Memory Management

Memory management

- We have seen how CPU can be shared by a set of processes
  - Improve system performance
  - Process management
- Need to keep several process in memory
  - Share memory
- Learn various techniques to manage memory
  - Hardware dependent

Memory management

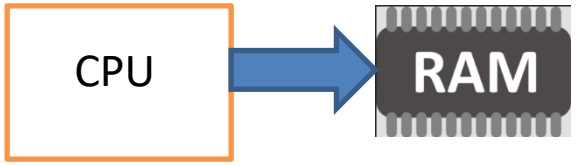
What are we going to learn?

- ***Basic Memory Management:*** logical vs. physical address space, protection, contiguous memory allocation, paging, segmentation, segmentation with paging.

- ***Virtual Memory:*** background, demand paging, performance, page replacement, page replacement algorithms (FCFS, LRU), allocation of frames, thrashing.

Background

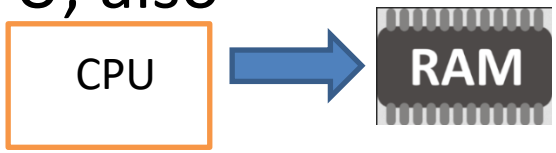
- Program must be brought (from disk) into memory
- Fetch-decode-execute cycle



- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Sequence of memory addresses generated by running program

Logical vs. Physical Address Space

Logical address – generated by the CPU; also referred to as **virtual address**



Physical address – address seen by the memory unit

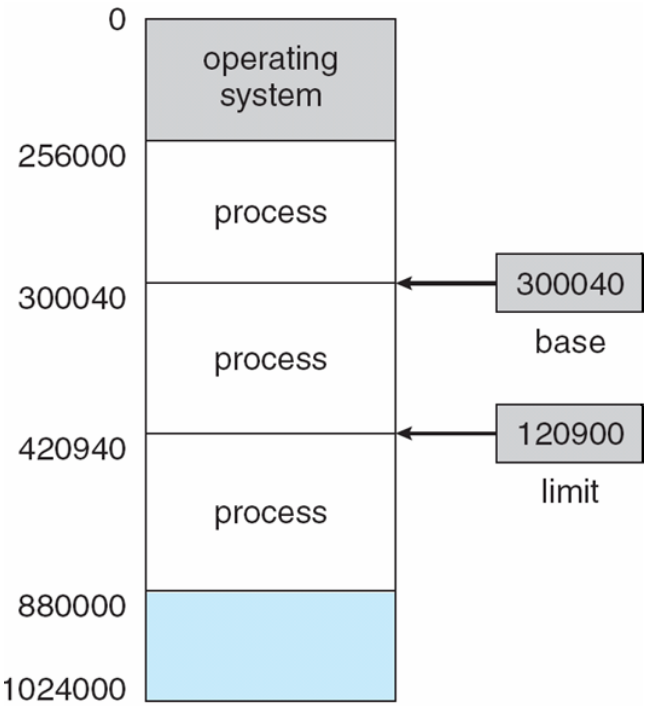
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Background

Multiple processes resides in memory

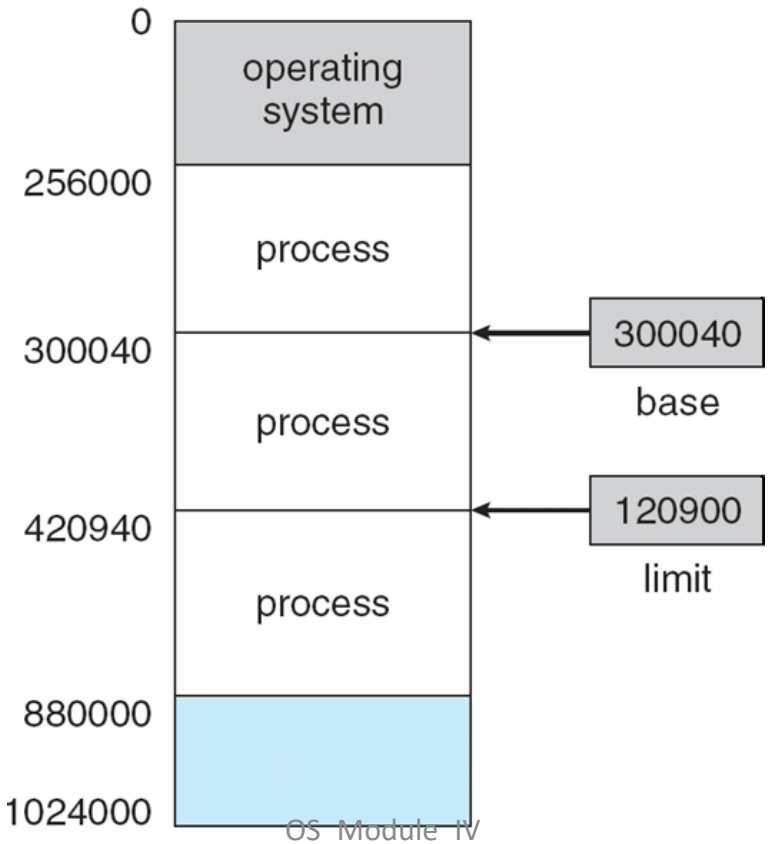
- Protection of memory required to ensure correct operation

1. Protect OS
2. Protect user processes

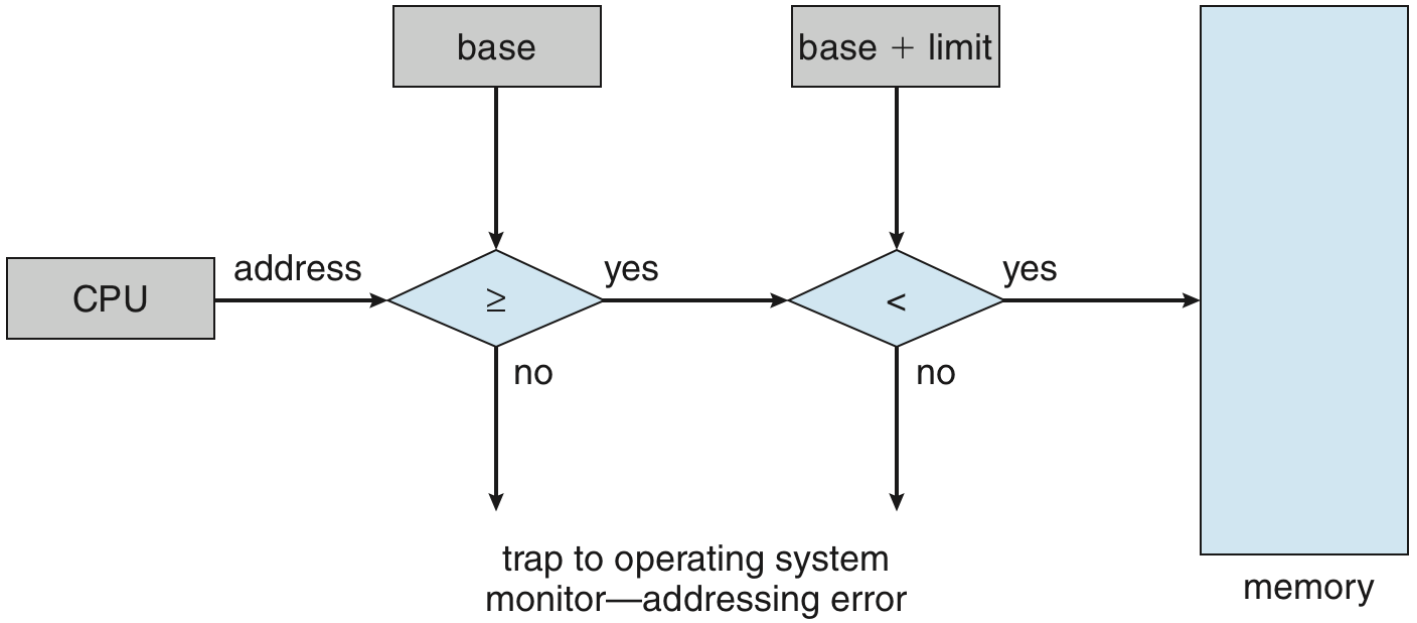


Base and Limit Registers

- A pair of **base** and **limit** registers define the logical address space



Hardware Address Protection with Base and Limit Registers

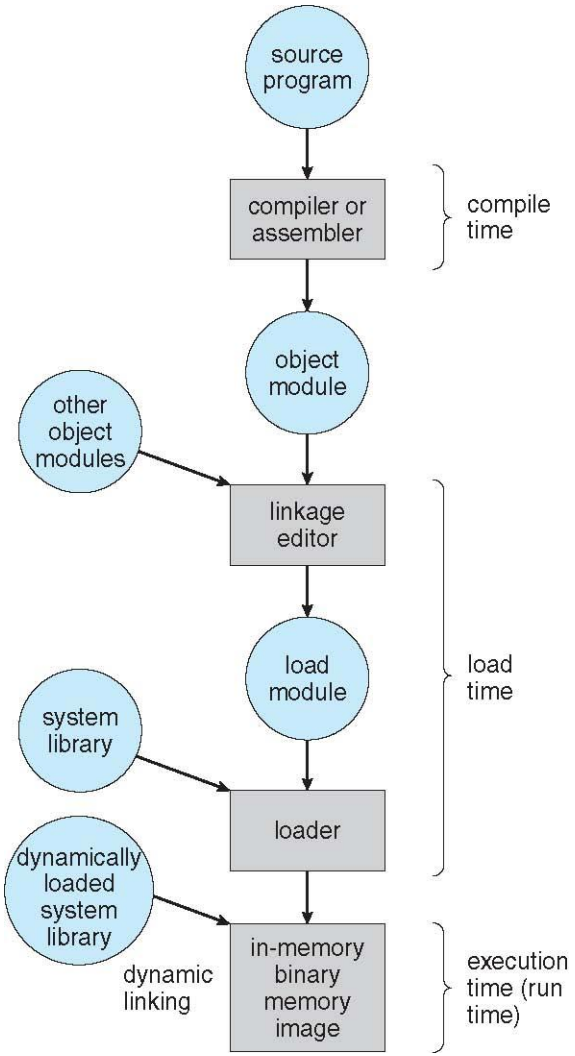


- OS loads the base & limit reg.
- Privileged instruction

Address Binding

- Process resides in main memory
- Associate each data element with memory address
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014

Multistep Processing of a User Program

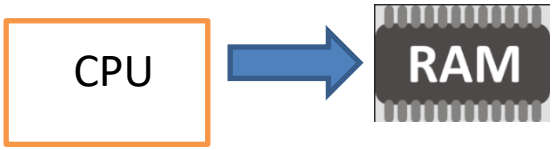


Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: If the process can be moved during its execution from one memory segment to another
    - Binding delayed until run time
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

Logical address – generated by the CPU; also referred to as **virtual address**



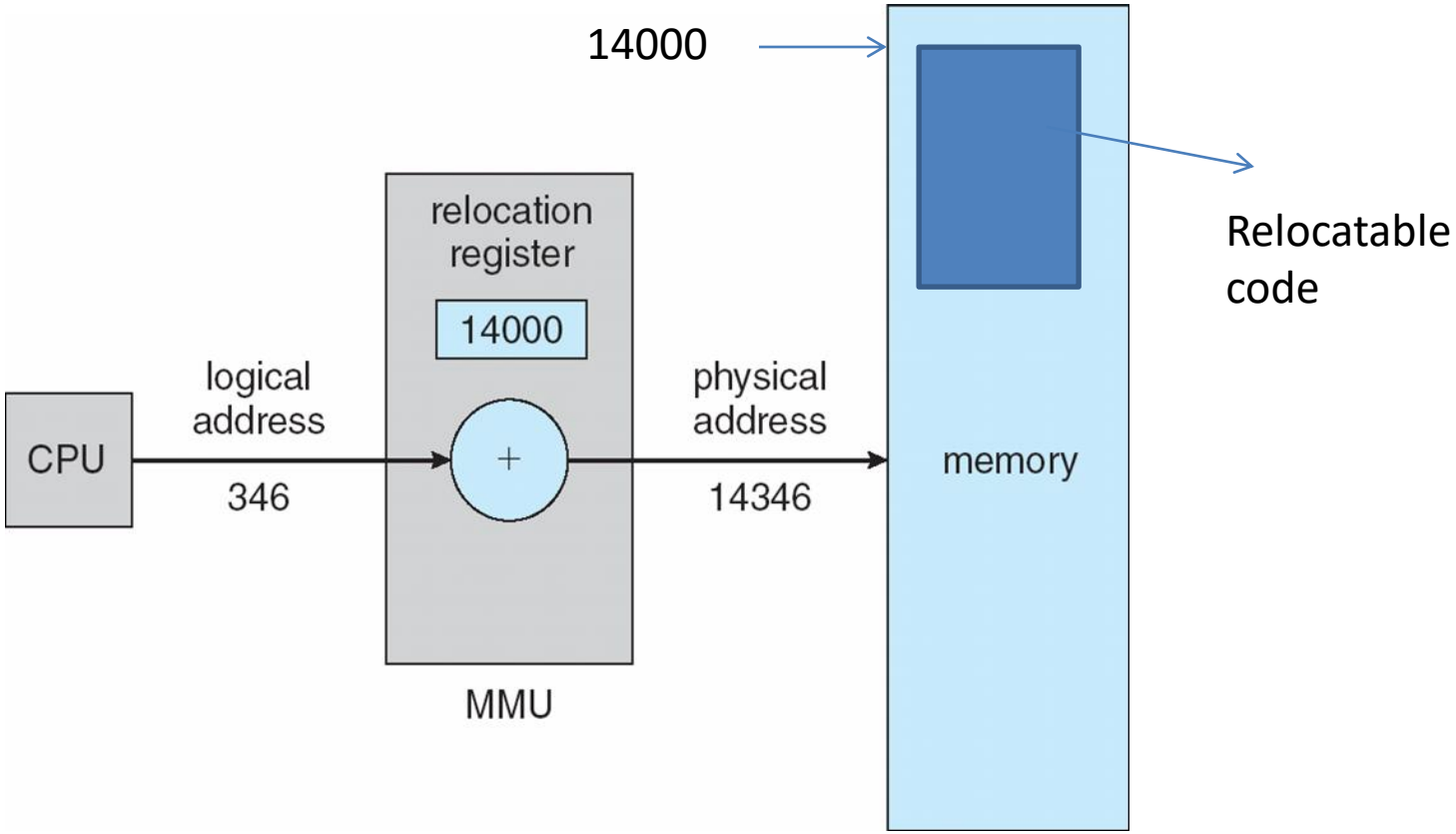
Physical address – address seen by the memory unit

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

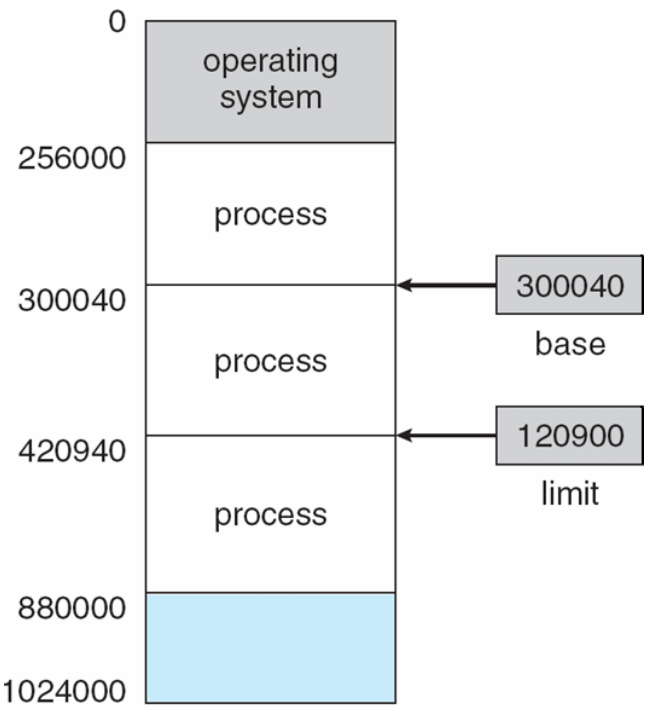
- Hardware device that at run time maps virtual to physical address
- Many methods possible
- To start, consider simple scheme where the value in the **relocation register** is added to every address generated by a user process at the time it is sent to memory
  - **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses (0 to max); it never sees the *real* physical addresses (R to R+max)
  - Say the logical address 25
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register



Contiguous Allocation

Multiple processes resides in memory



Contiguous Allocation

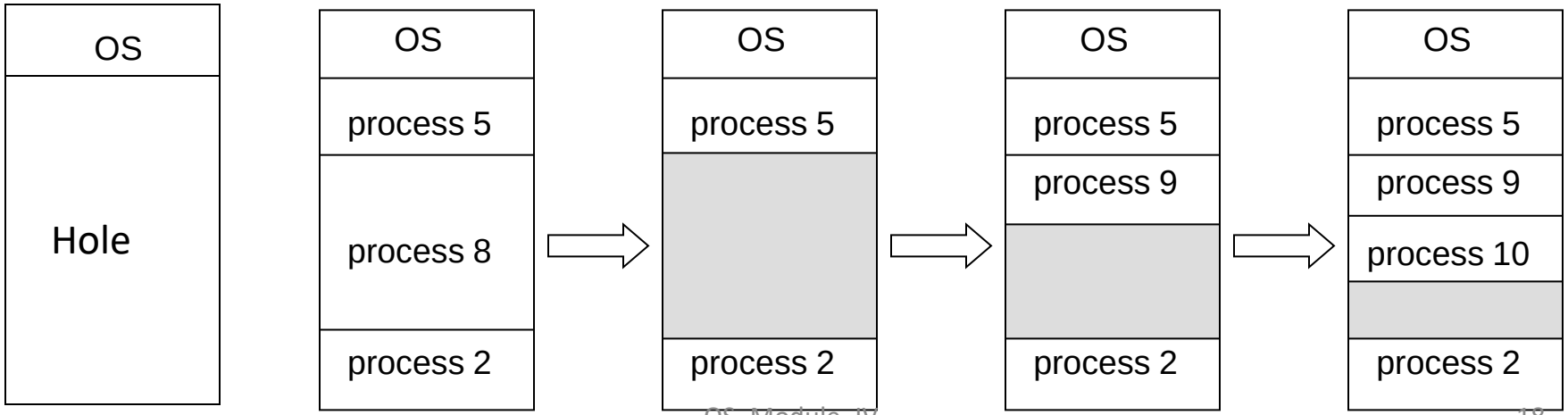
- Main memory usually divided into two partitions:
  - Resident operating system, usually held in low memory
  - User processes then held in high memory
  - Each process contained in **single contiguous section** of memory

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - Divide memory into several **Fixed size partition**
  - Each partition stores one process
  - Degree of multiprogramming limited by number of partitions
  - If a partition is free, load process from job queue
  - MFT (IBM OS/360)

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - **Variable partition scheme**
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - Keeps a table of free 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)



Dynamic Storage-Allocation Problem

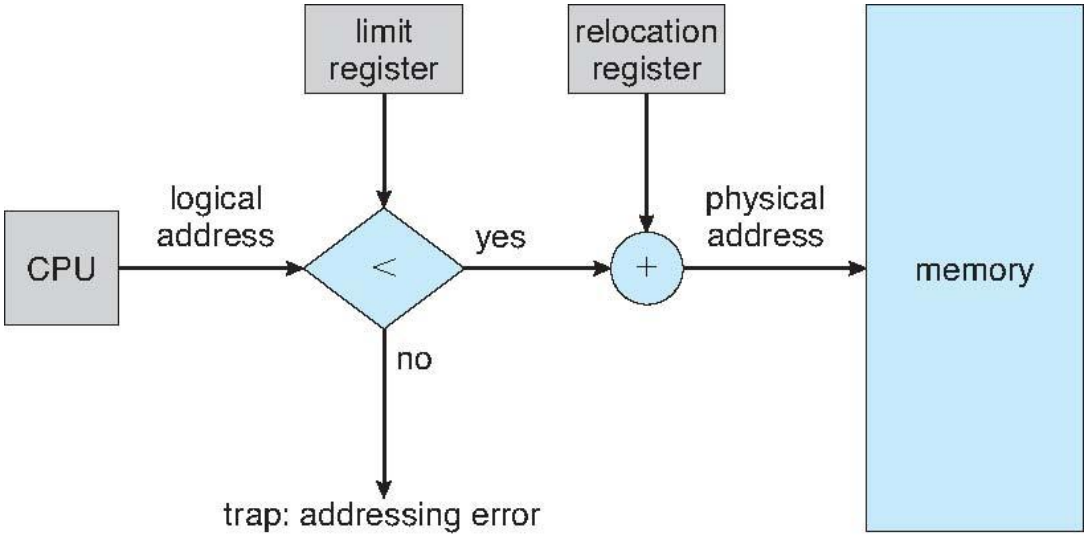
How to satisfy a request of size *n* from a list of free holes?

Dynamic storage allocation problem

- **First-fit**: Allocate the *first* hole that is big enough
- **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

Hardware Support for Relocation and Limit Registers

- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - Context switch
  - MMU maps logical address *dynamically*



Fragmentation

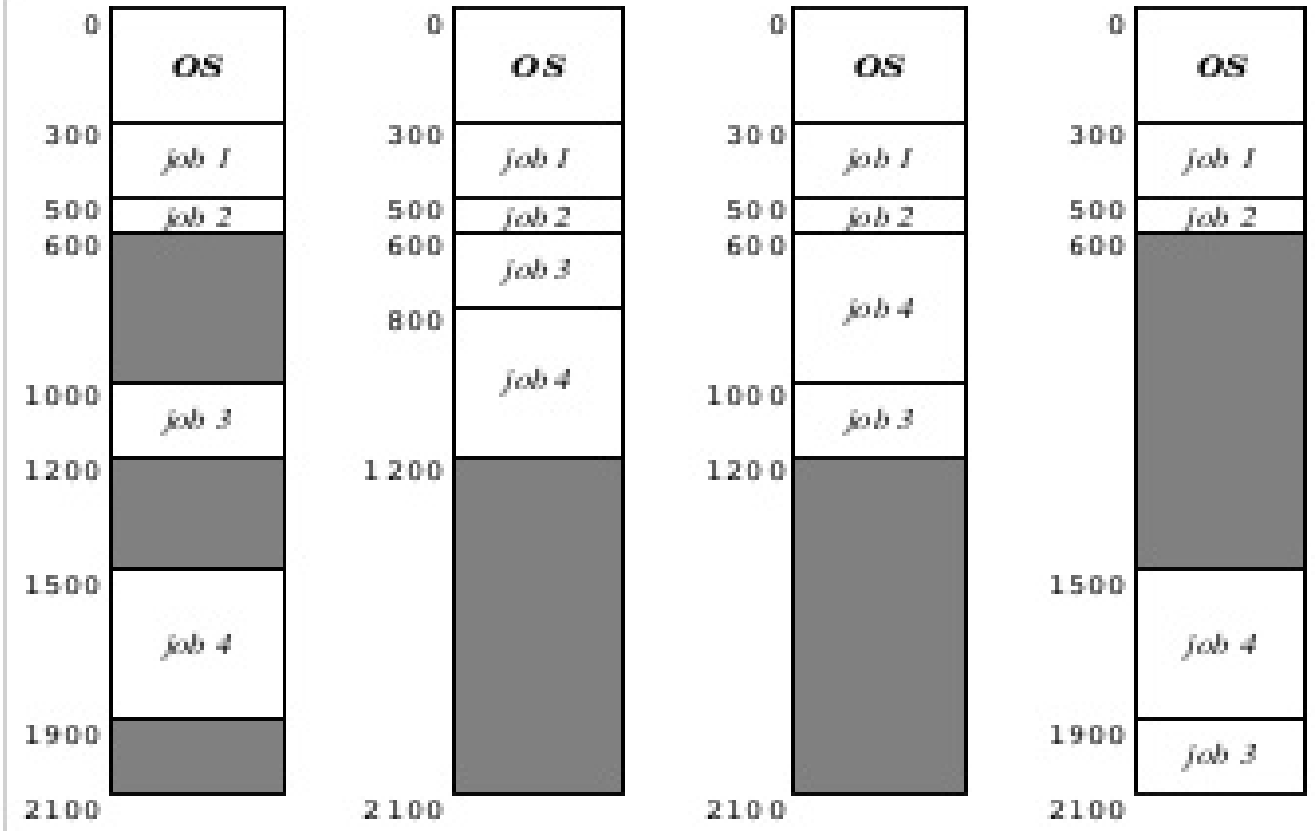
- Processes loaded and removed from memory
  - Memory is broken into little pieces

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

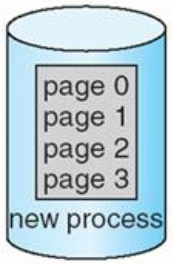
Fragmentation (Cont.)

