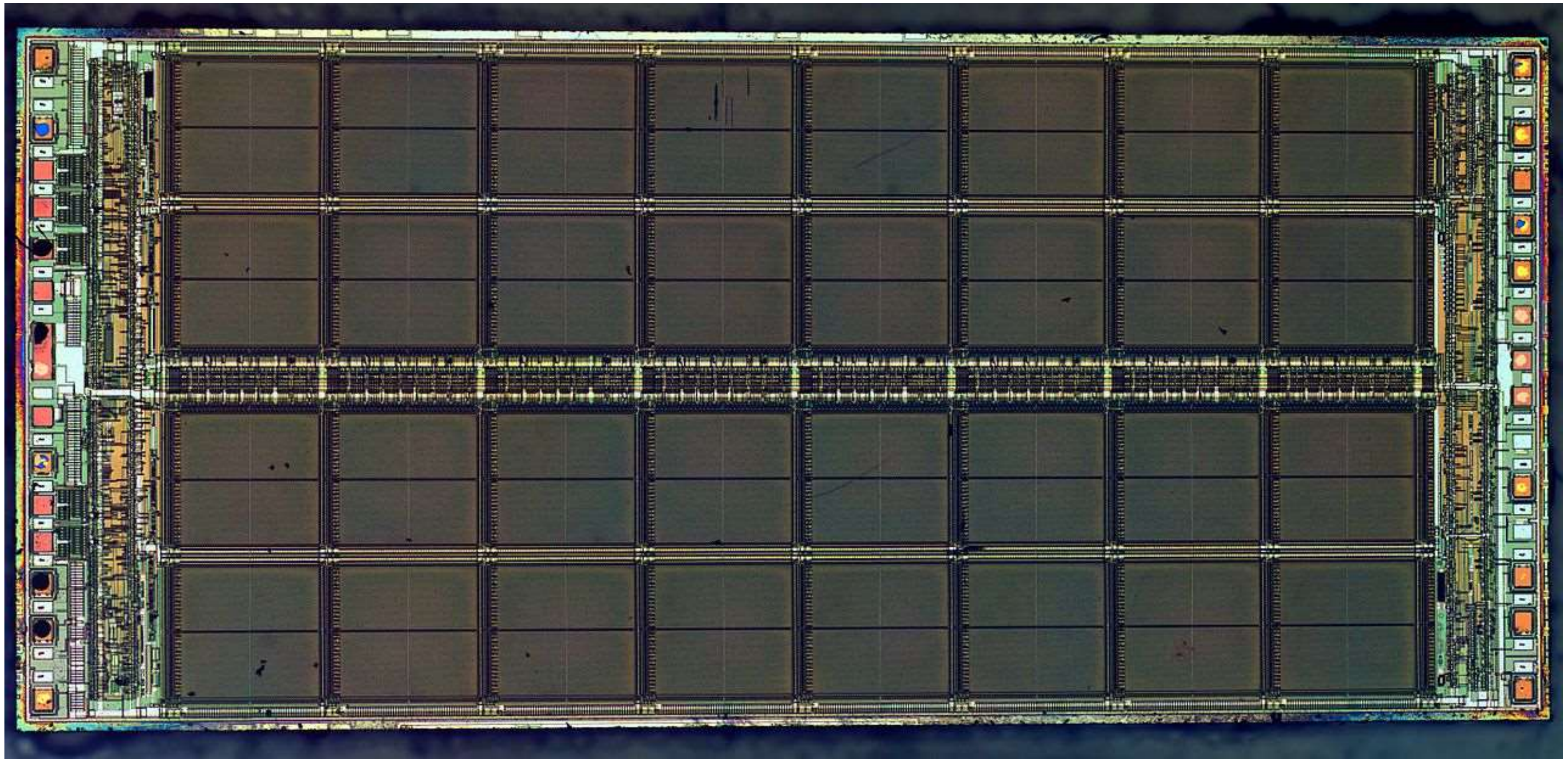
Chapter 8: Memory Management Strategies

- **Background**
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table

Memory Technology (2012)

- SRAM semiconductor memory - **Cache**
  - **<u>0.5ns – 2.5ns</u>**, $500 – $1000 per GB
- **DRAM semiconductor** memory – **Main memory**
  - 50ns – 70ns, $10 – $20 per GB
- Flash semiconductor memory – **Flash disk/memory**
  - 5,000 – 50,000ns, $0.75 - $1.00 per GB
- Magnetic disk – **Hard disk**
  - 5ms – 20ms, $0.05 – $0.1 per GB
- Ideal memory: Fast, large and cheap?
  - Access time of SRAM
  - Capacity and cost/GB of disk

DRAM: main memory

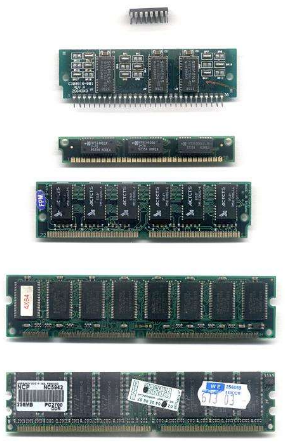


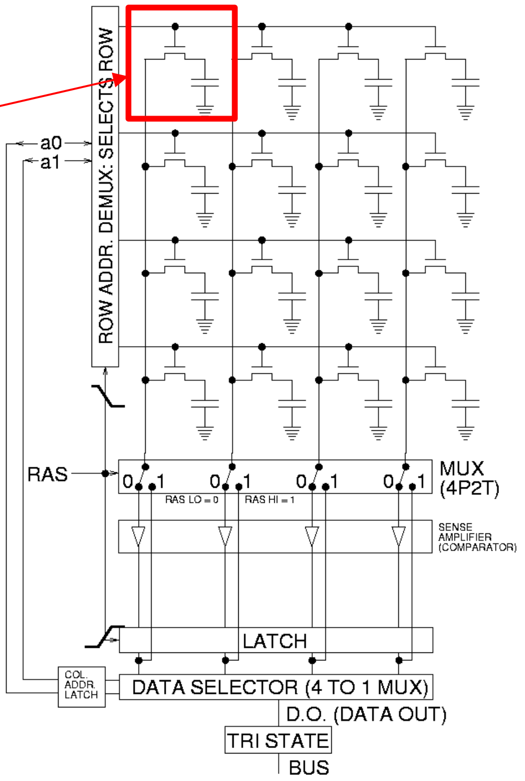
http://zeptobars.com/en/read/how-to-open-microchip-asic-what-inside

DRAM: main memory

- **Dynamic random access memory**
  - Storage cells: One capacitor and one transistor = one data bit

- As software engineers,
  - Let the Electrical Engineers worry about these complexities...
  - We are only interested in <u>memory address</u> and <u>byte data</u>

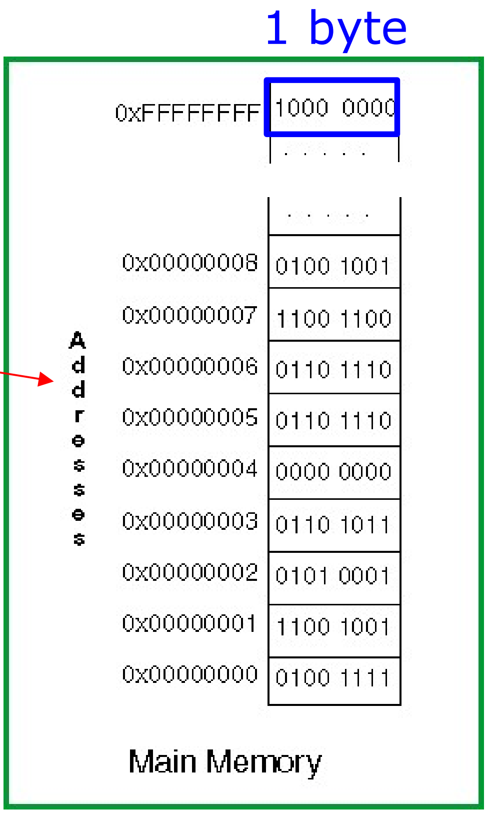




https://en.wikipedia.org/wiki/Dynamic_random-access_memory

DRAM: main memory

- Dynamic random access memory
  - Give address to DRAM
    - Read/write data
  - **Byte address**
    - Each data is 1-byte



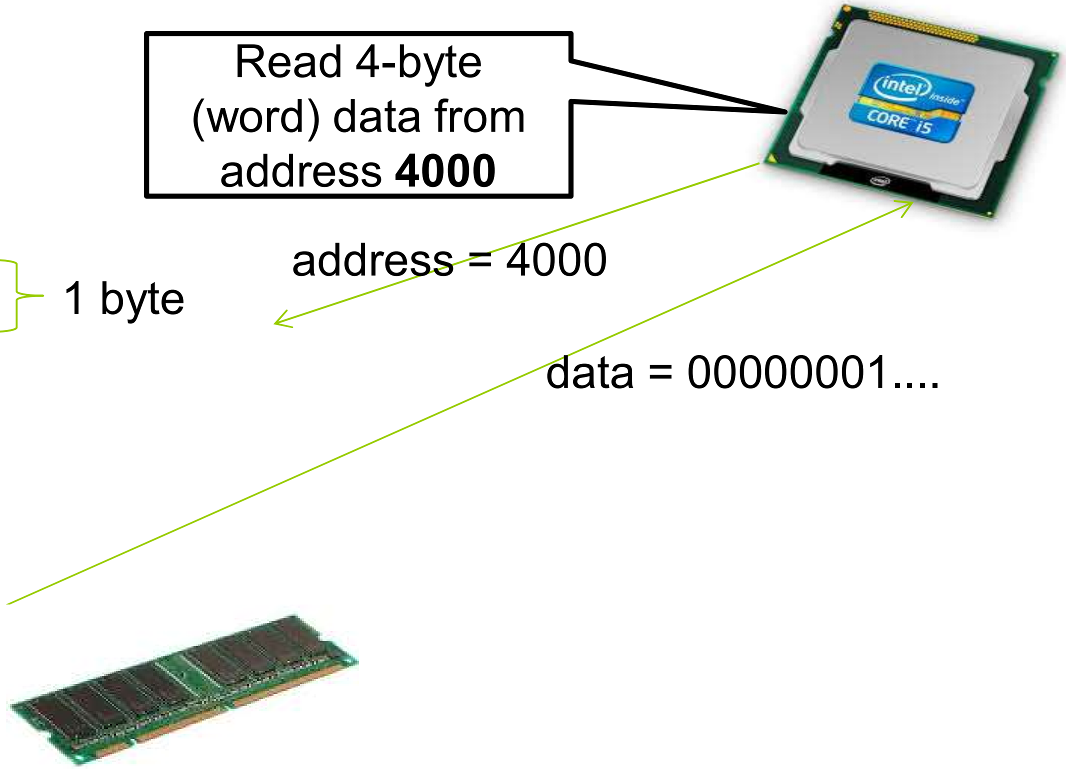
https://chortle.ccsu.edu/AssemblyTutorial/

DRAM: main memory

- Dynamic random access memory
  - Byte address

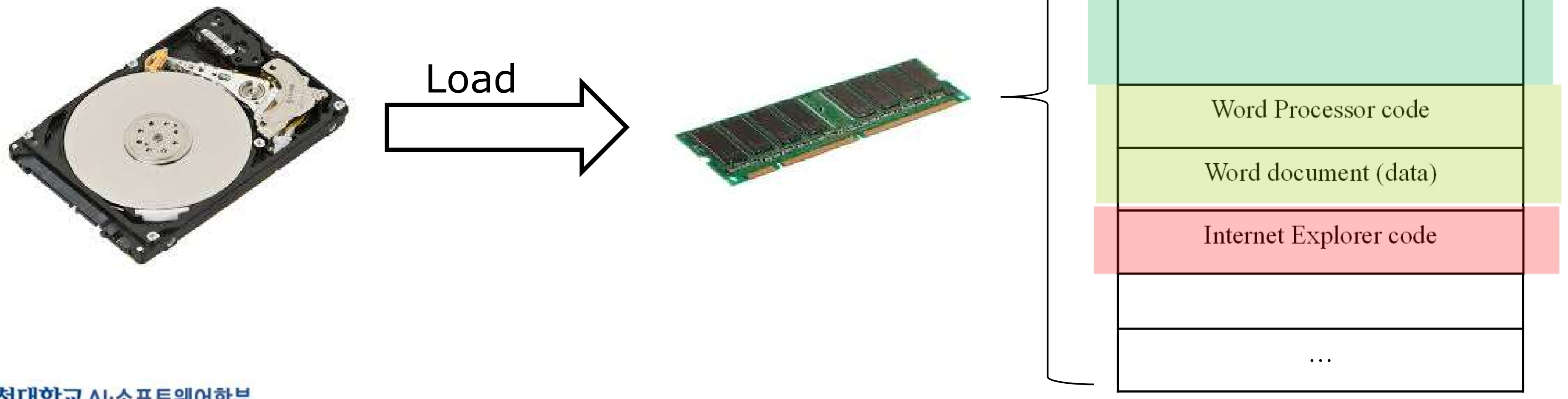
address	data
0	01101111
1	10011111
2	00000000
...	…
4000	00000001
4001	…
…	…

1 byte



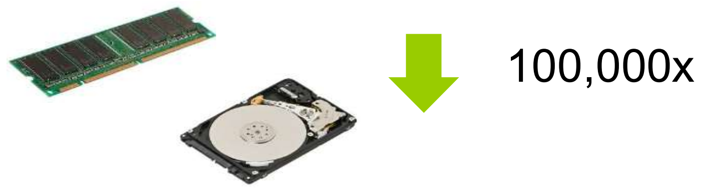
DRAM: main memory

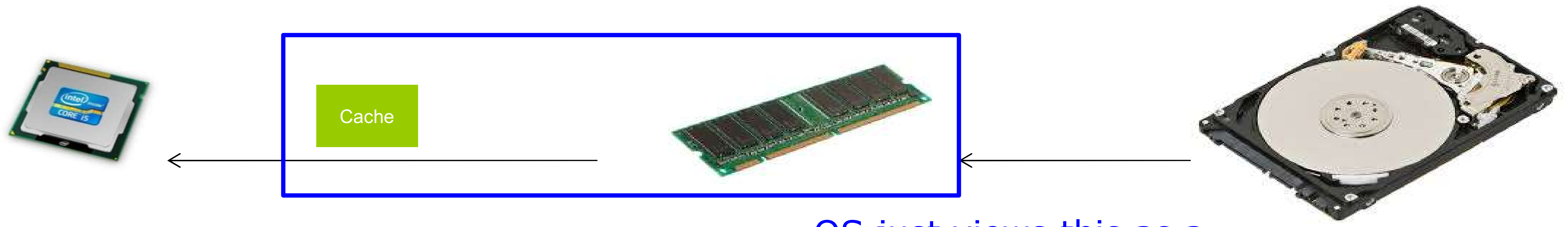
- Dynamic random access memory
  - **Programs** are usually stored in the **hard disk**
    - When programs are executed, it is loaded on the **main memory** – faster than disk
  - Similarly, when **file** is **open**, it is loaded on the main memory



DRAM: main memory

- Why load program/data from disk to memory?
  - DRAM access time ≈ 50ns
  - Disk access time ≈ 5,000,000ns
  - If we use disks to run programs it will be **100,000 times** slower
- But DRAM is still much slower than CPU
  - It takes about **100~400 CPU cycles** to access DRAM
  - Use a small but very fast memory = **Cache**!!



