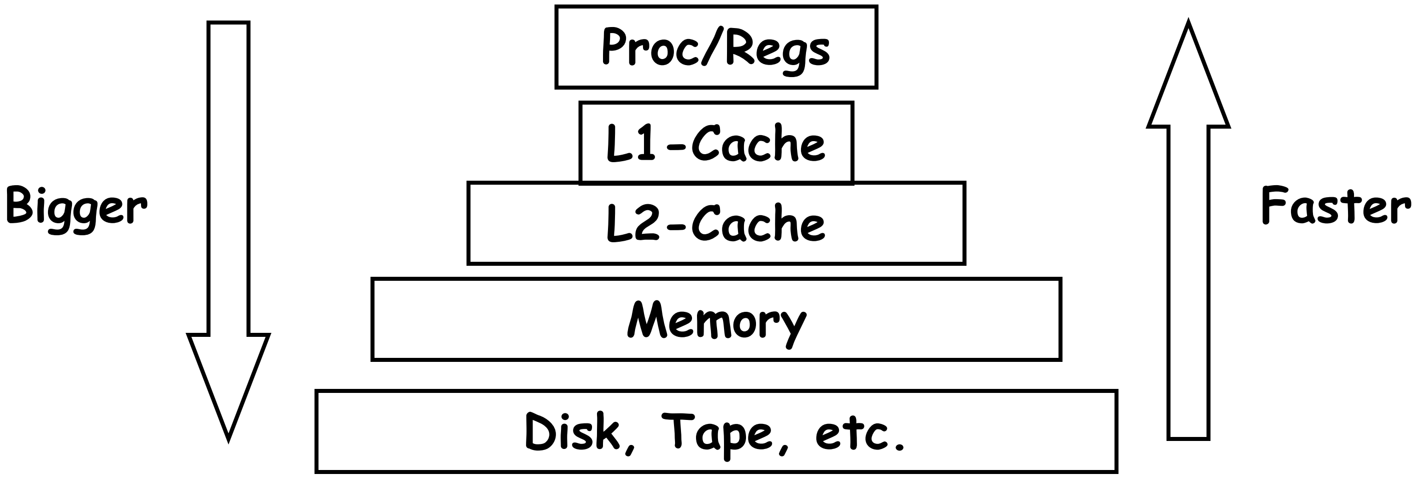
Memory Management

Dr. B.Majhi, Professor CS
NIT Rourkela


Proc/Regs

L1-Cache

L2-Cache

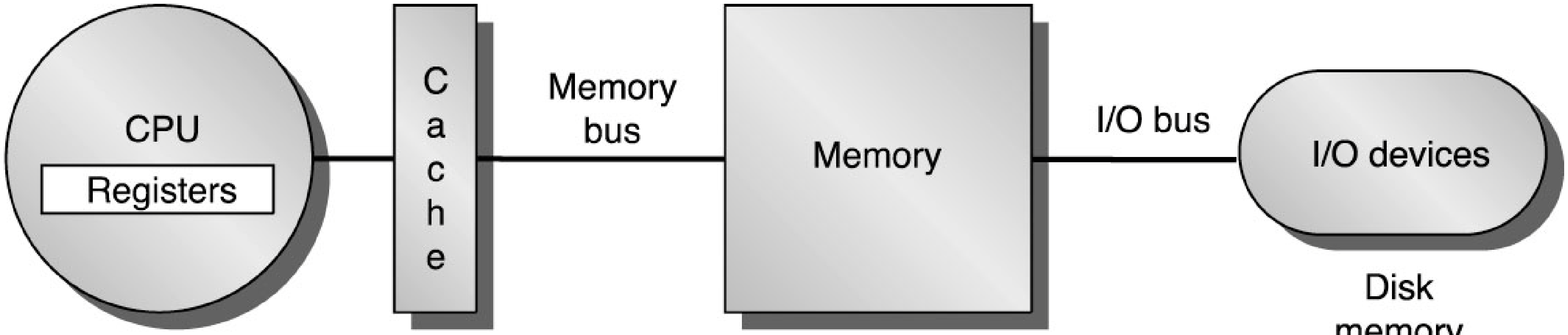
Memory

Disk, Tape, etc.

Bigger

Faster

Typical Memory Hierarchy

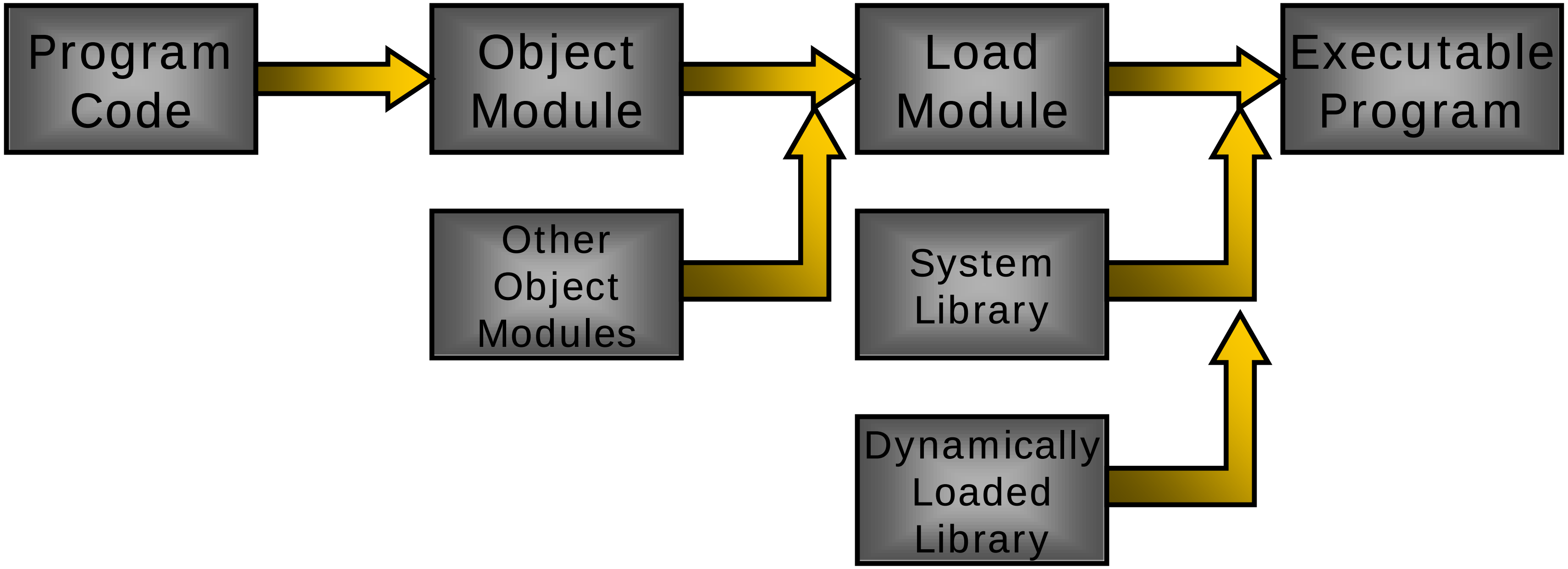


	Register reference	Cache reference	Memory reference	Disk memory reference
Size:	500 bytes	64 KB	512 MB	100 GB
Speed:	0.25 ns	1 ns	100 ns	5 ms