- 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
    - Change relocation reg.
  - Cost
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used


0
OS
300
job 1
500
job 2
600
1000
job 3
1200
1500
job 4
1900
2100
0
OS
300
job 1
500
job 2
600
job 3
800
job 4
1200
2100
0
OS
300
job 1
500
job 2
600
job 4
1000
job 3
1200
2100
0
OS
300
job 1
500
job 2
600
1500
job 4
1900
job 3
2100

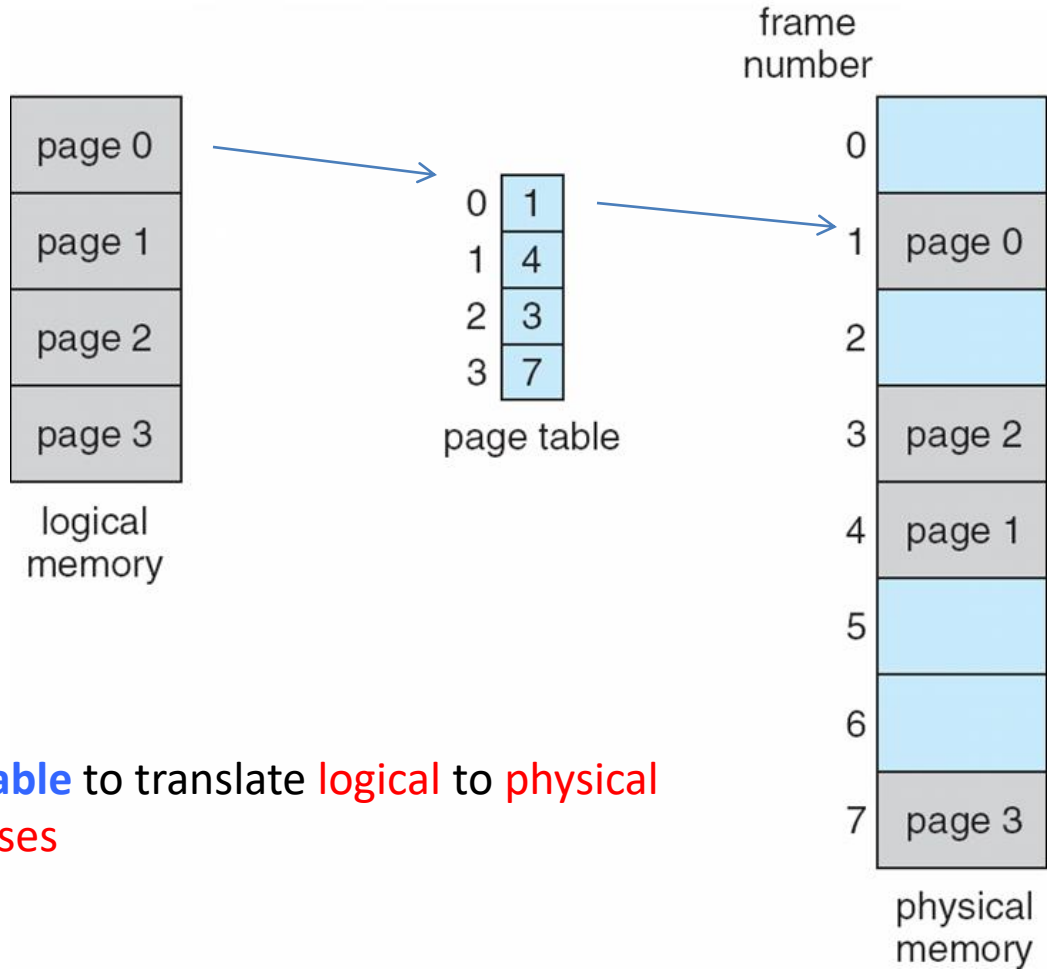
Paging

- Physical address space of a process can be noncontiguous;
  - process allocates physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
  - To run a program of size *N* pages, need to find *N* free frames and load program
- Backing store likewise split into pages



- Set up a **page table** to translate logical to physical addresses
- System keeps track of all free frames

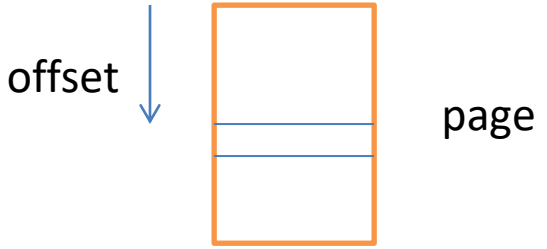
Paging Model of Logical and Physical Memory

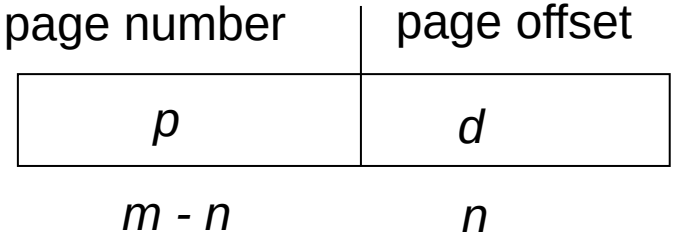


page table to translate logical to physical addresses

Address Translation Scheme

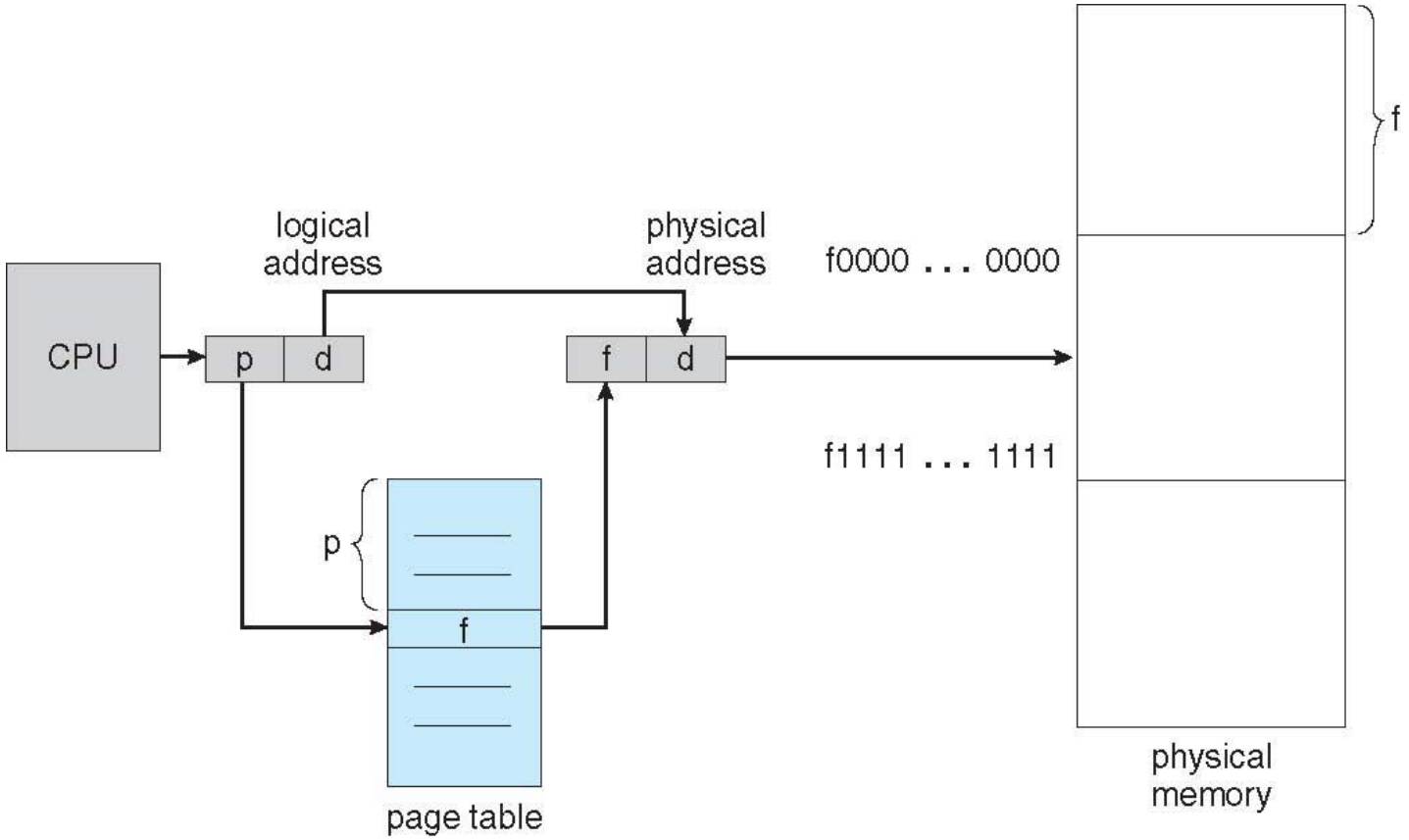
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a **page table**
    - which contains base address of each page in physical memory
  - **Page offset (*d*)** – offset within a page
    - combined with base address to define the physical memory address that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





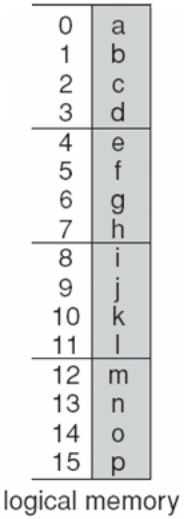
  - **For given logical address space 2^m and page size 2^n**

Paging Hardware

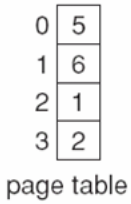


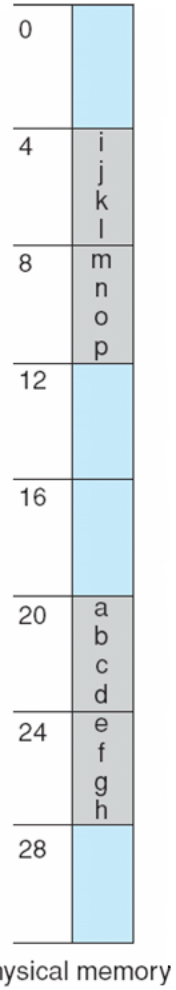
Paging Example

Logical address = 16
Page size=4
Physical memory=32



User's view





Run time address binding

n=2 and *m*=4 32-byte memory and 4-byte pages

Logical address 0
(0*4+0)
Physical address:
(5*4+0)=20

Logical address 3
(0*4+3)
Physical address:
(5*4+0)=23

Logical address 4
(1*4+0)
Physical address:
(6*4+0)=24

Logical address 13
(3*4+1)
Physical address:
(2*4+1)

Paging

- External fragmentation??
- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- So small frame sizes desirable?
  - But increases the page table size
  - Poor disk I/O
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- User's view and physical memory now very different
  - user view=> process contains in single contiguous memory space
- By implementation process can only access its own memory
  - protection

- Each page table entry 4 bytes (32 bits) long
- Each entry can point to 2^{32} page frames
- If each frame is 4 KB
- The system can address 2^{44} bytes (16TB) of physical memory

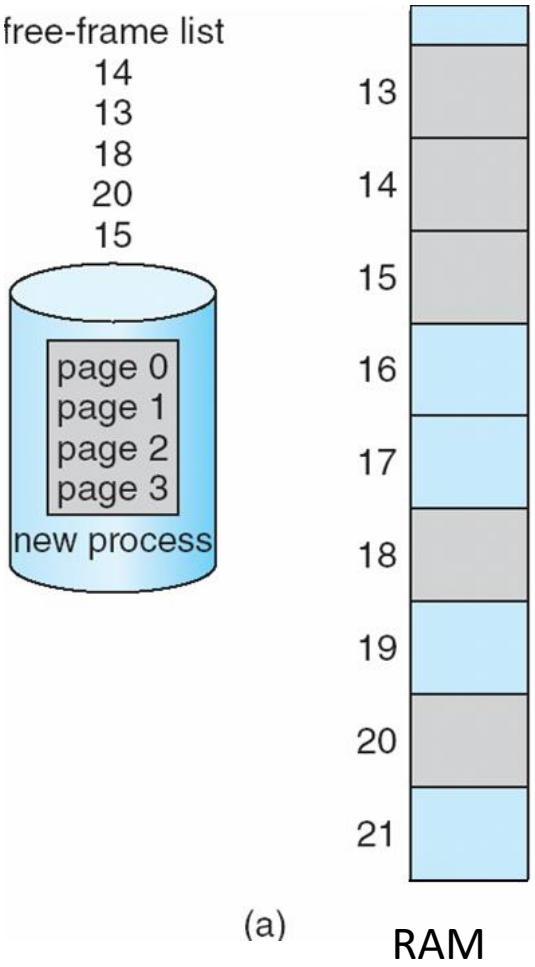
Virtual address space 16MB.
Page table size?

- Process P1 arrives
- Requires n pages => n frames must be available
- Allocate n frames to the process P1
- Create page table for P1

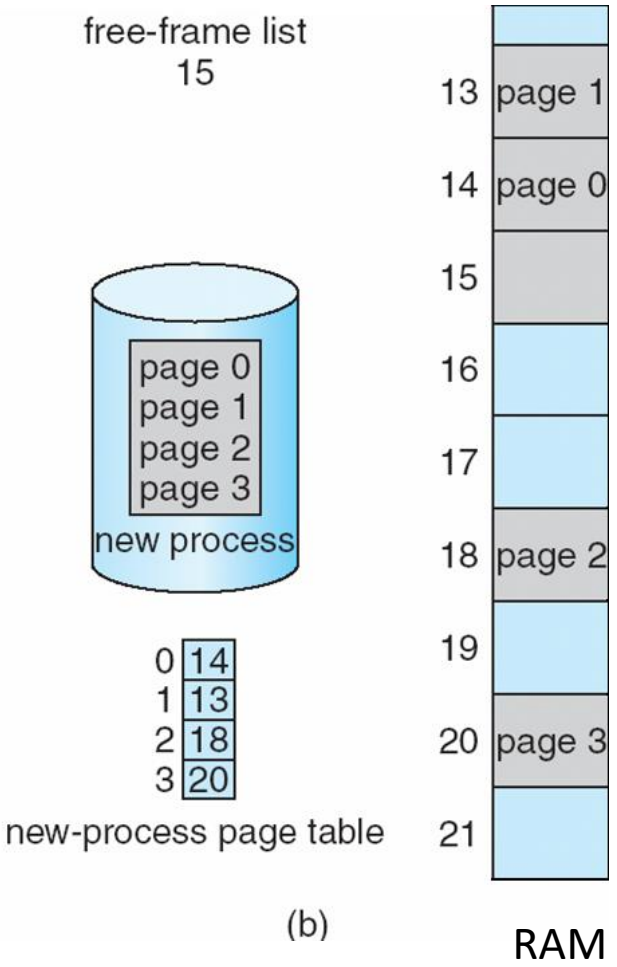
Free Frames

Frame table

Use's view



Before allocation



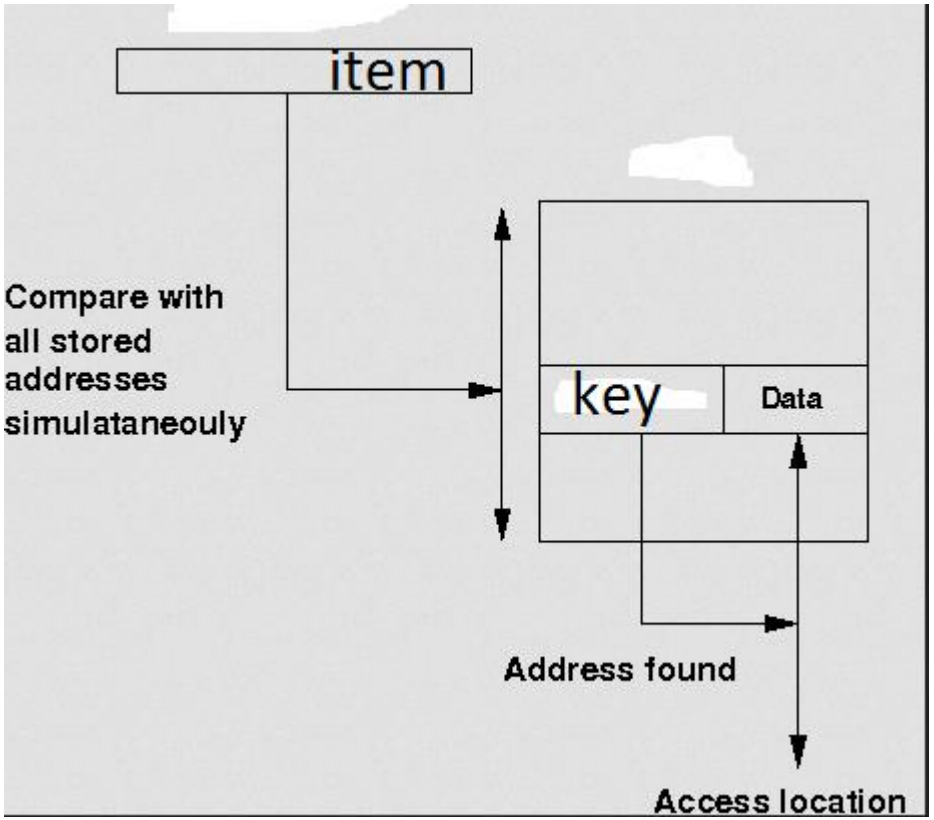
After allocation

System's view

Implementation of Page Table

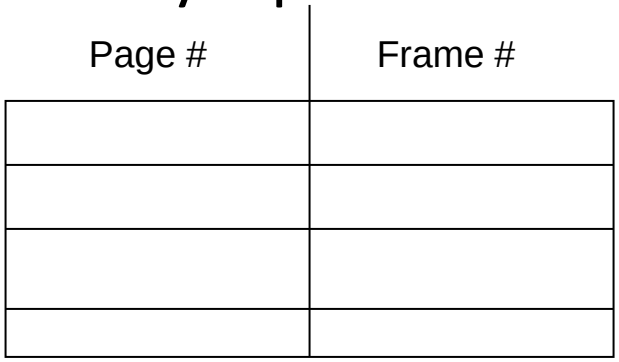
- For each process, Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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)**

Associative memory



Associative Memory

- Associative memory – parallel search



- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Implementation of Page Table

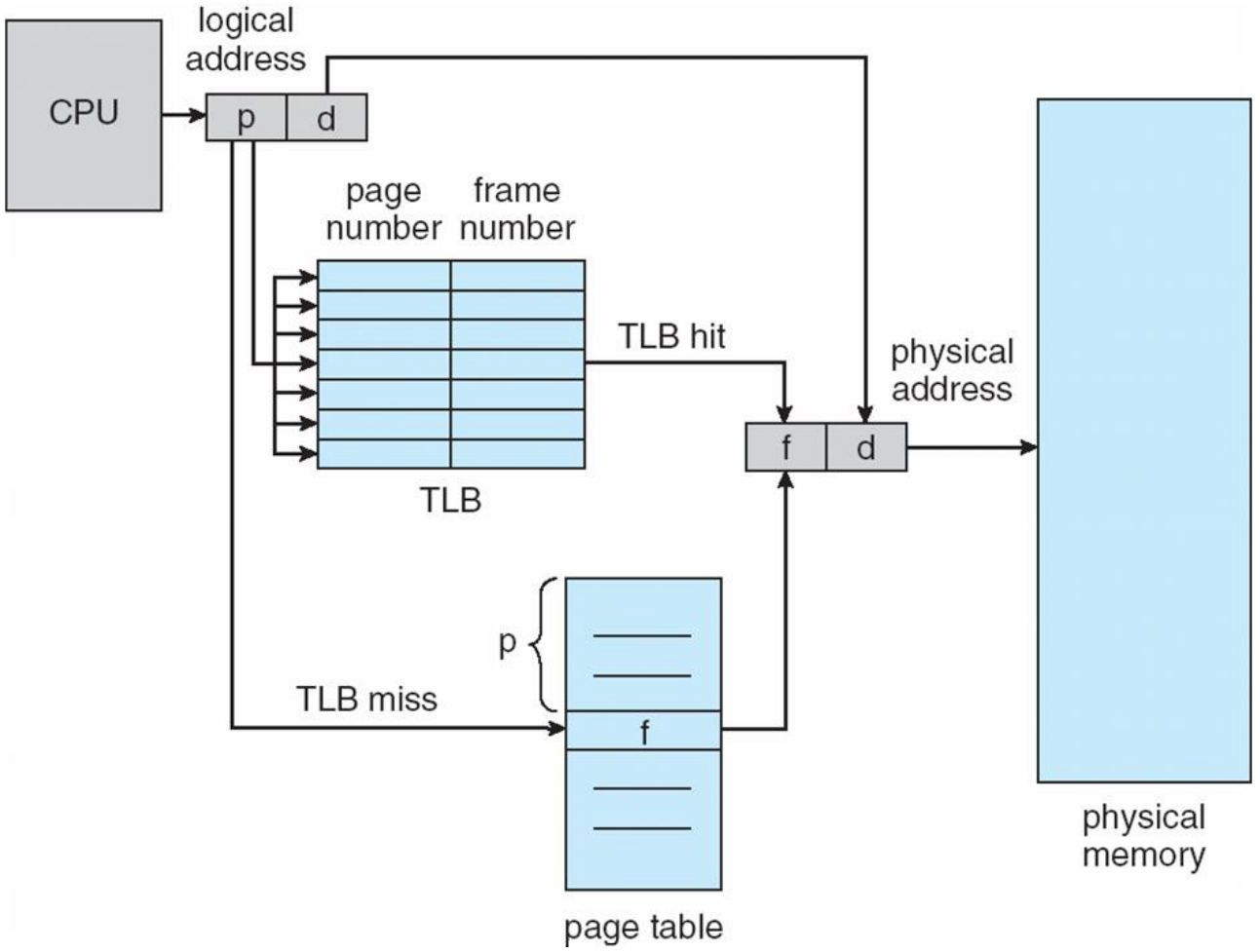
- For each process, Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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)**

- TLBs typically small (64 to 1,024 entries)

- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered (LRU)
  - Some entries can be **wired down** for permanent fast access

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process (PID) to provide address-space protection for that process
  - Otherwise need to flush at every context switch

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to size of TLB
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time (EAT)**

$$EAT = (100 + \varepsilon)\,\alpha + (200 + \varepsilon)(1 - \alpha)$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider better hit ratio -> α = 98%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.98 x 120 + 0.02 x 220 = 122ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated 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
  - Or use PTLR
- Any violations result in a trap to the kernel

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

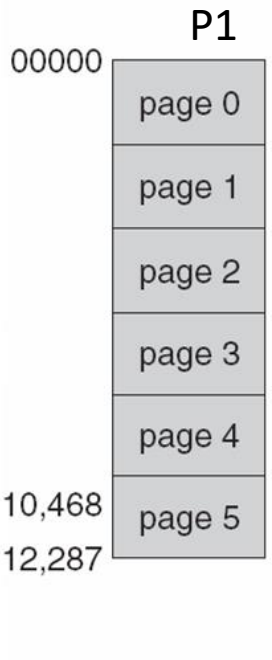
14 bit address space (0 to 16383)

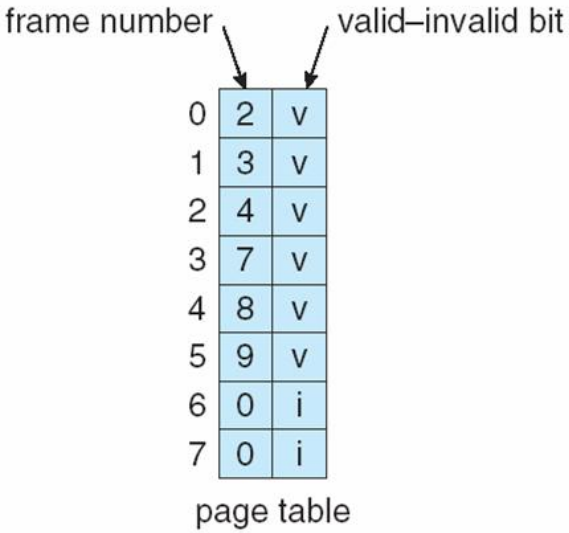
Page size 2KB

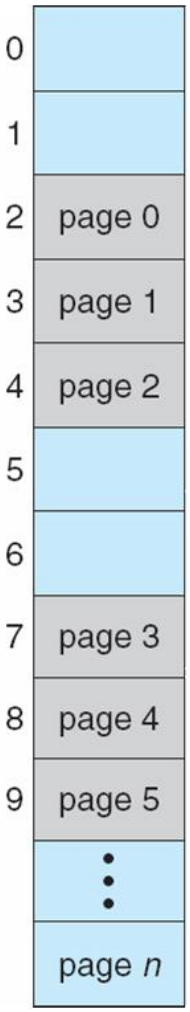
Process P1 uses only 0 to 10468

P2

Page 0
Page 1
Page 2
Page 3







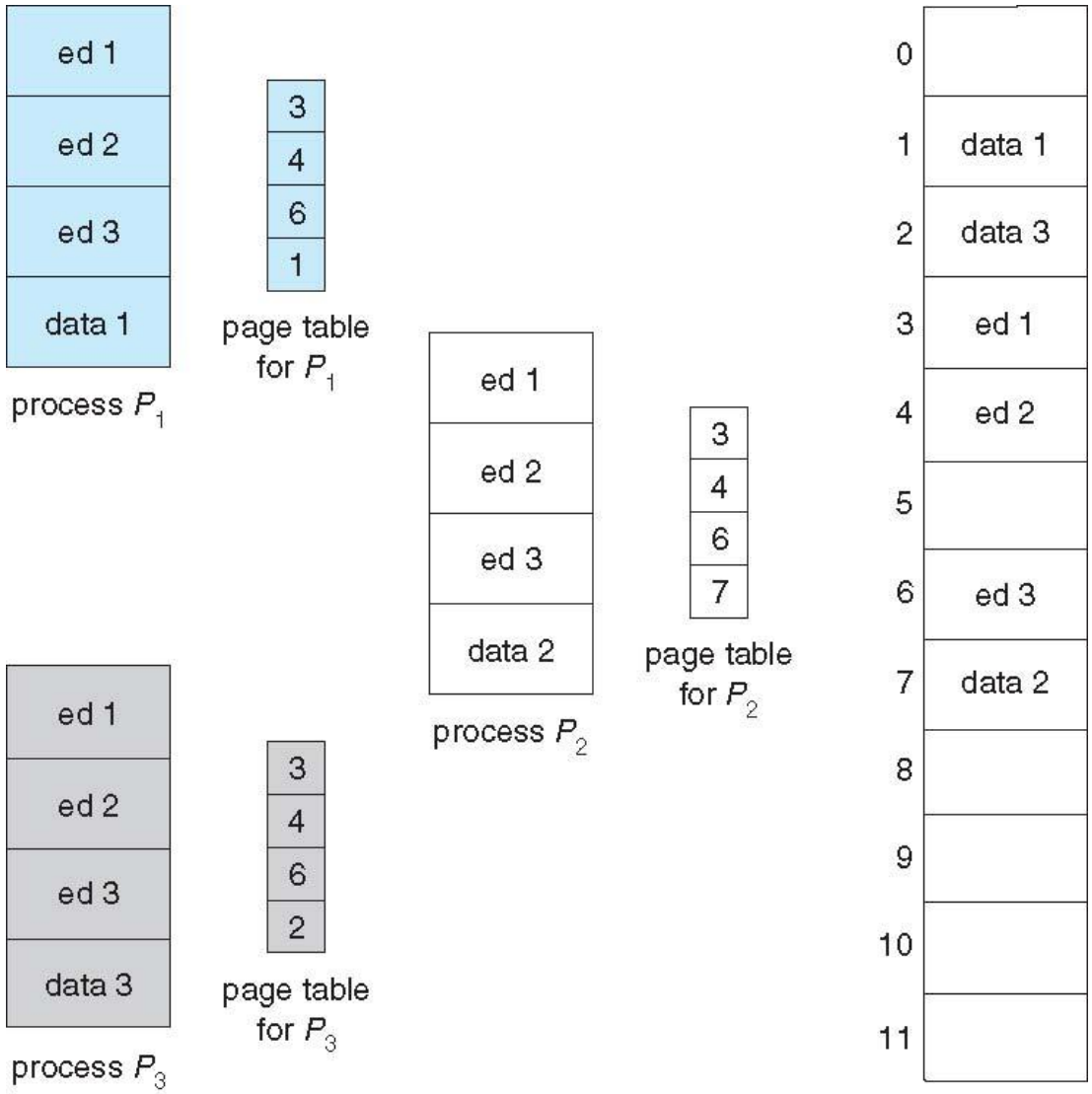
Internal fragmentation

Use of PTLR (length)