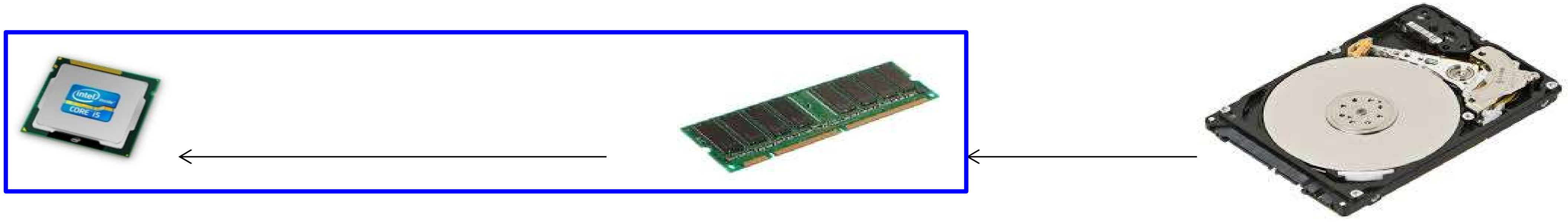


OS just views this as a single memory

Background

- Memory is ***central*** to the operation of a computer system

- A typical **CPU instruction-execution cycle**
  - 1. The CPU **fetches** an **instruction** from **memory**
  - 2. The instruction is then decoded and may cause operands (data) to be fetched from memory
  - 3. the result may be **stored back in memory**



Background Summary

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory is the only storage that CPU can access directly
  - CPU cannot directly access disk, network, printers, ... – ***must*** be copied to memory first
- Speed?
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
    - Note: Faster **cache memory** sits between main memory and CPU registers. **OS view of cache is same as main memory!!**

Sharing/Protecting the Memory

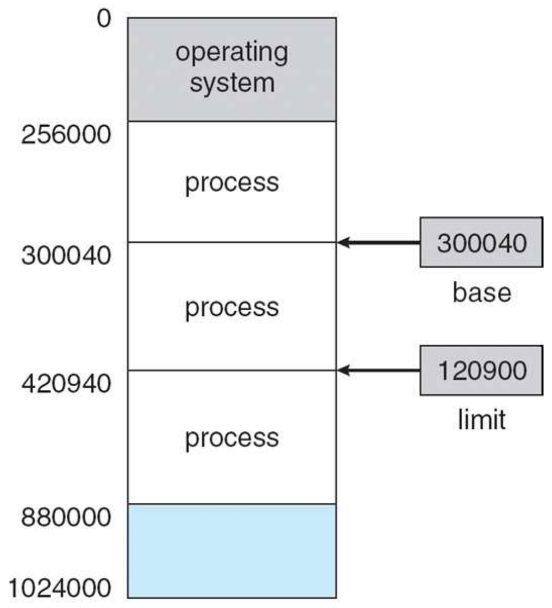
- Kernel, System/user Processes, Data share the main memory

- **Q1:** How can the processes share the main memory?
- **Q2:** How can the processes be protected from each other?

- Answer: Each process uses **Logical (or Virtual) address** instead of actual physical memory address!

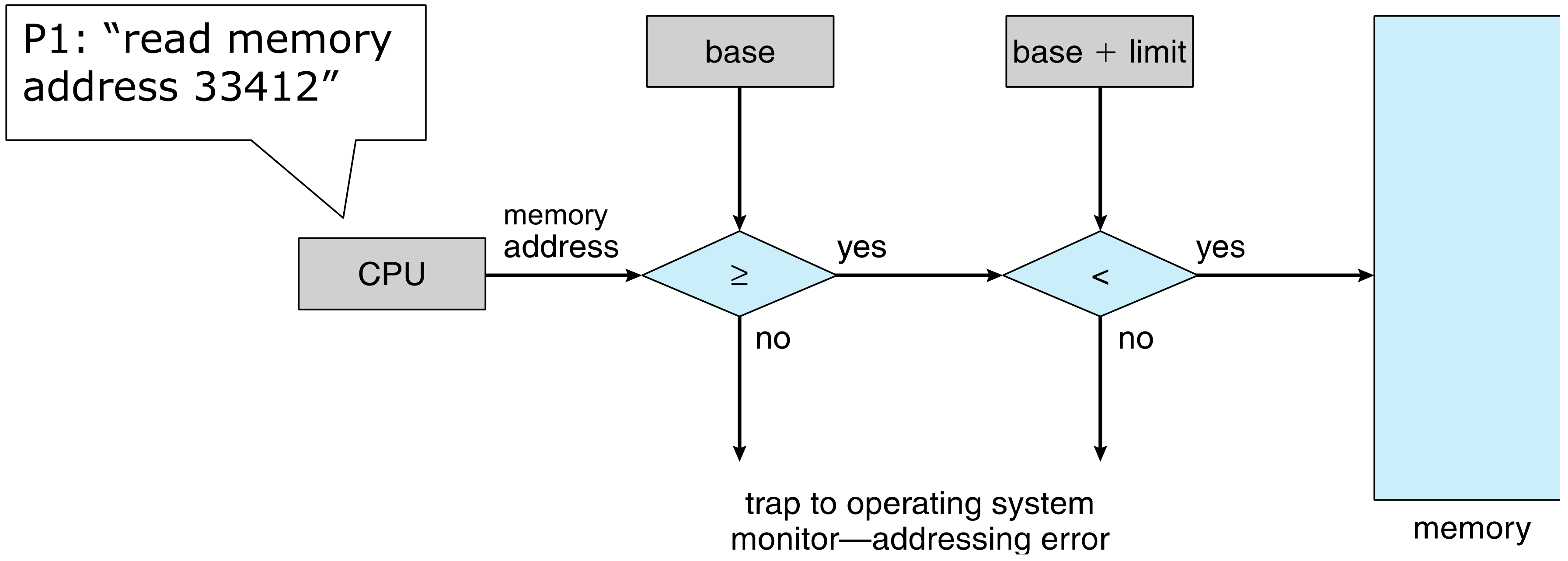
Base and Limit Registers

- **Protection** of memory required to ensure correct operation
  - Protect OS from user programs
  - Protect user programs from one another
- Each process must have separate memory space
- Hardware protection: a pair of **base register** and **limit register** define the **logical address space** of each process
  - Memory access outside the logical space?
    - Trap to OS



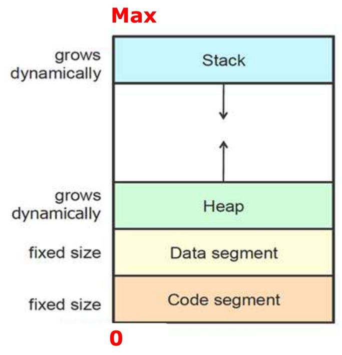
Hardware Address Protection with Base and Limit Registers

CPU must check every memory access generated in user mode to be sure it is between **base** and **base+limit** for that user

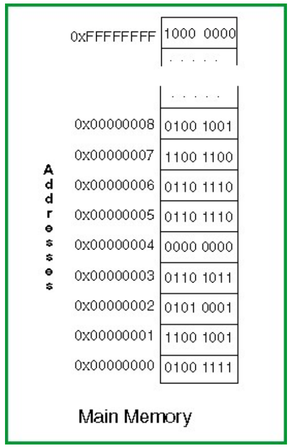

P1: "read memory address 33412"
base
base + limit
CPU
memory address
≥
yes
<
yes
no
no
trap to operating system monitor—addressing error
memory

Logical vs. Physical Address Space

- **Logical address** (= **virtual address**)
  - An address generated by **CPU:** meaningful only to the **user process**
  - Always starts from **address 0**
  - **Logical Address space**
    - **Set** of all logical addresses generated by a program

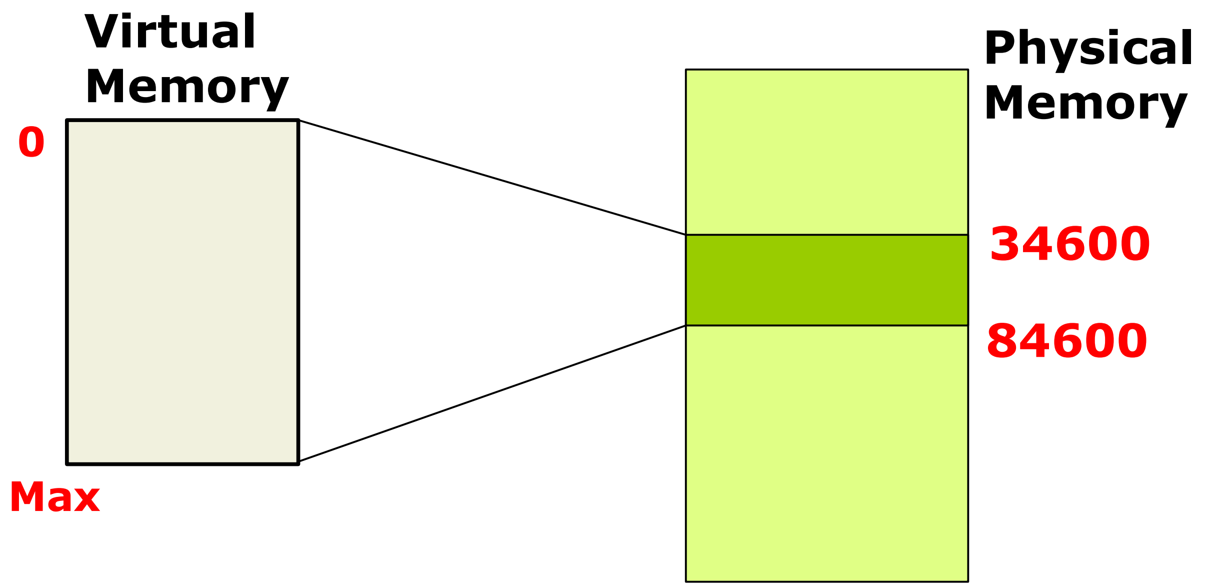


- **Physical address**
  - Address seen by the **physical memory unit**
  - **Physical Address space**
    - Set of all physical addresses



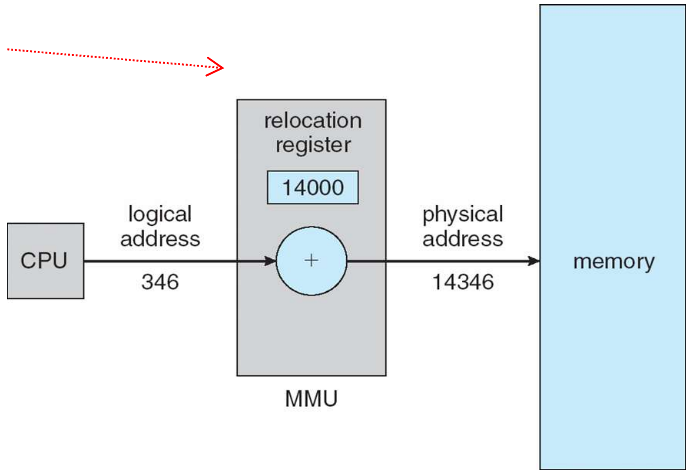
Address Space

- A **logical (virtual) address** is a memory address that a process uses to ac cess its own memory
  - Virtual address ≠ actual physical RAM address
  - When a process accesses a virtual address, the **memory-management unit (MMU)** **translates** the virtual address into a physical address
  - The OS determines the mapping from virtual address to physical address



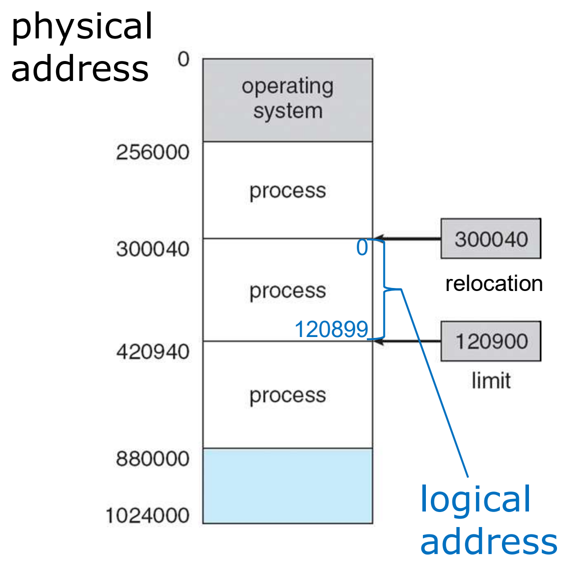
Memory-Management Unit (MMU)

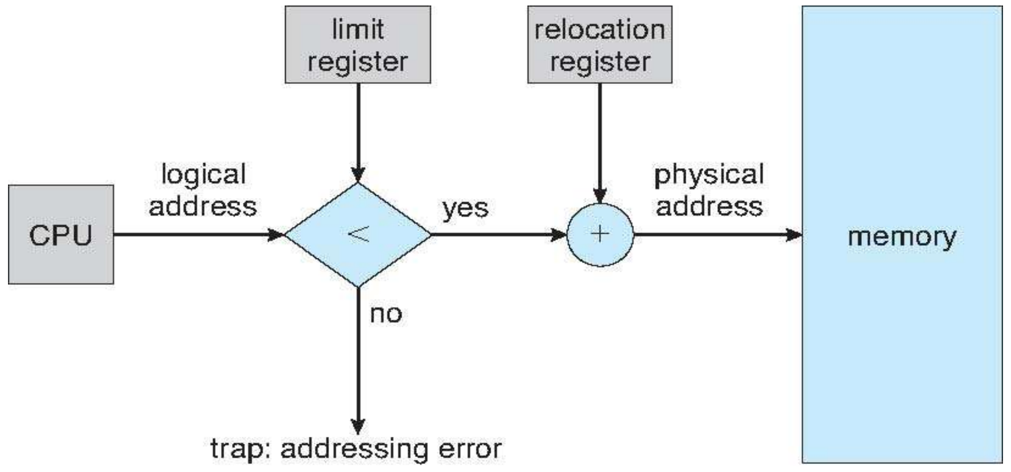
- **Memory-Management Unit (MMU)**
  - Maps virtual to physical address at **run time** with **hardware support**.
  - Many methods possible, covered in this chapter (e.g., relocation register, paging, segmentation)
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
- The user program deals only with ***logical*** addresses; it never sees the real ***physical*** addresses



Memory Protection

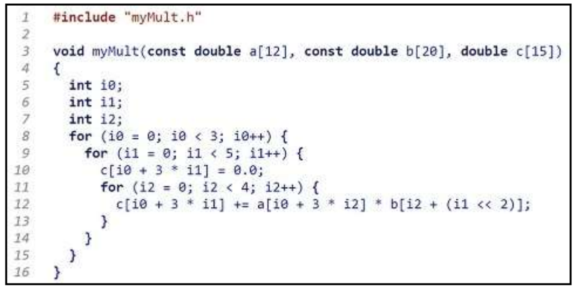
- Need to **protect** user processes from each other, and from changing operating-system code and data
- **Memory Protection: Relocation-register scheme**
  - A relocation register and a limit register define a logical address space
  - Address protection with relocation and limit registers





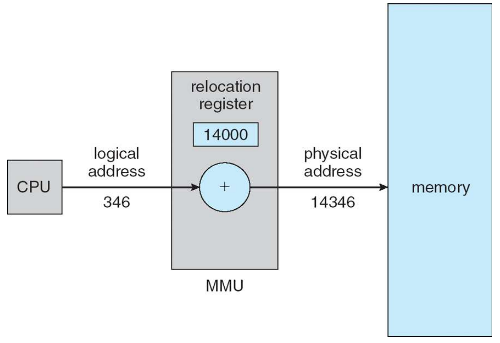
Address Binding

- Program on disk as binary executable file
  - ready to be brought into memory to execute as process
  - And then system **loads** the process in physical memory
- **Memory addresses** represented in different ways at different stages of a program’s life
  - Source code addresses usually **symbolic**
    - e.g., “int **count**”
  - Binary code needs **address binding** between *logical* and *physical* addresses:
    - e.g., “14 bytes from beginning of this program” (logica l) vs. 74014 (physical)
- When should we do address binding?