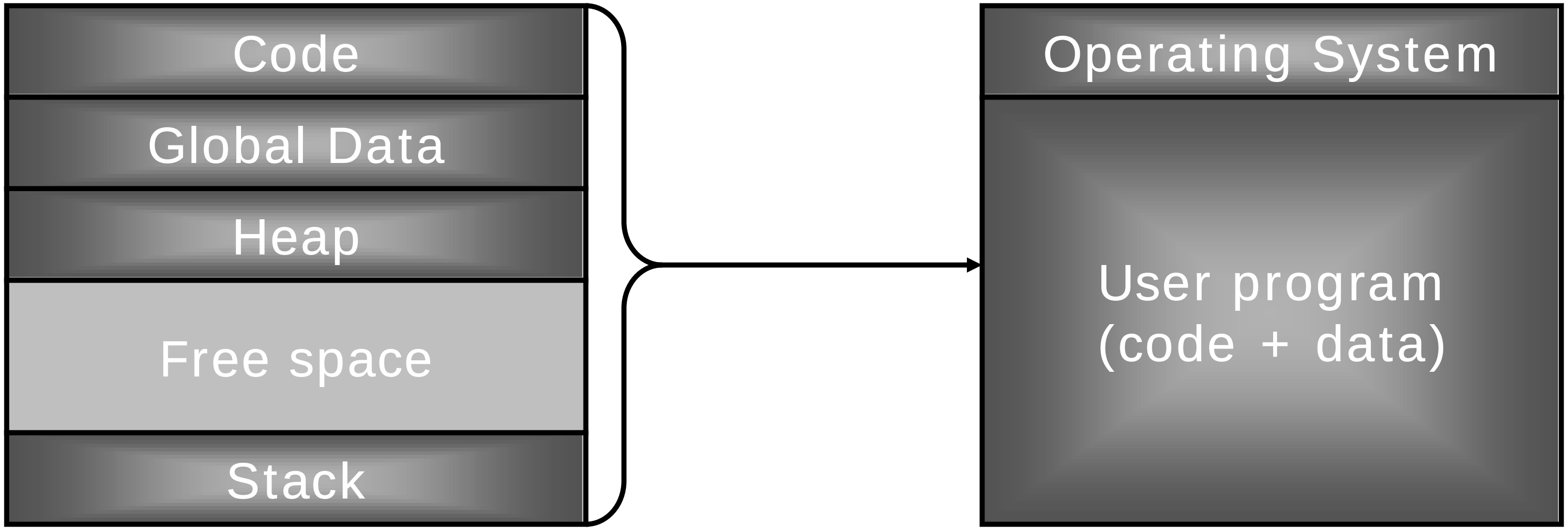
Memory Management

- Memory management is about how the OS manages the allocation of memory to the various processes in a system
- Topics:
- Logical versus Physical Address Space
- Contiguous Allocation
- Paging
- Segmentation

From Source Code to an Executing Program

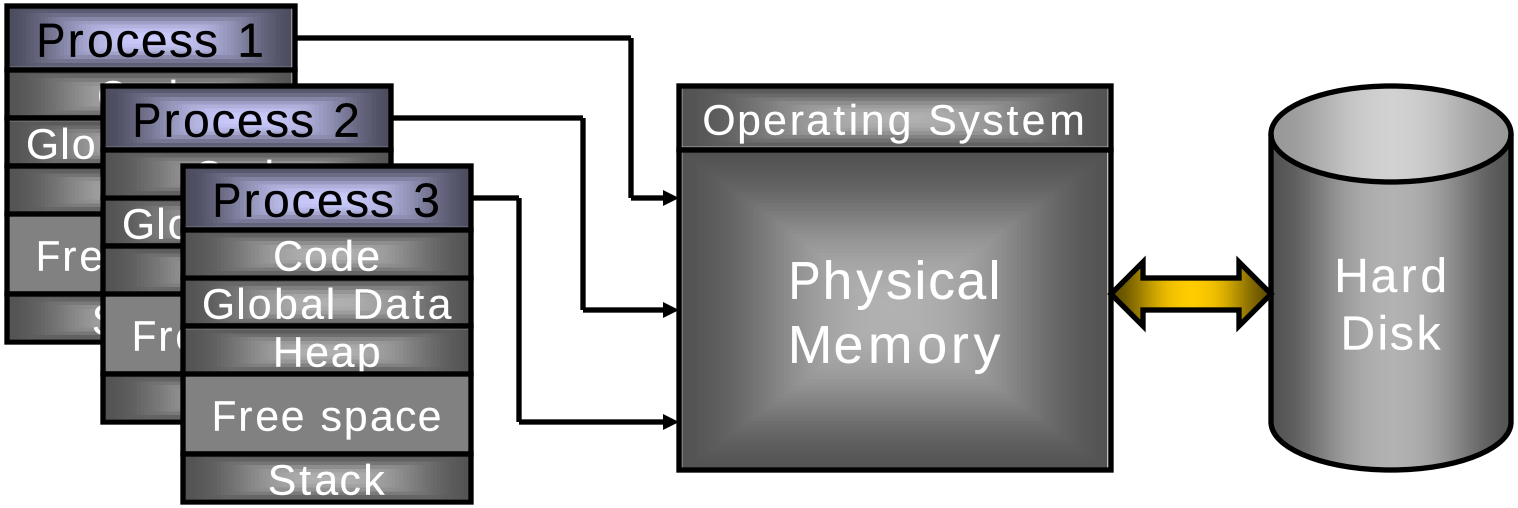


Single Contiguous Store Allocation



- No special hardware needed
- Program must fit into main store
  (or use an Overlay)
- Compiler can generate *absolute code*

Multiprocessing System



- Processes can reside anywhere in memory
  - – Or even be swapped out to disk

Address binding

- Program must be brought into memory and converted into a process for it to be executed.
- More than one process is held in physical memory at the same time
- During execution processes make references to physical memory locations i.e. locations in chips
- BUT - Program code represents locations symbolically e.g. variable names
- Mapping of one type of address to another type of address known as **address binding**
- Address binding can happen at three different stages.

Binding of Instructions/Data to Memory

1. **Compile time**: If the location of where the process is to be placed in memory is known when the program is compiled, then compiler can replace symbolic references to data in program code with actual addresses in memory where that data is to be held. You must recompile code if location of process in memory is to change.

2. **Load time**: If the location of where the process is to be placed in memory is NOT known when the program is compiled - compiler replaces symbolic references to data with **relocatable addresses.** These are addresses that are relative to address of begining of program code I.e. first word of program code is given address 0 and then all other locations within program are given addresses as offset from start of program. The actual address of beginning of program code when it is placed in memory is added to all addresses when program is loaded into memory.

3. Execution time: the location of where the process is to be placed in memory is only known when the process is to be executed (run time binding) - as before compiler replaces symbolic references to data with **relocatable addresses.** However, process may be moved during its execution from one part of memory to another. Hardware (e.g., *base* and *limit registers*) is used to generate actual address to identify location in memory chip - e.g. relocatable address + address in some relocation register gives real address to be used. This is done for each memory reference.

Logical vs. Physical Address Space

- Execution time binding of addresses means that the address that is generated by the process when it is running on the CPU may be different to the address of the actual physical location that holds the instruction or data required - the address generated by the process running on CPU (known as **logical address** or **virtual address**) has to be translated into **physical address** of location in memory
- Logical and physical addresses are the same in compile-time and load-time address-binding, but may be different in run-time binding

Memory-Management Unit (MMU)

- MMU is a hardware device that maps virtual to physical address.
- In MMU every address generated by a user process is modified (e.g. by adding a relocation register value to address) before sending it to memory.
- Base and limit registers are an example of a simple MMU.

Memory allocation

- memory allocation is the process by which the OS decides what memory to allocate to a process for its code, data and system data
- There are 3 general schemes for allocating memory to processes:
  - contiguous allocation
  - paging
  - segmentation

Contiguous Allocation

- The physical memory space of a process is held in one area of memory i.e. all locations are contiguous (next to each other)

- Main memory divided into two parts:
  – Resident operating system - held in low memory.
  – Memory for processes in high memory

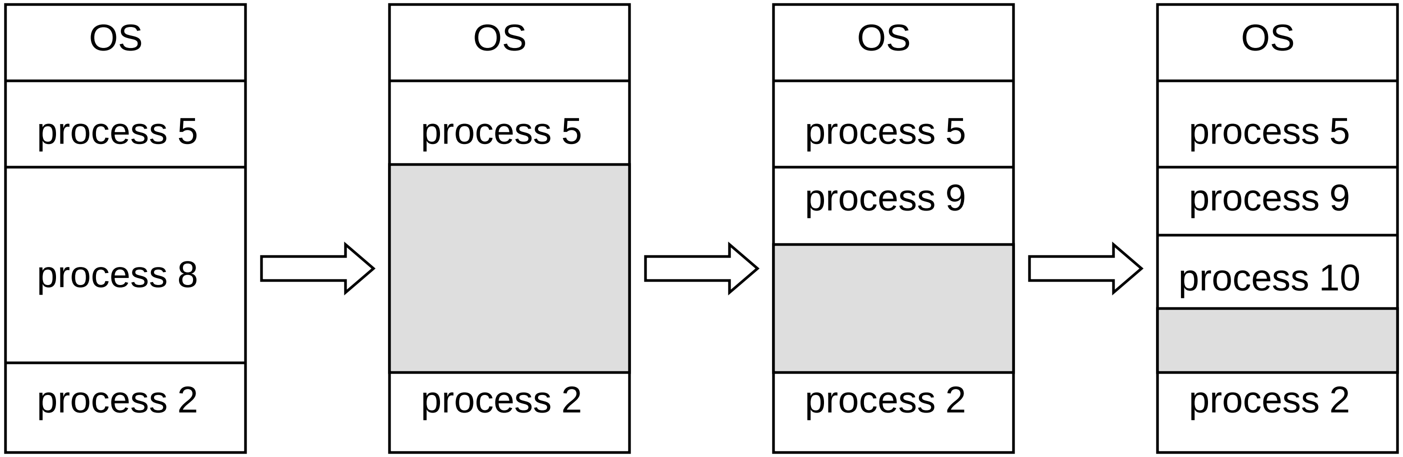
- memory area for processes can be organised either
  – to hold the address space of a single process at a time (called single partition allocation)
  – to hold the address space of several processes at the same time (multiple partition allocation) - one partition for each process

Single-partition allocation

- Relocation-register scheme used to protect operating-system code and data.
- Base register (also sometimes called relocation register) contains value of physical address of start of process memory area, limit register contains range of logical addresses – each logical address must be less than the limit register.

Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - **Hole** – this is a block of memory that is currently available (free) to be allocated to a process. Holes of various sizes are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (holes)



Dynamic Storage-Allocation Problem

- How to satisfy a request for memory of size *n* from a list of free holes:

- **First-fit**: Allocate the *first* hole from list 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.

Fragmentation

- **External fragmentation** – enough total memory space exists for a process, but it is not all in one block i.e. not contiguous. The free memory is divided into small areas (many small holes) - thus process cannot be loaded into memory even though total free space is enough.
- The unusable memory space exists outside of the memory partitions allocated to the running processes, thus it is called external fragmentation.

- Reduce external fragmentation by compaction
  - Shuffle memory contents in memory to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- **Internal fragmentation** – memory allocated to a process may be slightly larger than amount of memory requested. This may be because if only the area of memory actually needed is allocated then the area leftover is too small to be of any use so all block is allocated. The unused area of memory is internal to a partition. Hence it is called internal fragmentation.

Paging

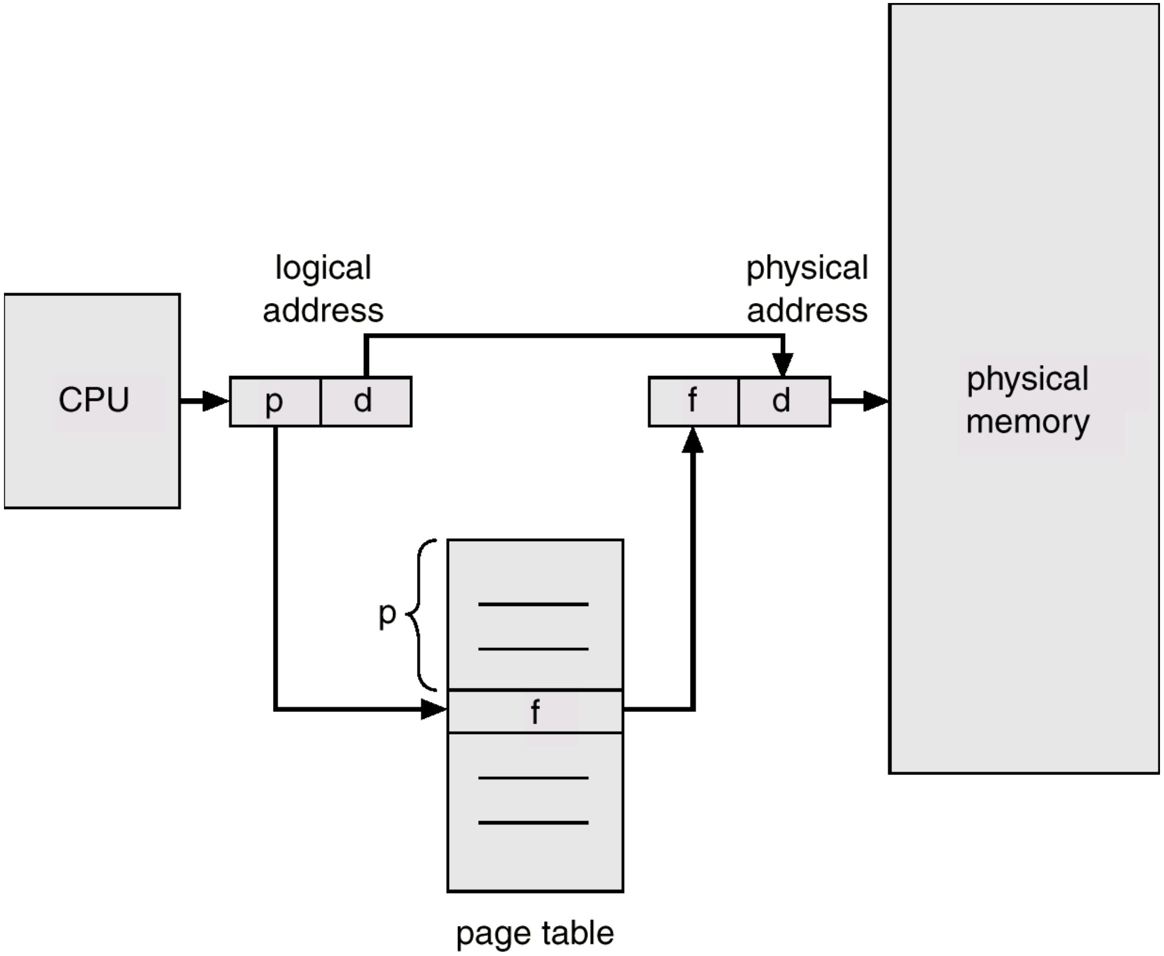
- In paging physical address space of a process can be noncontiguous i.e. addresses of consecutive logical locations in process may have non-consecutive physical addresses. The memory of the process may be divided up into a number of separate blocks in memory, the physical address is relative to the start address of the block of memory in which the required memory resides. As a result a process can be allocated physical memory wherever the latter is available.

- In paging scheme physical memory is divided up into into fixed-sized blocks called frames (normally between 1/2K and 8K in size).
- logical memory of a process is also divided into blocks of same size as the page frames. These blocks called pages.
- OS keeps track of all free frames.
- To run a program of size n pages, need to find *n* free frames and load program into them.
- Then set up a page table to translate logical to physical addresses. Separate page table for each process.