Shared Pages Example

- System with 40 users
  - Use common text editor
- Text editor contains 150KB code 50KB data (page size 50KB)
  - 8000KB!
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
    - Code never changes during execution
- Only one copy of the editor in the memory
- Total memory consumption
  - 40*50+150=2150KB

Shared Pages Example



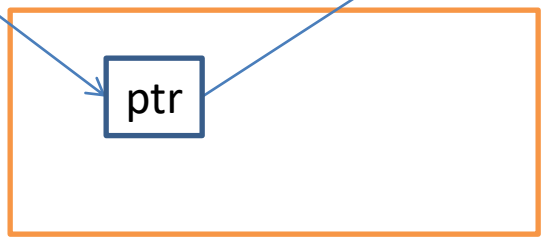
Data share: example

writer.c

```
int main()
{

      int shmid,f,key=3,i,pid;
      char *ptr;

      shmid=shmget((key_t)key,100,IPC_CREAT|0666);
      ptr=shmat(shmid,NULL,0);
      printf("shmid=%d ptr=%u\n",shmid, ptr);
      strcpy(ptr,"hello");
     i=shmdt((char*)ptr);

}
```

reader .c

```
int main()
{

      int shmid,f,key=3,i,pid;
      char *ptr;

      shmid=shmget((key_t)key,100,IPC_CREAT|0666);
      ptr=shmat(shmid,NULL,0);
      printf("shmid=%d ptr=%u\n",shmid, ptr);
      printf("\nstr %s\n",ptr);
}
```



Shared memory

Structure of the Page Table

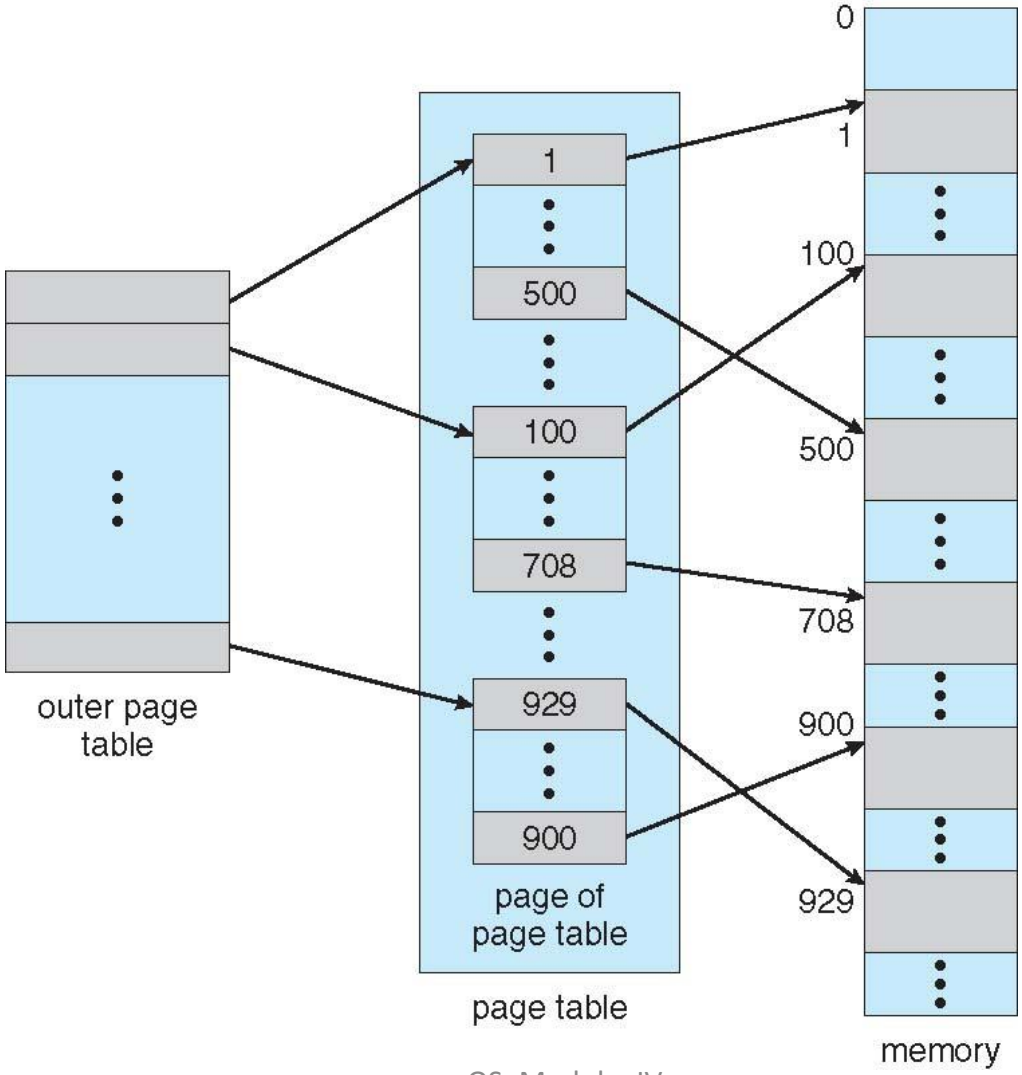
- Memory requirement for page table can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries 2^{20} (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

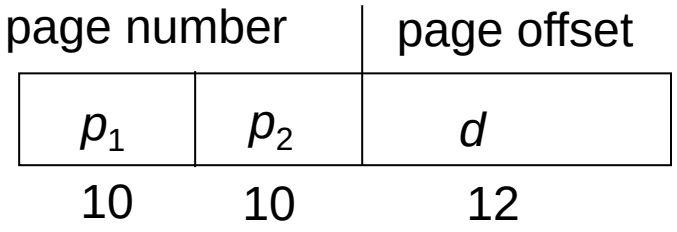
- Break up the page table into multiple pages
- We then page the page table
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



Two-Level Paging Example

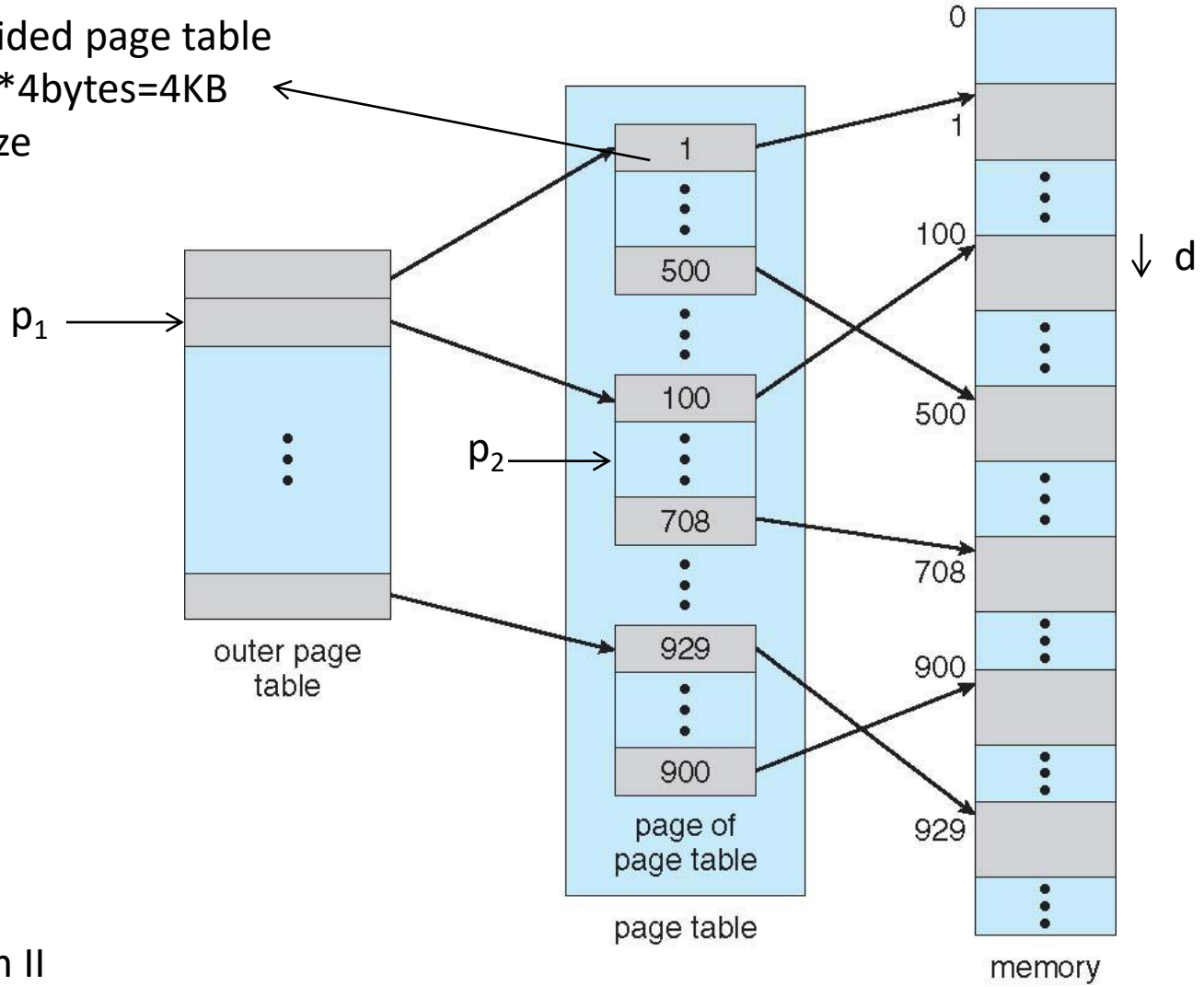
- A logical address (on 32-bit machine with 4KB page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

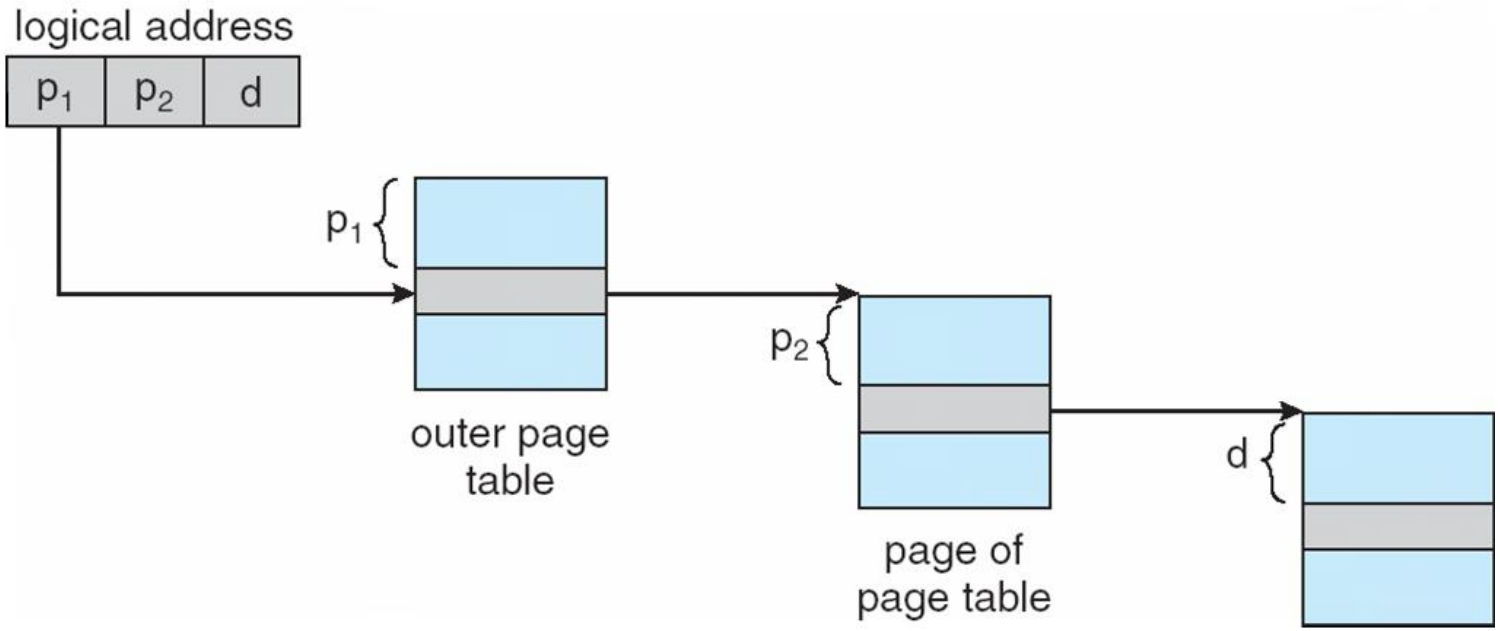
Two-Level Page-Table Scheme

Each divided page table size=2^{10} *4bytes=4KB =Page size



Pentium II

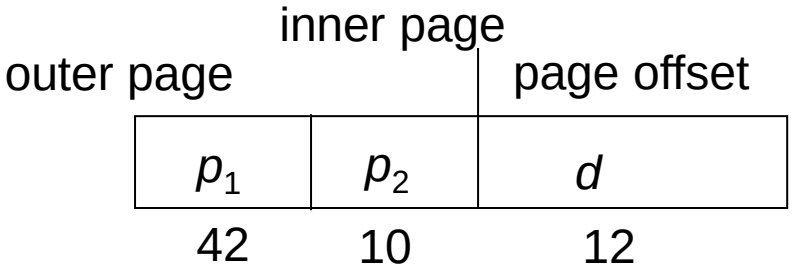
Address-Translation Scheme



Pentium II

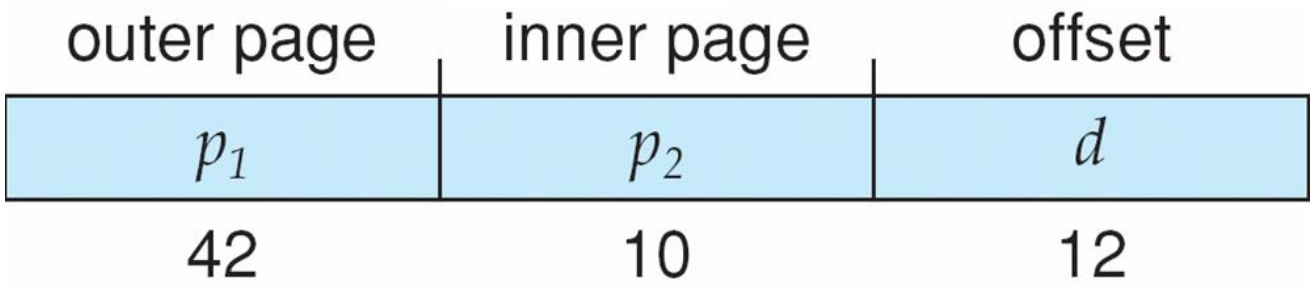
64-bit Logical Address Space

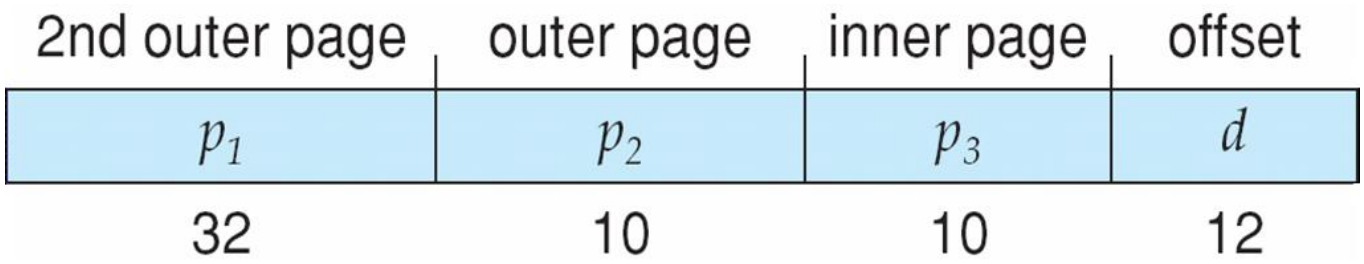
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme



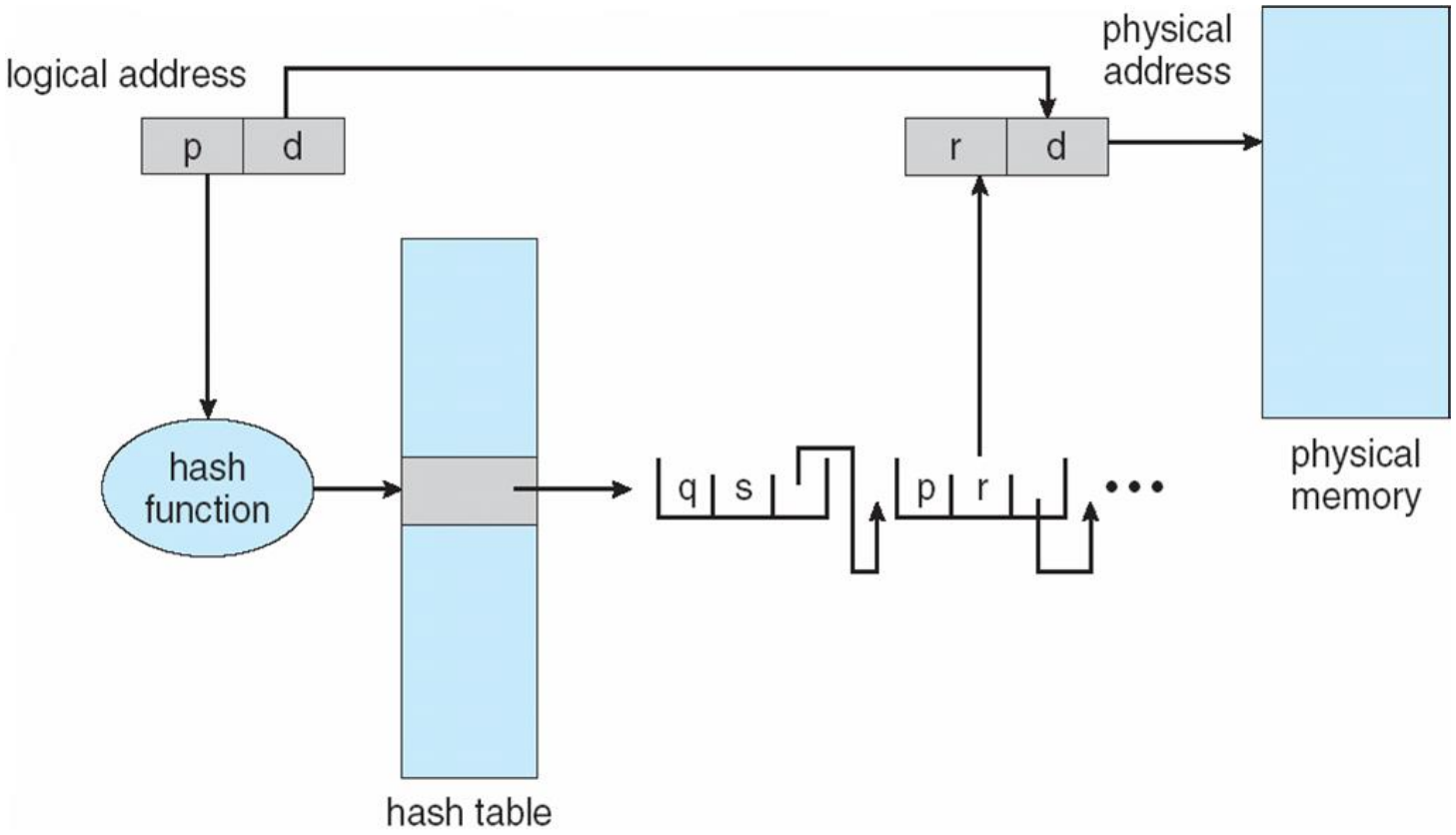


SPARC (32 bits), Motorola 68030 support three and four level paging respectively

Hashed Page Tables

- Common in virtual address spaces > 32 bits
- The page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the page number (2) the value of the mapped page frame (3) a pointer to the next element
- 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

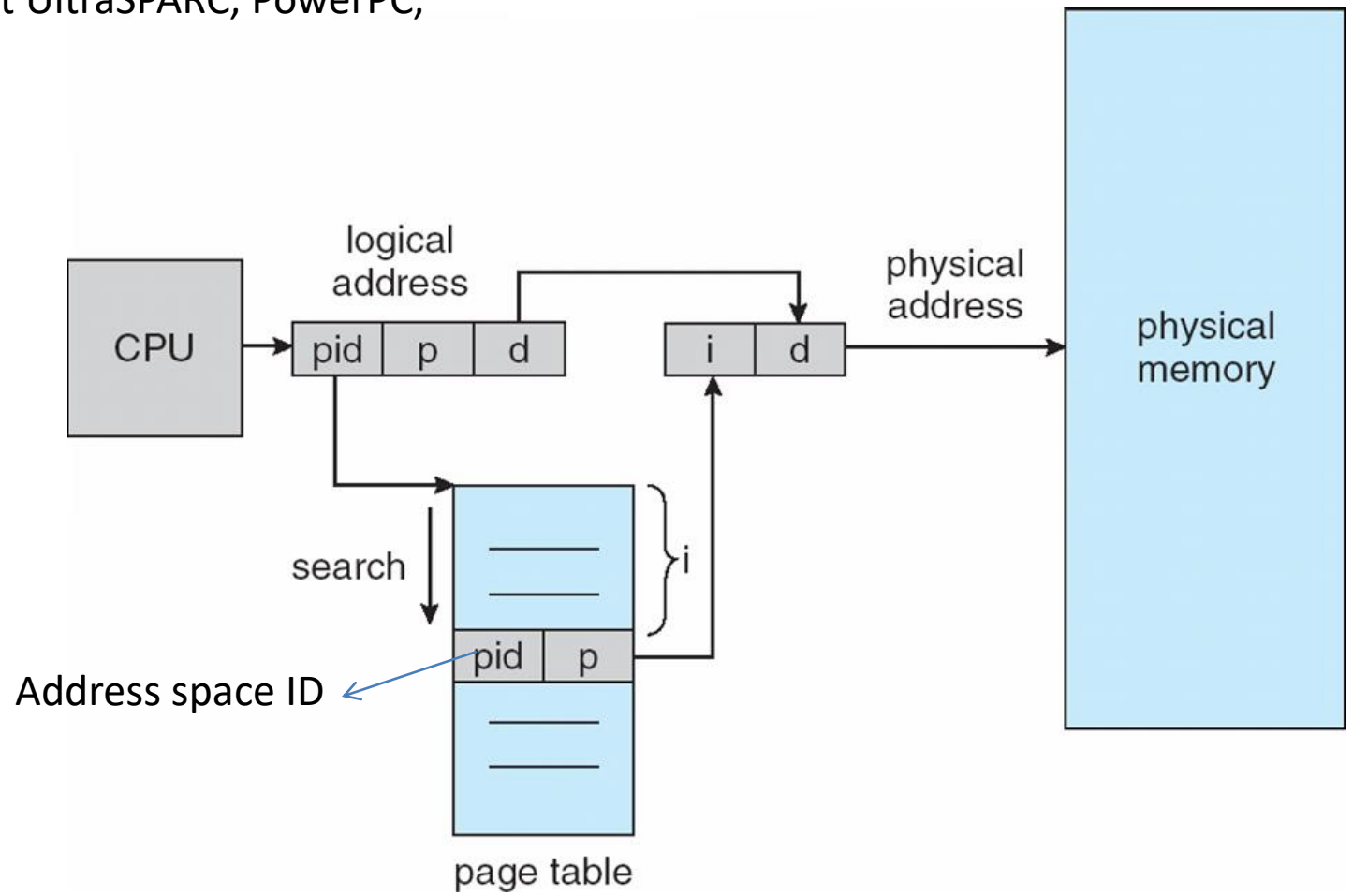


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all frames
- One entry for each frame
- Entry consists the page number stored in that frame, with information about the process that owns that page
- Decreases memory needed to store each page table,
  - but increases time needed to search the table when a page reference occurs