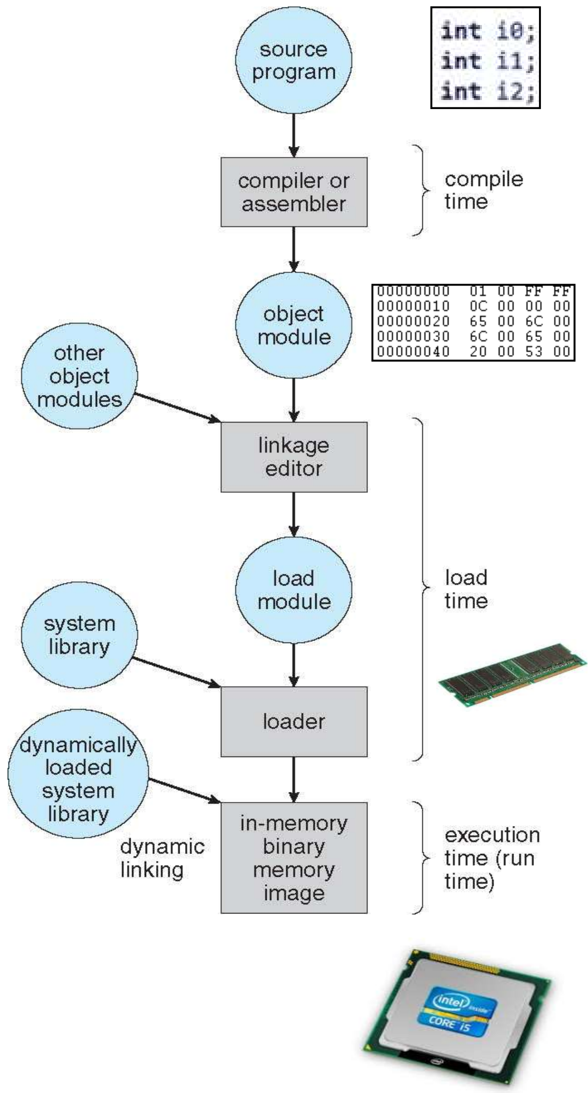
```
#include "myMult.h"

void myMult(const double a[12], const double b[20], double c[15])
{
  int i0;
  int i1;
  int i2;
  for (i0 = 0; i0 < 3; i0++) {
    for (i1 = 0; i1 < 5; i1++) {
      c[i0 + 3 * i1] = 0.0;
      for (i2 = 0; i2 < 4; i2++) {
        c[i0 + 3 * i1] += a[i0 + 3 * i2] * b[i2 + (i1 << 2)];
      }
    }
  }
}
```



Binding of Instructions and Data to Memory

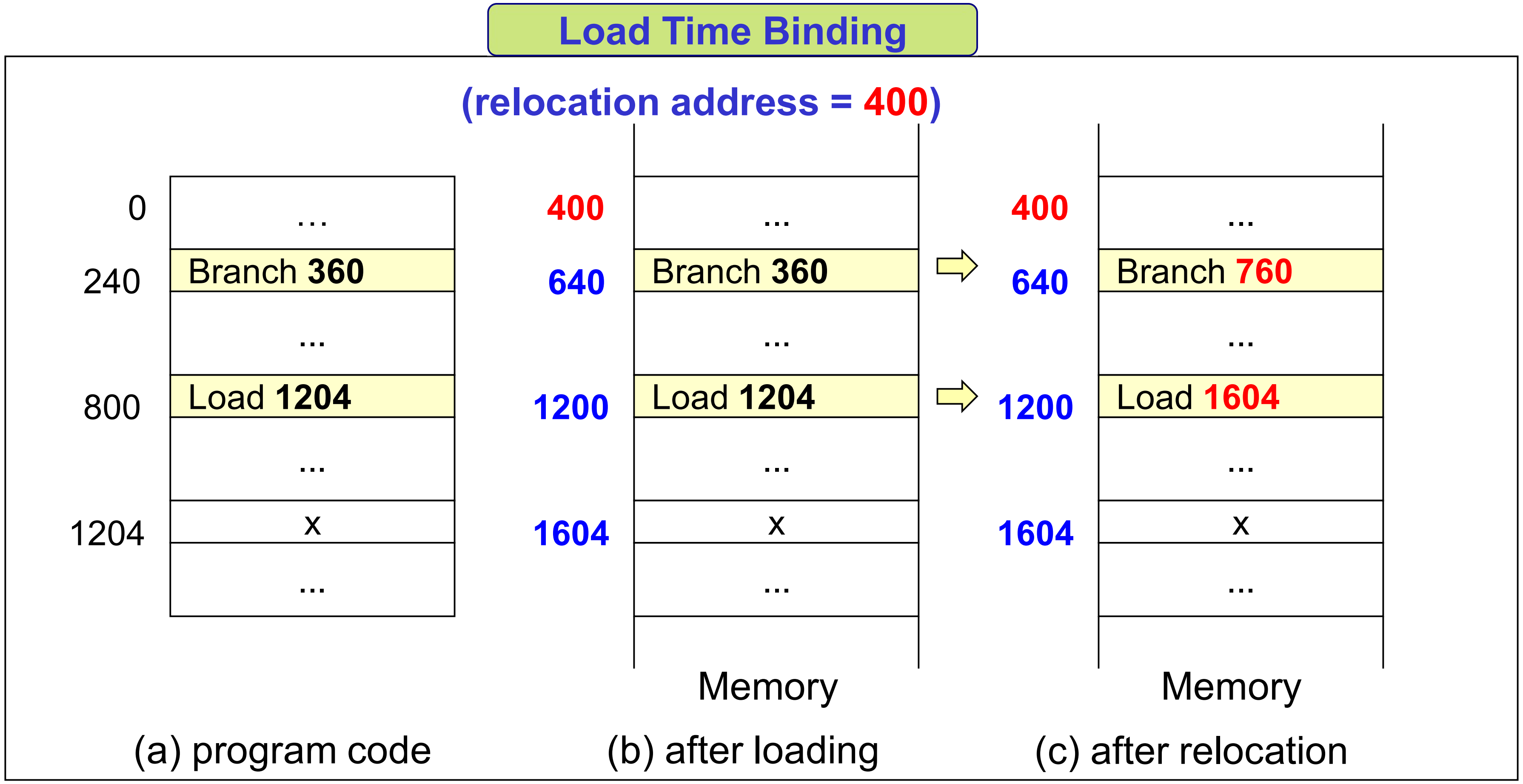
- **Address binding** can happen at three different stages: compile time, load time, execution time
- **Compile time**
  - If memory location known at compile time, **absolute code** can be generated
    - e.g., user process will reside starting at location R_1, compiled code will start at R_1
  - If starting location changes (e.g., R_2), must **recompile** code
- **Load time**
  - If memory location is not known at compile time, compiler must generate **relocatable code**
  - Address binding is delayed until **load time**
  - If starting address changes, need to **reload** user code



```
int i0;
int i1;
int i2;
```

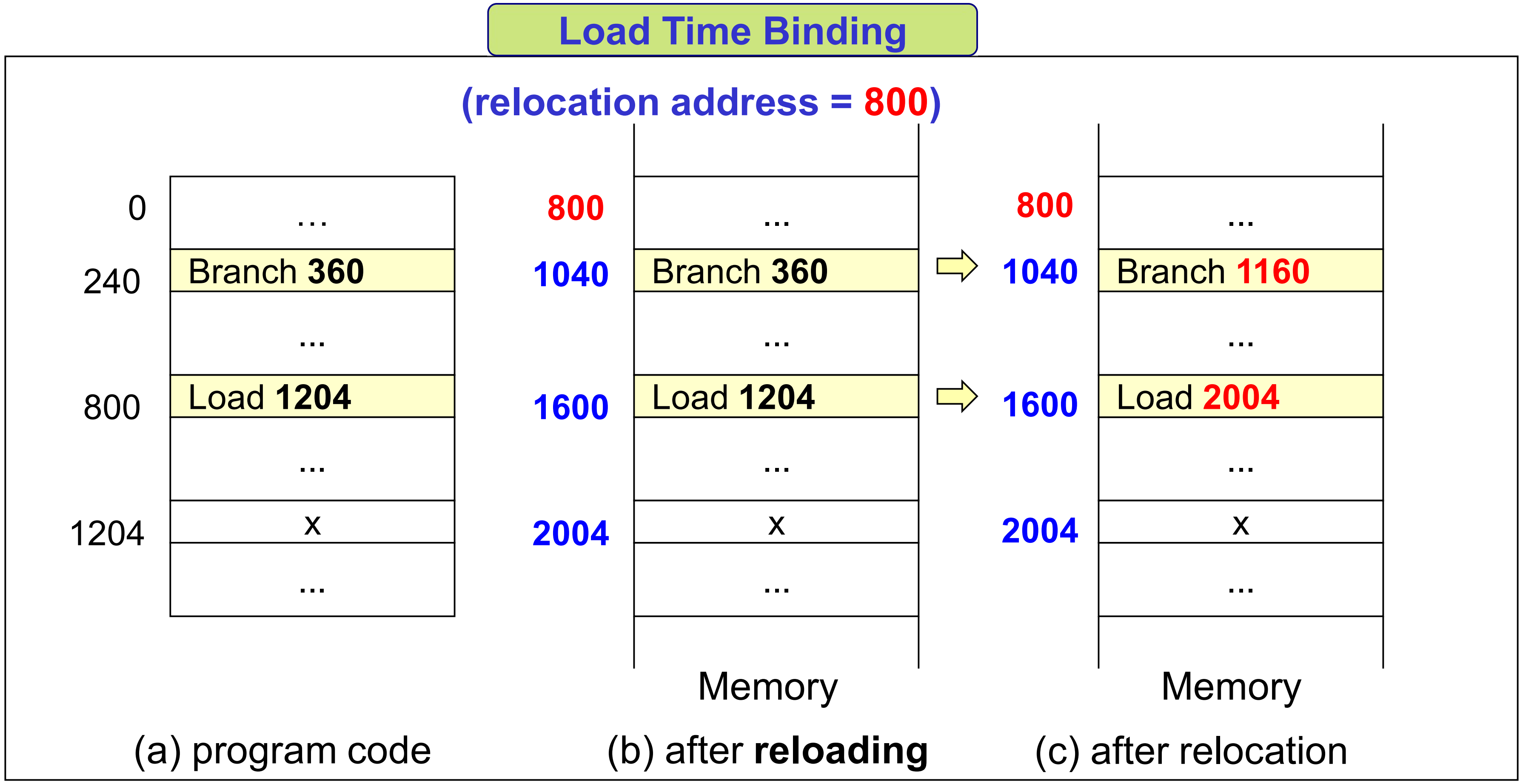
```
00000000   01 00 FF FF
00000010   0C 00 00 00
00000020   65 00 6C 00
00000030   6C 00 65 00
00000040   20 00 53 00
```

Example: Load time binding



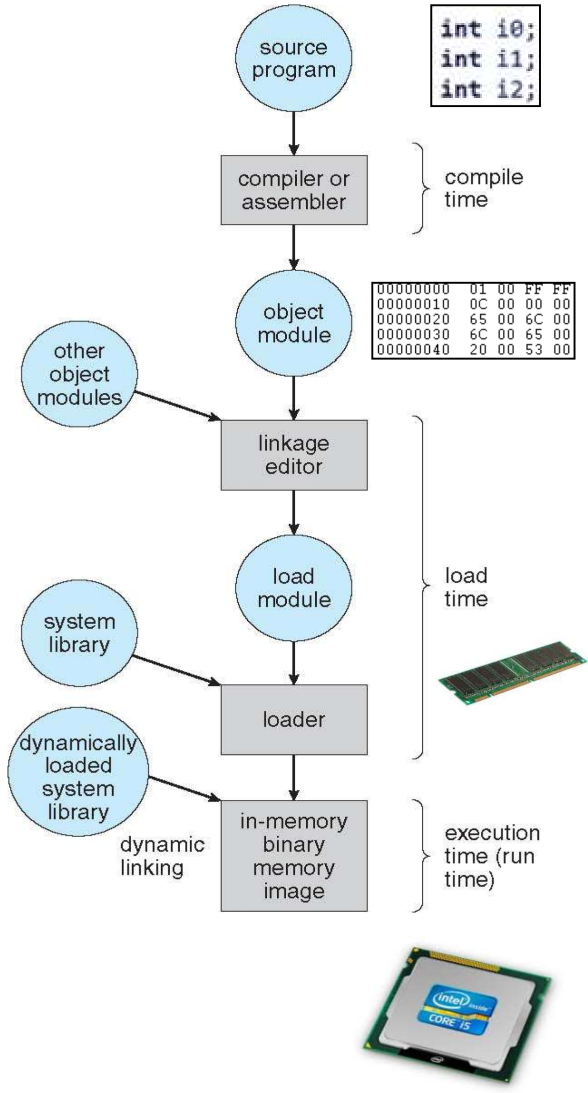
Example: Load time binding

Q: What if relocation address changes to 800? Need to reload

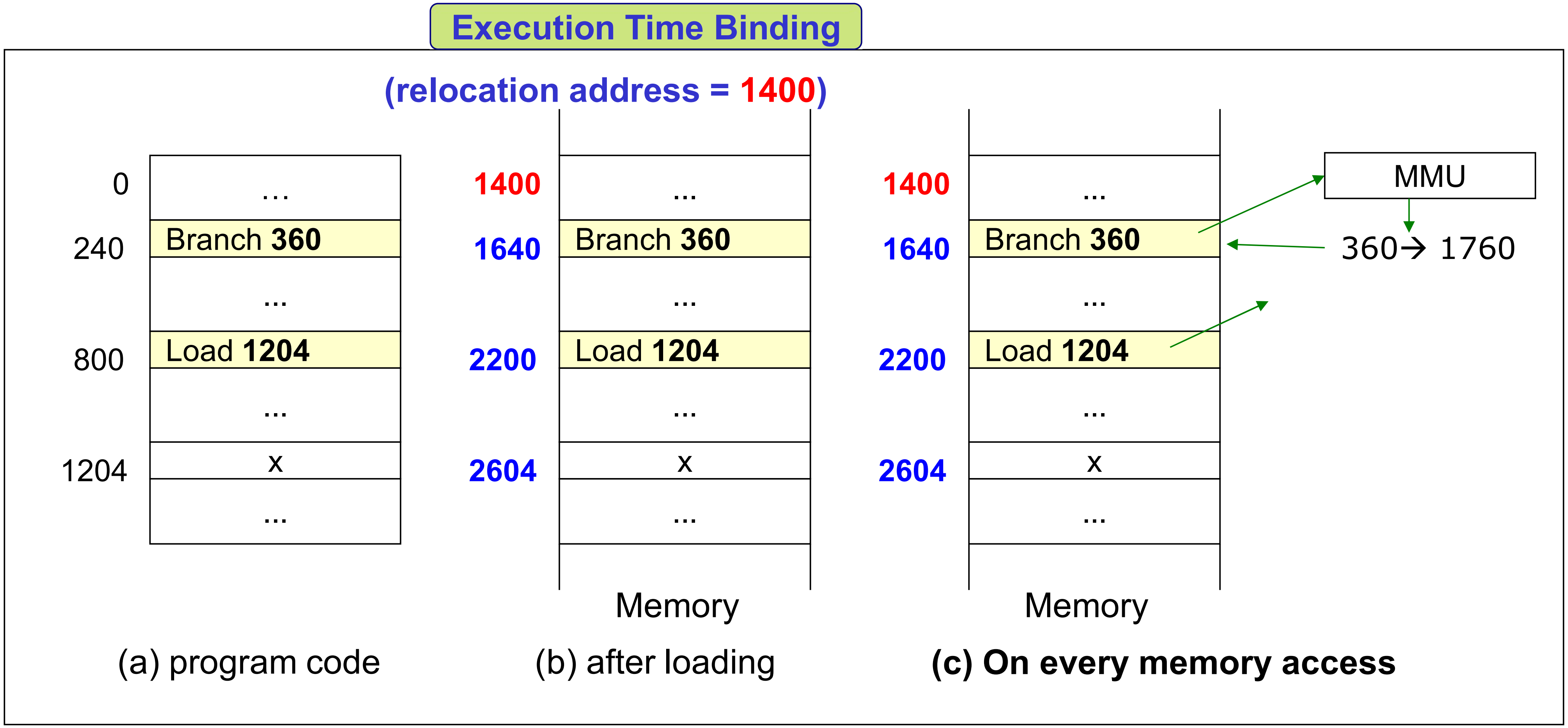


Binding of Instructions and Data to Memory

- **Execution time**
  - process memory location can be moved during execution
  - Address binding delayed until **run time**
  - Need hardware support for address maps (e.g., MMU such as base and limit registers)
  - Most OSs uses this method