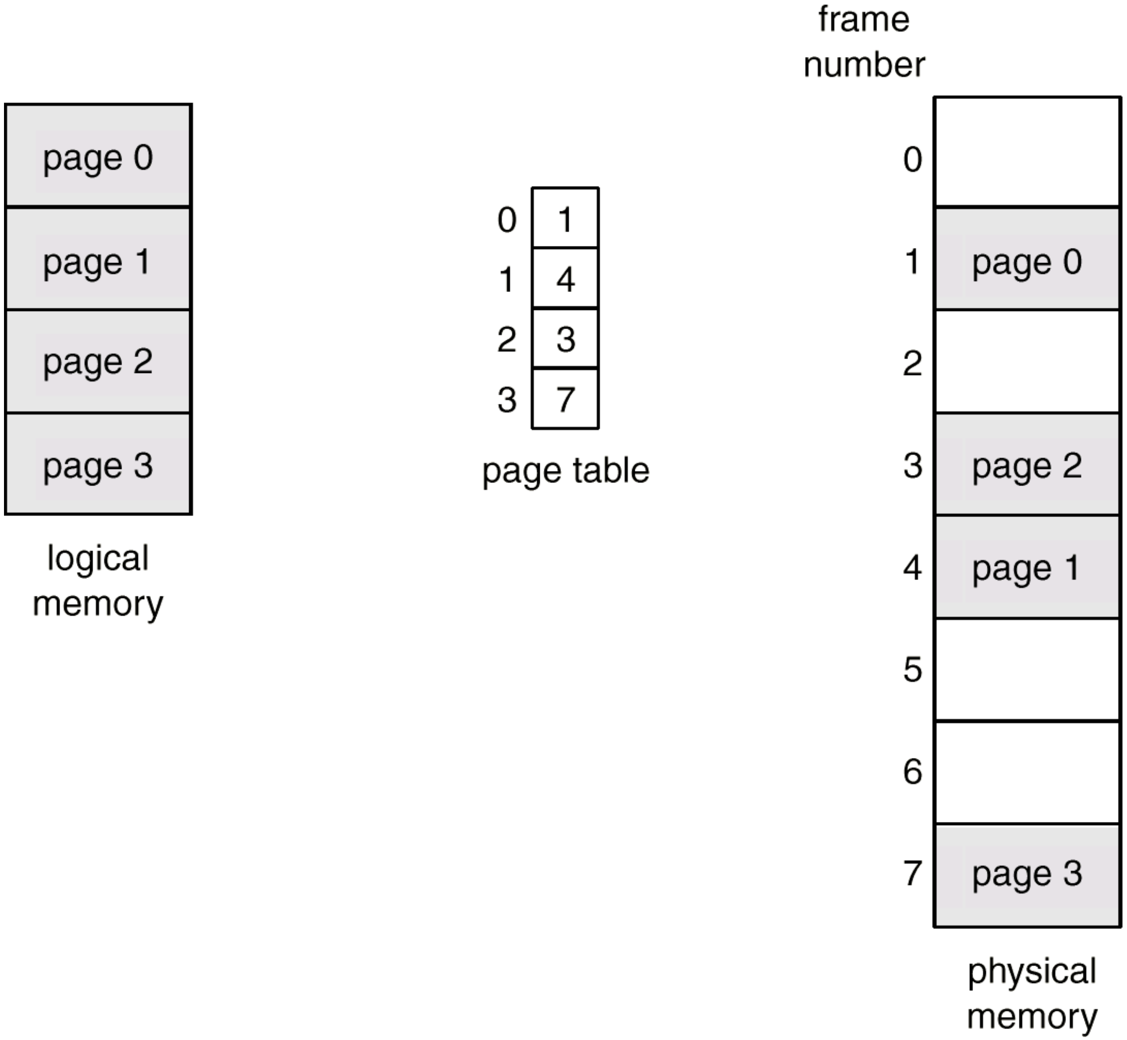
Address Translation Scheme

- 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)* – combined with base address to define the physical memory address that is sent to the memory unit.
- Paging eliminates external fragmentation, but internal fragmentation can still exist.