Inverted Page Table Architecture

64 bit UltraSPARC, PowerPC,

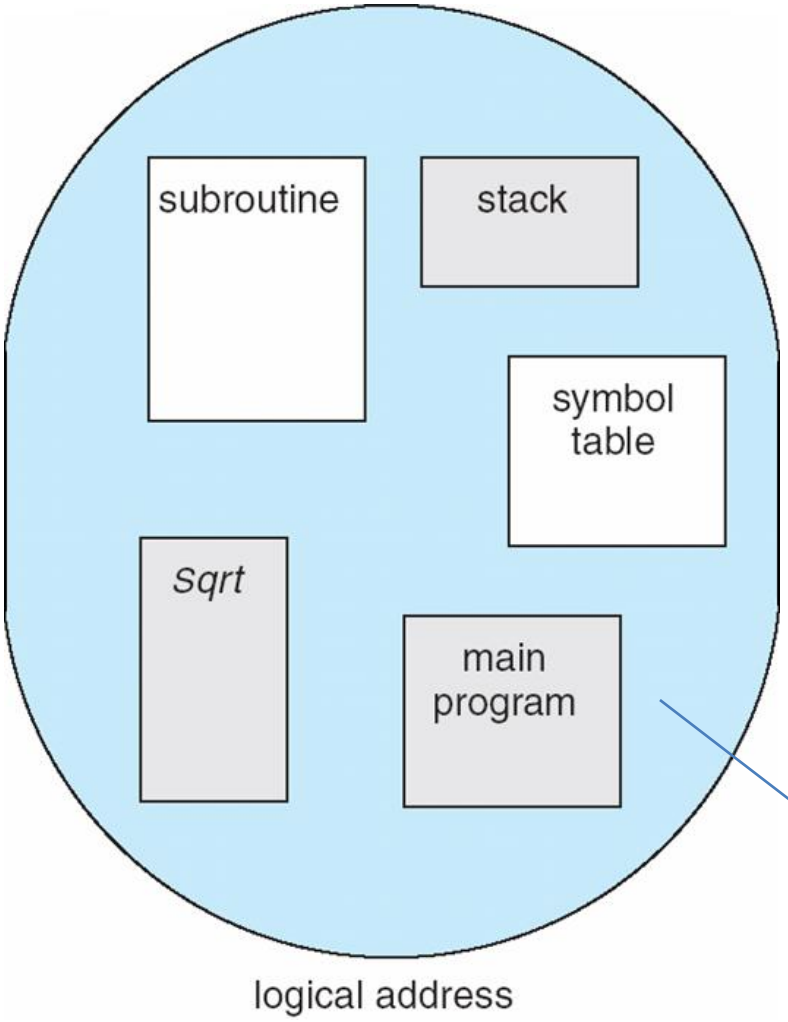


Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:
    - main program
    - procedure
    - function
    - method
    - object
    - local variables, global variables
    - common block
    - stack
    - symbol table
    - arrays

Compiler generates the segments
Loader assign the seg#

User's View of a Program

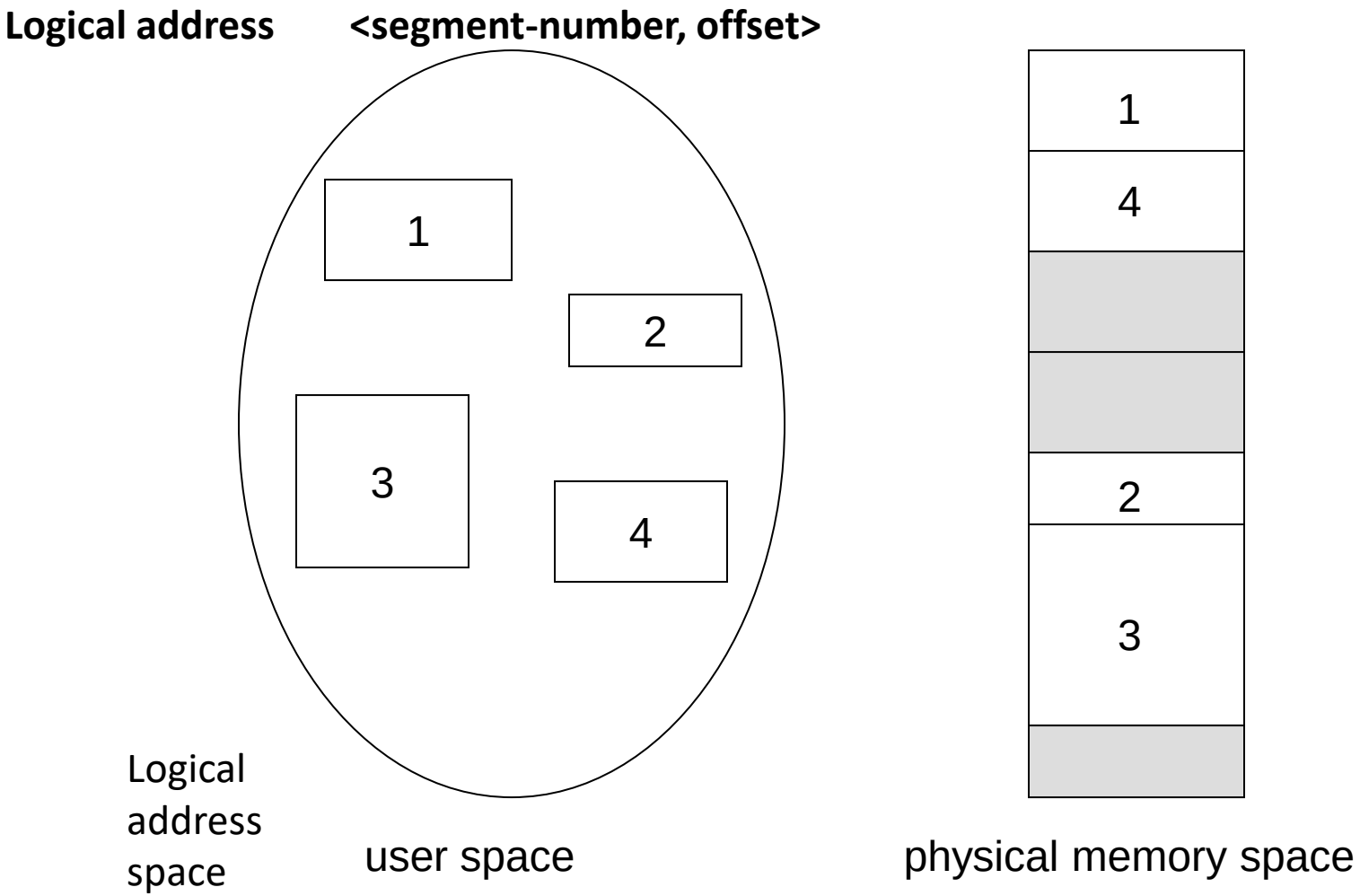


User specifies each address by two quantities
(a) Segment name
(b) Segment offset

Logical address contains the tuple
<segment#, offset>

- Variable size segments without order
- Length=> purpose of the program
- Elements are identified by offset

Logical View of Segmentation


Logical address
<segment-number, offset>
1
2
3
4
Logical address space
user space
1
4
2
3
physical memory space

- Long term scheduler finds and allocates memory for all segments of a program
- Variable size partition scheme

Windows XP Memory Usage

Segment	First Address	Last Address	Size
Code	401000x	403000x	002000x ~ 8 Kbytes
Static (Global) Data	403000x	703000x	300000x ~ 3 megabytes
Heap	760000x	3A261000x	39800000x ~ 950 megabytes
Stack	22EF00x	16EF00x	1C0000x ~ 2 megabyte

LINUX Memory Usage

Segment	First Address	Last Address	Size
Code	8048400x	8049900x	001500x ~ 6 Kbytes
Static (Global) Data	8049A00x	8349A00	300000x ~ 3 megabytes
Heap	B7EE,B000x	01CE,4000x	B6000000x ~ 3 gigabytes
Stack	BFFB,7334x	29BA,91E0x	9640,0000x ~ 2.5 gigabyte

Memory image

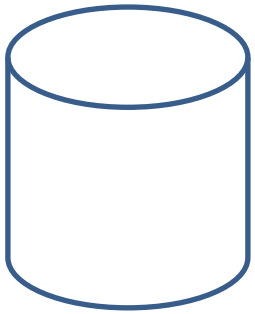
```
0x08048368 <main+0>:    55                      push   %ebp
0x08048369 <main+1>:    89 e5                   mov    %esp,%ebp
0x0804836b <main+3>:    83 ec 08                sub    $0x8,%esp
0x0804836e <main+6>:    83 e4 f0                and    $0xfffffff0,%esp
0x08048371 <main+9>:    b8 00 00 00 00          mov    $0x0,%eax
0x08048376 <main+14>:   83 c0 0f                add    $0xf,%eax
0x08048379 <main+17>:   83 c0 0f                add    $0xf,%eax
0x0804837c <main+20>:   c1 e8 04                shr    $0x4,%eax
0x0804837f <main+23>:   c1 e0 04                shl    $0x4,%eax
0x08048382 <main+26>:   29 c4                   sub    %eax,%esp
0x08048384 <main+28>:   83 ec 0c                sub    $0xc,%esp
0x08048387 <main+31>:   68 c0 84 04 08          push   $0x80484c0
0x0804838c <main+36>:   e8 1f ff ff ff          call   0x80482b0
0x08048391 <main+41>:   83 c4 10                add    $0x10,%esp
0x08048394 <main+44>:   e8 02 00 00 00          call   0x804839b <b>
```

```
1   void b();
2   void c();
3   int  main( )
4   {
5      printf( "Hello from main\n");
6      b();
7   }
8   // This routine reads the opcodes from memory and prints them out.
9   void  b()
10  {
11     char  *moving;
12
13     for ( moving = (char *)(&main); moving < (char *)(&c); moving++ )
14        printf( "Addr = 0x%x, Value = %2x\n", (int)(moving), 255 & (int)*moving );
15  }
16  void  c()
17  {
18  }
```

```
0x0804839b <b+0>:       55                      push   %ebp
0x0804839c <b+1>:       89 e5                   mov    %esp,%ebp
0x0804839e <b+3>:       83 ec 08                sub    $0x8,%esp
0x080483a1 <b+6>:       c7 45 fc 68 83 04 08    movl   $0x8048368,0xfffffffc(%ebp)
0x080483a8 <b+13>:      81 7d fc d9 83 04 08    cmpl   $0x80483d9,0xfffffffc(%ebp)
0x080483af <b+20>:      73 26                   jae    0x80483d7 <b+60>
0x080483b1 <b+22>:      83 ec 04                sub    $0x4,%esp
0x080483b4 <b+25>:      8b 45 fc                mov    0xfffffffc(%ebp),%eax
0x080483b7 <b+28>:      0f be 00                movsbl (%eax),%eax
0x080483ba <b+31>:      25 ff 00 00 00          and    $0xff,%eax
```

Executable file and virtual address

a.out



Symbol table	
Name	address
SQR	0
SUM	4

Virtual address space

Paging view

0	Load	0
4	ADD	4

Segmentation view

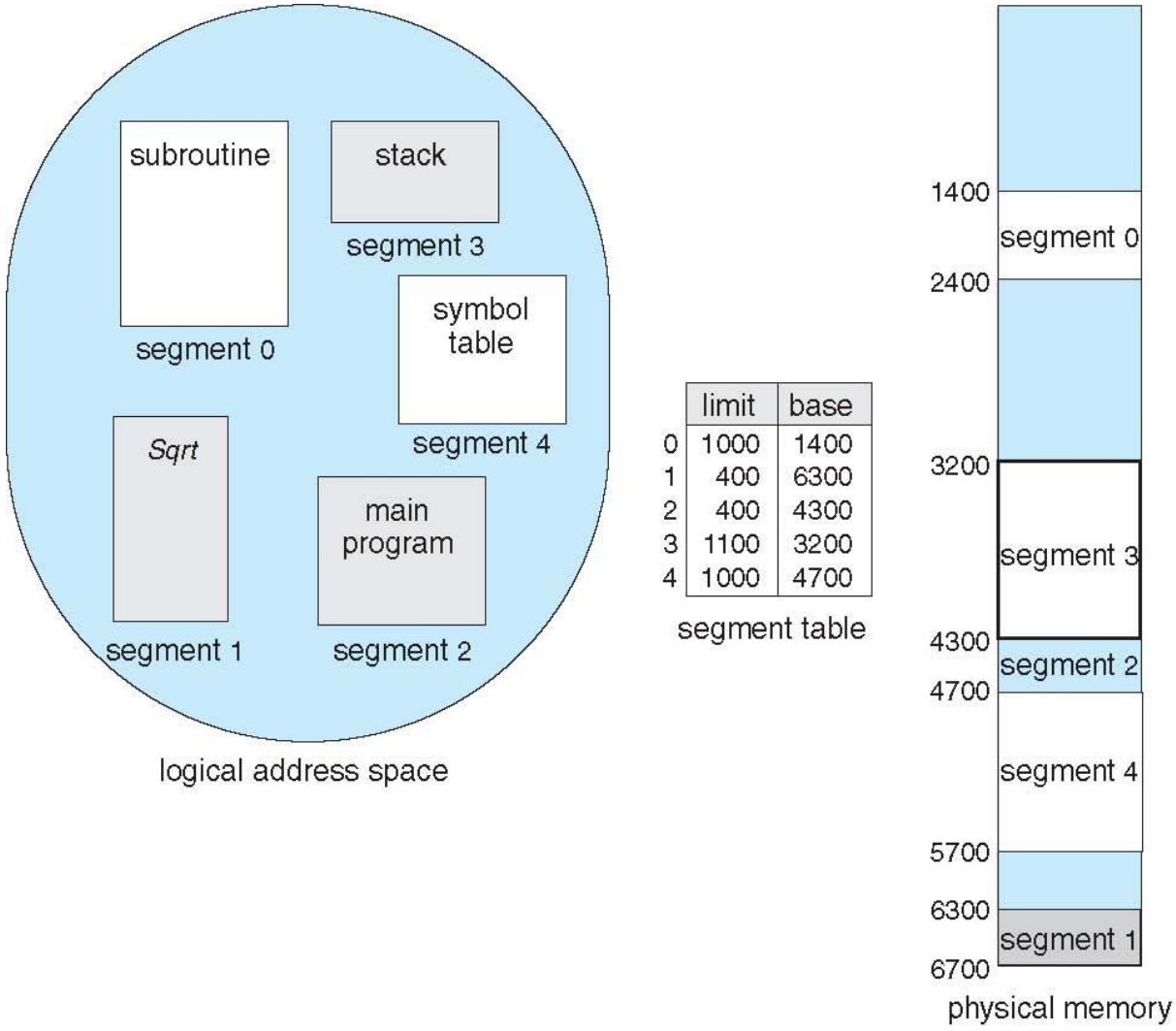
<CODE, 0>	Load	<ST,0>
<CODE, 2>	ADD	<ST,4>

Segmentation Architecture

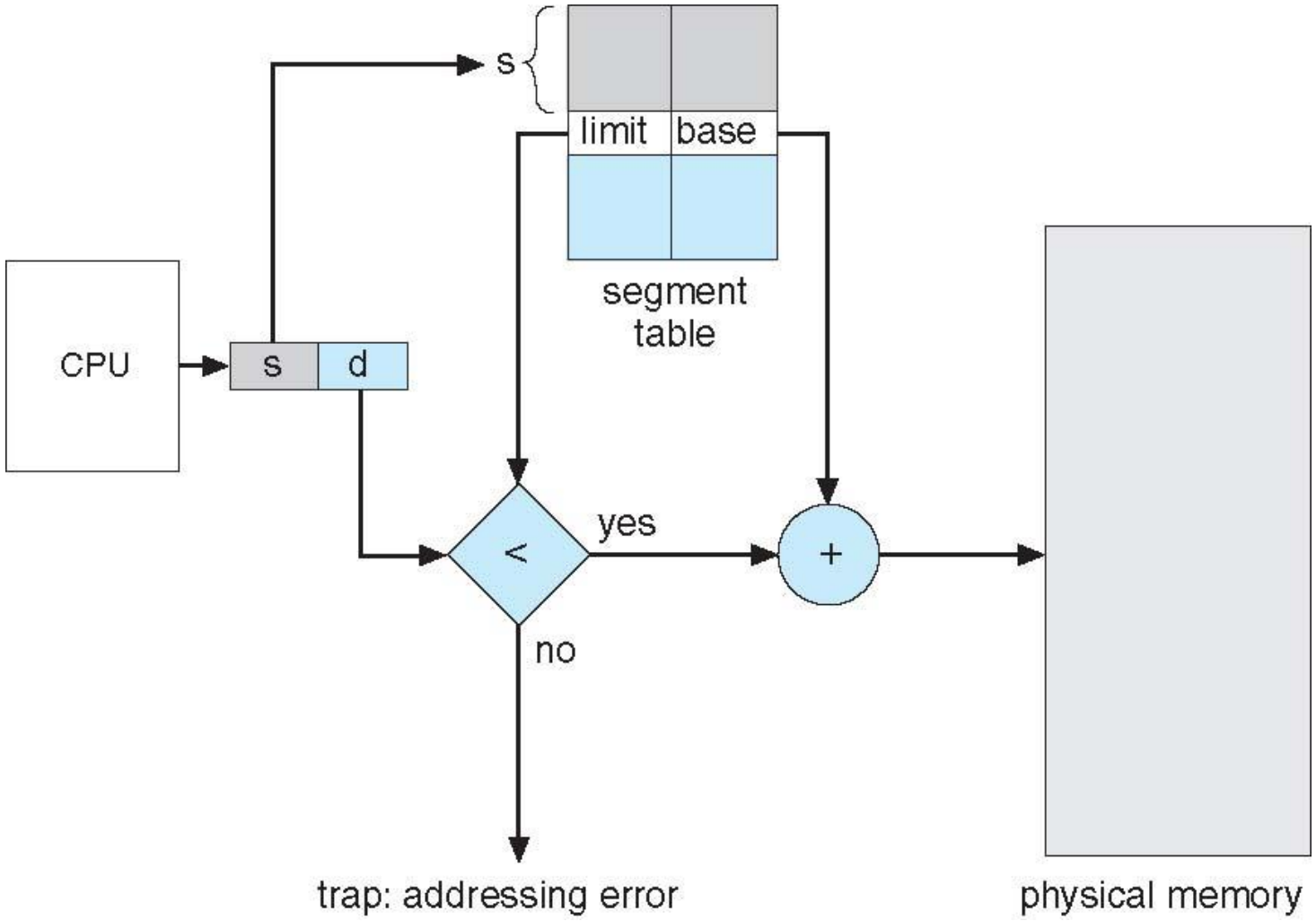
- Logical address consists of a two tuple:
  <segment-number, offset>
- **Segment table** – maps two-dimensional logical address to physical address;
- Each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length 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**

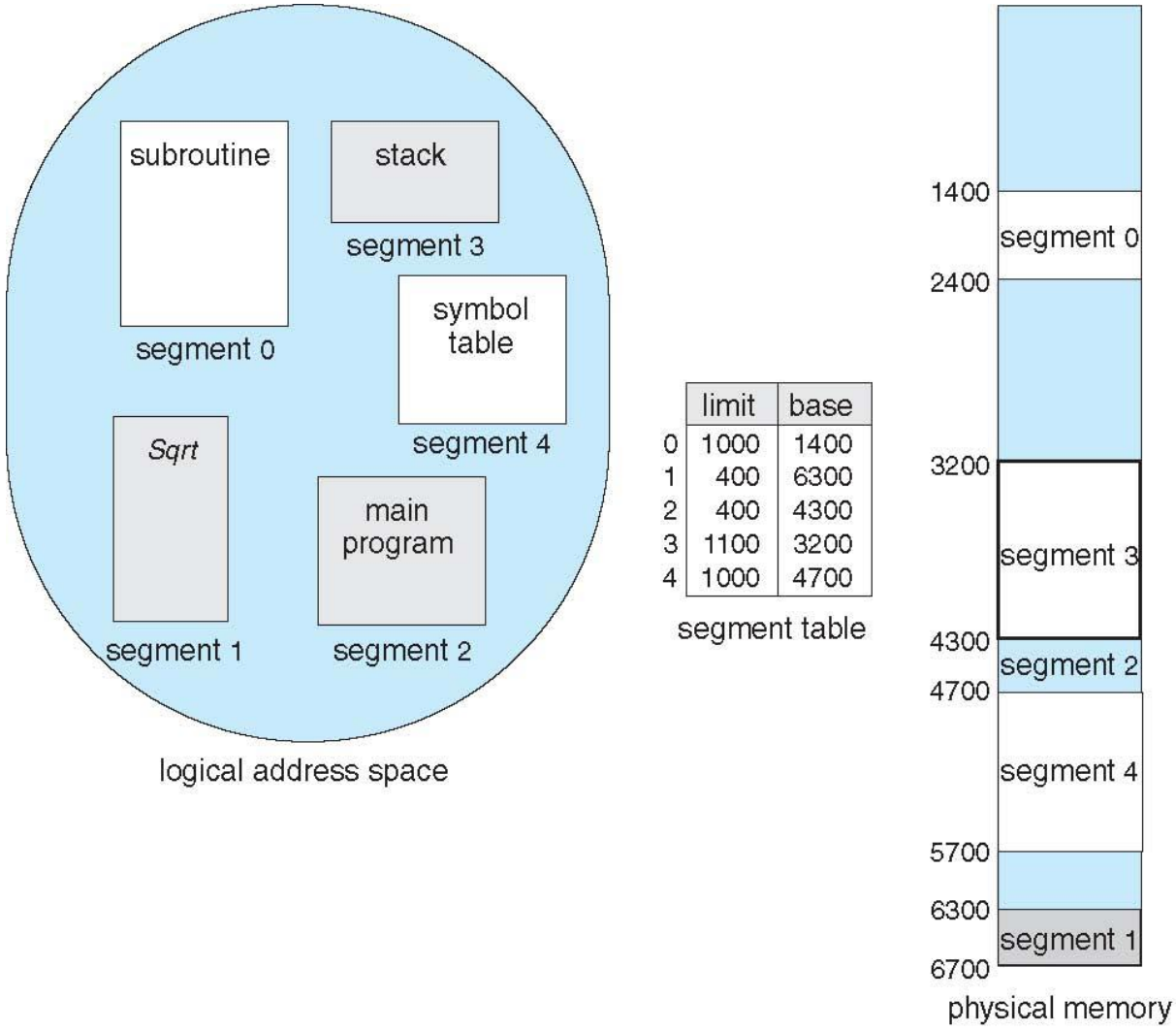
Example of Segmentation



Segmentation Hardware



Example of Segmentation



Segmentation Architecture

- Protection
- Protection bits associated with segments
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
  - Long term scheduler
  - First fit, best fit etc
- Fragmentation

Segmentation with Paging

Key idea:

Segments are splitted into multiple pages

Each page is loaded into frames in the memory

Segmentation with Paging

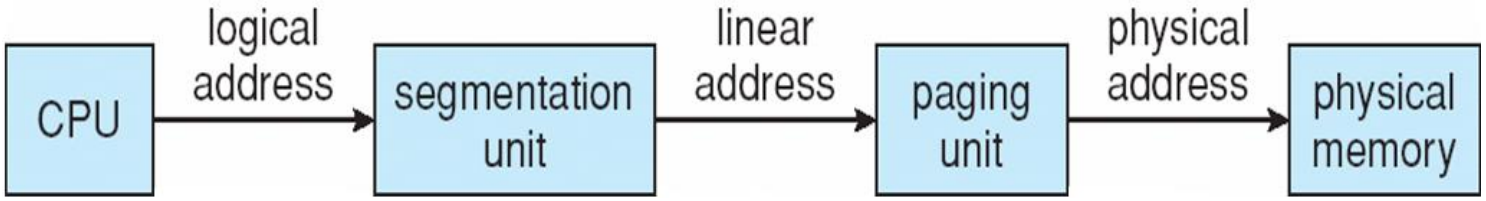
- Supports segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - <selector(16), offset (32)>
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table LDT**)
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table GDT**)