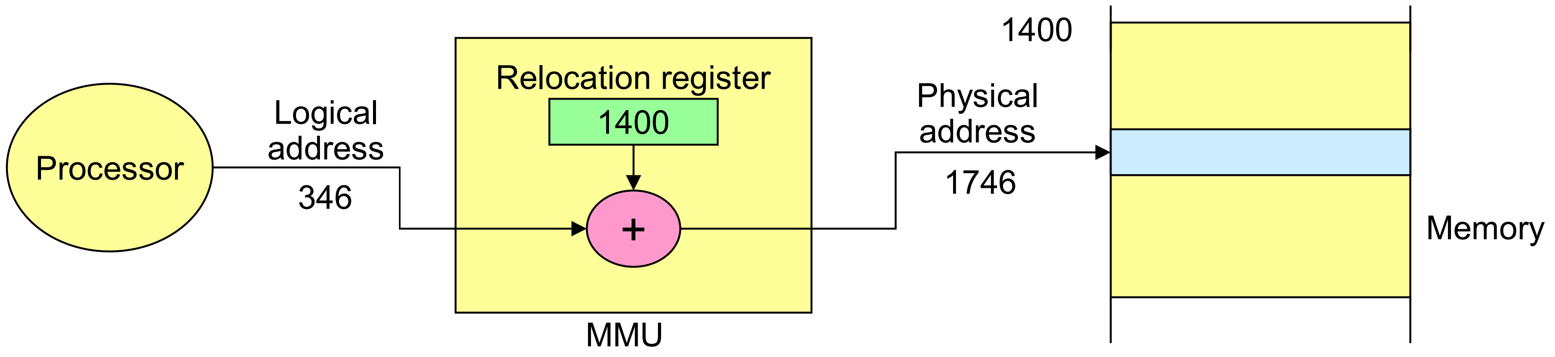


Example: Execution (Run) time binding



Dynamic Relocation

- **Run-time binding**
  - Run-time mapping <u>from</u> logical(virtual) address <u>to</u> physical address is performed by **MMU**(Memory Management Unit)



Q: What if relocation address changes to 1800 in previous slide?

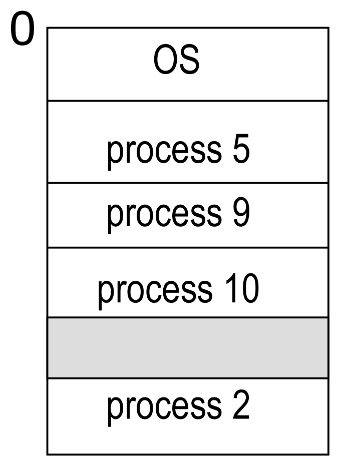
Chapter 8: Memory Management Strategies

- Background
- **Contiguous Memory Allocation**
- Paging
- Structure of the Page Table

Memory Protection

- **Memory Allocation**
  - Allocate memory to OS and user processes in an efficient way possible

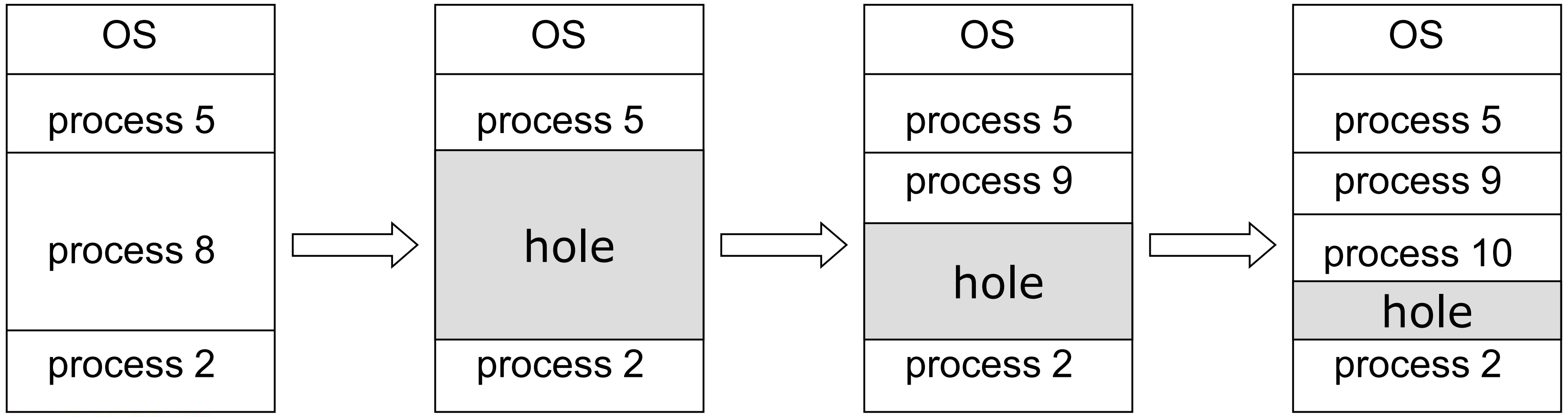
- Main memory usually into two partitions:
  - Resident **operating system**
    - usually held in low memory with interrupt vector
  - User **processes**
    - held in high memory



- **Contiguous Memory Allocation**
  - Allocate available memory to processes
  - Each process is contained in a single section of **contiguous** memory

Memory Allocation: Contiguous Allocation

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)



Contiguous Allocation: Example

- Example

Memory allocation and partition scenario

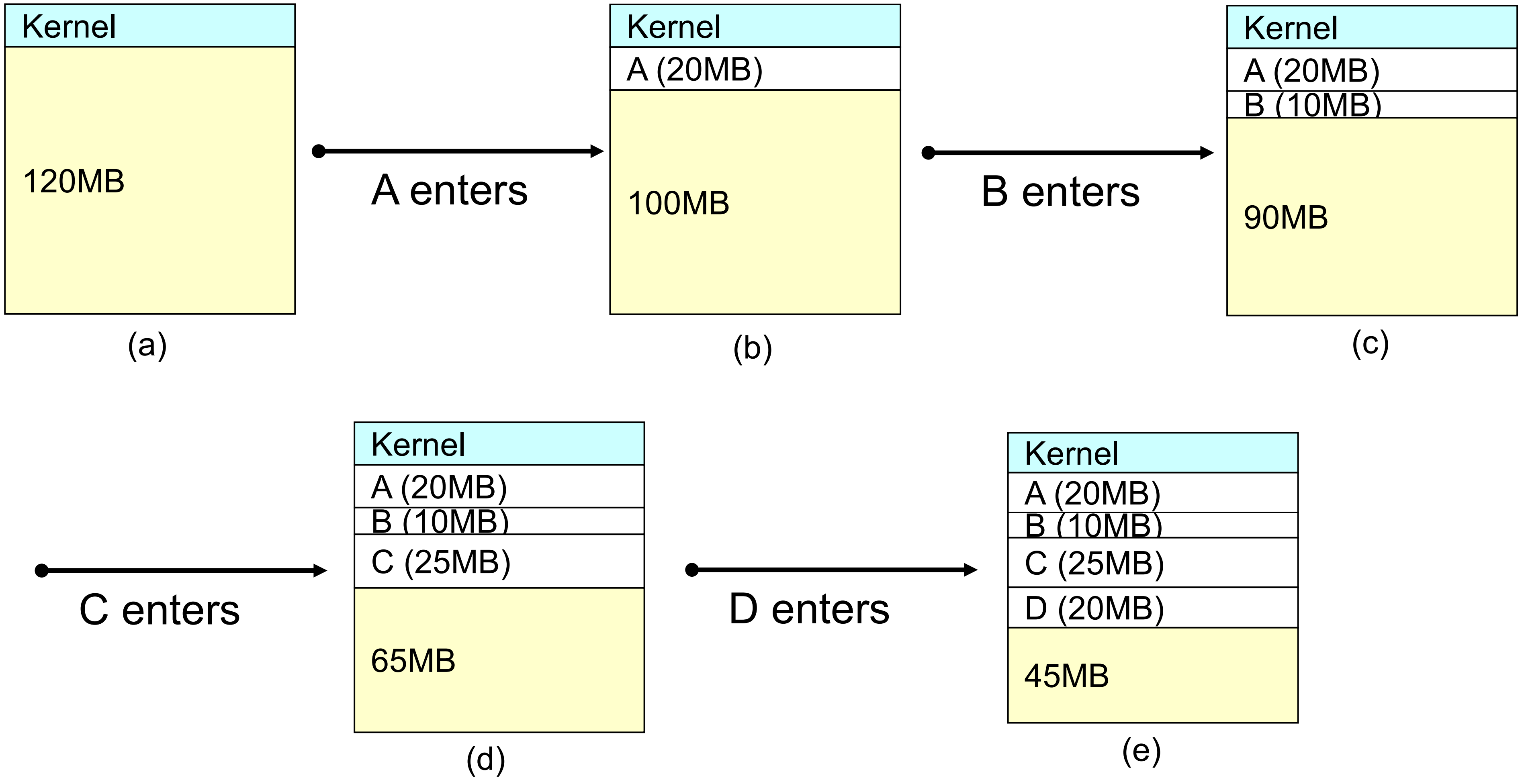
- Assumption

  - Memory space: 120 MB

- (a)	: Initial state
- (b)	: After loading process A(20MB)
- (c)	: After loading process B(10MB)
- (d)	: After loading process C(25MB)
- (e)	: After loading process D(20MB)
- (f)	: After process B releases memory
- (g)	: After loading process E(15MB)
- (h)	: After process D releases memory

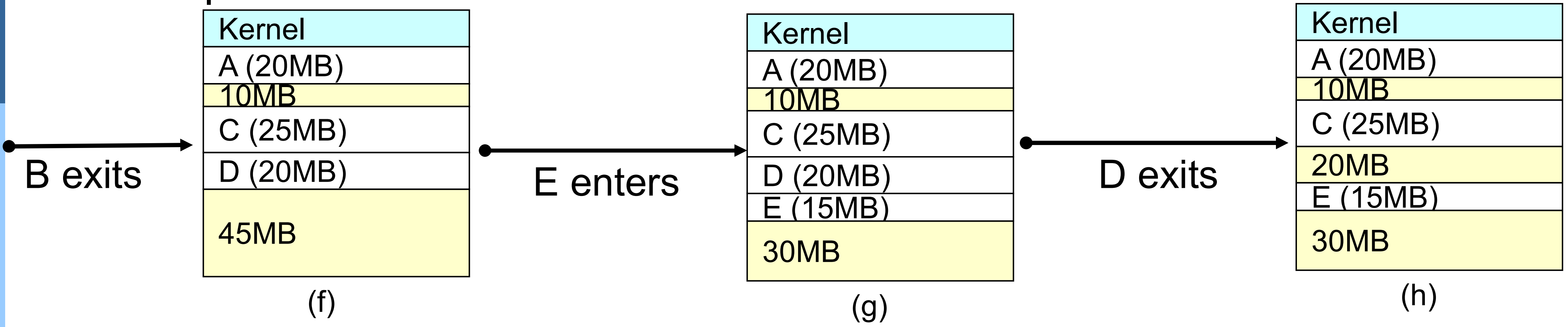
Contiguous Allocation: Example

- Example



Contiguous Allocation: Example (Cont'd)

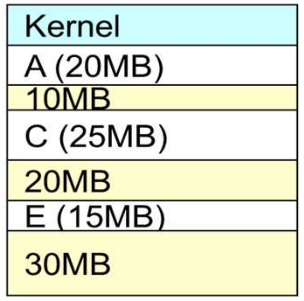
- Example



- ***Hole*** **– block of available memory; holes of various size are scattered throughout memory**

External Fragmentation

F (40MB)?



- **External Fragmentation**
  - Total memory space exists to satisfy a request, but it is not contiguous
- Analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to external fragmentation
  - 1/3 may be unusable -> **50-percent rule**
- We will discuss **internal fragmentation** later

External Fragmentation Solutions?

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
    - **NOTE:** We cannot preempt memory space if logical memory is determined in load time
- Another possible solution to the external-fragmentation problem
  - Permit the logical address space of the processes to be ***noncontiguous***
  - Allowing a process to be allocated physical memory wherever such memory is available

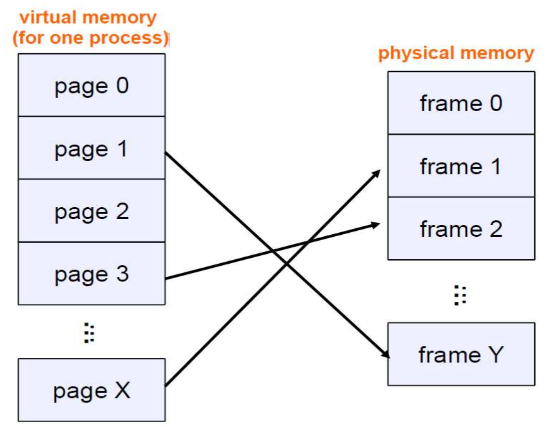
Chapter 8: Memory Management Strategies

- Background
- Contiguous Memory Allocation
- **Paging**
- Structure of the Page Table

Paging : Basic Method

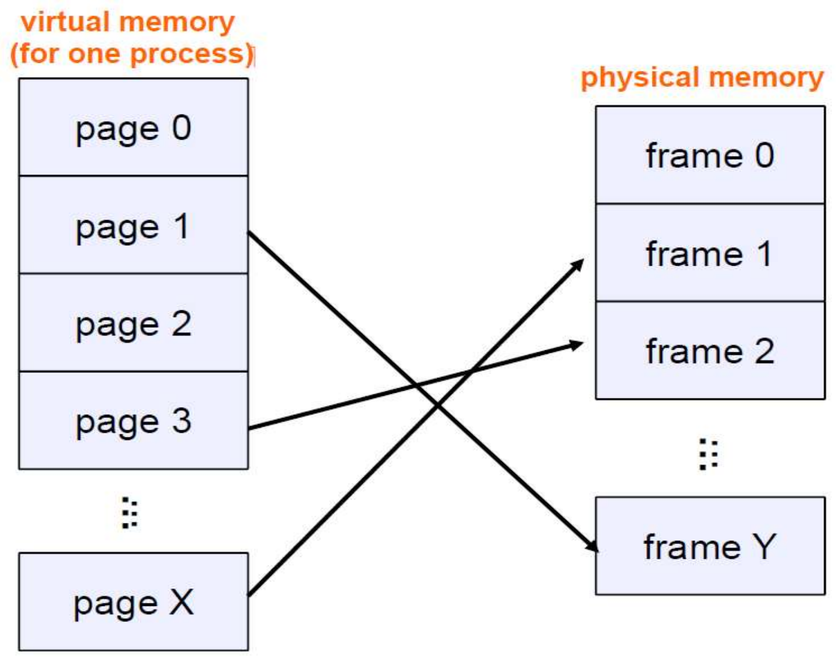
- **Paging**
  - ***Noncontiguous memory management*** *scheme*
    - No external fragmentation – no need for compaction
- Basic method
  - Divide
    - **physical memory** → *fixed-sized blocks :* **frames**
    - **logical memory** → *blocks of same size* : **pages**
    - size of one frame = size of one page