Address Translation Architecture



Paging Example



page 0
page 1
page 2
page 3

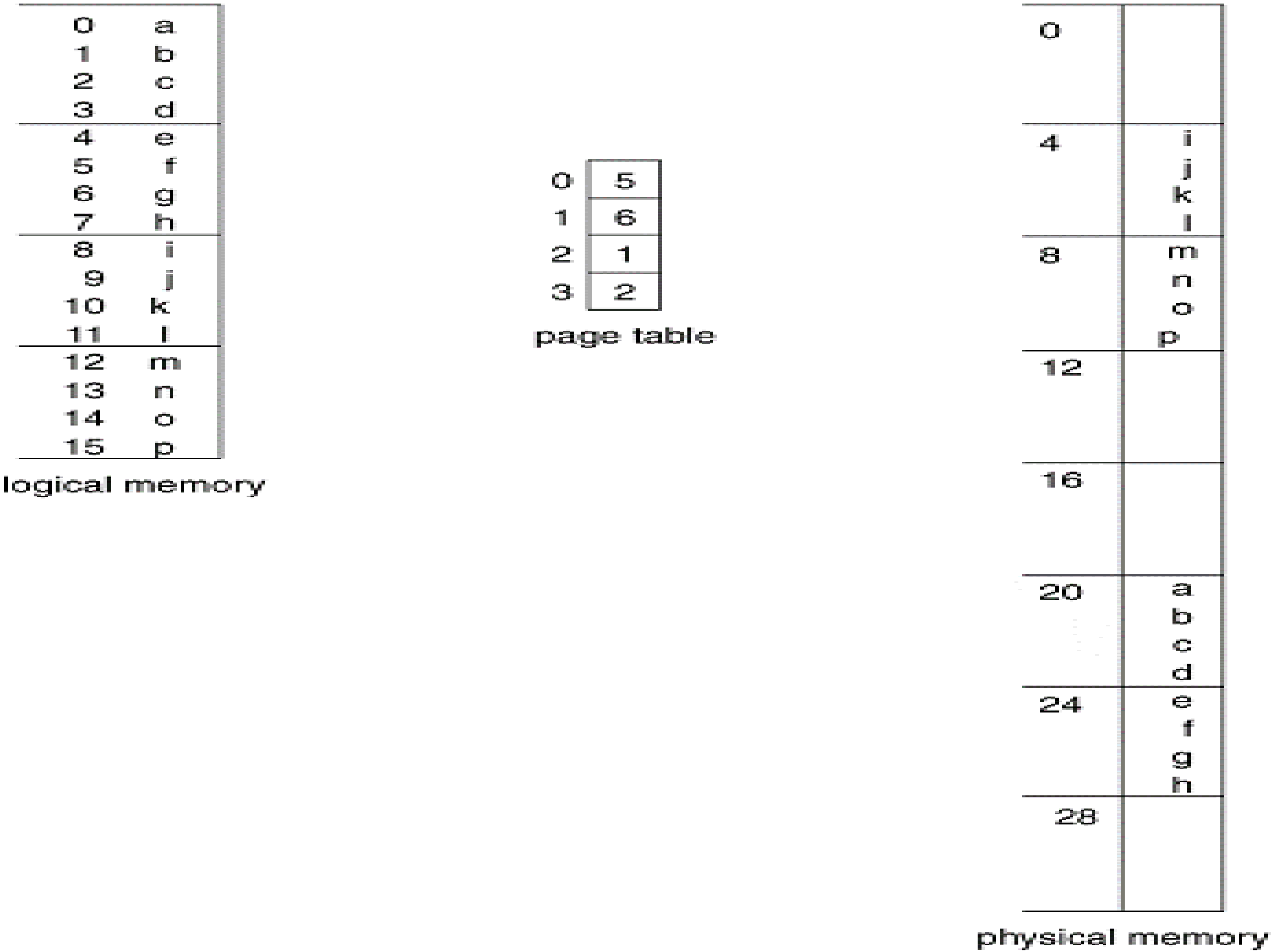
logical memory

0	1
1	4
2	3
3	7

page table

frame number	
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

physical memory



logical memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

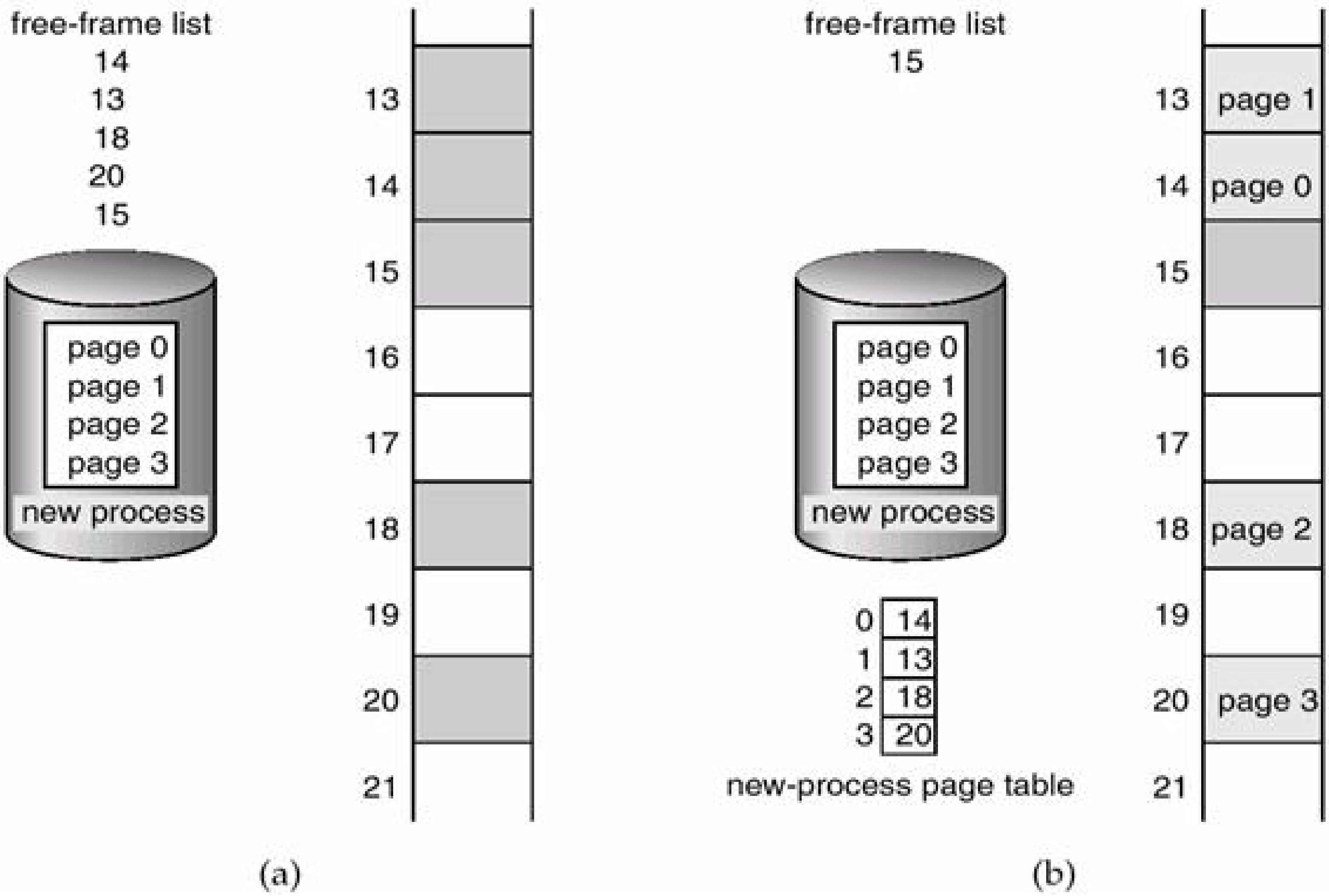
page table

0	5
1	6
2	1
3	2

physical memory

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

Free-frame list before and after allocation



0	14
1	13
2	18
3	20

Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One to get the page frame address from the page table and one to actually get the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs)*

Associative Register

- Associative registers – allow parallel search

Page #	Frame #

- If page number is in associative register, get frame number out.

- Otherwise get frame number from page table in memory and add it to associative registers

Effective Access Time

- Associative Lookup = T_a time units
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- If hit ratio = α, then Effective Access Time (EAT) can be found as follows:

if page no. in associative regs then use 1 microsecond to access memory for data + time for associative lookup (this will happen α fraction of the time)

if page no. not in associative regs then in addition to time for associative lookup, use 2 microseconds to access page frame address and data both in memory (this will happen 1- α fraction of the time)

$$\text{EAT} = \alpha(1 + T_a) + (1 - \alpha)(2 + T_a)$$

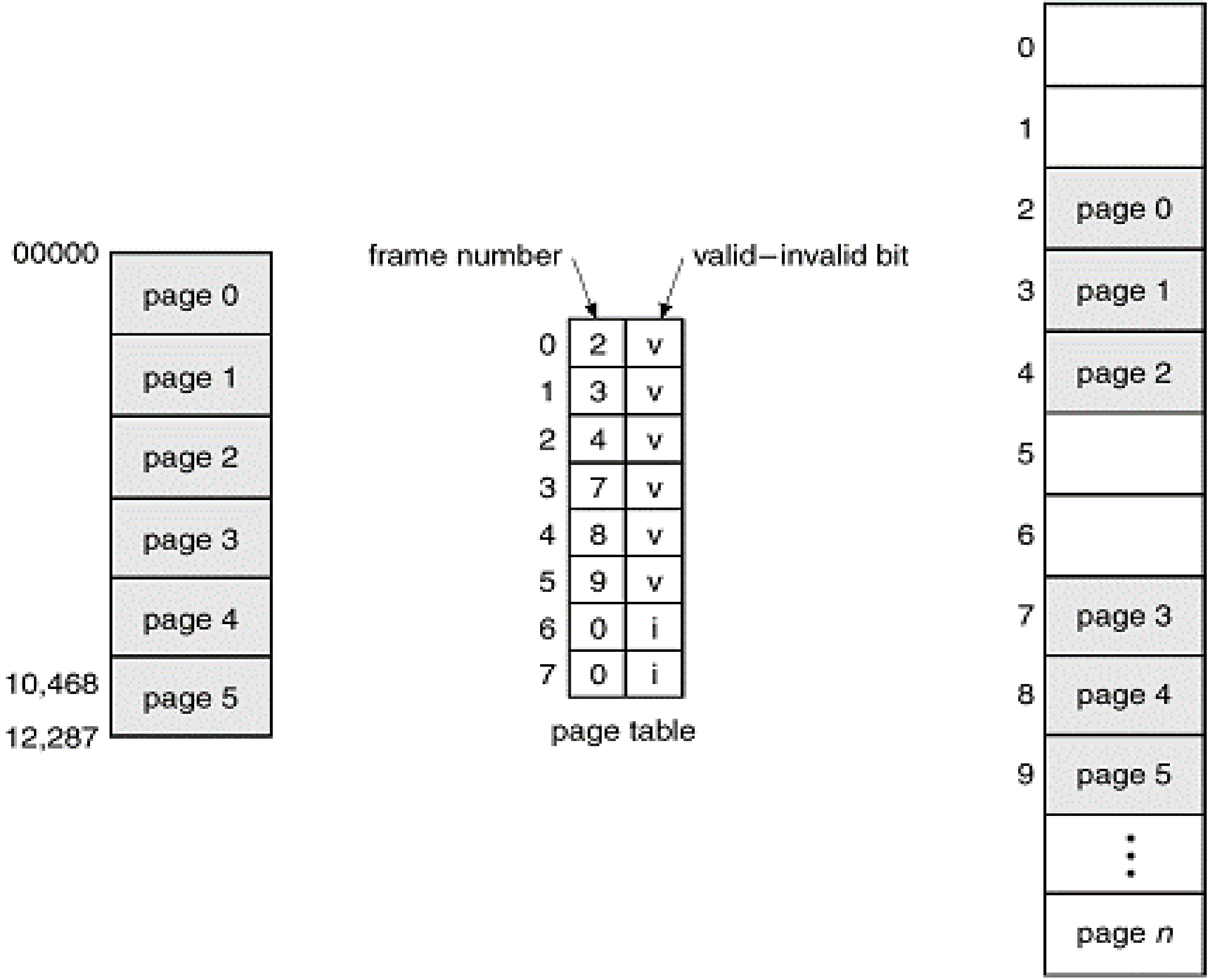
$$= 2 + T_a - \alpha$$

Memory Protection

- Memory protection is implemented by associating a protection bit with the page table entry for each frame - the bit specifies whether frame is read/write or read only

- Also *Valid-invalid* bit attached to each entry in the page table - bits loaded for each process as part of context switch for process:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page for the process to access.
  - "invalid" indicates that the page is not in the process' logical address space and should thus generate an error