- CPU generates logical address (six Segment Reg.)
  - Given to segmentation unit
    - Which produces linear addresses
  - Physical address 32 bits
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB

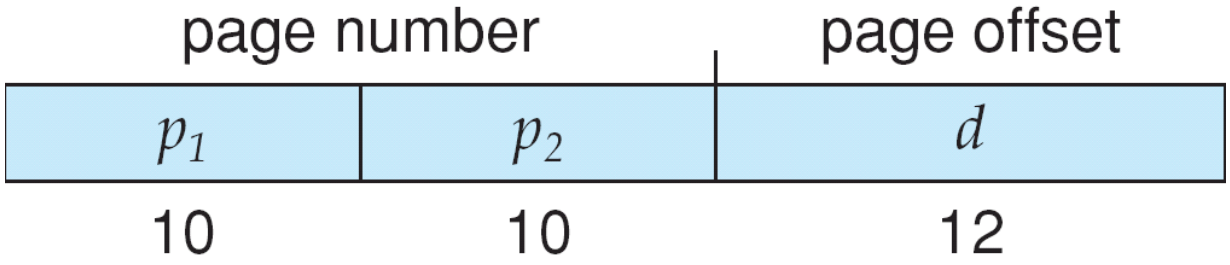
Intel 80386

IBM OS/2

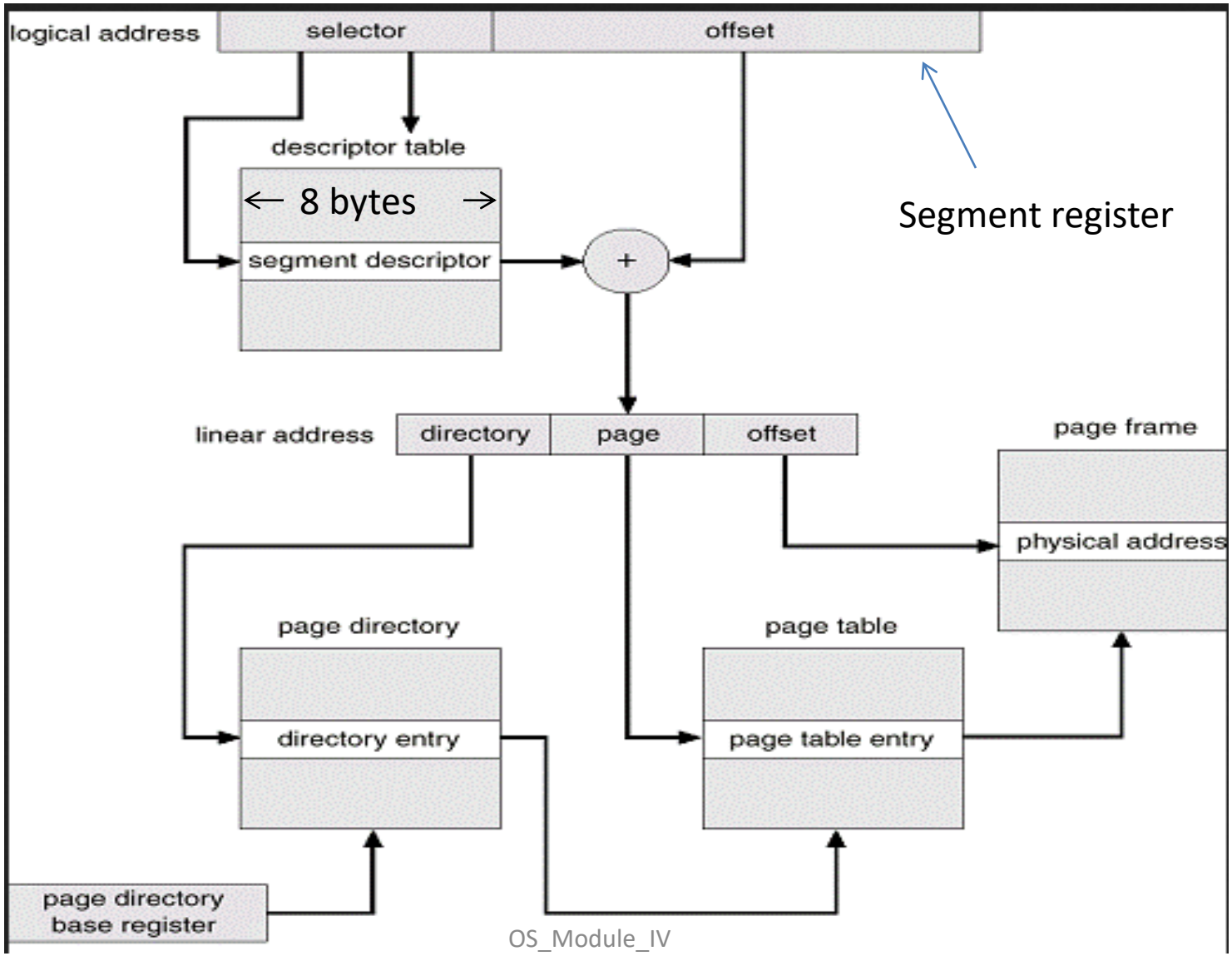
Logical to Physical Address Translation in Pentium



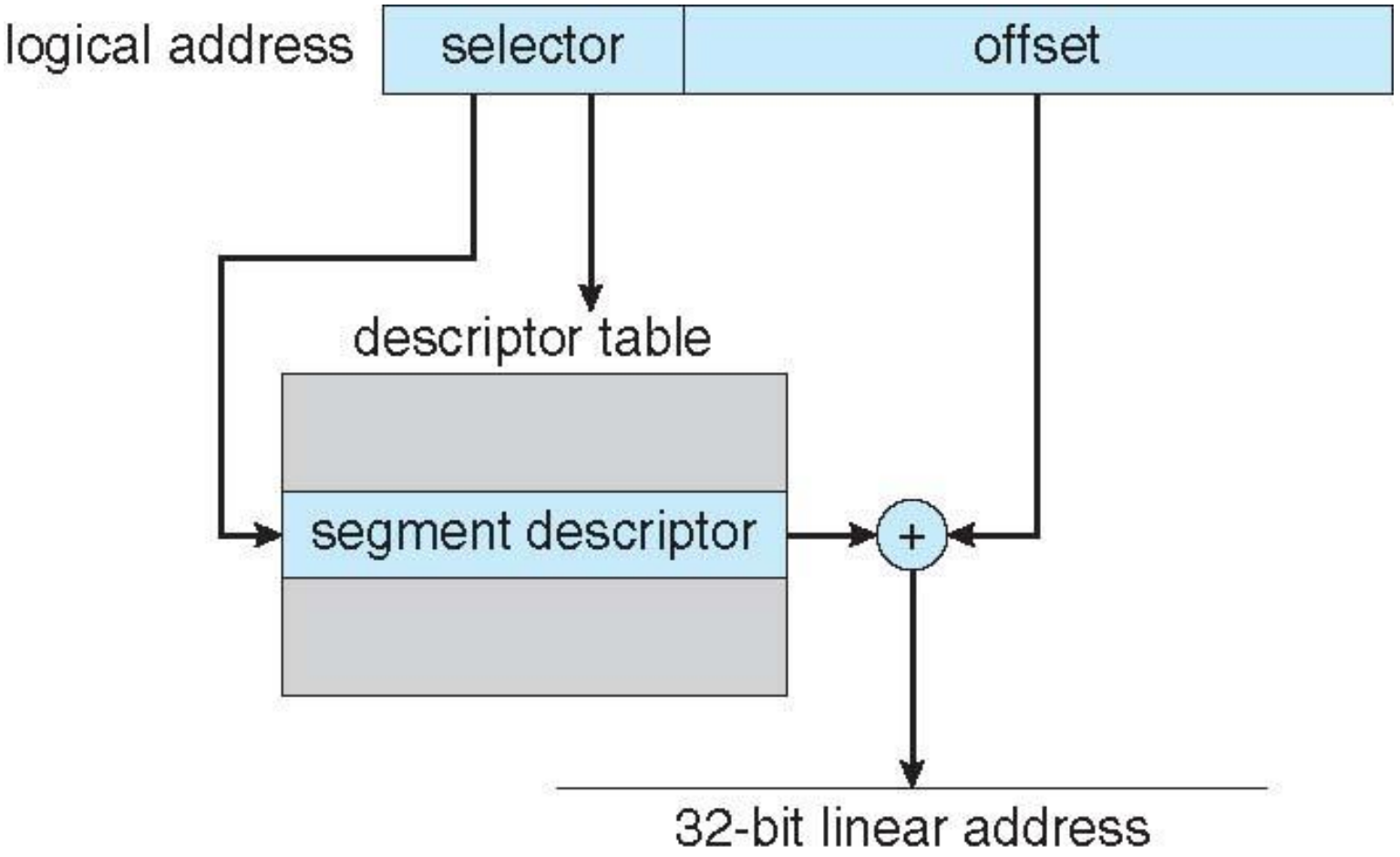
Page table=2^{20} entries



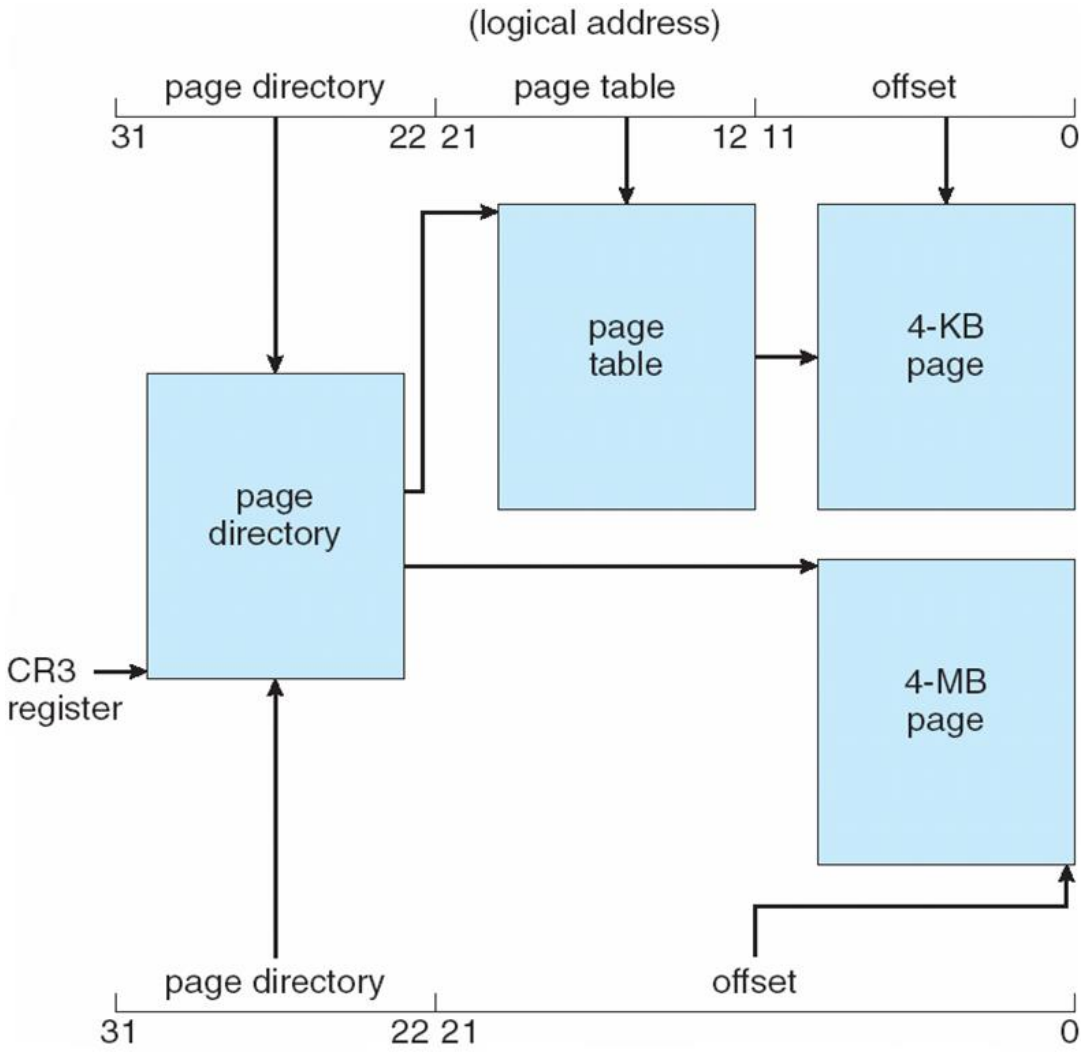
Example: The Intel Pentium



Intel Pentium Segmentation



Pentium Paging Architecture



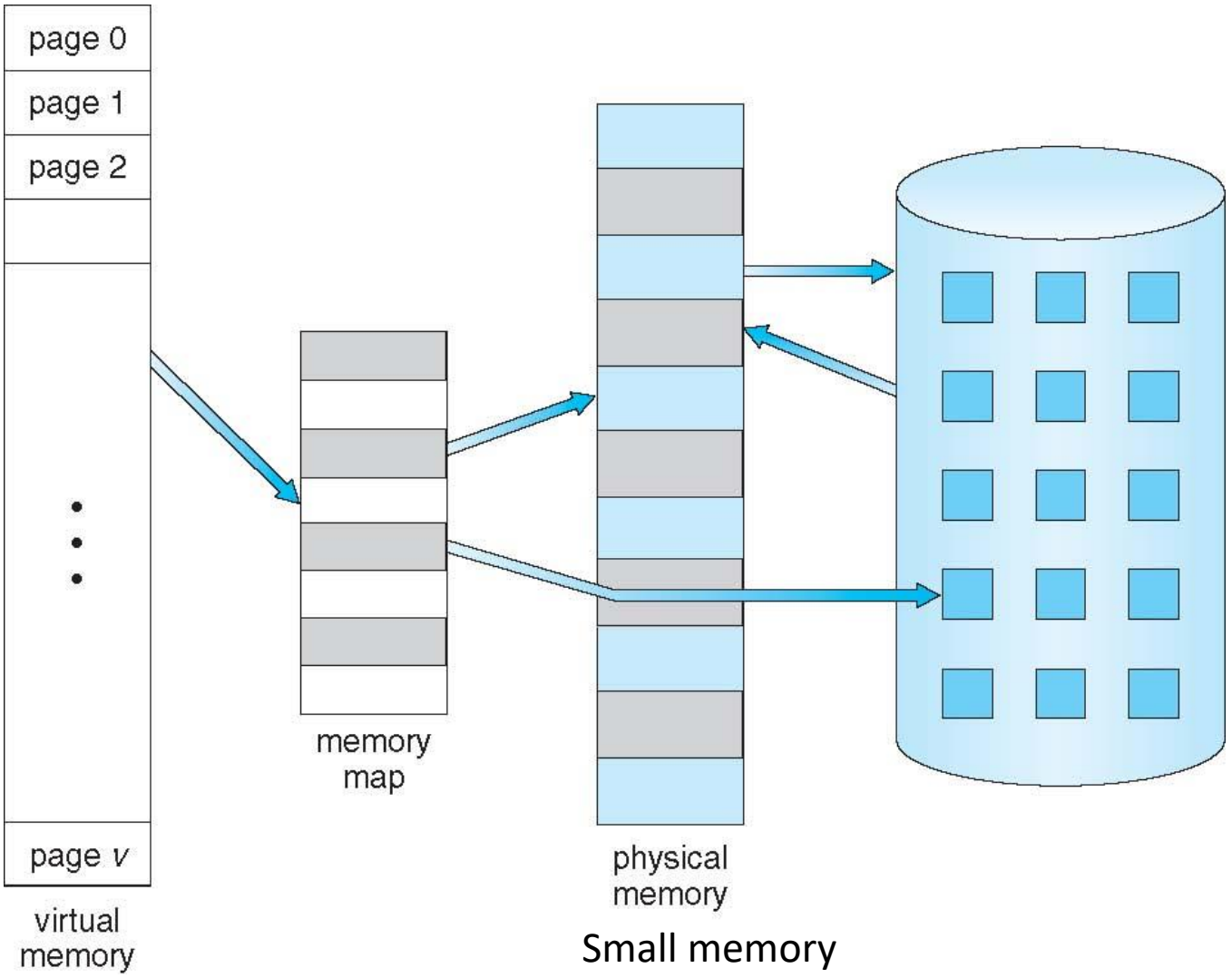
Virtual Memory

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - programs could be larger than physical memory
  - More processes can be accommodated

Virtual Memory That is Larger Than Physical Memory

Large virtual space



Classical paging

- Process P1 arrives
- Requires n pages => n frames must be available
- Allocate n frames to the process P1
- Create page table for P1

Allocate < n frames

Background

- **Virtual memory** – separation of user logical memory from physical memory
  - Extremely large logical space is available to programmer
  - Concentrate on the problem
- Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Starts with address 0, allocates contiguous logical memory
  - Physical memory
    - Collection of frame
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

- Bring a page into memory only when it is needed

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

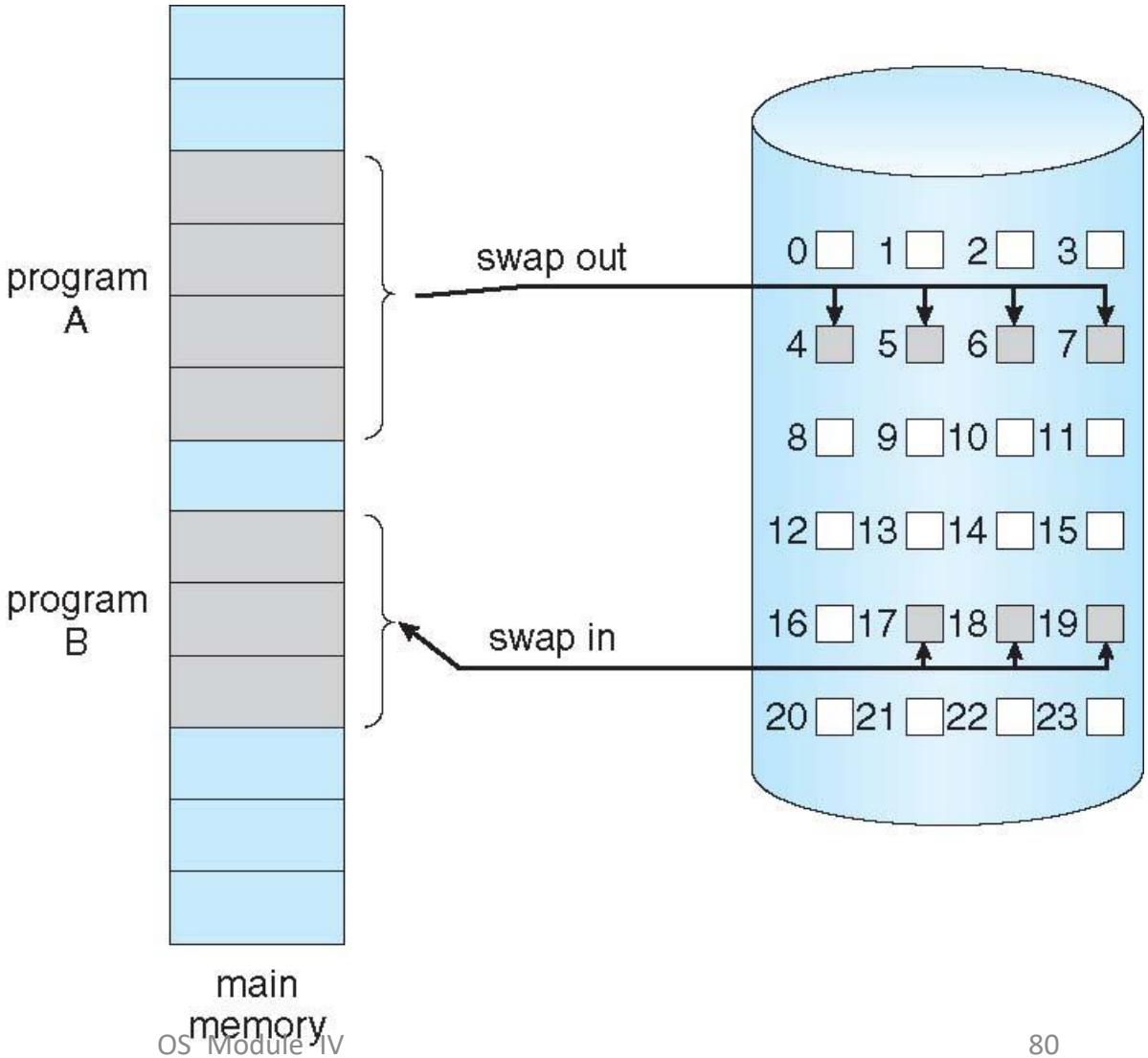
- Less I/O needed, no unnecessary I/O
  - Less memory needed
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

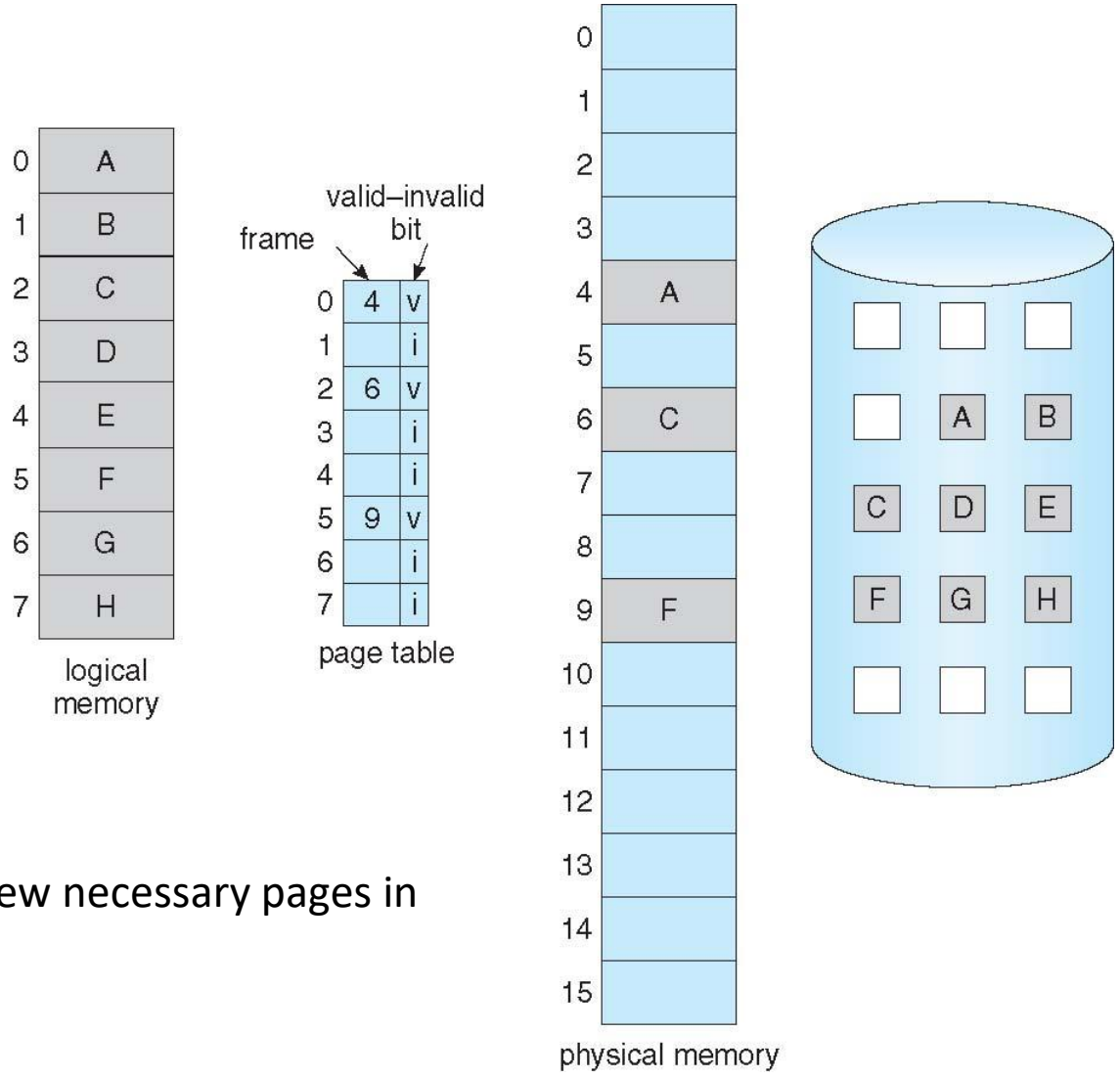
Valid address information is available in PCB

Transfer of a Paged Memory to Contiguous Disk Space

- When we want to execute a process, swap in
- Instead of swap in entire process, load page
- Pager



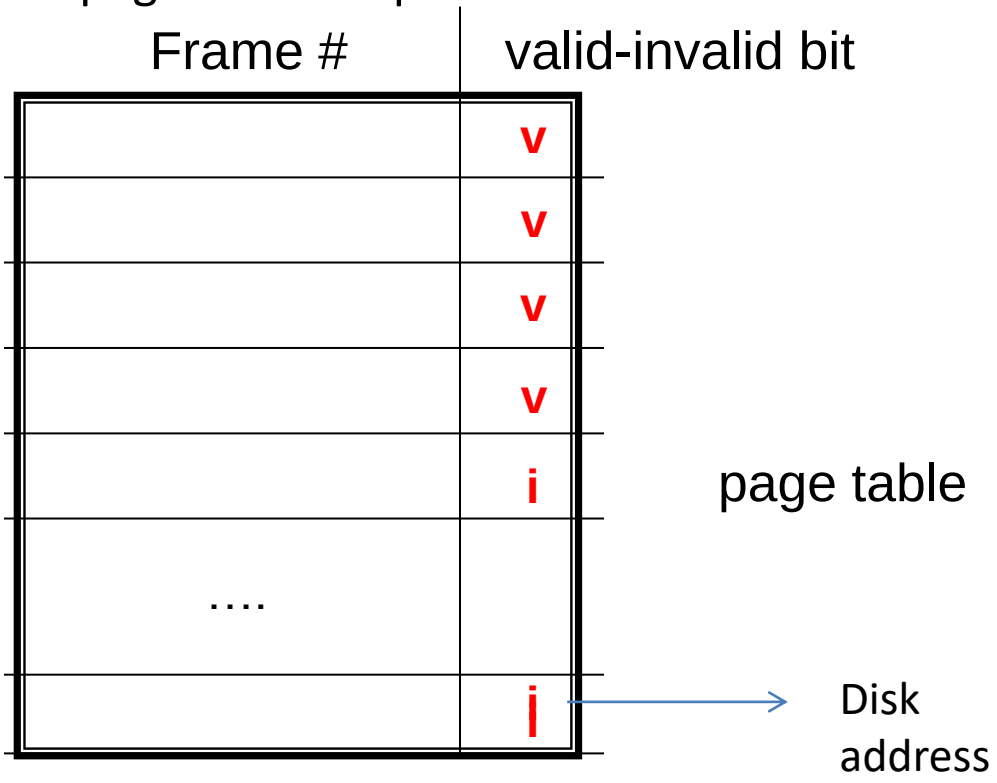
Page Table When Some Pages Are Not in Main Memory



Pager loads few necessary pages in memory

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Fault

- If the page in not in memory, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at PCB to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory (load the page)
2. Get empty frame
3. Swap page into frame via scheduled disk operation
4. Reset page table to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

What Happens if There is no Free Frame?