  - Set up a **page table** to translate logical to physical addresses

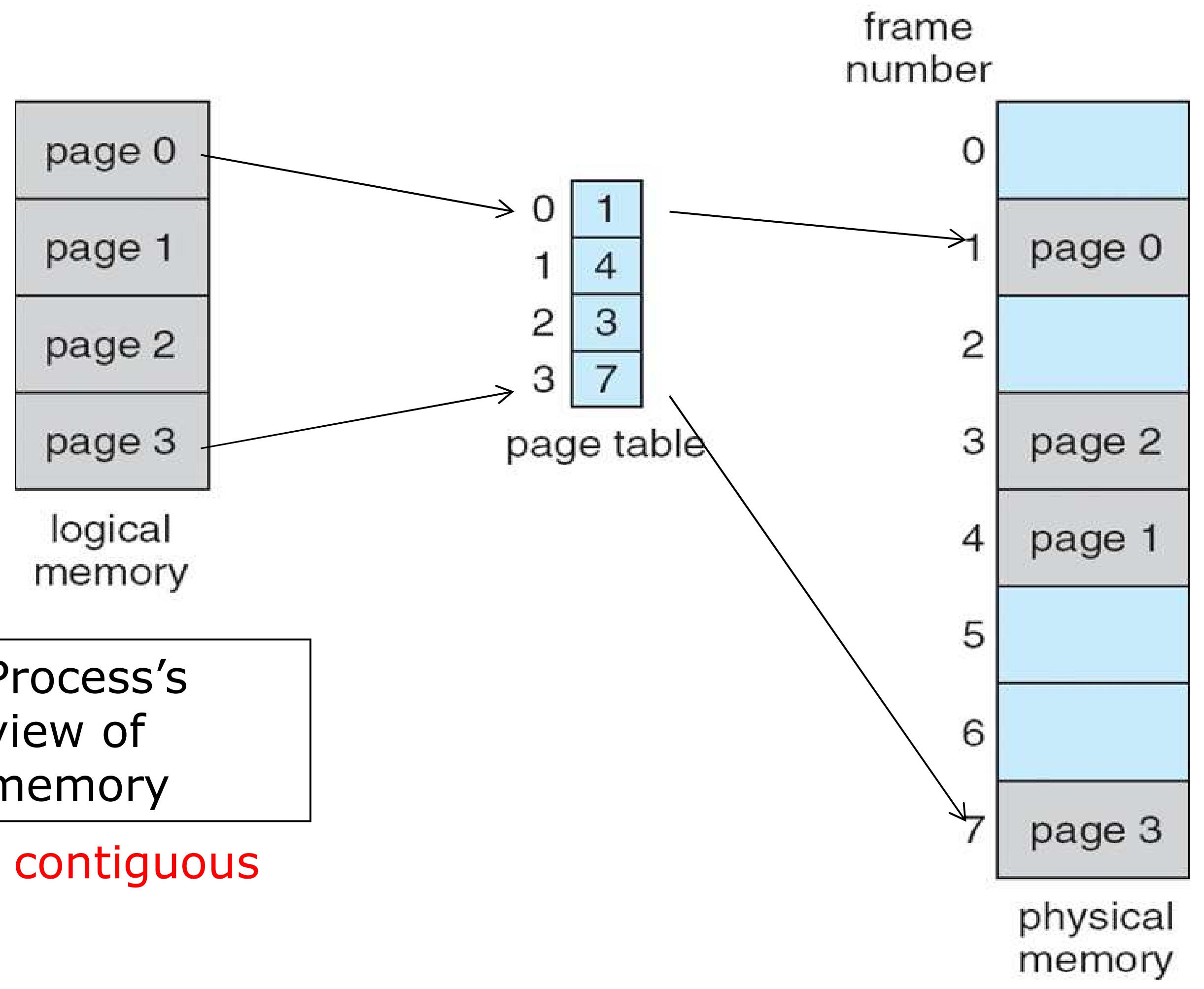


Cont'd.

- Solve the external fragmentation problem by using **fixed size chunks** of virtual (**page**) and physical memory (**frame**)



Paging Example



Process's view of memory

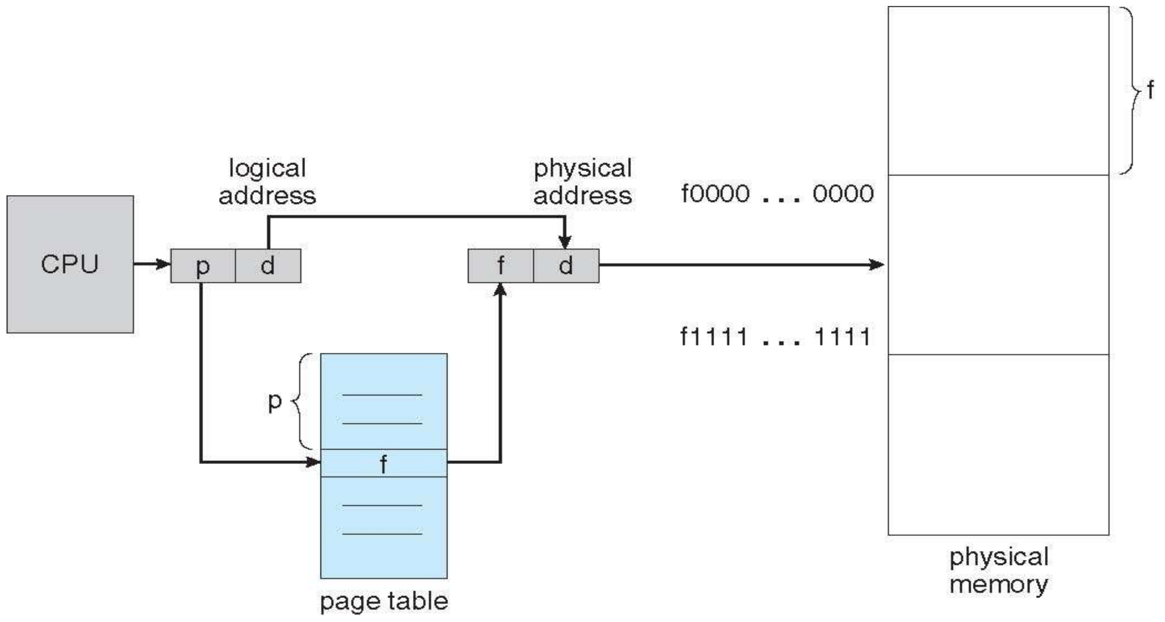
contiguous

Actual physical memory

Noncontiguous

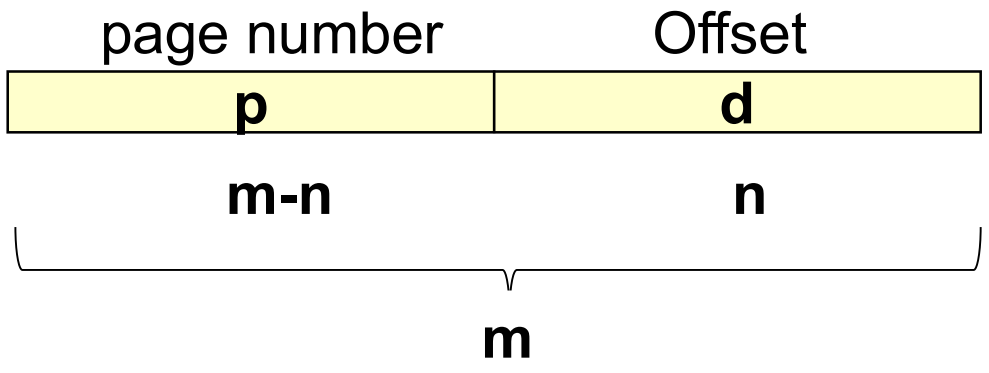
Page Number and Offset

- When a process is executed, its pages are loaded into any available memory frames
- Every address generated by CPU is divided into two parts: **page number** (**p**) and **page offset** (**d**)
  - **page number** (**p**): index into a **page table**
  - **page table**: contains **base address** of each page in physical memory
  - **page offset** (**d**): page number and page offset combines to define the physical memory address



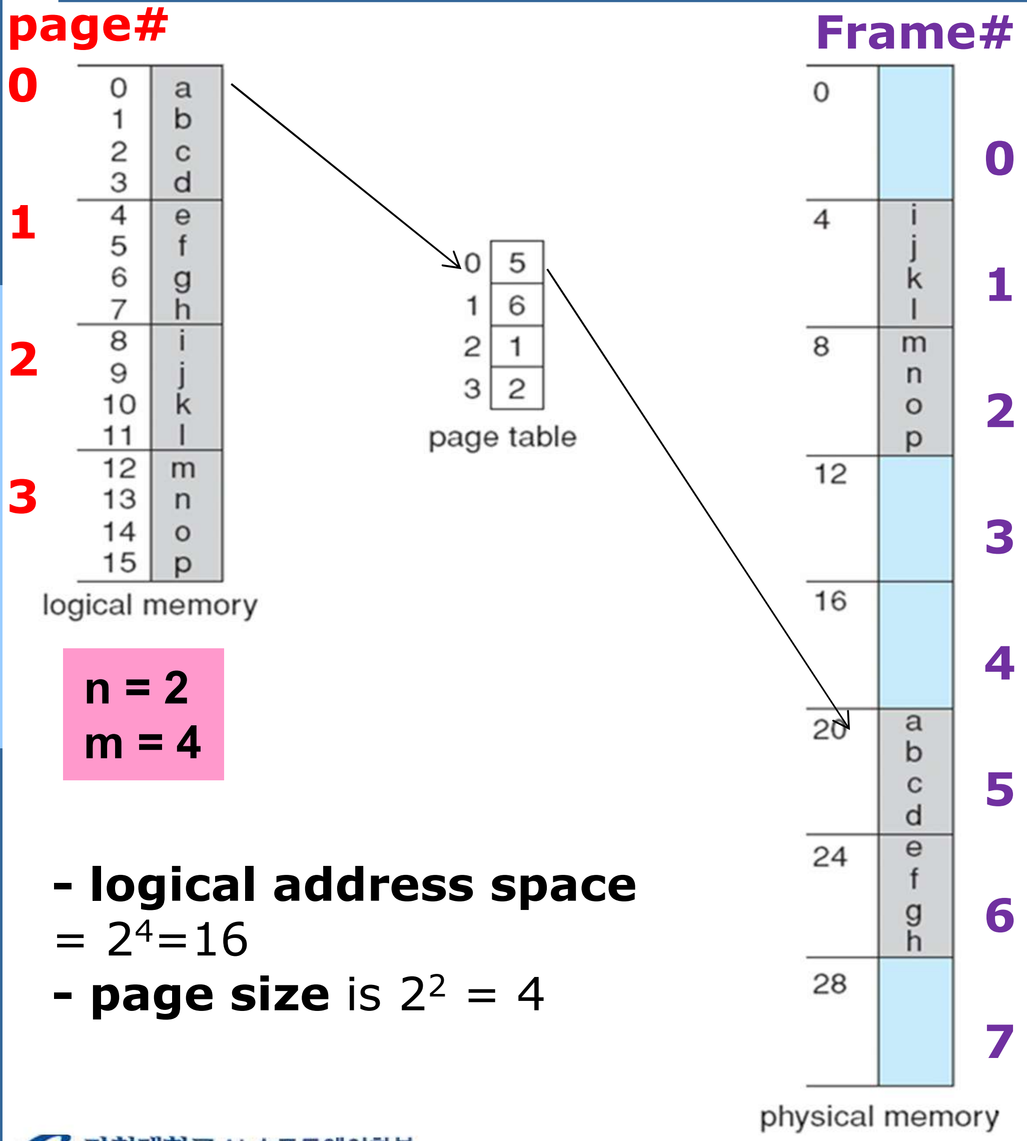
Page Size

- Fixed sized chunk: **page size** (=frame size) defined by hardware
  - vary between 512B (=2^9B) and 1GB (=2^{30}B) per page – depending on computer architecture
- If the size of **logical address space** is 2^m and **page size** is 2^n
  - **page number(p)**: high-order m-n bits of a logical address: **index** into a *page table*
  - **page offset(d)**: low-order n bits of a logical address



page size?

Paging in bits Example



n = 2
m = 4

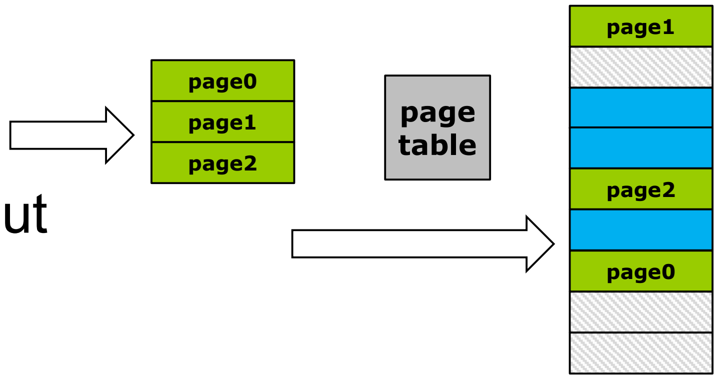
- **logical address space** $= 2^4 = 16$
- **page size** is $2^2 = 4$

- m = 4, n = 2
  - Logical address space: 2^m = 16 bytes
  - Page size : 2^n = 4 bytes
  - Page number: 2^{m-n} = 4 bytes

- **Example**
  - Logical address 0
    - ✓ Page #0 → Frame #5, offset# 0
    - ✓ physical address = (5x4)+0 = 20 **data "a"**
  - Logical address 1, 11?
    - ✓ Page# ? → Frame #? offset?
    - ✓ physical address? data?