Valid-invalid bit



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

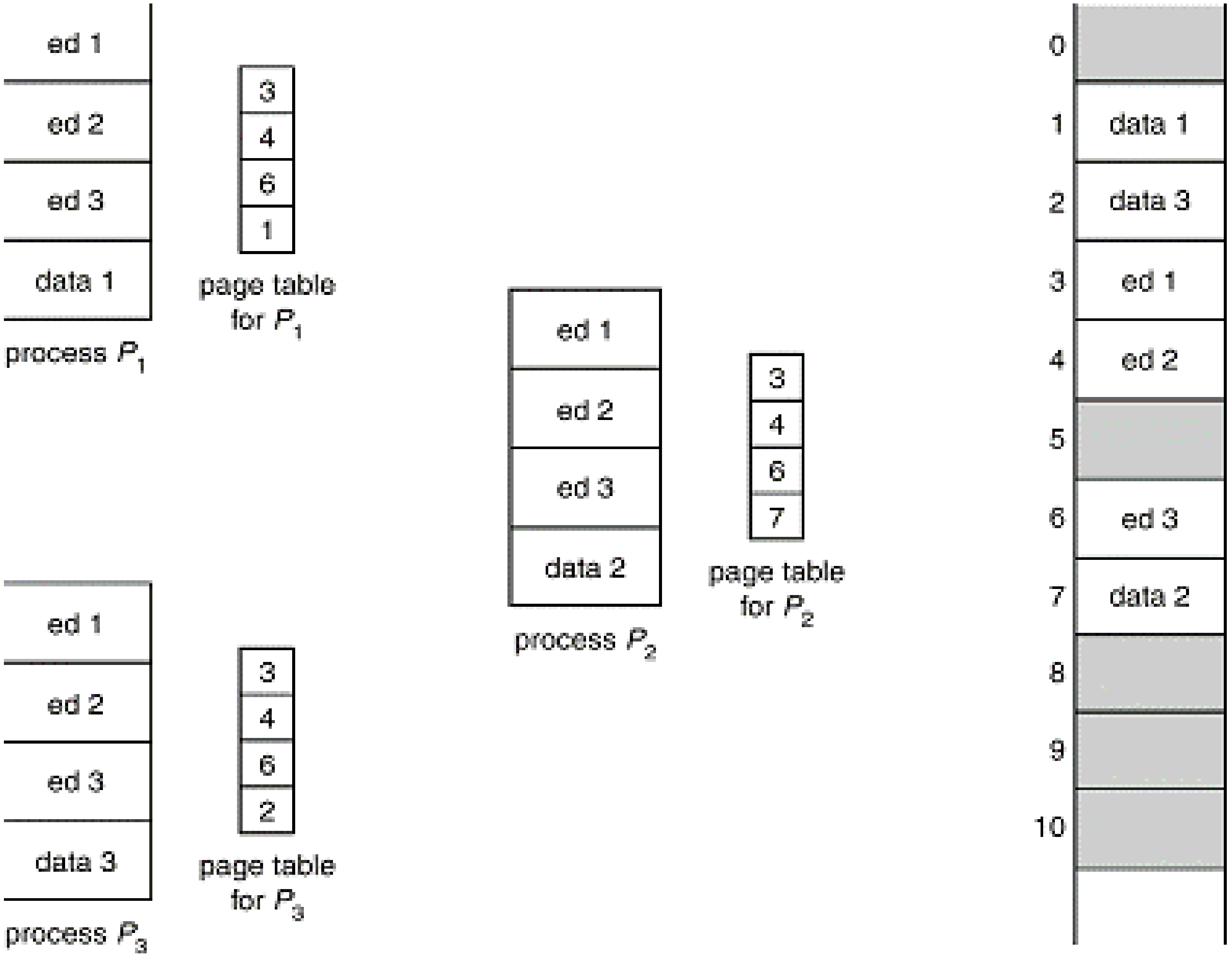
page table

Shared Pages

Shared code

1. One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
2. Shared code must appear in same location in the logical address space of all processes. Private code and data
3. Each process keeps a separate copy of the code and data.
4. The pages for the private code and data can appear anywhere in the logical address space.

Example


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1
data 1
2
data 3
3
ed 1
4
ed 2
5
6
ed 3
7
data 2
8
9
10

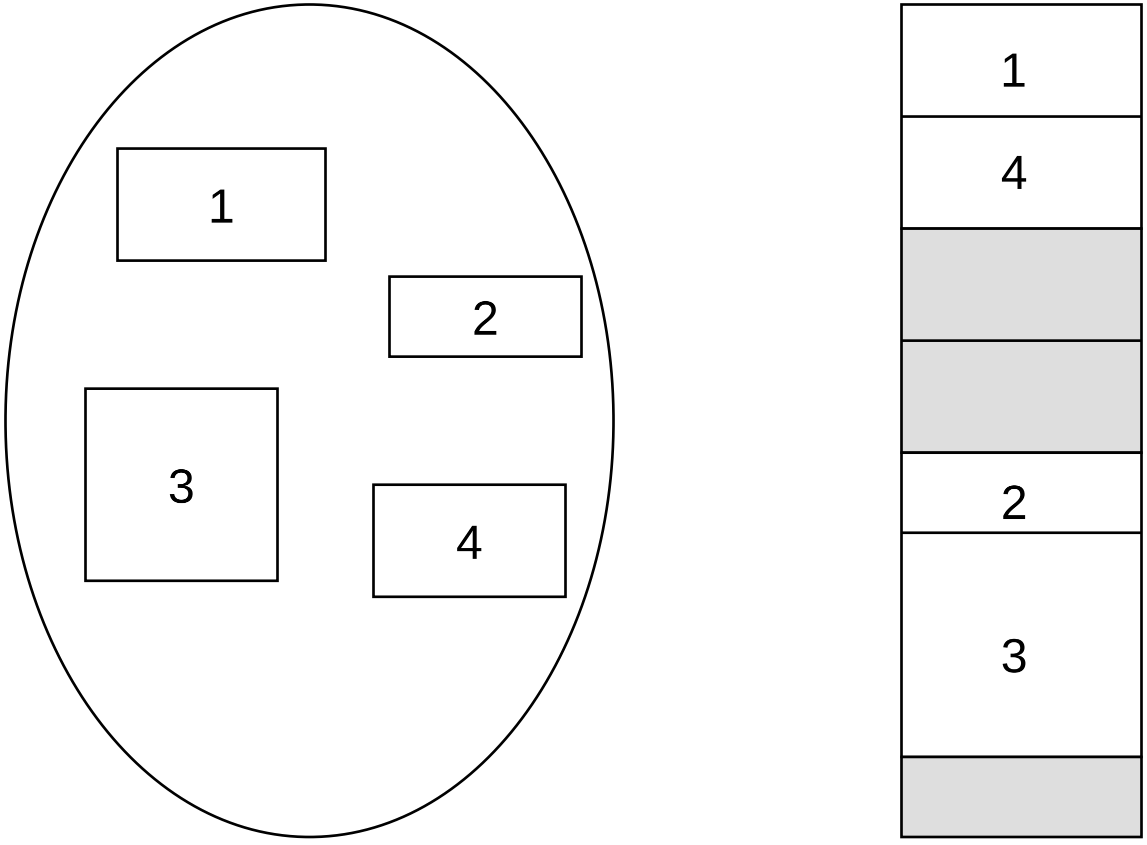
Segmentation

- Memory-management scheme that divides up processes’ address space into a number of blocks called segments

- segments are similar to pages, but the difference is that pages all have the same size, whereas segments may be a number of different sizes

- In this scheme a program is treated as a collection of segments. A segment is a logical unit e.g. main part of program, a group of functions, the set of global data values, arrays, a stack, in OO languages may be individual objects

Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

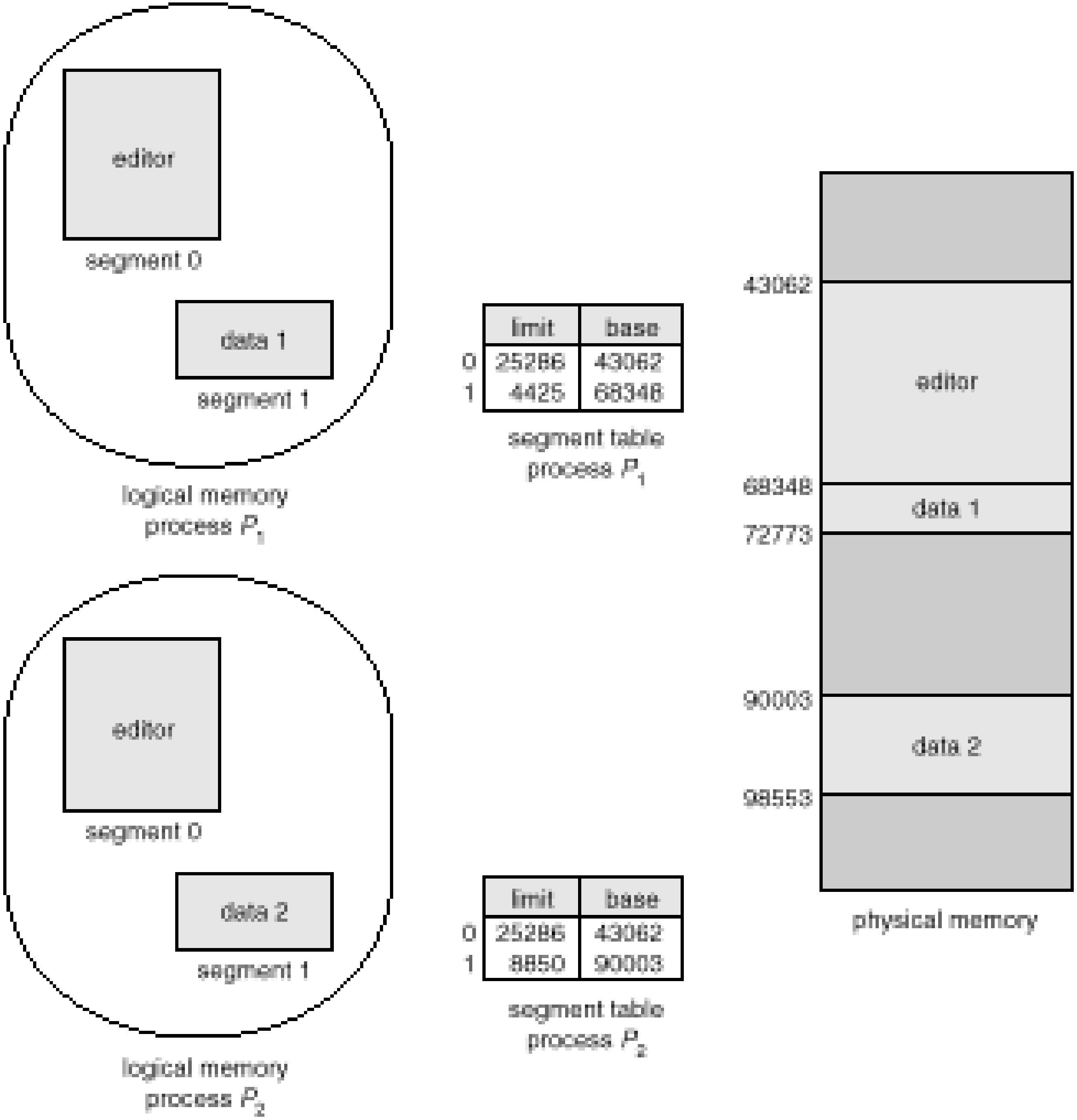
- Logical address consists of a pair: <segment-number, offset>,
  - Segment number is an index into the *segment table* - with one segment table for each process
  - offset is the value to be added to start address of segment to give real physical address
- each entry in segment table has:
  - base value – contains the starting physical address where the segment resides in memory.
  - *limit* – specifies the length of the segment.
- Offset address is legal if offset < limit value