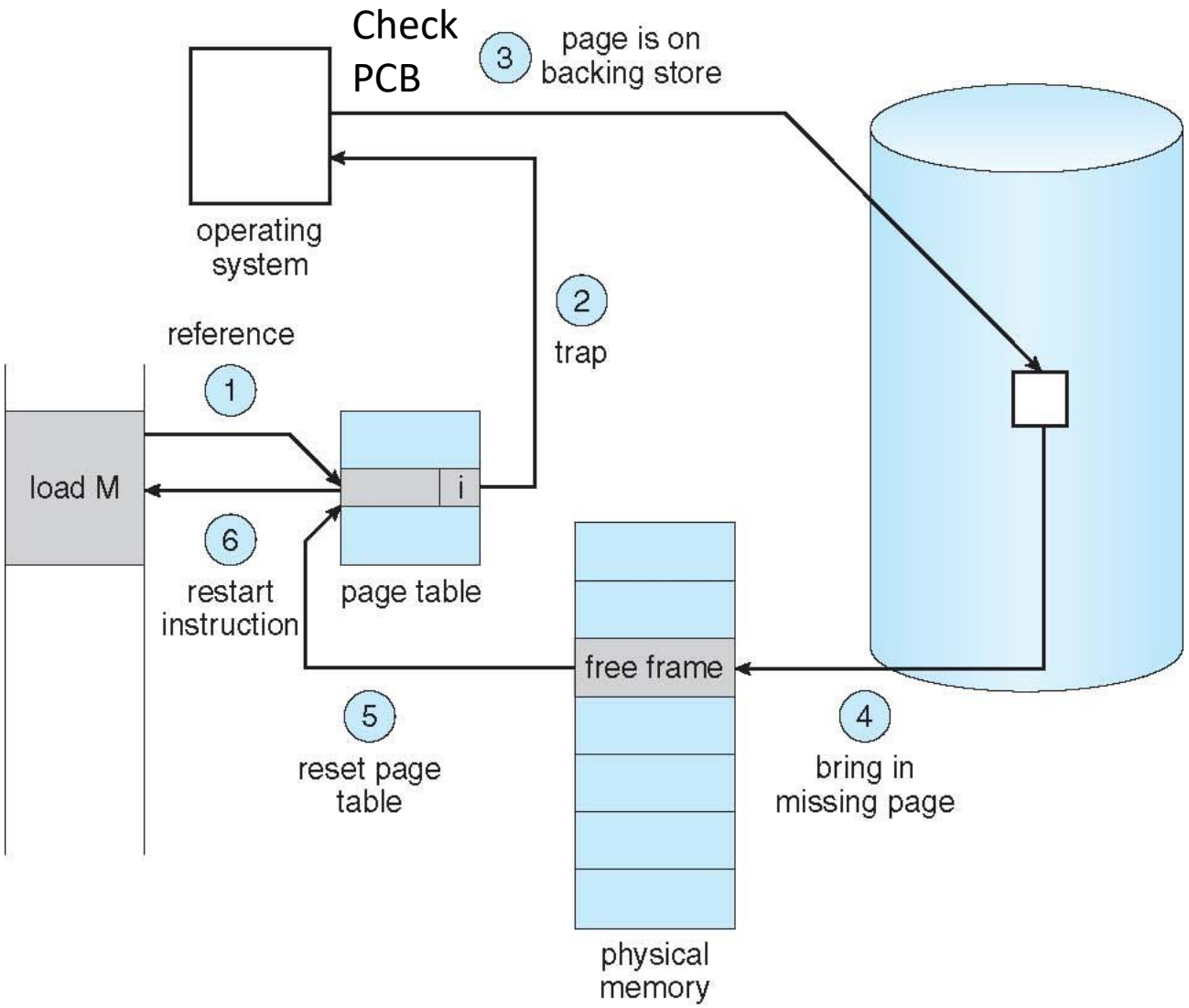
- Example
  - 40 frames in memory
  - 8 processes each needs 10 pages
  - 5 of them never used
- Two options
  - Run 4 processes (10 pages)
  - Run 8 processes (5 pages)
- Increase the degree of multiprogramming
  - Over allocating memory

- **Page fault**
  - **No free frame**
  - Terminate? swap out? replace the page?

- **Page replacement** – find some page in memory, not really in use, page it out
  - Performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times

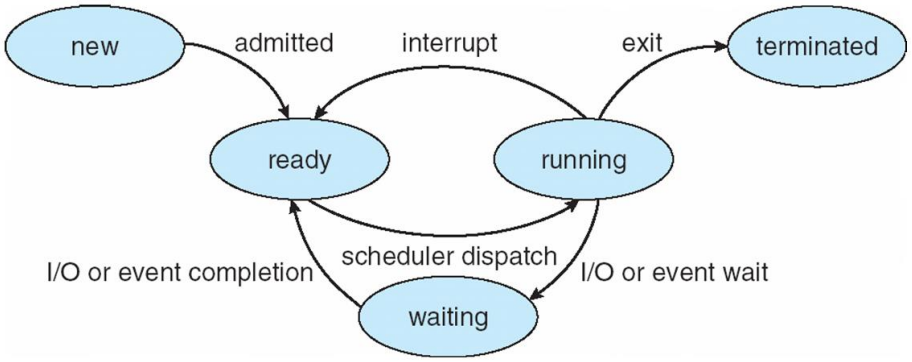
Steps in Handling a Page Fault



Pure Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - Swap in that page
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages (instruction + data) -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart after page fault

Steps in the ISR



- In Demand Paging

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Get a free frame
6. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
7. While waiting, allocate the CPU to some other user
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and process state of the running process
10. Determine that the interrupt was from the disk
11. Correct the page table and other tables to show page is now in memory
12. Wait for the CPU to be allocated to this process again
13. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

Demand paging affects the performance of the computer systems

- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

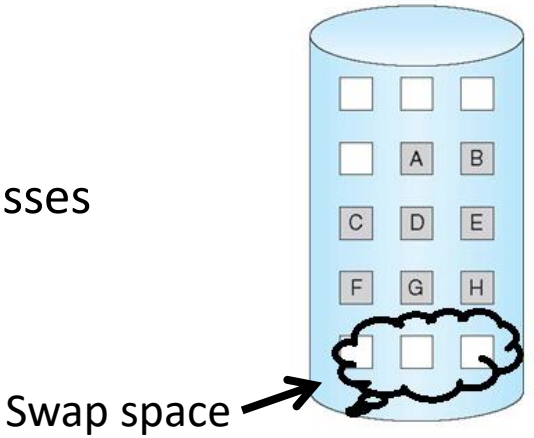
$$\begin{aligned} EAT = (1 - p) \times & \text{ memory access} \\ & + p \ (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead} \\ & \quad) \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Better utilization of swap space



Allocation of Frames

- How do we allocate the fixed amount of memory among various processes?

- Single user system
  - Trivial

Allocation of Frames

- Each process needs *minimum* number of frames
- Minimum number is defined by the instruction set
- Page fault forces to restart the instruction
  - Enough frames to hold all the pages for that instruction
- Example:
  - Single address instruction (2 frames)
  - Two address instruction (3 frames)
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - proportional allocation

Fixed and proportional Allocation

- Equal allocation – m frames and n processes
  - Each process gets m/n
- For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Unfair for small and large sized processes
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

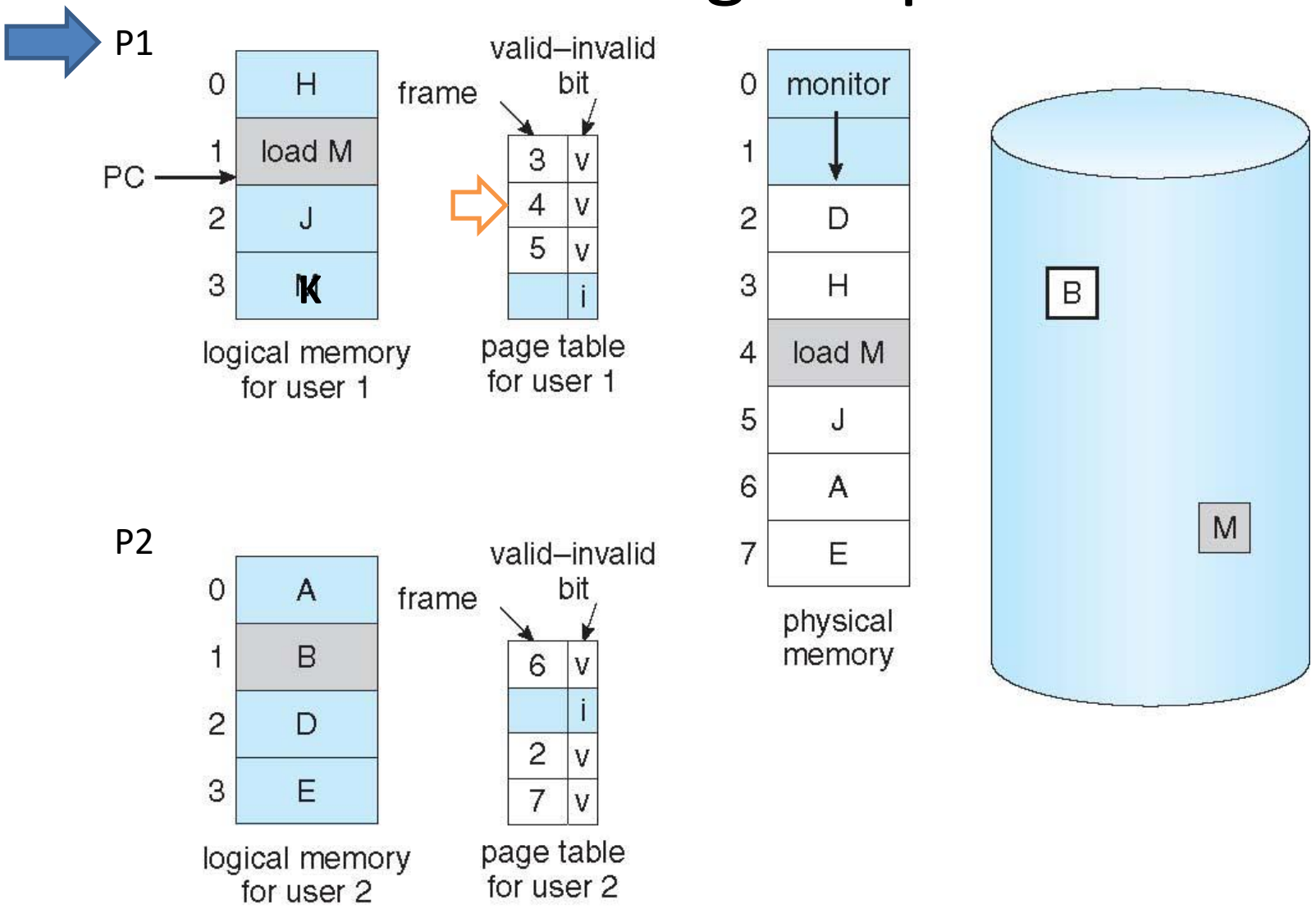
Priority Allocation

Allocation of frames

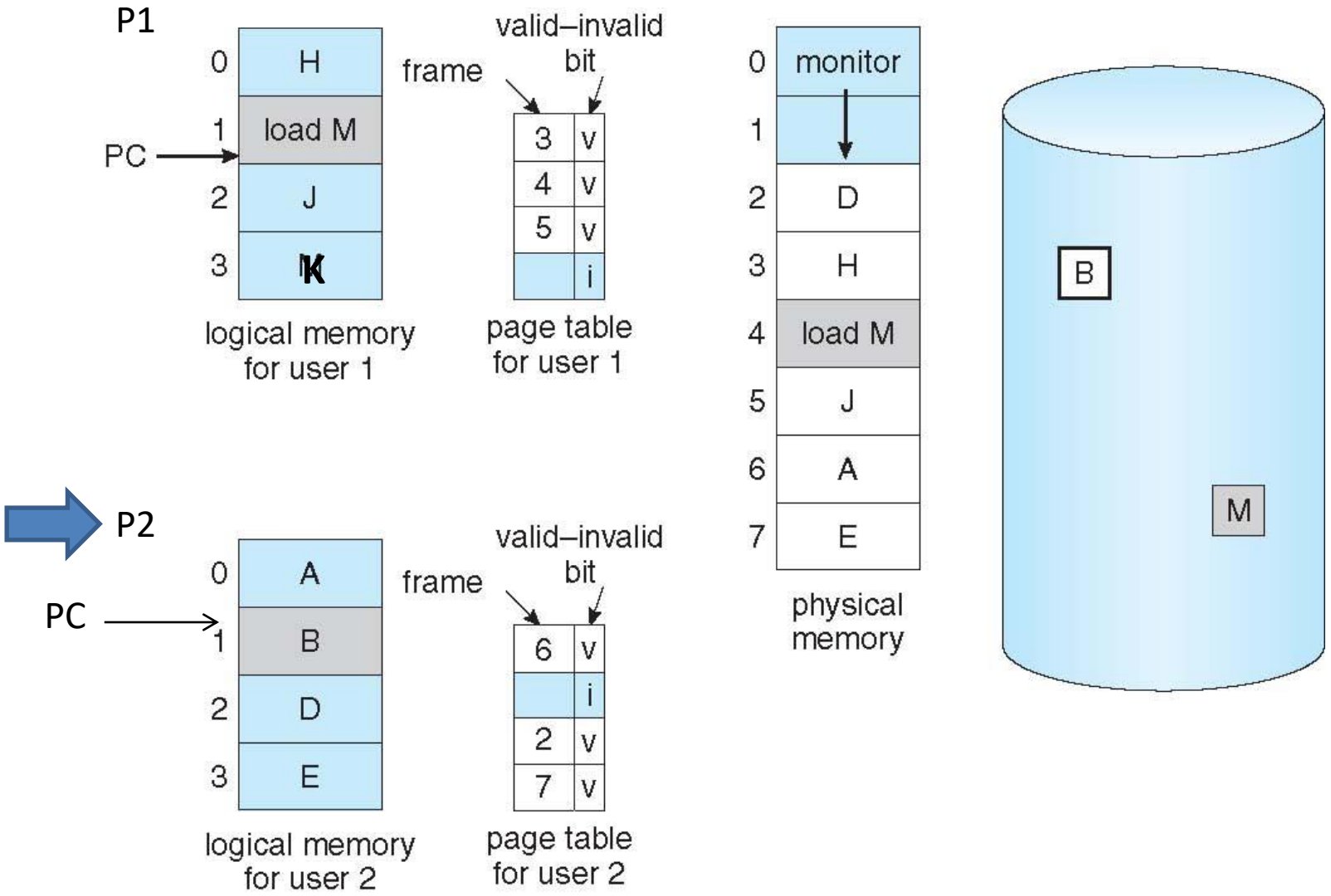
- Depends on multiprogramming level

- Use a proportional allocation scheme using priorities along with size

Need For Page Replacement



Need For Page Replacement



Basic Page Replacement

1. Find the location of the desired page on disk

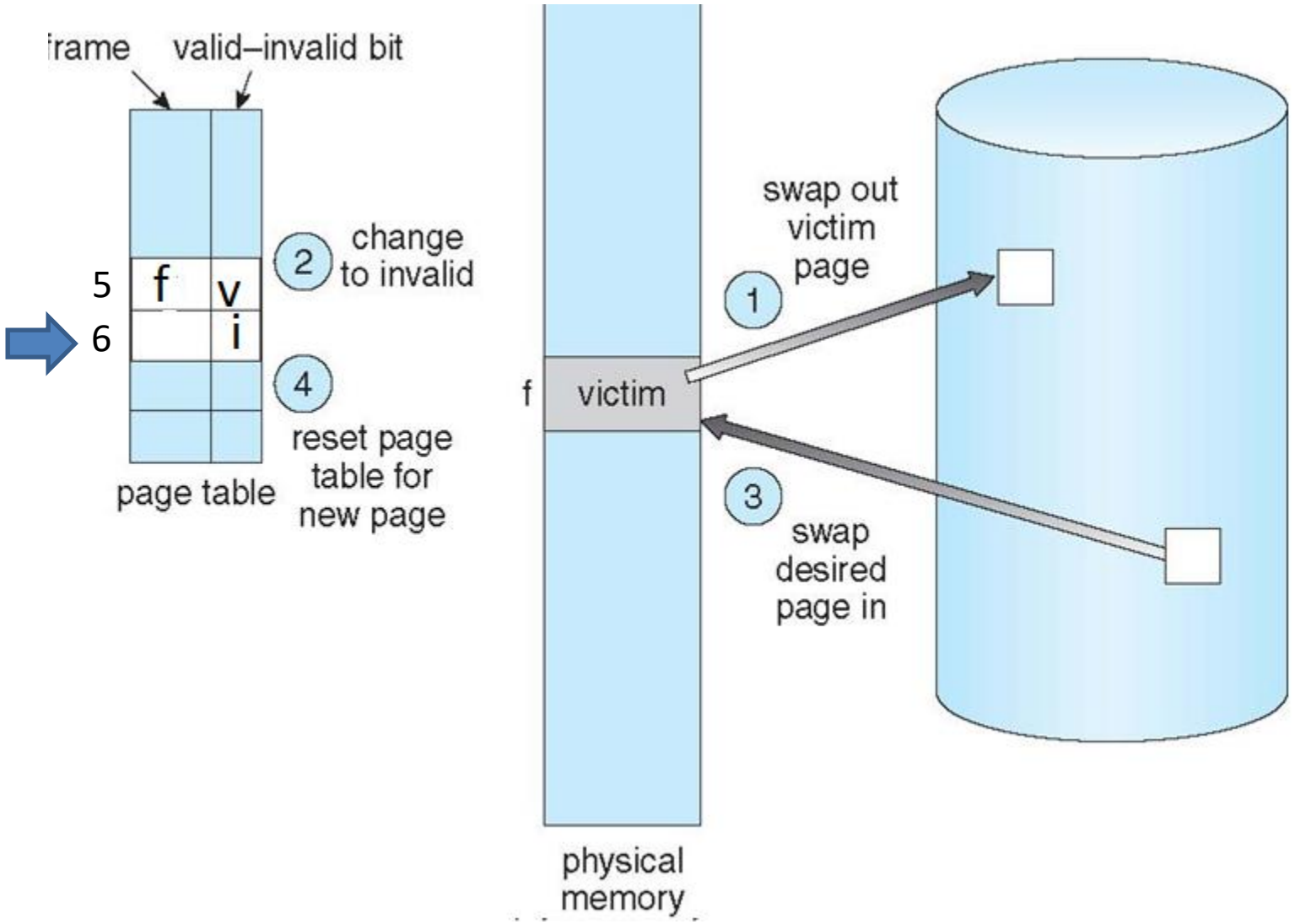
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame (of that process)**
     - Write victim frame to disk

3. Bring the desired page into the (newly) free frame; update the page and frame tables

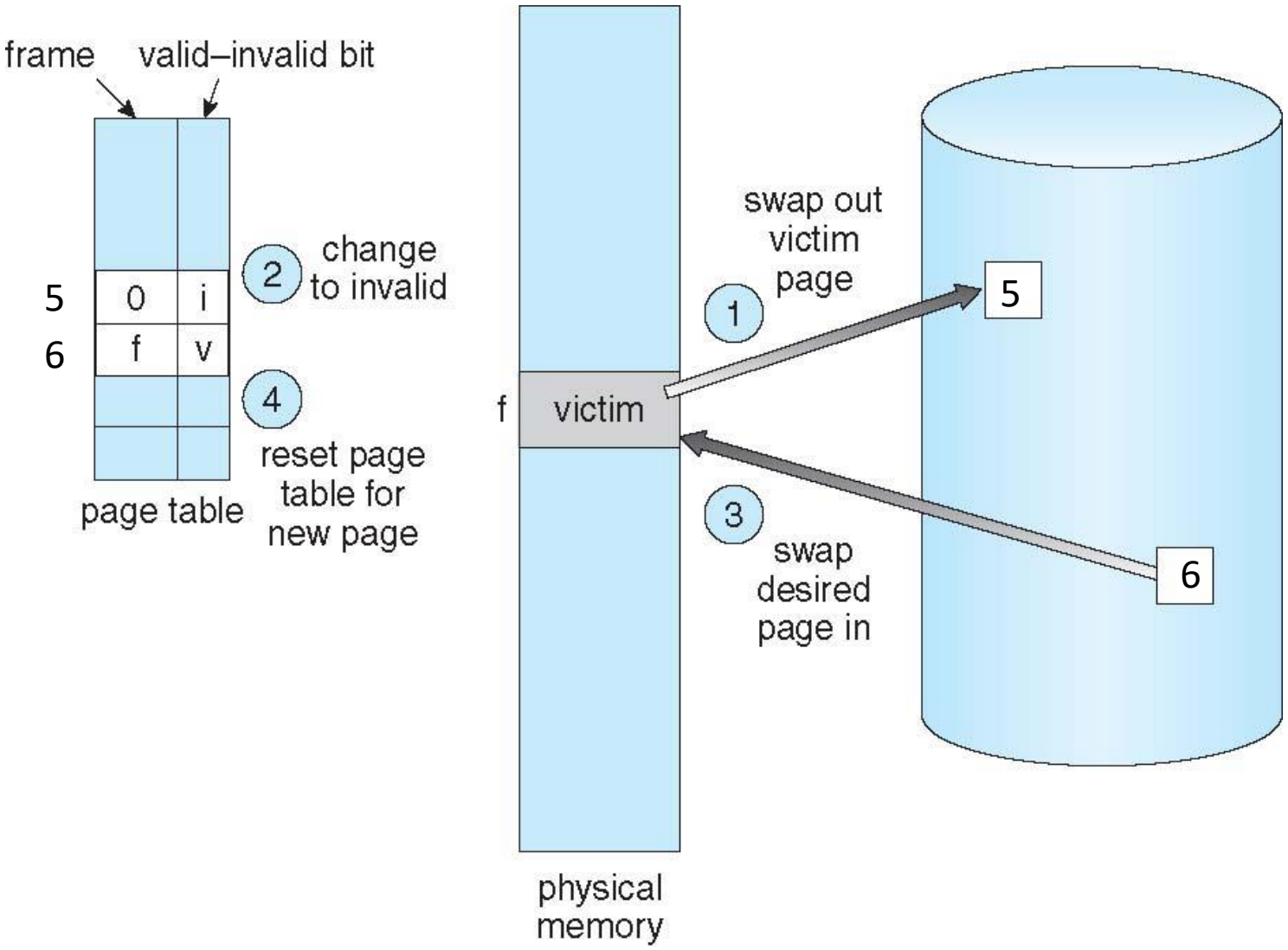
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing Effective memory access time

Page Replacement



Page Replacement



Evaluation

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- Trace the memory reference of a process