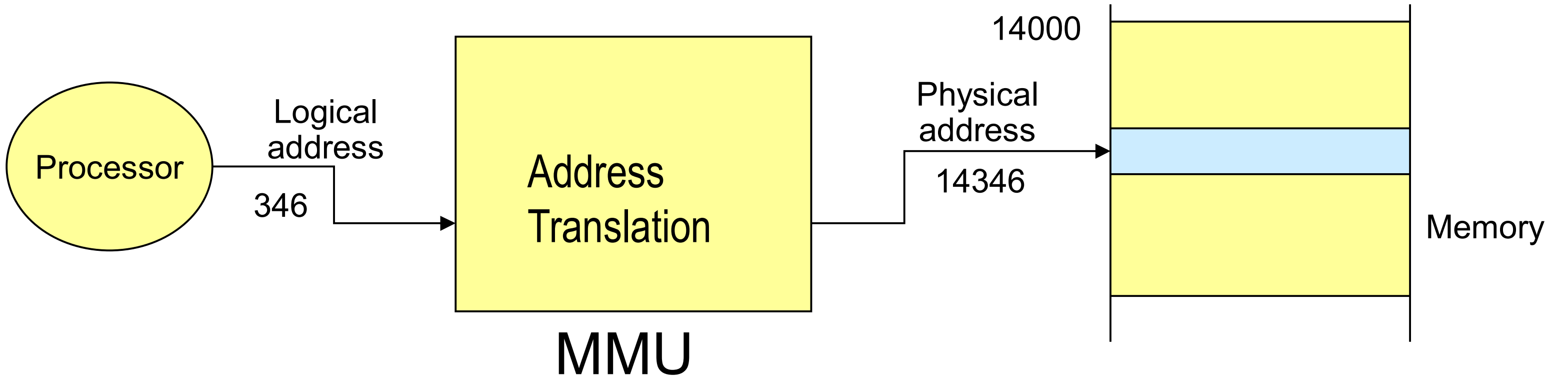
Why Paging? (1)

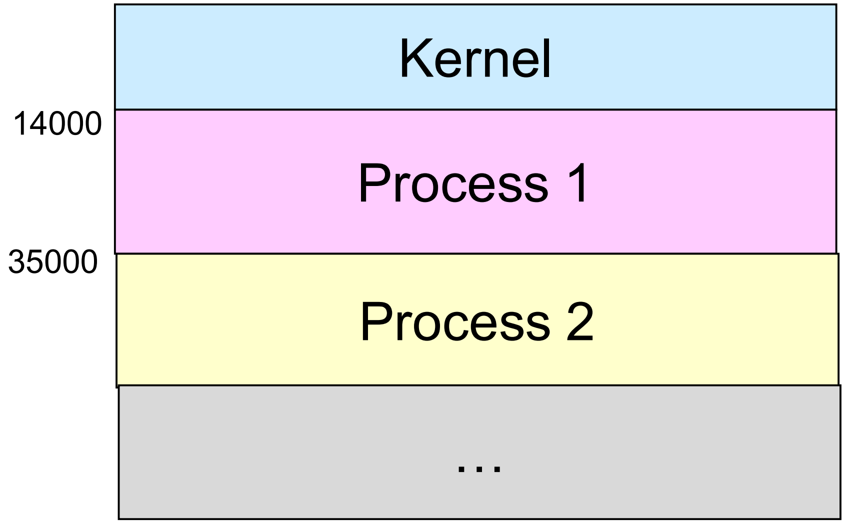
- Clear separation between the programmer's view of memory (logical memory) and physical memory
  - Programmer: One contiguous single space
  - Actual physical memory: scattered throughout the physical memory
  - Needs ***page table*** for address translation

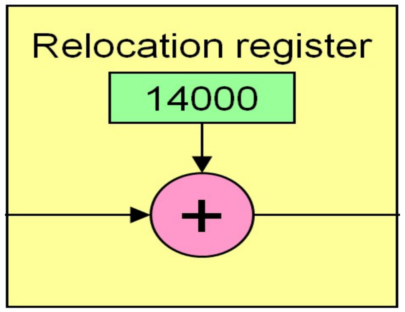


1. Provides **Transparency**
   - Programmer only sees *logical address*, never sees actual *physical address* – *physical address* is transparent to programmer
   - This mapping is *hidden* from programmer and controlled by OS

Contiguous vs. Paging

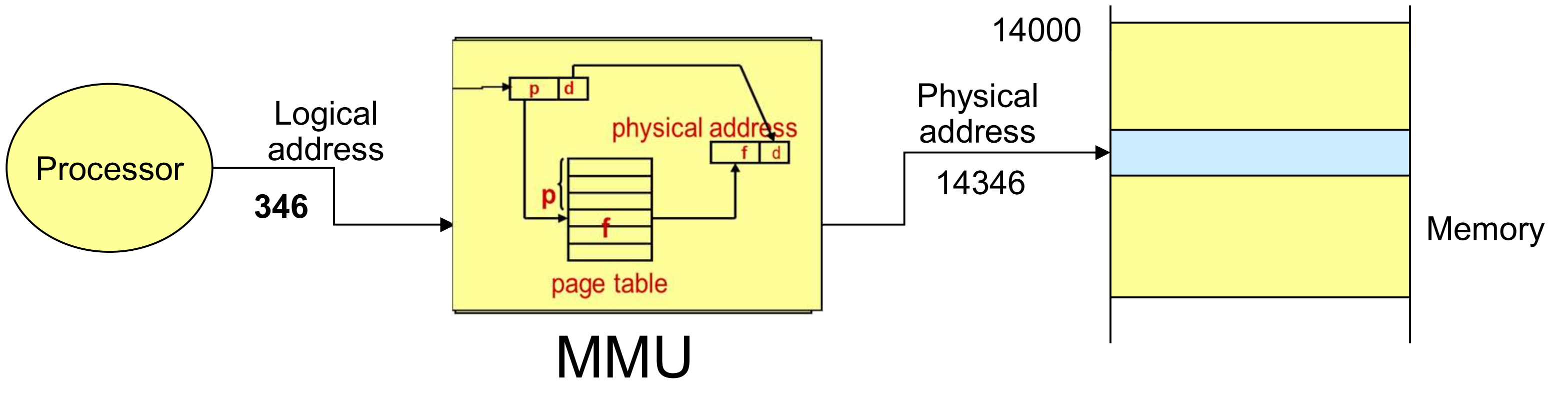


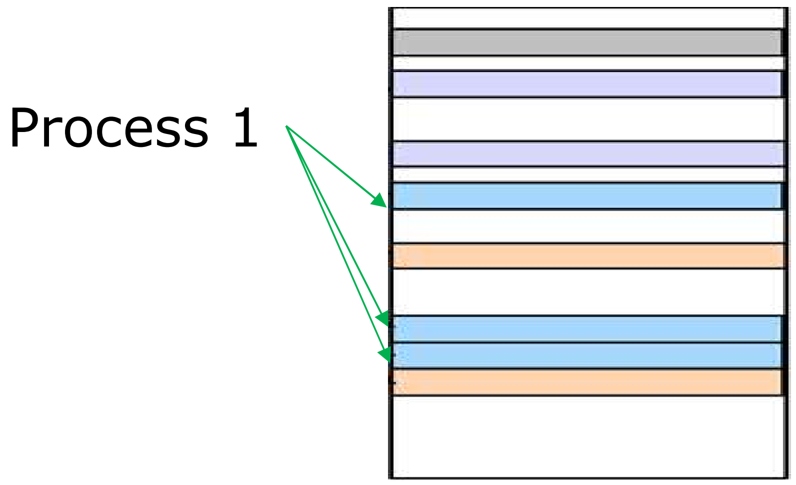


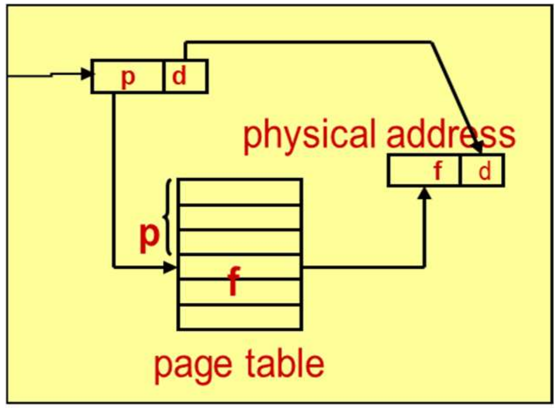


MMU for Contiguous Allocation

Contiguous vs. Paging



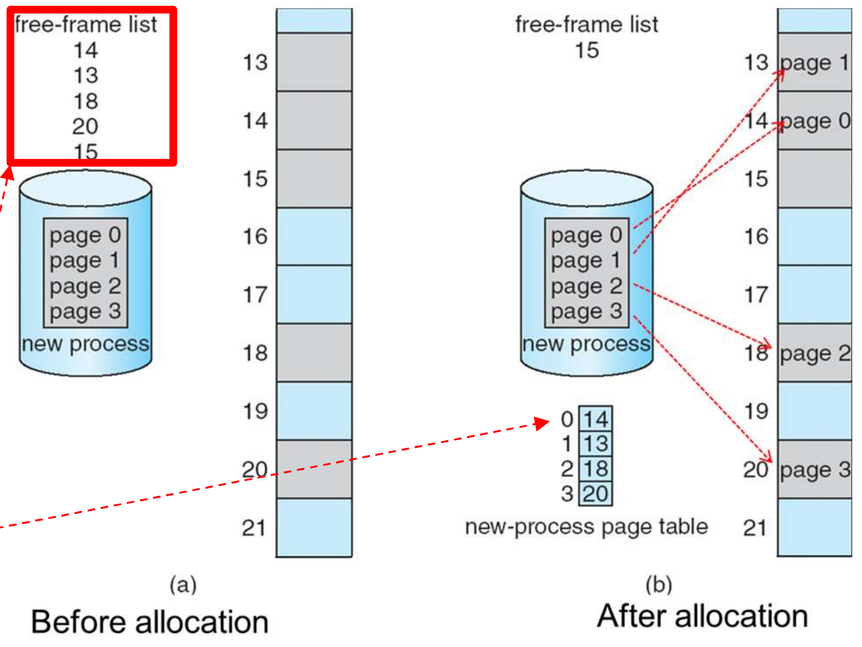




MMU for Paging

Free-frame management

- To execute a process whose size requires **n** pages, at least **n** free frames are required – How do we find the **free frames**?
  - The first page of the process is loaded into the first frame listed on **free-frame list**
  - and the allocated frame number is put into **page table**
  - The next page is loaded into another frame …



A new process with 4 pages

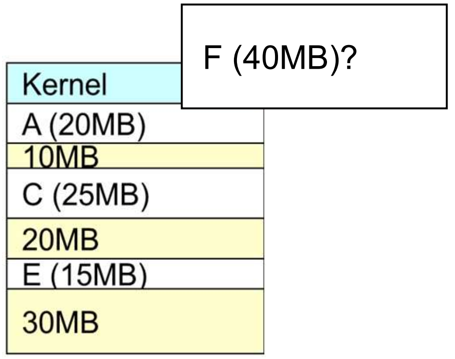
Why Paging? (2)

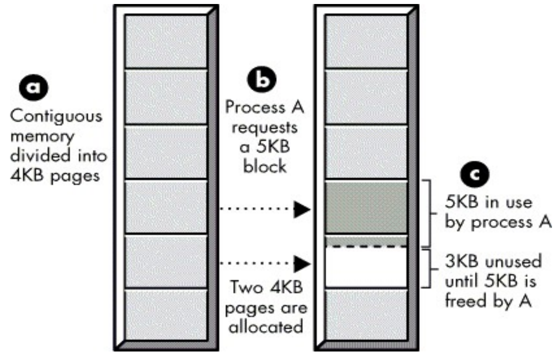
2. **No external fragmentation**

- Noncontiguous memory allocation - ANY free frame can be allocated to a process that needs it.
- However, we may have **internal fragmentation**
  - e.g.) if a page size is 2048 bytes, a process of 72766 bytes would need 35 pag es plus 1086 bytes
  - In worst case, a process would need n pages plus 1 byte, and it would be alloc ated n+1 frames – internal fragmentation in last frame

Internal Fragmentation

- **External Fragmentation**
  - Contiguous allocation: Total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**
  - Memory is allocated in fixed-sized **pages**
  - Allocated **pages** may be larger than requested memory;
    - page size 4kB,
    - process needs 9kB – allocated 3 pages (12kB)
  - this size difference is memory internal to a partition, but not being used
- How do we determine the Page Size?
  - Consider **tradeoff**: Internal fragmentation, page table size, disk I/O efficiency
  - Practically 4 – 8KB in size



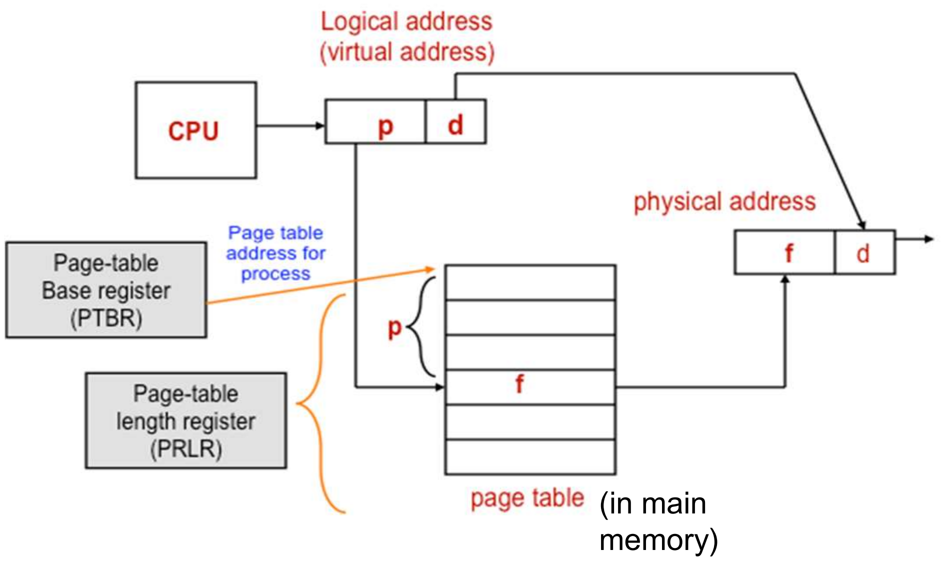

a
Contiguous memory divided into 4KB pages
b
Process A requests a 5KB block
Two 4KB pages are allocated
c
5KB in use by process A
3KB unused until 5KB is freed by A

Implementation of Page Table

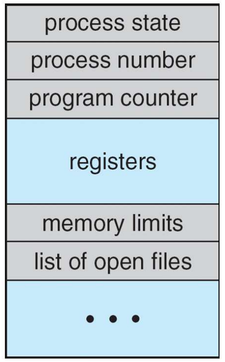
- There is a **page table** for **each process**!!
  - e.g., process P1’s page table, P2’s page table, ...
- **Option 1**: Keep in registers and make if fast
  - Implement page table as **registers**: translation very fast but expensive – need t o keep it small (e.g., 256 entries)
  - But page table can be very large! (e.g., 1 million entries)
- **Option 2:** Keep in memory with register help
  - **Page table** is kept in **main memory**
  - Use **CPU registers** to store location and size of page table – hardware support

Implementation of Page Table

- **Page table** is kept in **main memory** with **hardware support**:
  - ***Page-table base register*** (**PTBR**)
    - points to the page table for its process
  - ***Page-table length register*** (**PTLR**)
    - size of the page table (for the process)


Logical address (virtual address)
CPU
p
d
physical address
f
d
Page table address for process
Page-table Base register (PTBR)
p
f
Page-table length register (PRLR)
page table
(in main memory)

- When context switch happens,
  - Need to change the page table changing page tables requires changing the **PTBR** and **PTLR**

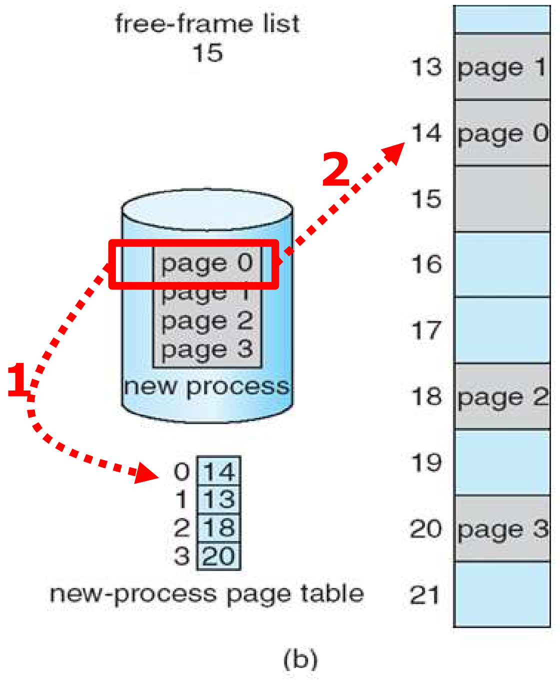


Page Table in Main Memory

- **Page table** is kept in **main memory**: Problems?
  - Every data/instruction memory access *actually* requires **two memory accesses**
    - 1. page table   2. data/instruction
  - Example
    - Process wants to read memory at logical address 010 (page 0, offset 10)

    1. First read page table to look for physical frame number of page 0 (=14)

    2. Read frame number 14



Page Table in Main Memory

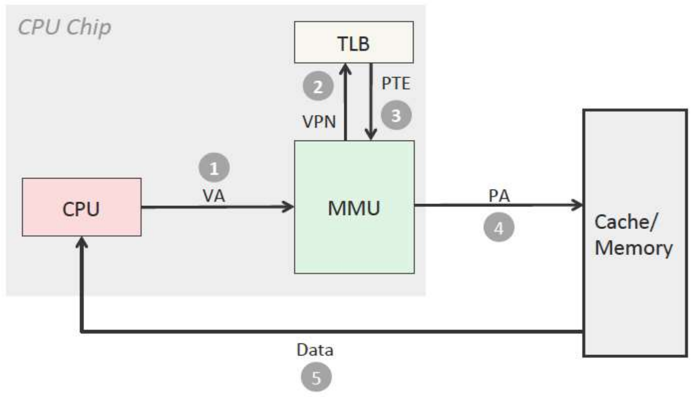
- Isn't it slow to have to go to memory twice every time?
  - It takes about 100~400 CPU cycles to access DRAM
  - Yes, it would be… so, real MMUs don't