- *Segment-table base register (STBR)* points to the segment table’s location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

segment number *s* is legal if *s* < STLR.

Segmentation Architecture (Cont.)

- Protection. Each entry in segment table contains:
  - validation bit - which if = 0 ⇒ illegal segment
  - specification of read/write/execute privileges
- Protection bits associated with segments
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram



	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

Virtual Memory

- Reason for virtual memory & what it is.
- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Page-Replacement Algorithms
- Allocation of Frames
- Thrashing

Virtual memory - the need for it.

- Memory is a limited resource
- the memory required by the various processes that want to run on the system may be greater than the total physical memory
- using the memory allocation mechanisms we have seen so far, this would mean that the long term scheduler (one that decides whether to allow processes into the system) would have to refuse to run some of the processes until some of the processes have finished and can release memory back to the OS

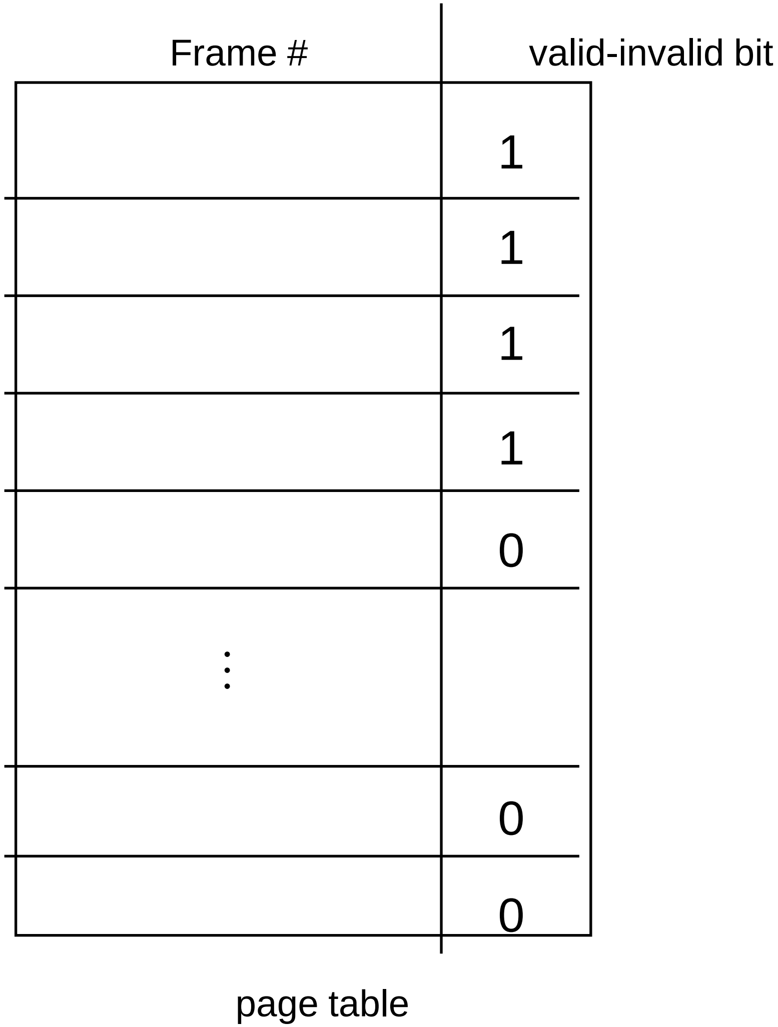
- virtual memory is used to get round this problem - it is an attempt to allow a computer with ***X*** amount of physical memory to be able to run even when the memory required by processes running on the system is greater than ***X***
- to do this only a proportion of a process’ memory is held in physical memory (some of it is left on hard disk), so more proceses can be fitted into a given area of physical memory
- So need some scheme which will determine which areas of the process’ memory needs to be in physical memory - allocation scheme

- in general the idea is to only keep in physical memory those items of data and instructions that are currently being used or are likely to be used in the near future - implemented using **demand paging** or **demand segmentation**
- also when physical memory that is available to a process becomes full and new memory needs to be brought in, then OS needs some schme which will determine which memory needs to be replaced with the new memory that is being brought in - this is called a **replacement policy**

Demand Paging

- Bring a page into memory only when it is needed.
- Uses a lazy approach to bringing into memory
- a process makes a reference to a page of its memorywhile running on the CPU
  - if reference is invalid (e.g. no such program address) then abort process
  - if page not in memory then fetch into memory

Valid-Invalid Bit



Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

- A valid-invalid bit is associated with each page table entry (1 means the page is in memory, 0 means the page is not in memory)
- Initially valid–invalid bit is set to 0 on all entries.
- During adddress translation, if valid–invalid bit in page table entry is 0 then a **page fault** is generated

Page Fault

- **Page fault** is an interrupt that signals the OS that a page that has been referenced by a running process is not in memory
- OS looks at table in Process Control Block to decide whether:
  - the reference is invalid (address not in processes address space) then abort OR
  - Address not in memory in which case it needs to be brought into memory
- The first reference to a page will generate a page fault

Page Fault (Cont.)

- To bring in new page requires:
- Finding empty frame.
- Swapping page into frame.
- Updating page table and PCB table, setting valid bit = 1.
- Restart instruction that made memory reference, but now page with address in it is in memory

What happens if there is no free frame?

- **Page replacement** – find some page in memory that is not being used very much and replace it with new page.
  - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought in/out of memory several times during lifetime of process

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference generates a fault
- Effective Access Time (EAT)

  EAT = $(1 – p)$ * physical memory access time

  + p * (page fault overhead + time to copy page out (if modified) + time to copy page in + restart overhead + physical memory access time)
- Page fault/restart overheads include - switching process contexts, handling interrupt, finding free frame, initiating disk transfer, updating tables, etc.

Demand Paging Example

- For example we take memory access time = 100 nanoseconds and 50% of the time the page that is being replaced has been modified and therefore needs to be copied back out to disk. We will ignore page fault and restart overheads.