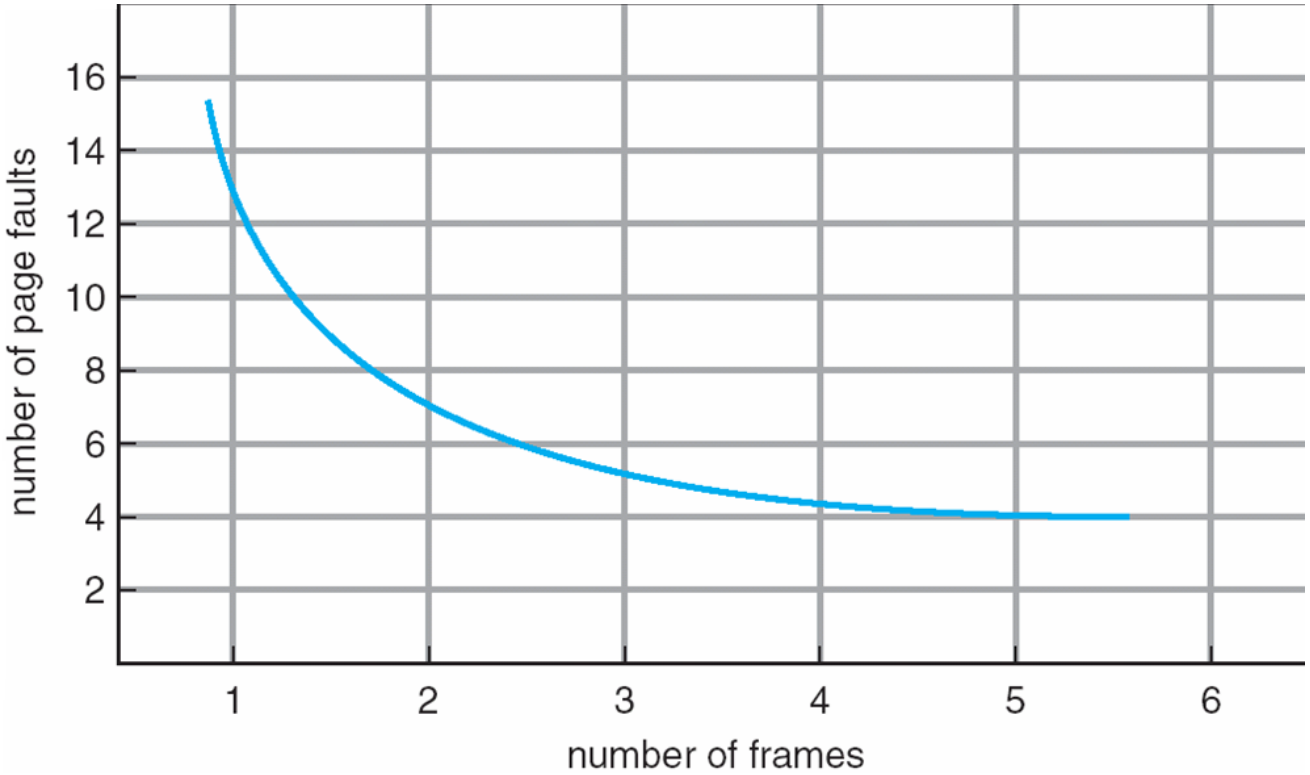
0100, 0432, 0101, 0612, 0102, 0104, 0101, 0611, 0102

- Page size 100B
- Reference string 1, 4, 1, 6, 1, 6

- In all our examples, the reference string is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

Associates a time with each frame when the page was brought into memory

When a page must be replaced, the oldest one is chosen

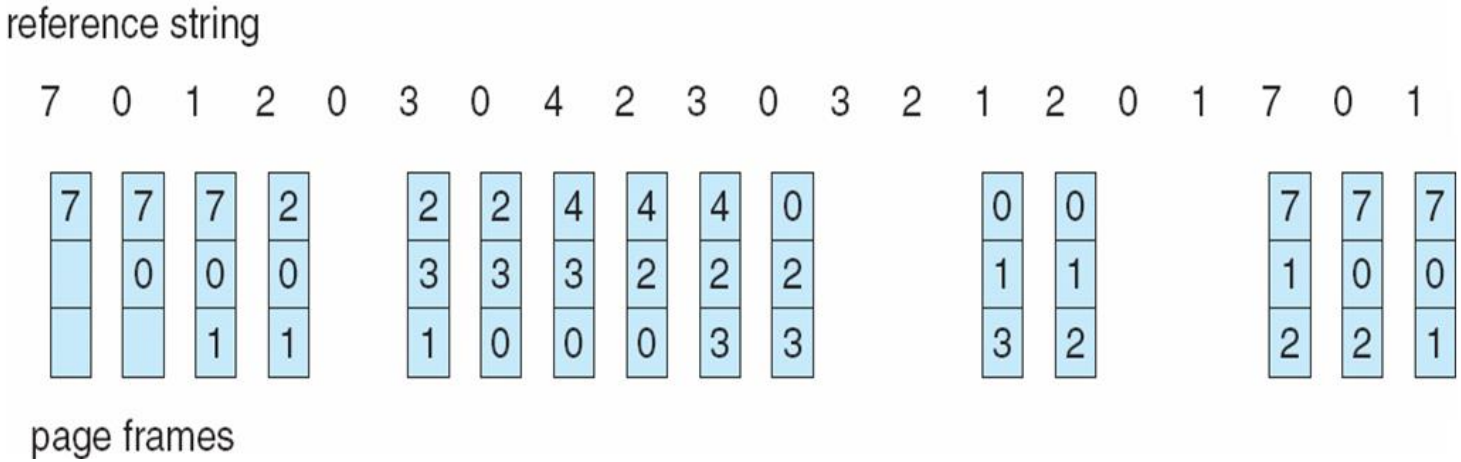
Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

Limitation:

A variable is initialized early and constantly used

FIFO Page Replacement

3 frames (3 pages can be in memory at a time)



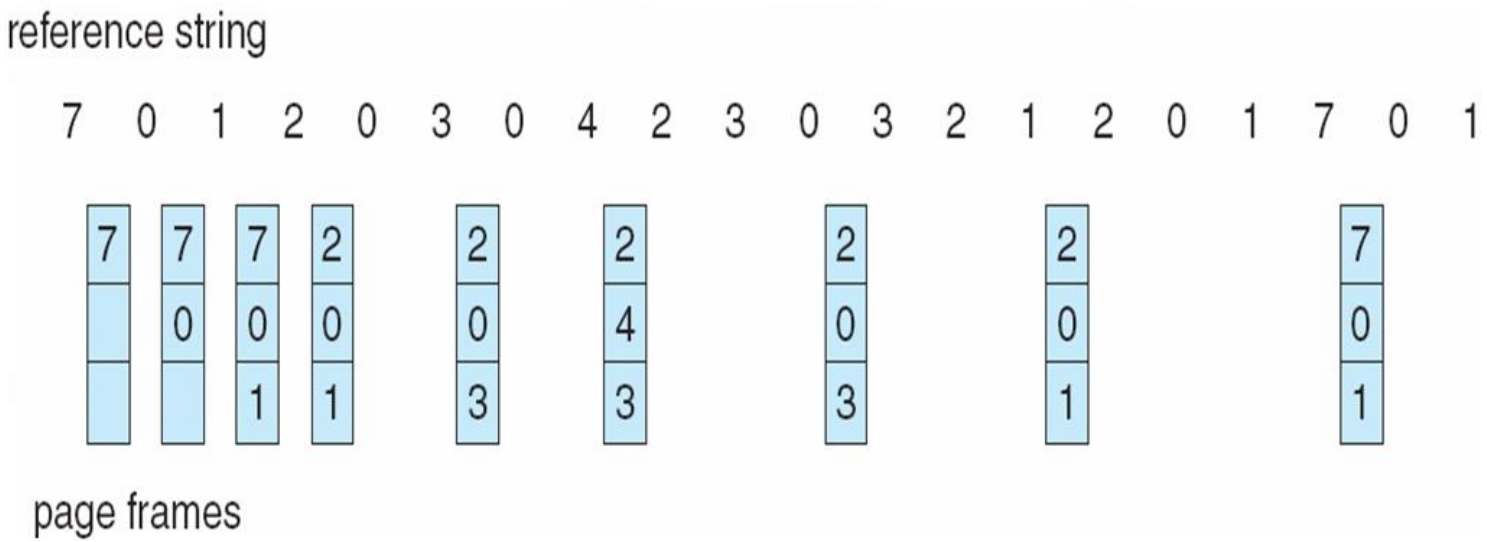
15 page faults

- How to track ages of pages?
  - Just use a FIFO queue to hold all the pages in memory
  - Replace the page at the head
  - Insert at tail

Optimal Algorithm

- Replace page that will not be used for longest period of time
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

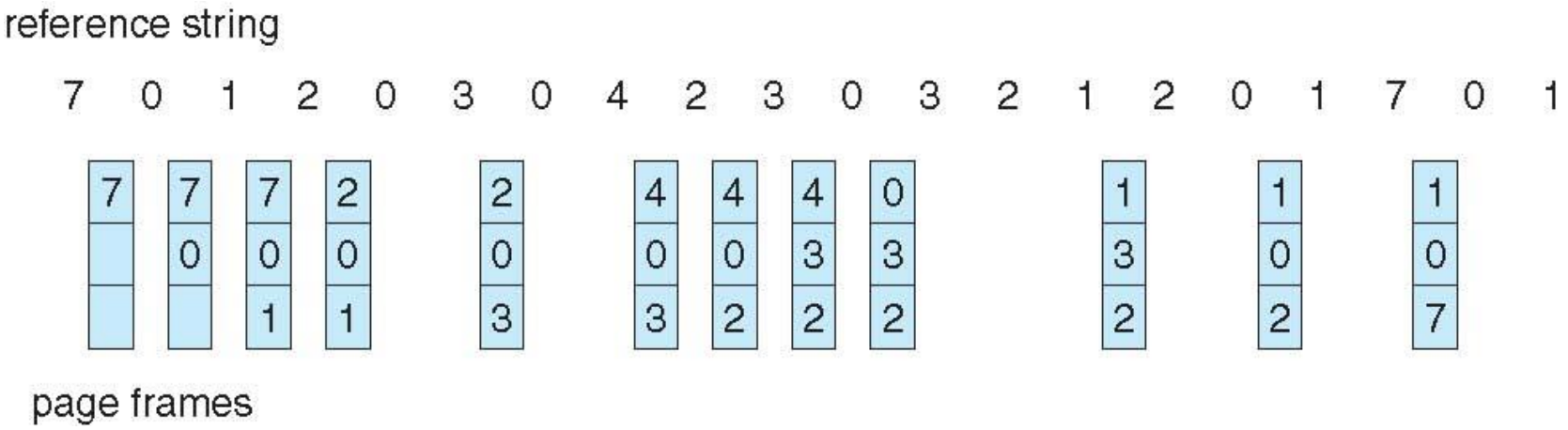
Optimal Page Replacement



9 page faults

Least Recently Used (LRU) Algorithm

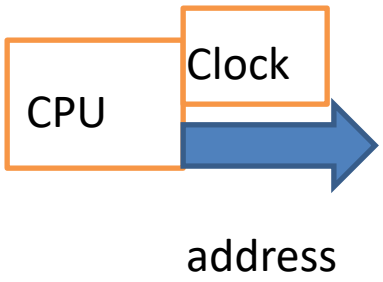
- **Use past knowledge rather than future**
  - Past is the proxy of future
- Replace page that has not been used in the most of the time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm-Implementation

- **Counter implementation**
  - CPU maintains a clock
  - Every page entry has a Time of use;
    - every time page is referenced, copy the clock into the time of use
  - When a page needs to be replaced, look at the "Time of use" to find smallest value
    - Search through table needed

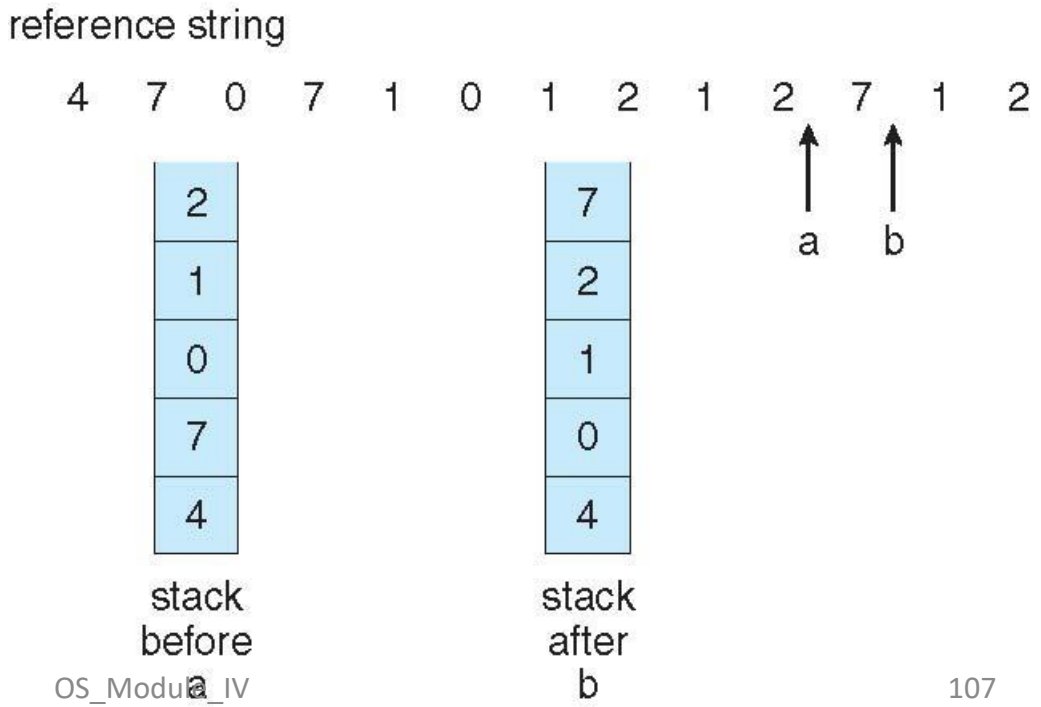


Page table

			Time of Use
0			
1			
2			
3			

LRU Algorithm-Implementation

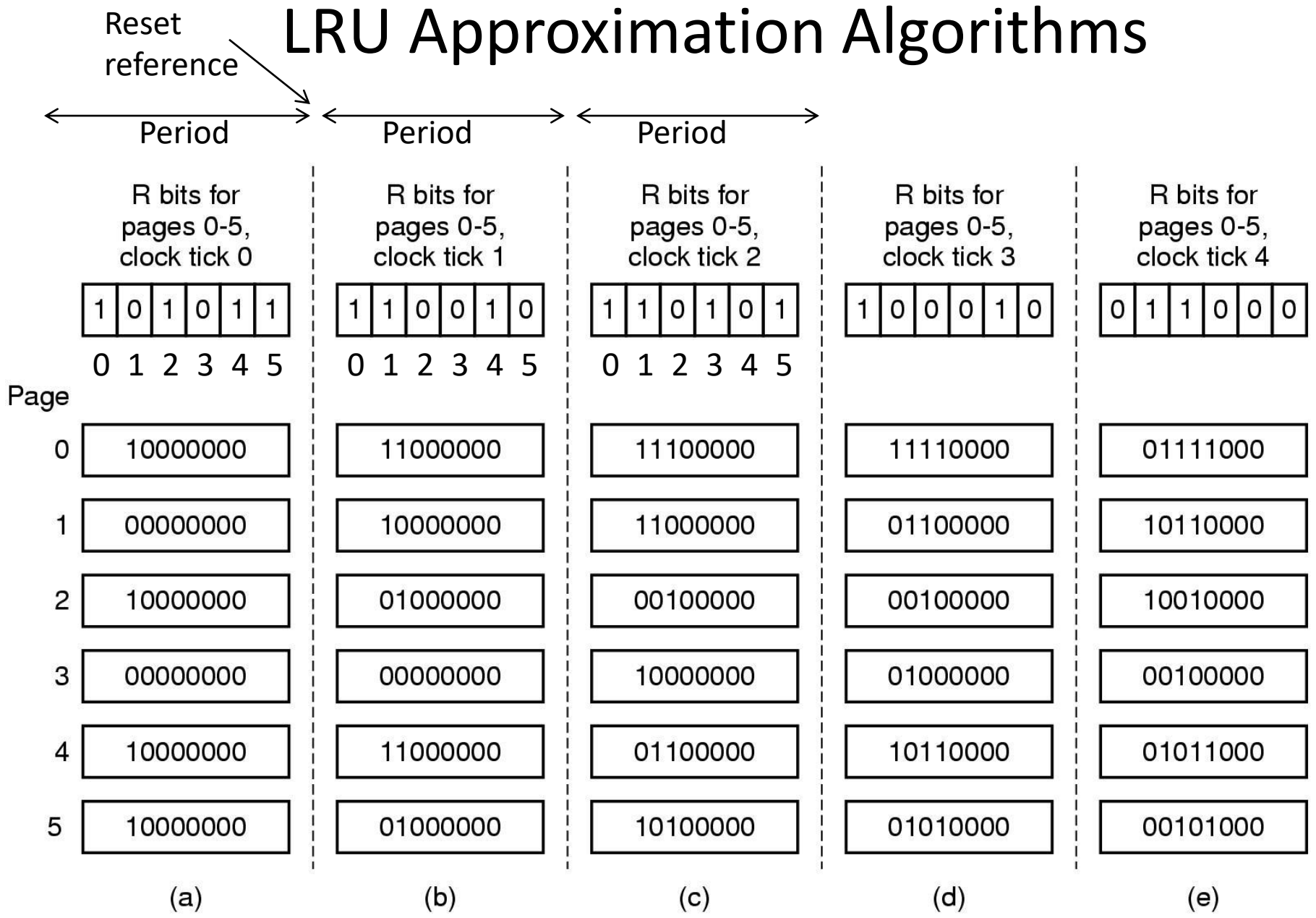
- **Stack implementation**
  - Keep a stack of page numbers in a double linked list form:
  - Page referenced:
    - move it to the top
  - Victim page is the bottom page



LRU Approximation Algorithms

- **LRU needs special hardware and still slow**
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
- Additional reference bit algorithm
  - Record the reference bits in regular interval
  - Keep a 8 bit string for each page in memory
  - At regular interval, timer copies the reference bit to the high order bit (MSB) of the string.
  - Shift the other bits right side by one bit

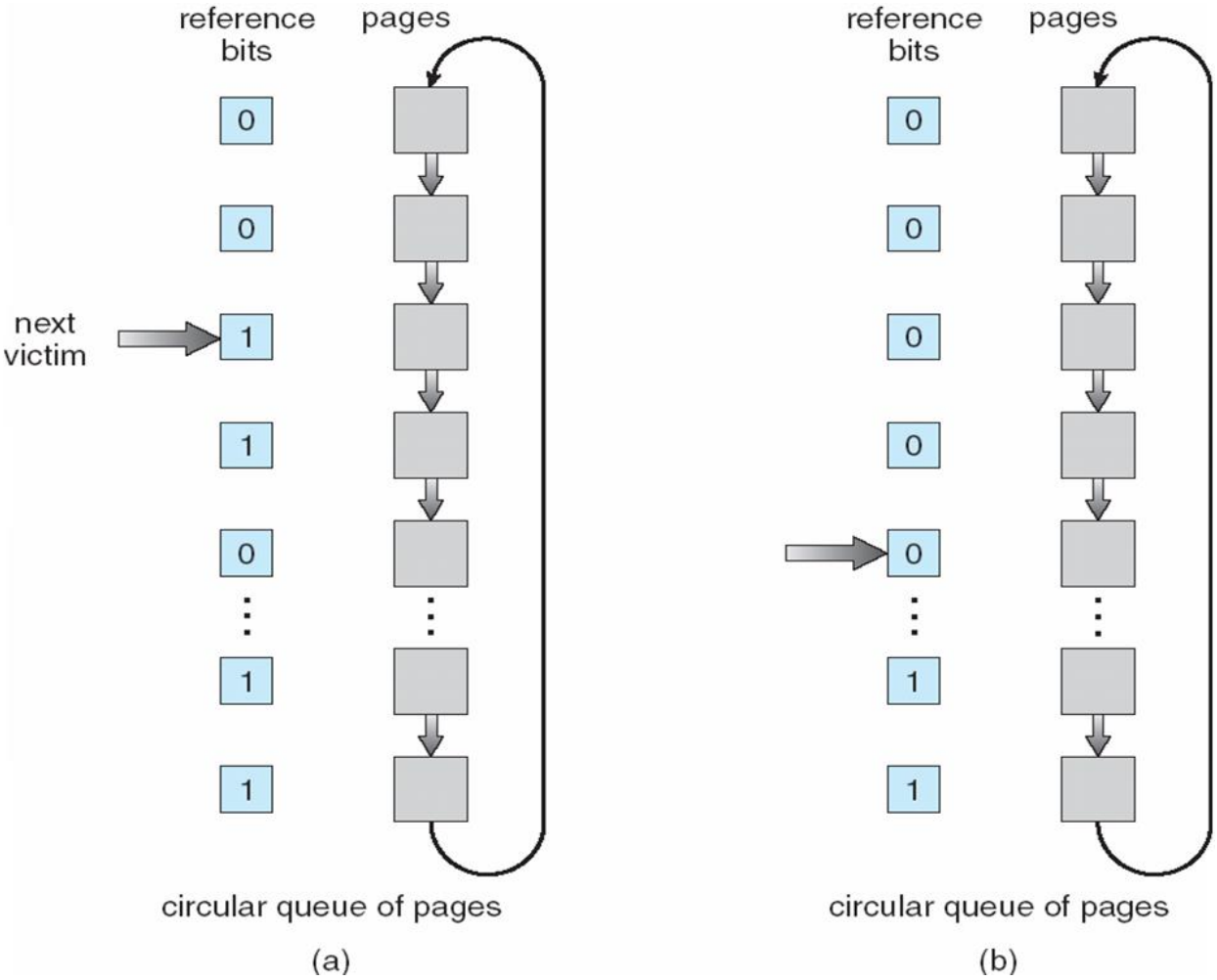
LRU Approximation Algorithms



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
- Additional reference bit algorithm
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory (reset the time)
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



{Reference bit, Dirty bit} combination

Counting Algorithms

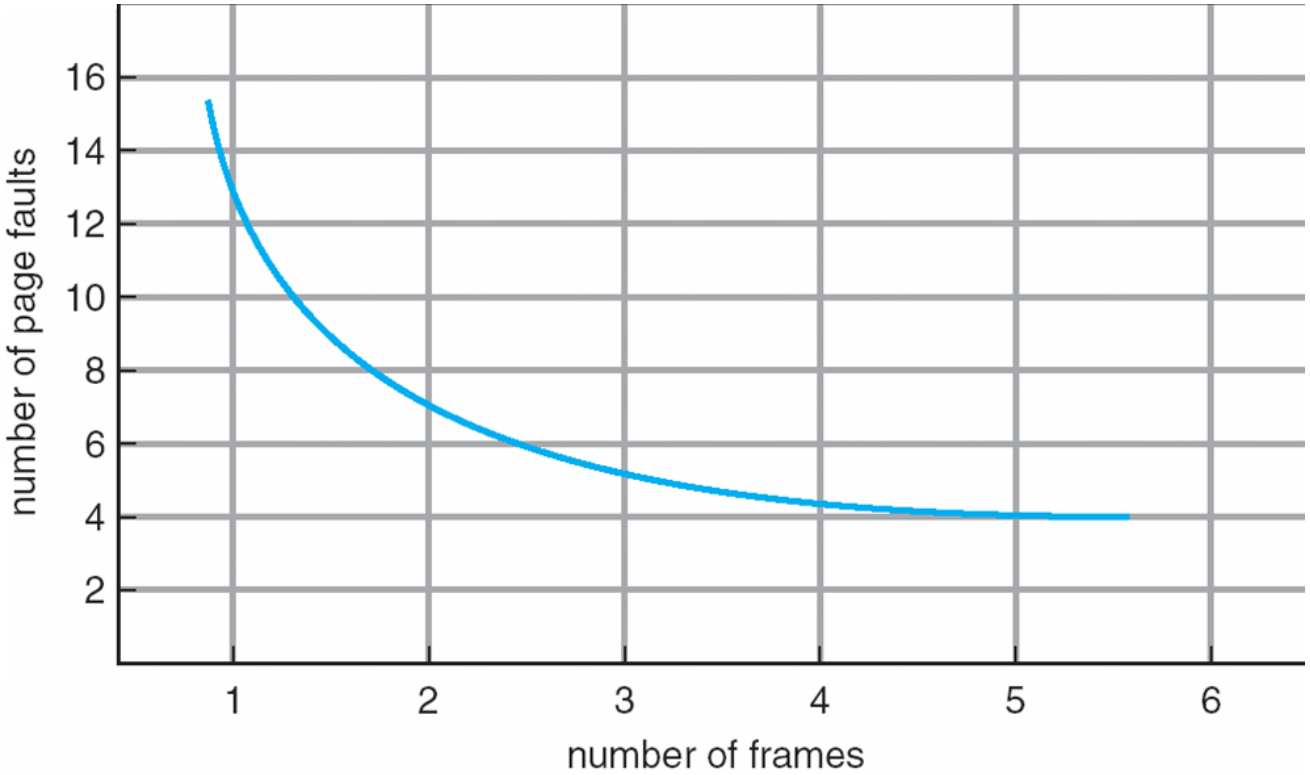
- Keep a counter of the number of references that have been made to each page