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
  - **TLB** is like a cache to the page table
    - e.g., cache memory is a cache to the main memory (RAM)
  - Store recently referenced page table entries in the TLB

TLB (Translation Look-aside Buffer)

- **Small, dedicated, super-fast hardware cache in MMU**
  - High speed memory technology, approaching register speeds
    - Implement using the dedicated registers or cache memories
    - Parallel search
  - Each TLB entry contains a page number and the corresponding page-tabl e entry
    - Search key = logical address, result = frame number

- Memory references that can be translated using TLB entries are much faster to resolve
  - high speed, but Expensive hardware

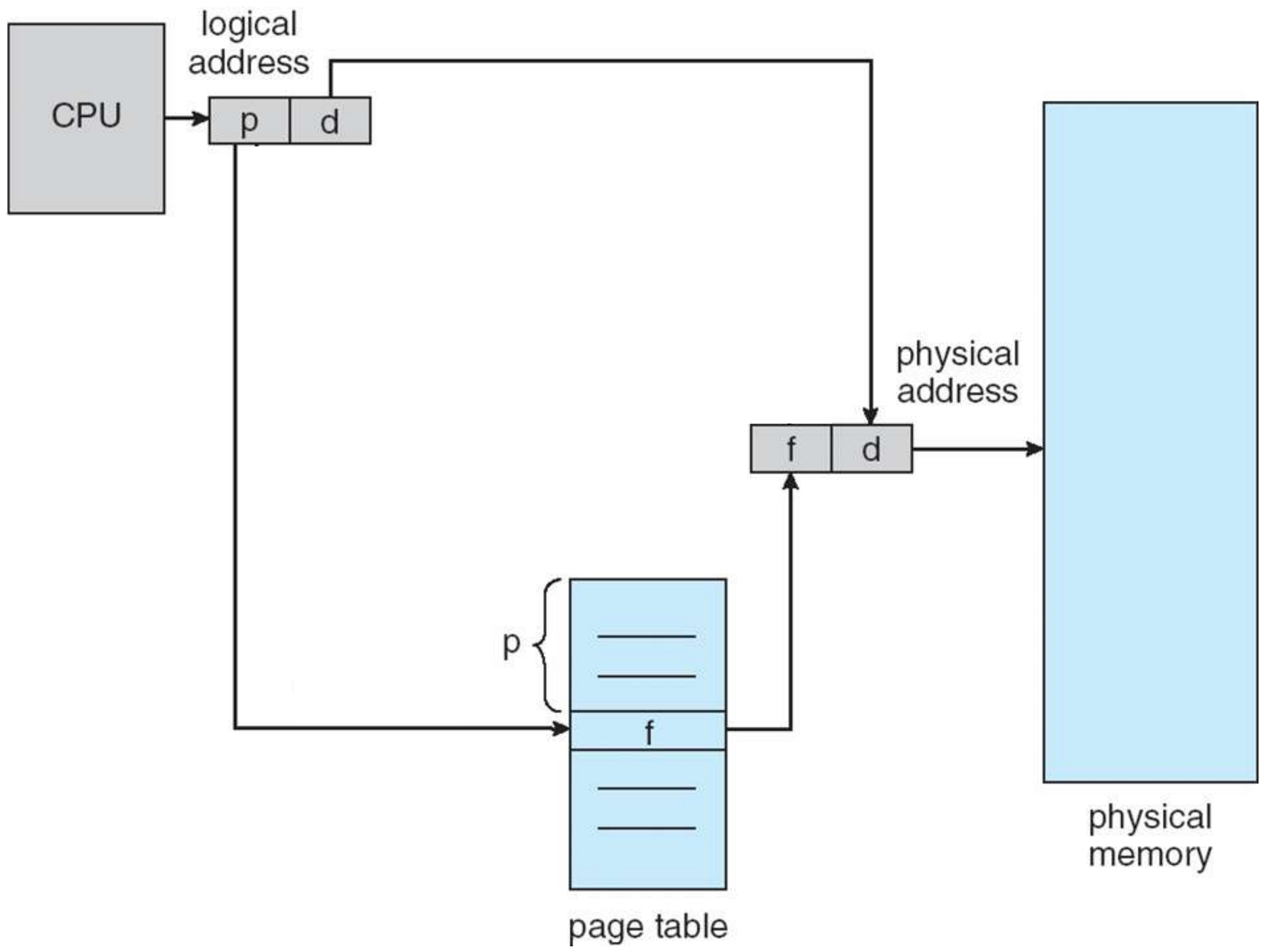


TLB

- **Recently-referenced** Page Table Entries (PTE's) are stored in the TLB
  - only small part of the page table is kept in TLB, typically 32 ~ 1,024 entries

- **TLB hit**: memory reference is in TLB
- **TLB miss**: memory reference is not in TLB
  - A TLB miss incurs an additional memory access to the page table
  - In this case, the page number/frame number is *added* to TLB for next page reference

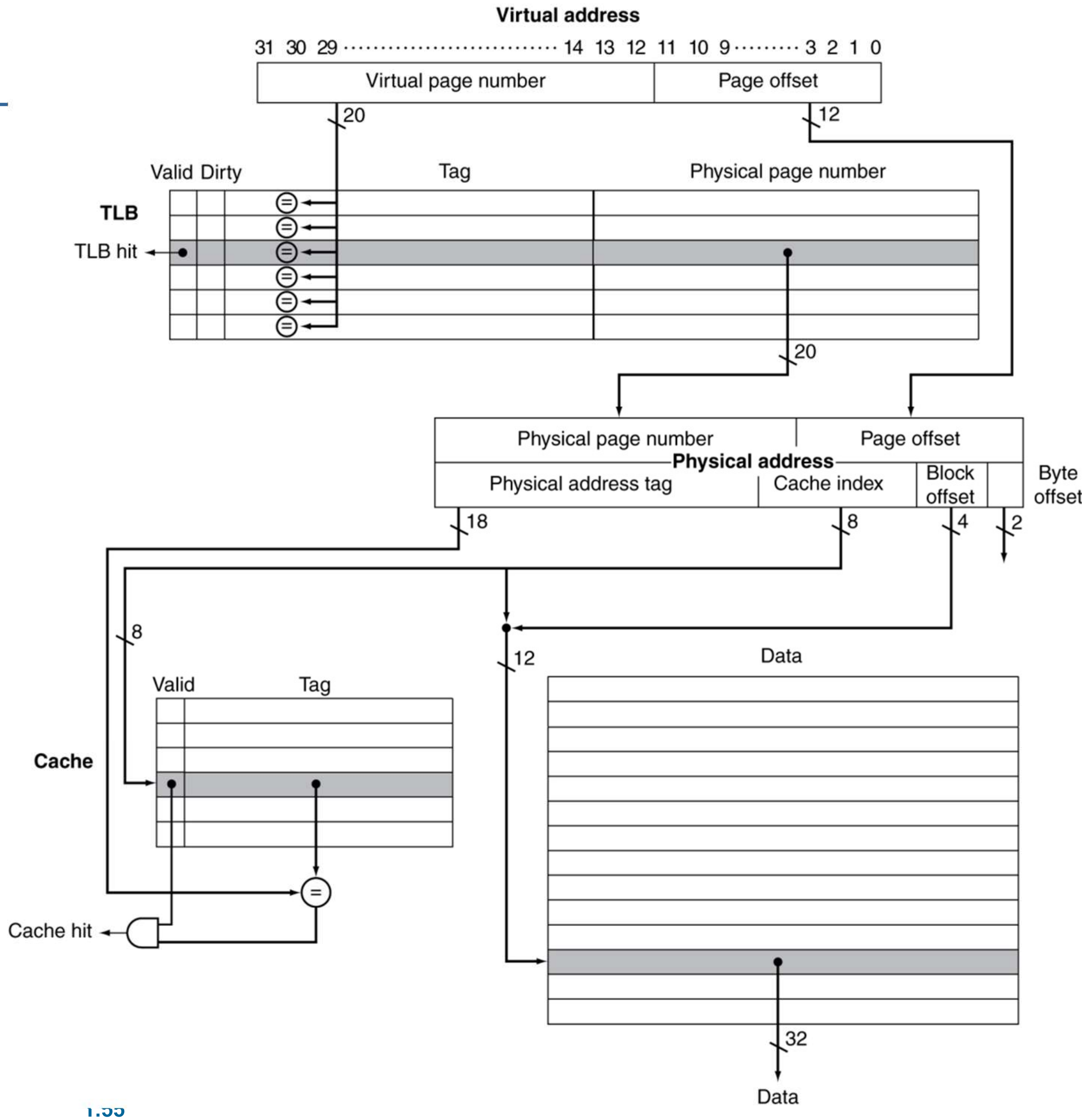
TLB



CPU
logical address
p
d
physical address
f
d
p
f
page table
physical memory

TLB with cache

- Memory access order
  - **Logical (virtual) address**
  - Check **TLB**
    - **TLB hit**: get address from TLB
    - **TLB miss**: check page table in memory
  - Acquire **physical address**
  - Check **cache**
    - **cache hit**: get data from cache
    - **cache miss**: get data from main memory
  - Acquire data



Effective Access Time (EAT)

- **Hit ratio**
  - percentage of times that a page number is found in **TLB**
  - e.g., 80% hit ratio: find the page in TLB 80% of the time
- **Assume**
  - memory access time = 100 ns, TLB access time = 1ns
  - Hit ratio = α
- **Effective Access Time (EAT)**

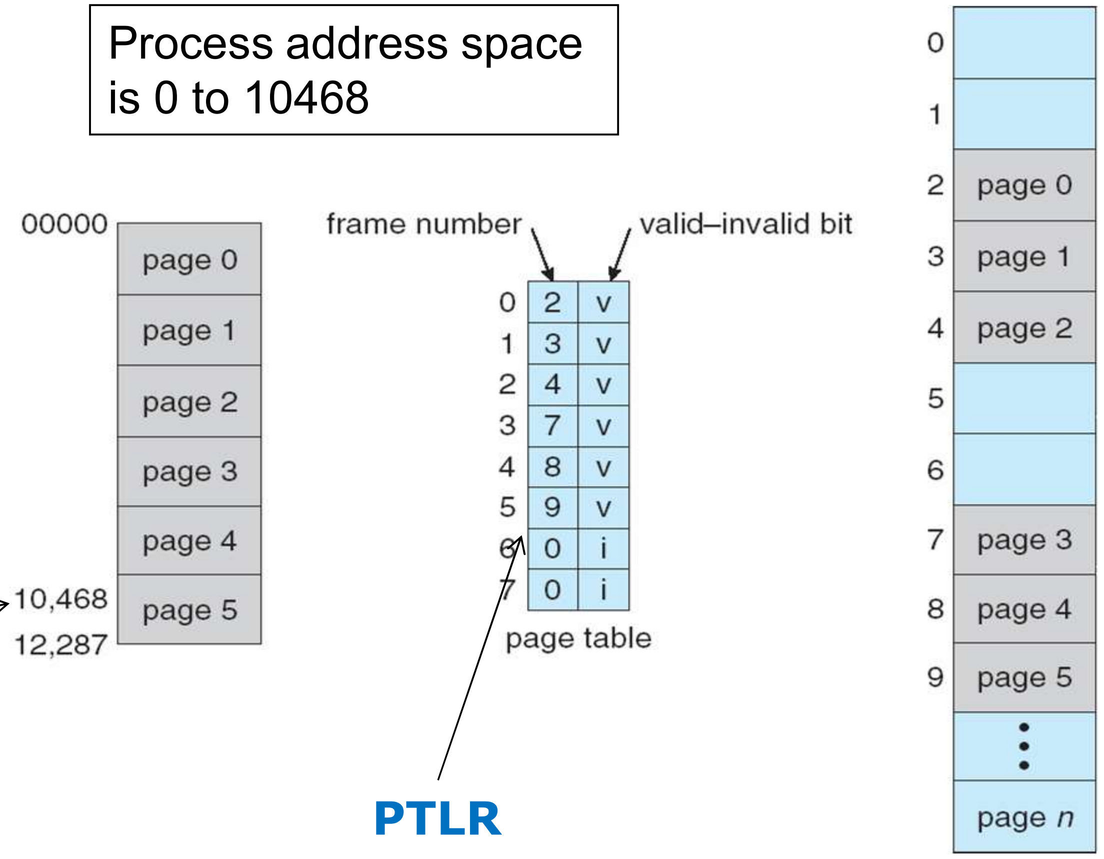
$$\textbf{EAT} = \alpha \text{ x } (\underbrace{1}_{\text{TLB}} + \underbrace{100}_{\text{data/instruction}})\text{ns} + (1 - \alpha) \text{ x } (\underbrace{1}_{\text{TLB}} + \underbrace{100}_{\text{page table}} + \underbrace{100}_{\text{data/instruction}})\text{ns} = 201 - 100\alpha$$

if α = 0.8 then EAT = 121ns, if α = 0.99 then EAT = 102ns

Memory Protection

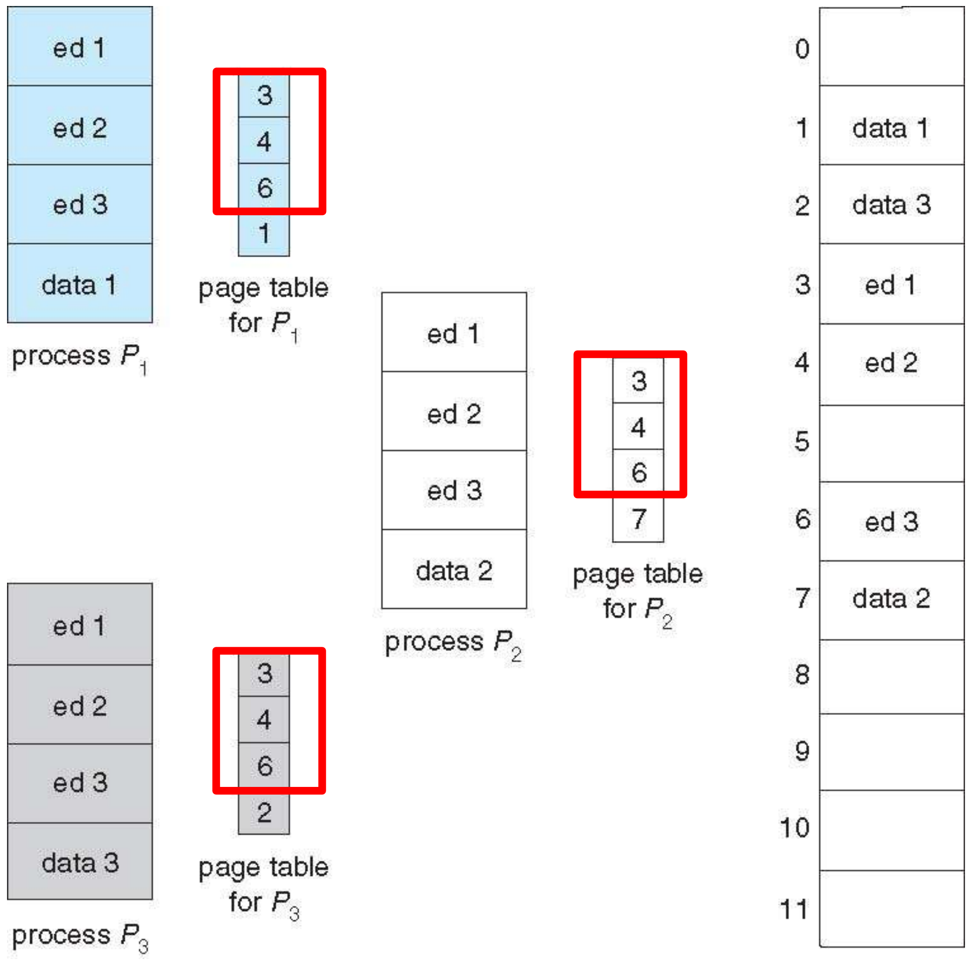
- **Valid-invalid** bit attached to each entry in the page table:
  - **valid bit** indicates that the associated page is in the process' logical address space, and is thus a *legal* page
  - **invalid bit** indicates that the page is *not* in the process' logical address space
- Internal fragmentation
  - page 5 is valid, but some parts of the page 5 are invalid

Process address space is 0 to 10468



Shared Page (Shared Memory)

- Shared code
  - One copy of code shared among processes
    - Example: 40 users run same program using 150KB code and 50KB data
      - need total 8,000KB physical memory
    - Share the code!
      - need total 2,000+150 = 2,150KB physical memory
  - IPC, Multiple threads sharing the same memory space



Why Paging? (3)

- Provides **Protection** to memory
  - Programmer can never access memory outside the page table.
  - Programmer can never access memory outside the page table length defined in **page-table length register (PTLR)**
- Provides **Shared memory**
  - Use shared pages

Chapter 8: Memory Management Strategies

- Background
- Contiguous Memory Allocation
- Paging
- **Structure of the Page Table**
  - Hierarchical Paging (Two-level page table)
  - Hashed Page Table
  - Inverted Page Tables

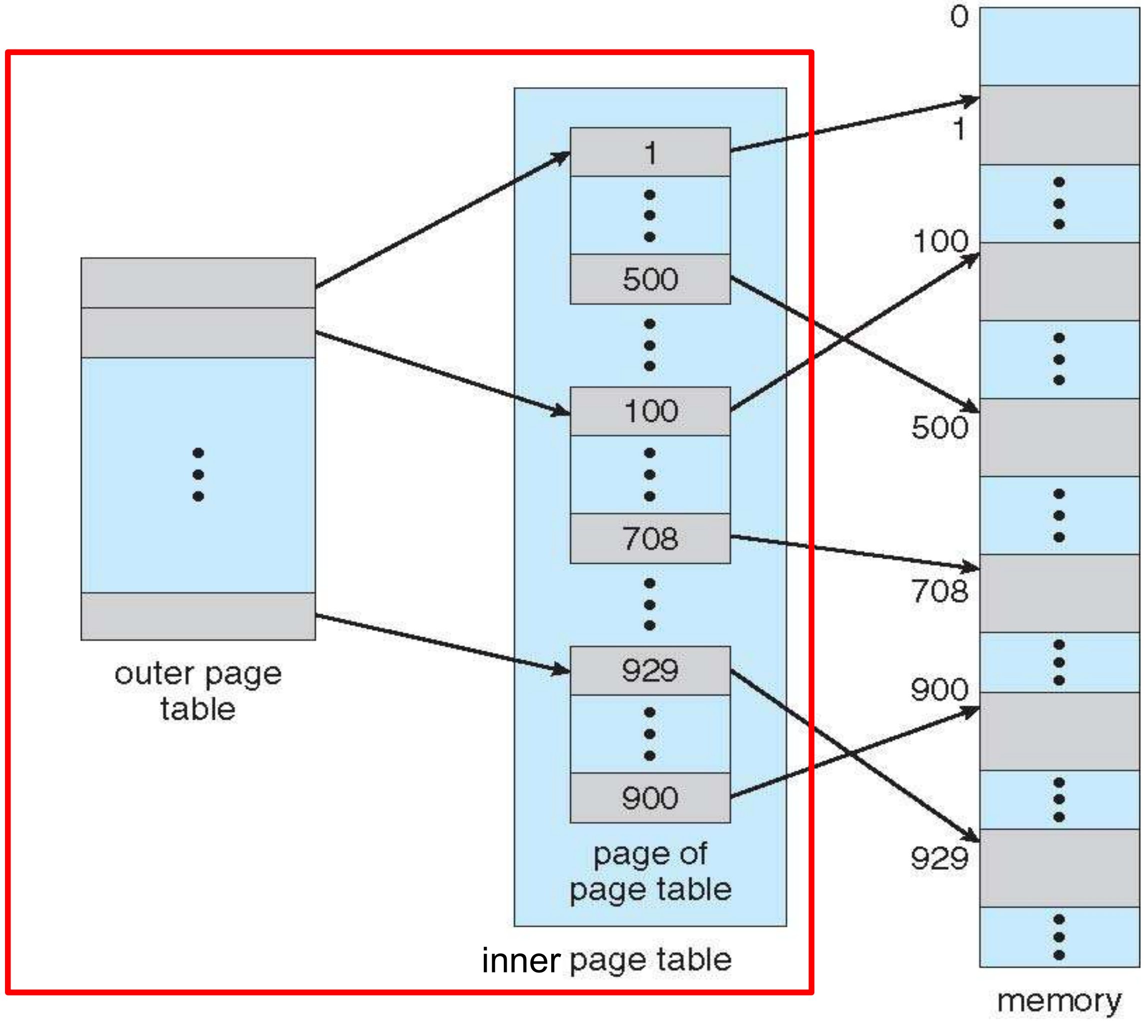
How to Store Page Table

- Consider a system with 32-bit logical address space
  - 32-bit (4GB) logical address space
  - If page size: 4KB (=2^{12}B)
  - Each page table needs up to 1 million (2^{20}) entries → 4MB ***contiguous*** **physical memory space** for page table
- Solution? **Hierarchical paging**
  - Multi-level paging, in which the page table is also paged

Hierarchical Page Tables

- Break up the logical address space into multiple page tables (logically)

- A simple technique is **a two-level page table**



Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - **a page number** consisting of 20 bits, **a page offset** consisting of 12 bits.
  - There should be $2^{20} \approx 10^6$ = 1 million page entries (4×10^6 = 4MB)
    - Page table becomes too large!

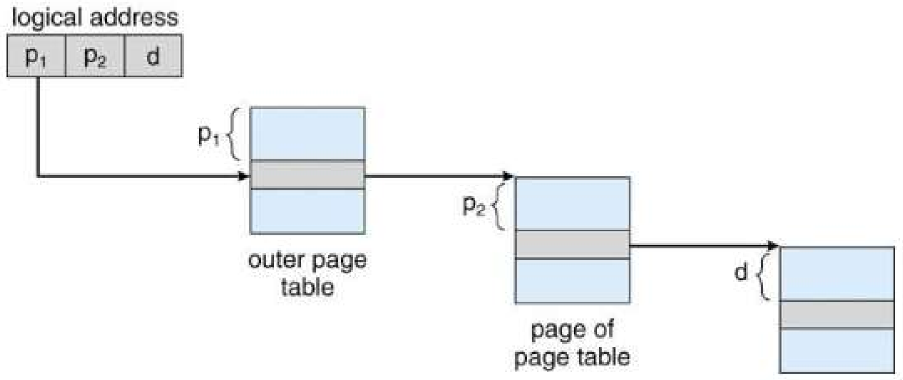
- Since **the page table is paged**, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.

Two-Level Paging Example

- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

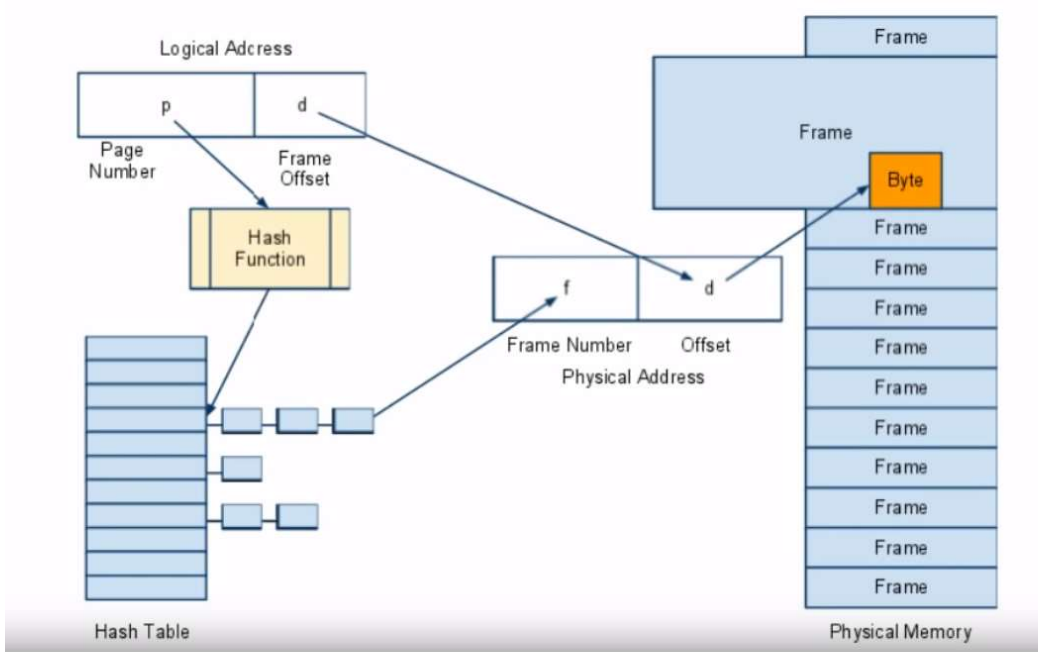
- where p_1 is an index into the **outer page table** (2^{10} = 1,024 entries: 4KB), and p_2 is the displacement within the page of the **inner page table** (2^{10} = 1,024 entries: 4KB).



Hashed Page Table

- Address spaces on 64-bit architectures are large
  - Page table search times can become excessive

- Hash table improves search times
- Page table stores linked list of page-to-frame mappings



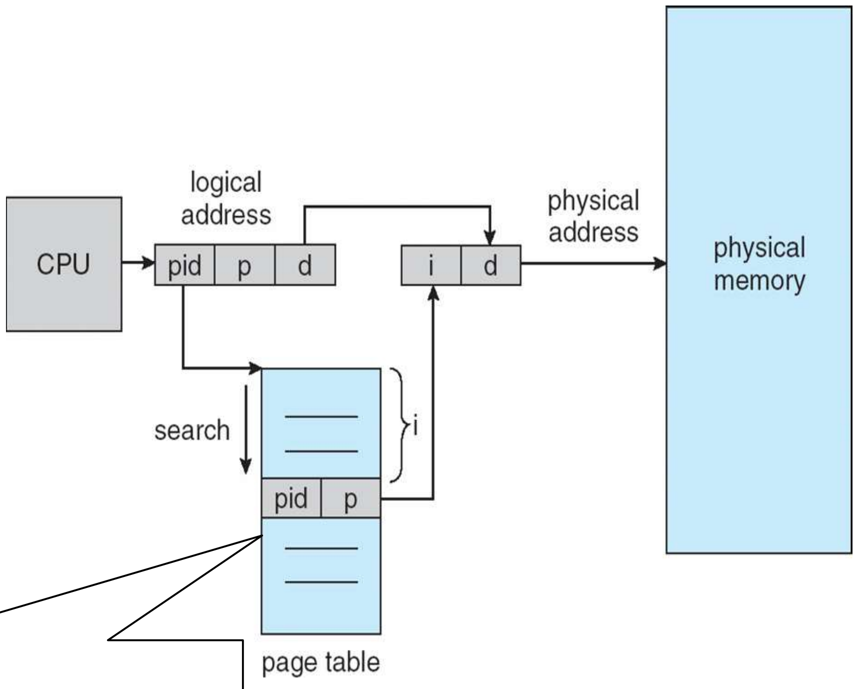
Inverted Page Table

- Usually, each process has a page table
  - Translate logical address (page) into physical address (frame)
  - Each page table can be very large! (as seen in last few slides)

- **Inverted page table**
  - Only one page table in OS!
  - One entry for each real frame of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

- Decreases memory needed to store each page table
  - Total table size = 1M entries (4MB size)

- Any disadvantages?



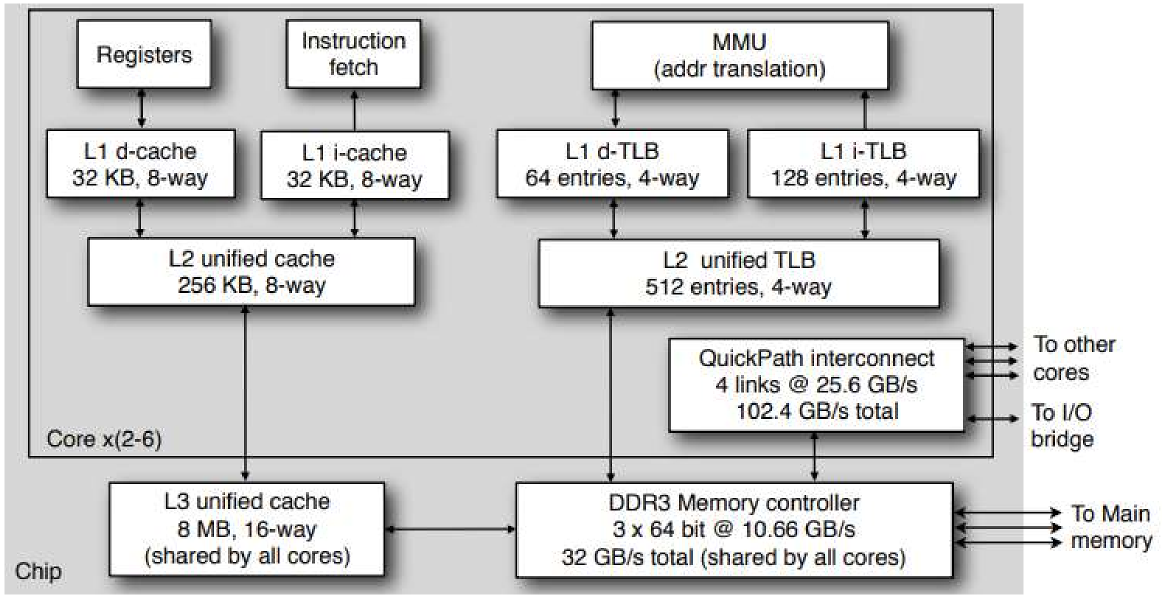
PT: page# → frame#
inverted PT: frame# → pid, page#

Inverted Page Table

- Increases time needed to search the table when a page reference occurs
  - If there are N entries, may need up to N searches!
- Solution
  - Use **hash table** to limit the search page-table entries
  - **TLB** can accelerate access
- Difficult to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Example: Intel Core i7

- Nehalem microarchitecture
  - Allows for full 64-bit virtual and physical address
  - Current implementations (Bloomfield, Lynnifield, etc.)
    - 48-bit virtual address (256 TB)
    - 52-bit physical address (4 PB)
  - Page size is either 4KB or 4MB (Linux uses 4KB)


Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L1 i-cache 32 KB, 8-way
L1 d-TLB 64 entries, 4-way
L1 i-TLB 128 entries, 4-way
L2 unified cache 256 KB, 8-way
L2 unified TLB 512 entries, 4-way
QuickPath interconnect 4 links @ 25.6 GB/s 102.4 GB/s total
To other cores
To I/O bridge
Core x(2-6)
L3 unified cache 8 MB, 16-way (shared by all cores)
DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s 32 GB/s total (shared by all cores)
To Main memory
Chip

Summary

- Various memory management schemes
  - Contiguous memory allocation
  - Noncontiguous memory allocation – paging
- H/W support for memory protection
  - Base and limit registers
  - TLB – associative mapping table
- Fragmentation problems
  - Internal and external fragmentation
- Paging system