- Swap Page Time = 10 msec = 10 million nanosec

  EAT = (1 – p) * 100 + p (15 million + 100) nanosec

  = 100 - p*100 + p*15 million + p*100   (in nanosec)

  = 100 + p * 15 million   (in nanosec)

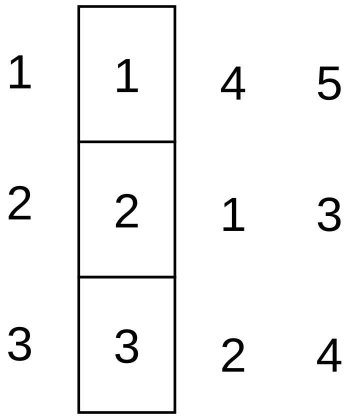
- size of page fault rate has huge affect on effective access time

Page-Replacement Algorithms

- Want lowest page-fault rate.
- We can examine performance of an algorithm by running it on a particular string (sequence) of memory references (reference string) in which we are only interested in the page number of the memory reference
- we can then compute for each algorithm the number of page faults that would occur with that particular sequence of memory references.
- In all examples that follow, the reference string is

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

First-In-First-Out (FIFO) Algorithm

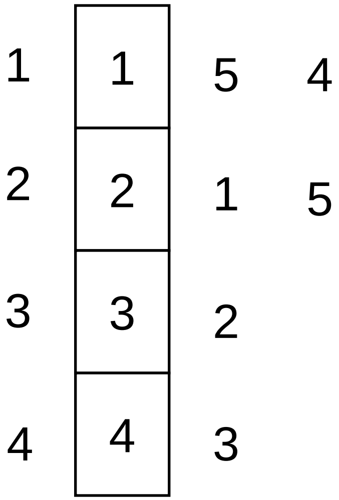


1 1 4 5

2 2 1 3

3 3 2 4

9 page faults



1 1 5 4

2 2 1 5

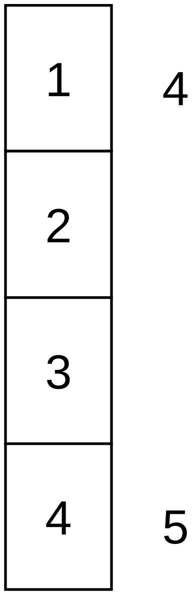
3 3 2

4 4 3

10 page faults

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)
- 4 frames - sometimes using more actual frames can produce more page faults – known as Belady’s Anomaly

An Optimal Algorithm



6 page faults

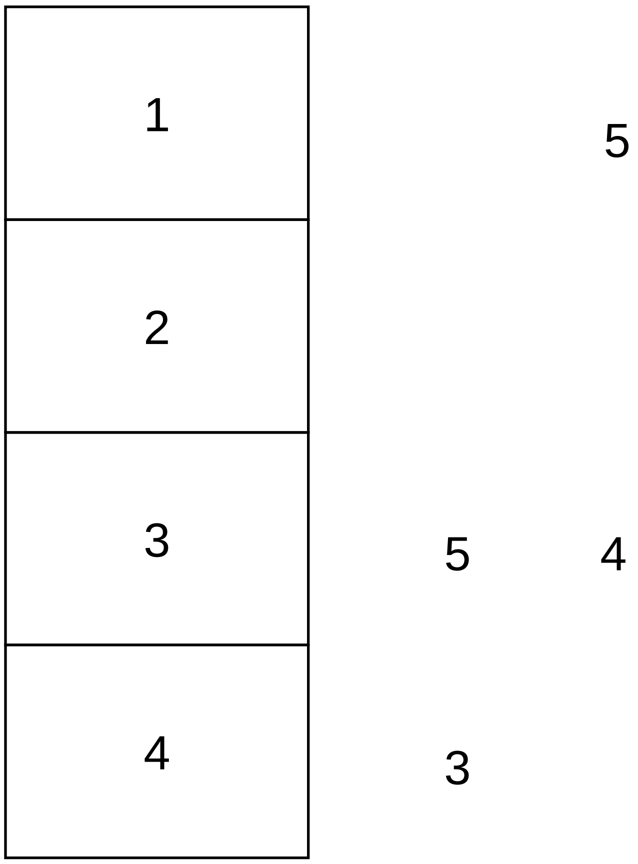
- Replace page that will not be used for longest period of time in the future.
- 4 frames example

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- How do you which page will be the one that is not referenced for the longest time in the future? - Can't know!
- It is used as benchmark in comparison of algorithms.
- But it can be approximated by taking recent past as guide to future and replacing page that has not been used for longest period of time - Least Recently Used

Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



8 Page Faults

LRU Algorithm (Cont.)

- implementation of LRU using counters
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which is oldest.
- implementation of LRU using a stack
  - keep a stack of page numbers in a double linked list:
  - when a page is referenced - move it to the top of the stack
  - Bottom of stack always LRU page
  - Updating stack requires changing set of pointer values

Allocation of Frames

- Each process needs minimum number of pages to be able to operate effectively.
- Two major allocation schemes.
  - Equal allocation
  - proportional allocation
- Equal allocation between processes – e.g. if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

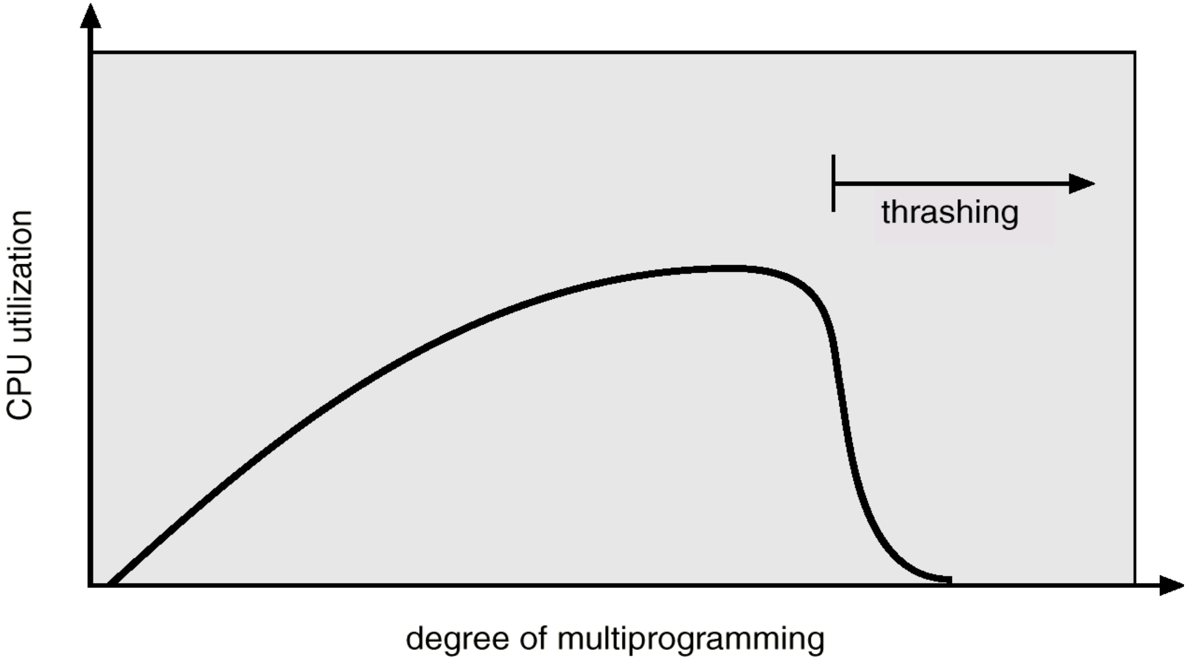
Global vs. Local Replacement

- Global replacement – replacement mechanism selects a page for replacment in a frame from the set of all frames; replacement of pages can use frames for replacement that currently belong to a different process from the one generating the page fault.
- Local replacement – replacement mechanism only selects a page for replaceemnt from a set of frames that have been allocated to the process generating the page fault.

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This can lead to low CPU utilization.

- Thrashing ≡ a process is busy swapping pages in and out from disk - very time consuming.



Thrashing

- Locality model
  - Set of addresses/pages a process accesses over a period of time during execution tend to form a stable set **-** known as a **locality**
  - after a period of time the process will start acessing a new set of addresses/pages which in turn will form a new stable set.
  - Localities may overlap i.e. some pages used throughout lifetime of program
- Why does thrashing occur?
  Sum of sizes of current localities of all processes on system > total hardware memory size

Demand Segmentation

- Used when insufficient hardware to implement demand paging.
- OS/2 allocates memory in segments, which it keeps track of through segment descriptors
- Segment descriptor contains a valid bit to indicate whether the segment is currently in memory.
  - If segment is in main memory, access continues,
  - If not in memory, segment fault.