Least and most frequently used

- **LFU Algorithm**: replaces page with smallest count

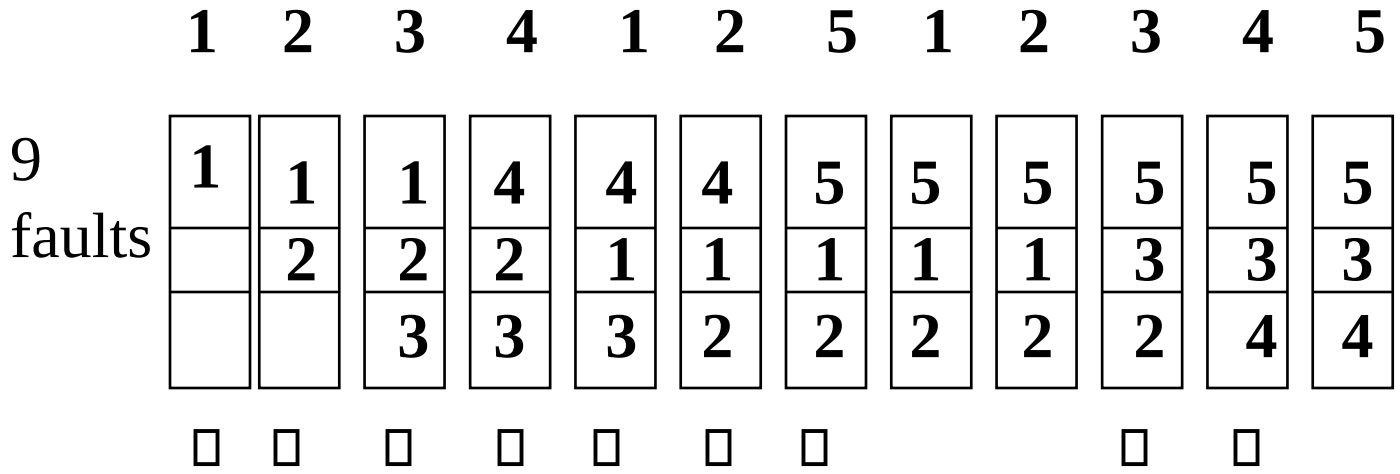
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Graph of Page Faults Versus The Number of Frames

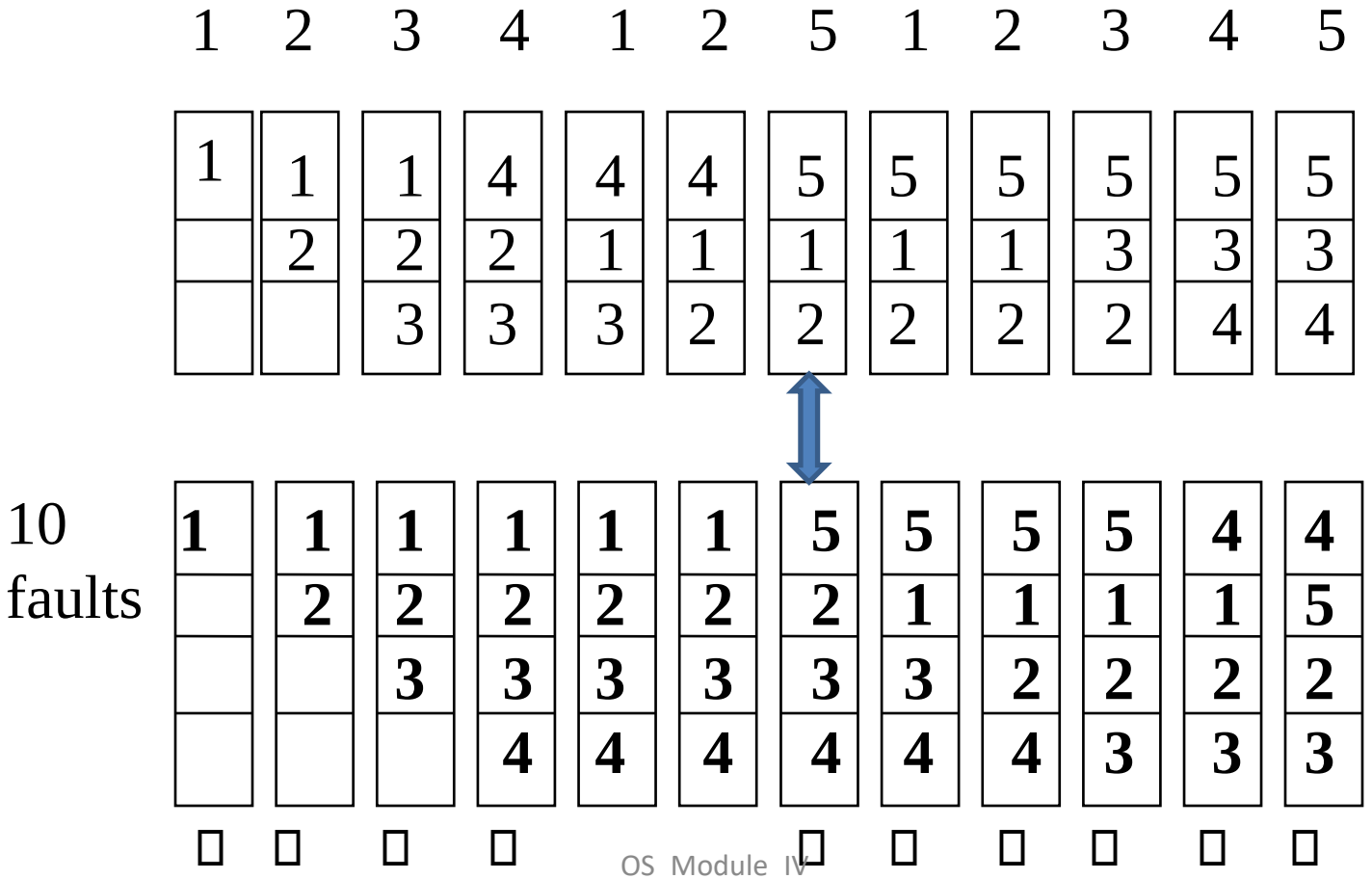


Increase in page frame decreases page fault rate?

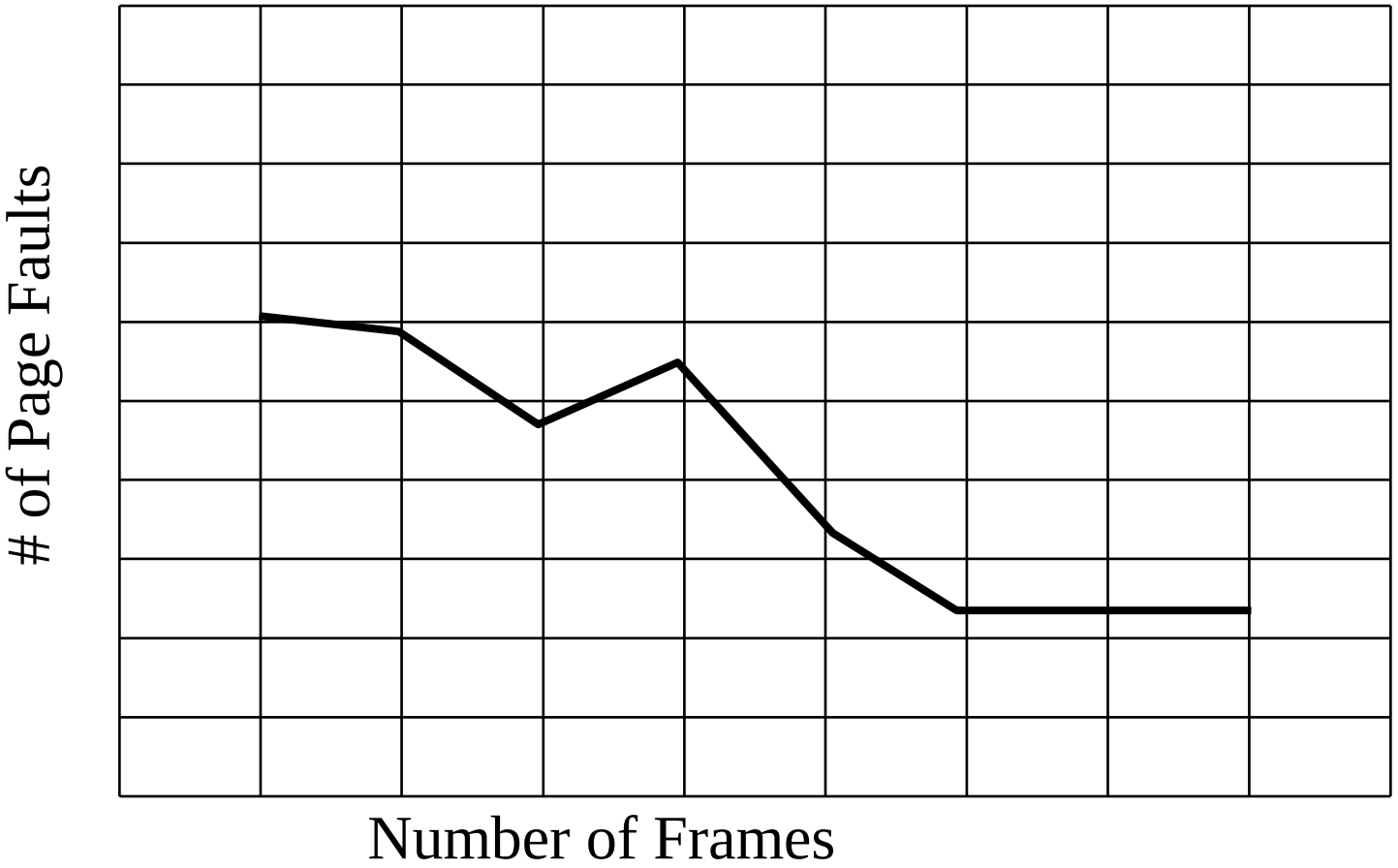
FIFO Example – Program - 5 pages, 3 frames of Memory allocated



FIFO Example – Program - 5 pages, 4 frames of Memory allocated



Belady's Anomaly



Belady's Anomaly

- This most unexpected result is known as Belady's anomaly – for some page-replacement algorithms, the page fault rate may increase as the number of allocated frames increases

- Is there a characterization of algorithms susceptible to Belady's anomaly?

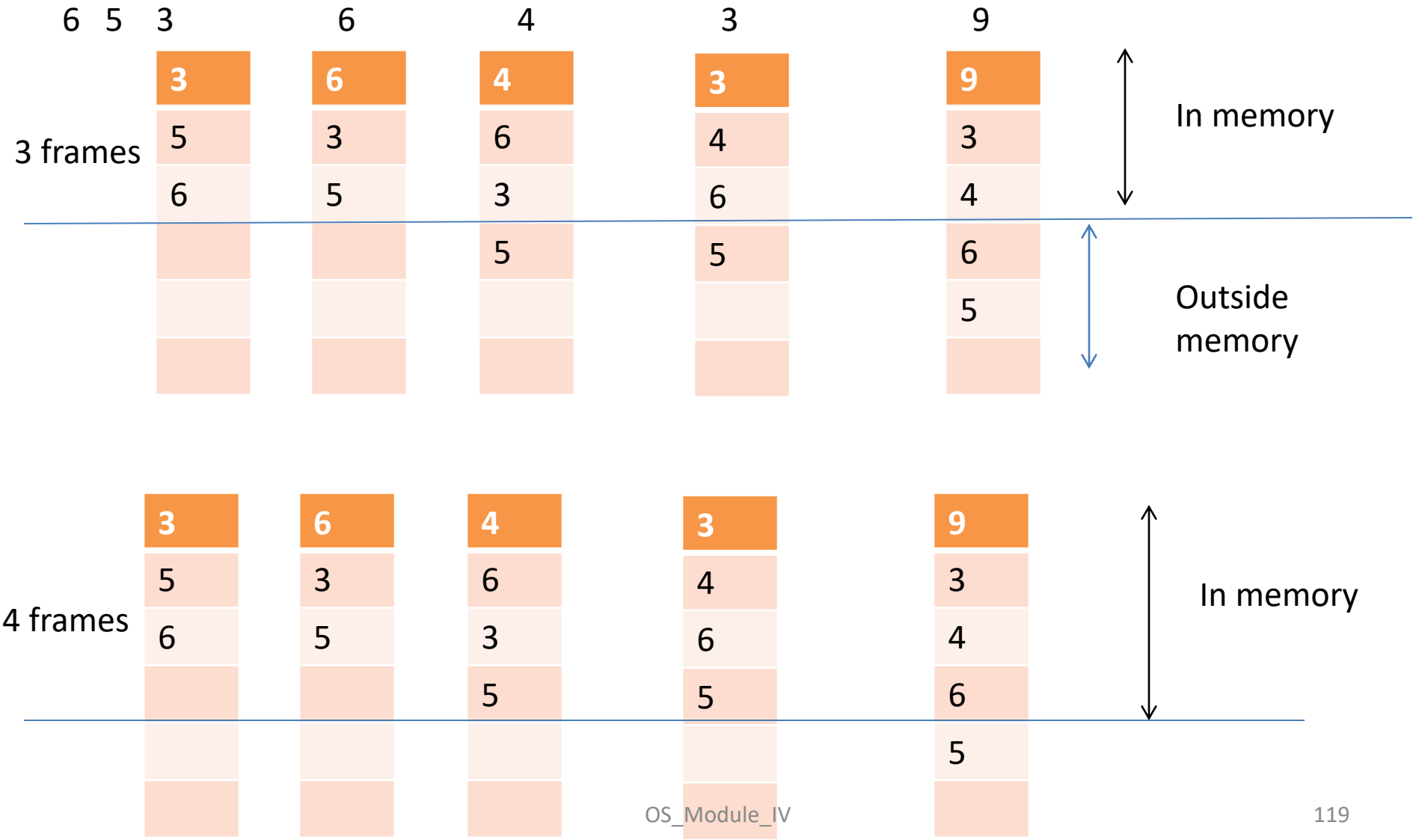
Stack Algorithms

- Certain page replacement algorithms are more “well behaved” than others
- (In the FIFO example), the problem arises because the set of pages loaded with a memory allocation of 3 frames is not necessarily also loaded with a memory allocation of 4 frames

- There are a set of paging algorithms whereby the set of pages loaded with an allocation of *m* frames is always a subset of the set of pages loaded with an allocation of *m* +1 frames. This property is called the inclusion property

- Algorithms that satisfy the inclusion property are not subject to Belady’s anomaly. FIFO does not satisfy the inclusion property and is not a stack algorithm

LRU algorithm



Global vs. Local Allocation

- Frames are allocated to various processes

- If process P_i generates a page fault
  - select for replacement one of its frames
  - select for replacement a frame from another process

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput ----- so more common
- Processes can not control its own page fault rate
  - Depends on the paging behavior of other processes

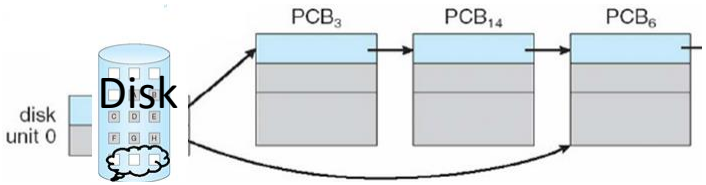
Thrashing

- If a process uses a set of “active pages”
  - Number of allocated frames is less than that
- Page-fault
  - Replace some “active” page
  - But quickly need replaced “active” frame back
  - Quickly a page fault, again and again
  - **Thrashing** ≡ a process is busy swapping pages in and out

- OS monitors CPU utilization
  - If low? Increase the degree of multiprogramming

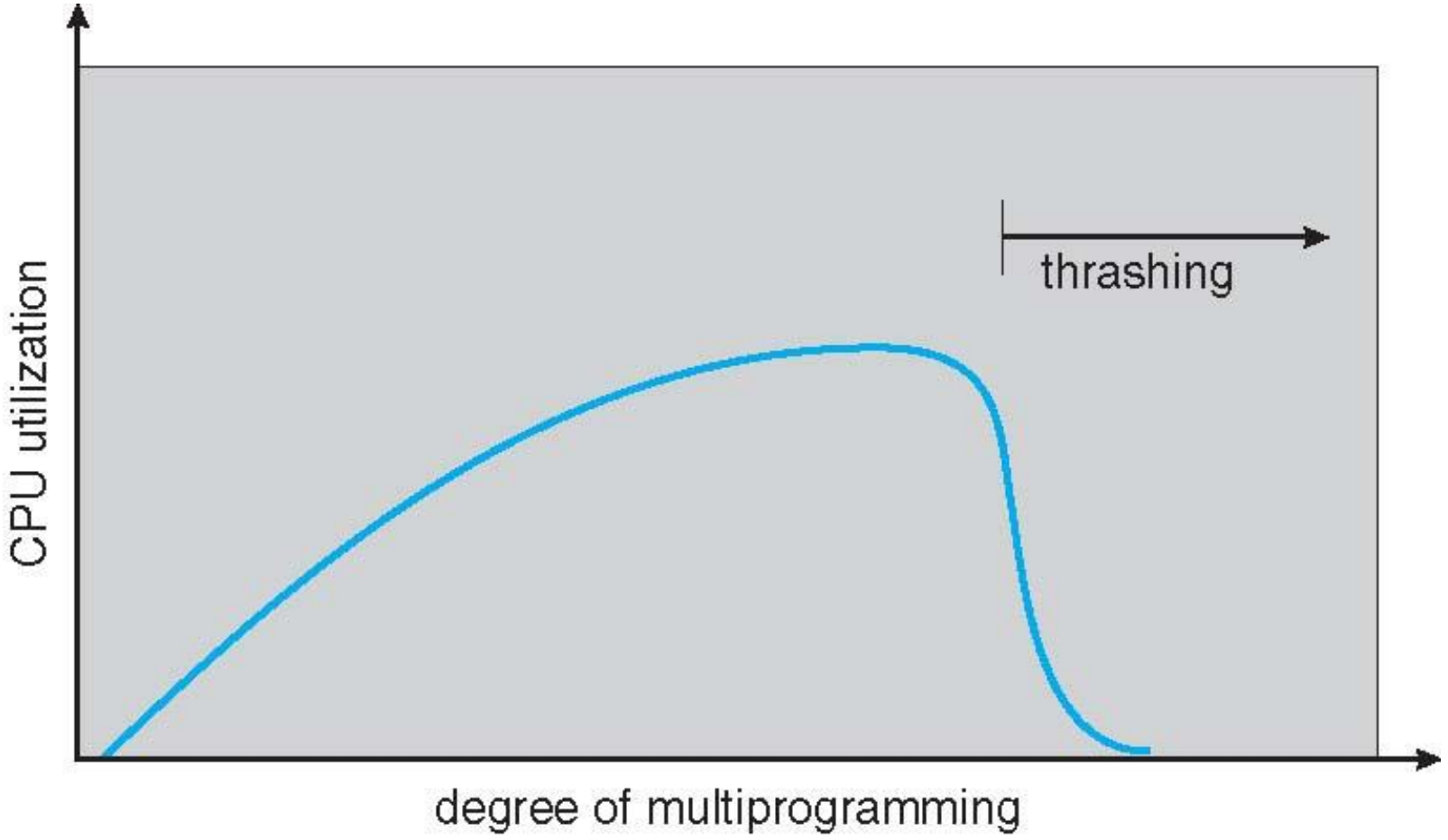
Thrashing

- Global page replacement
  - Process enters new phase (subroutine call) execution
  - Page fault
  - Taking frames from other processes
    - Replace “active” frames of other processes
  - These processes start page fault
  - These faulting processes wait on the device queue for disk
    - Ready queue empty
  - CPU utilization decreases



- CPU scheduler increases the degree of multiprogramming
  - More page faults
  - Drop in CPU utilization
- Page fault increases tremendously

Thrashing (Cont.)



Thrashing

- Solution
  - Local replacement
  - One process cannot steal frames from other processes
- Provide a process as many frames as needed
  - Able to load all active pages
  - How do we know?
  - Locality model

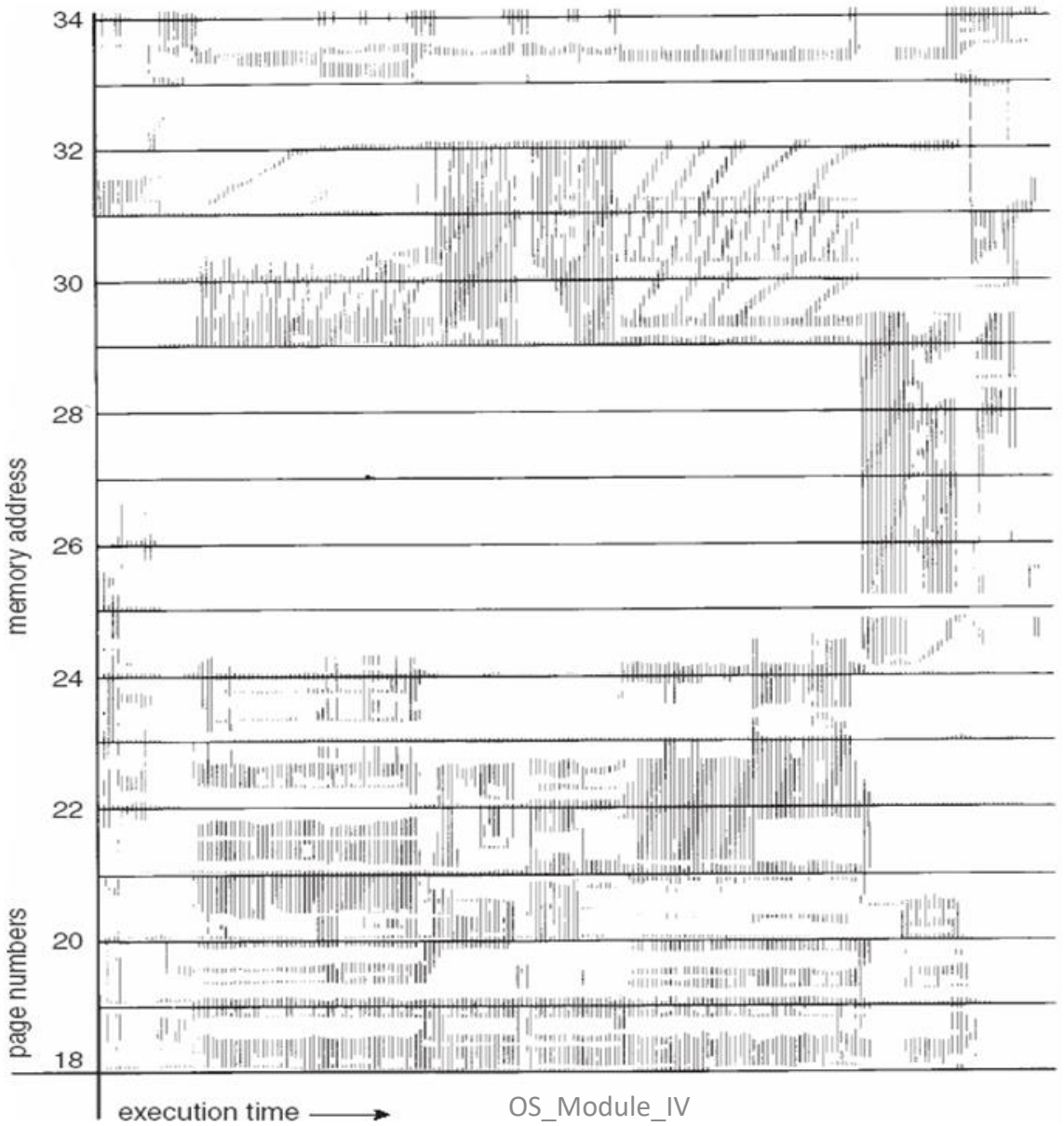
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

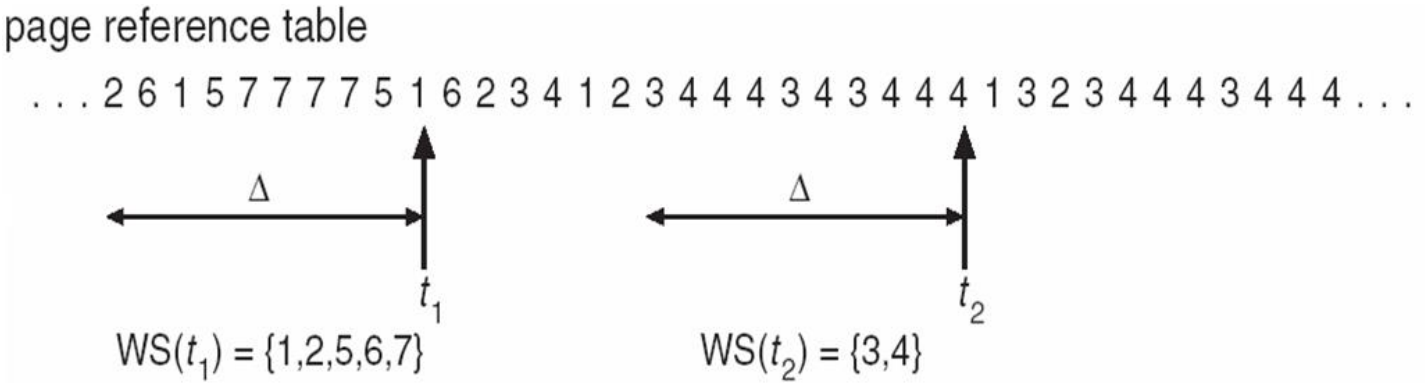
- Allocate enough frames to a process to accommodate its locality

- Why does thrashing occur?
  Σ size of locality > total allocated frames
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working-set model



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame