

METU - EE 442 - Operating Systems

Memory Management

Chapter 3

Cüneyt F. Bazlamaçcı (METU-EEE)

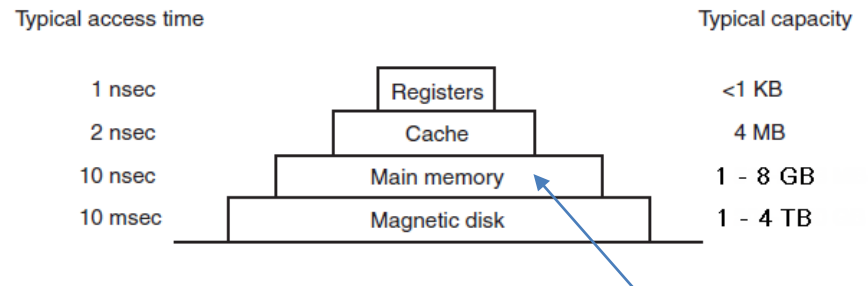
Memory

- Average home computer today has 10,000 times more memory than the largest computer in the early 1960s.
- Parkinson's Law, *“Programs expand to fill the memory available to hold them.”*
- Ideal computer memory:
 - Private, infinitely large, infinitely fast
 - Nonvolatile (does not lose its contents when the electric power is switched off)
 - Inexpensive

Memory Hierarchy in a Typical Computer System

- A few megabytes of very fast, expensive, volatile cache memory,
- A few gigabytes of medium-speed, medium-priced, volatile main memory,
- A few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage (+ some removable storage, such as DVDs and USB sticks)

OS abstracts this hierarchy into a useful model and then manages this abstraction.



Focus of chapter 3

Memory Manager

- Keeps track of which parts of memory are in use,
- allocates memory to processes when they need it,
- deallocate memory when they are done.

Memory Management

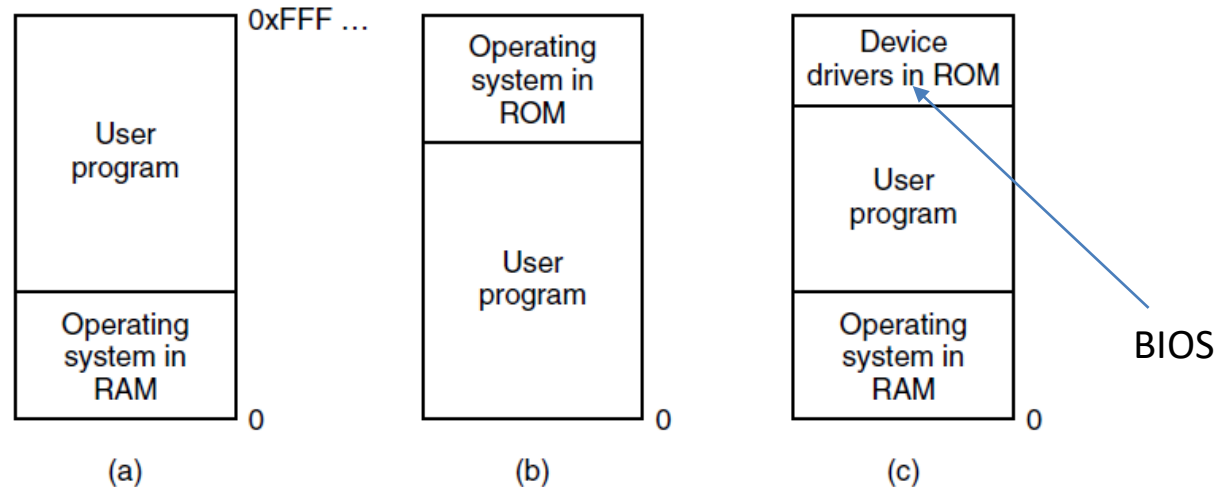
- In a multiprogramming system, a number of processes must be kept in memory in order to share the processor.
- Partitioning, paging and segmentation approaches are among possible choices used for memory abstraction.
- Memory has to be managed through memory management algorithms, which may require some hardware support.

Memory Management

- In order to be able to load programs anywhere in memory, the compiler must generate relocatable object code.
- Also, we must make sure that a program in memory addresses only its own area but no other program's area.
- Therefore, some protection mechanism is also needed.

No Memory Abstraction

- Memory presented to the programmer is simply physical memory.
 - a set of addresses from 0 to some maximum
 - each address corresponding to a cell containing some number of bits, commonly eight.



Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- Drawbacks
 - multiprocessing not possible (multiple threads may be possible).
 - User programs can easily trash OS.

Running Multiple Programs Without a Memory Abstraction

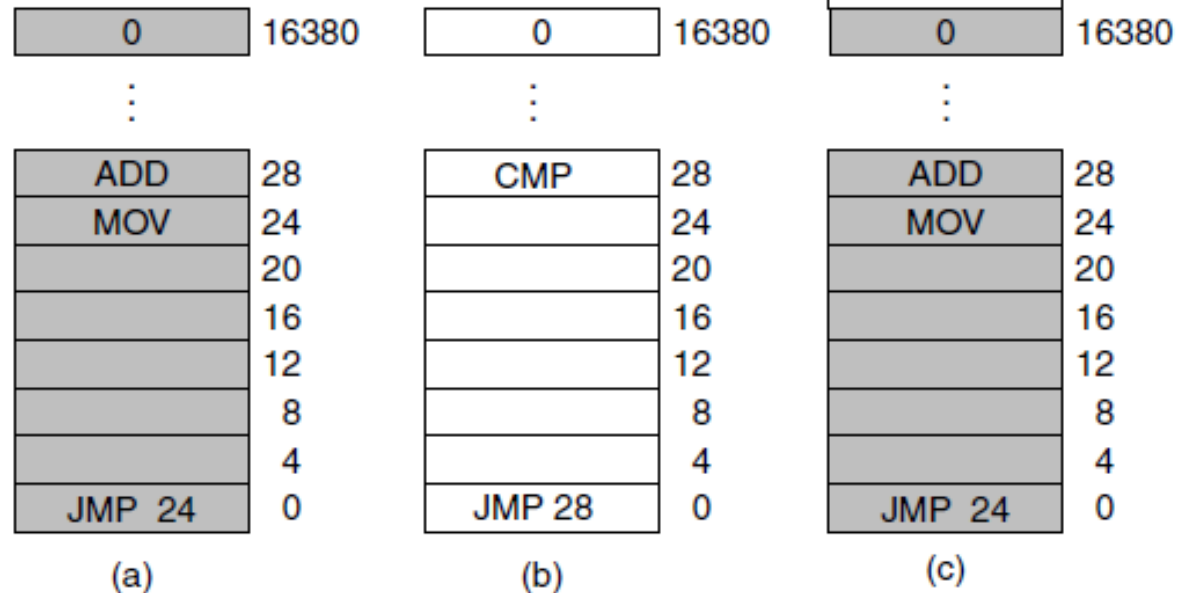


Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

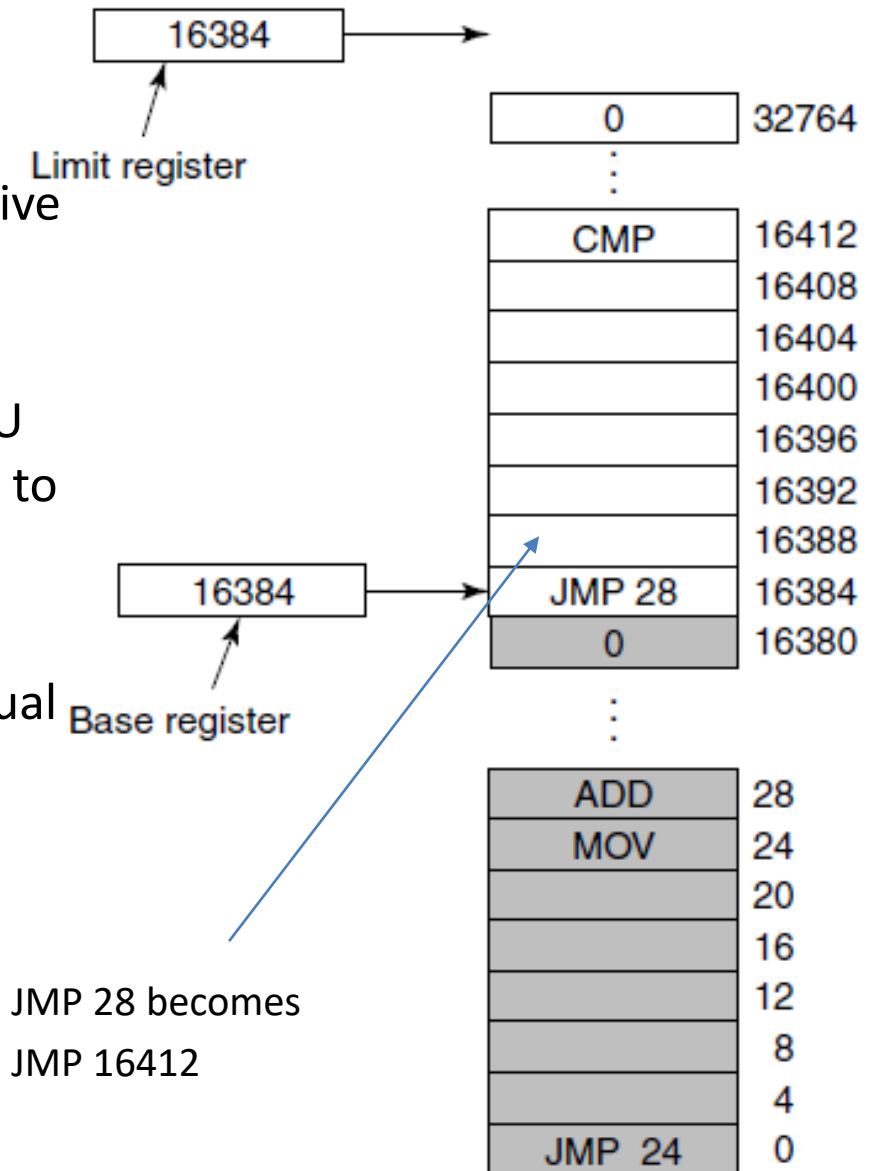
No Memory Abstraction

- History repeats itself in computer world.
 - direct addressing (no memory abstraction) is not used anymore in
 - mainframes, minicomputers, desktop computers, notebooks, and smartphones
 - no memory abstraction is still common in
 - embedded and smart card systems such as
 - radios, washing machines, and microwave ovens, etc.
- No memory abstraction is OK in some embedded systems since for example users are not free to run their own software on their toaster.

Address Space

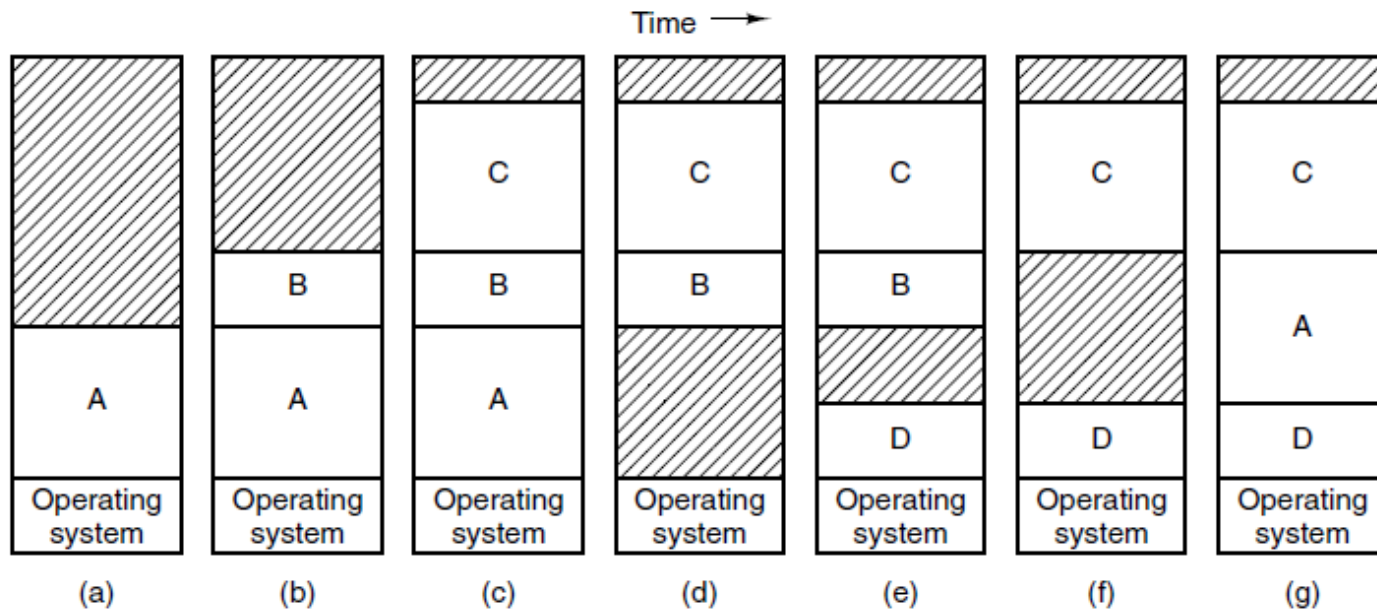
- Base and Limit Registers -

- Base and limit registers can be used to give each process a separate address space.
- When a process references memory, CPU hardware automatically adds base value to the address generated by the process.
- It also checks whether the address is equal to or greater than the limit register.
- If so a fault is generated.



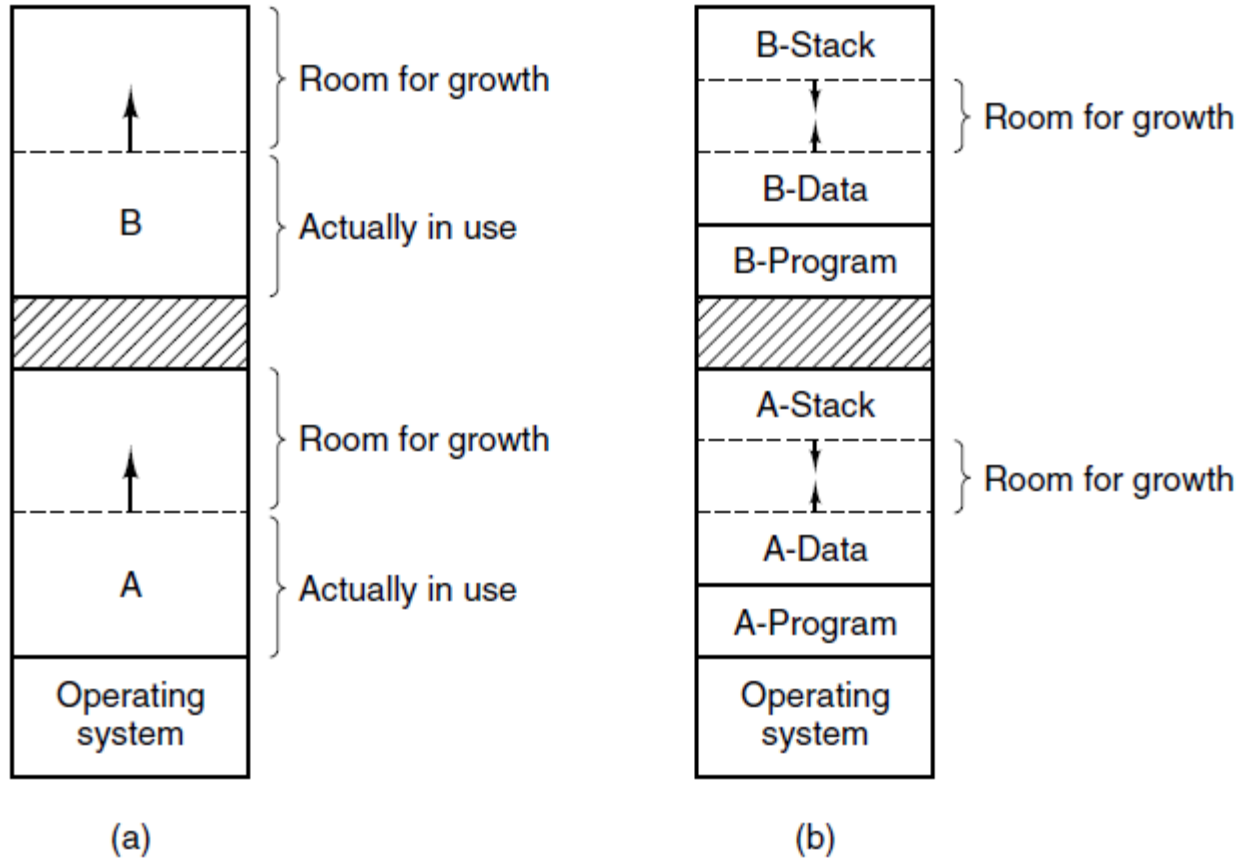
Swapping (1)

- Keeping all processes in memory all the time requires a huge amount of memory and cannot be done if there is insufficient memory.
- Swapping: bring in each process in its entirety, run it for a while, then put it back on the disk.



Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory

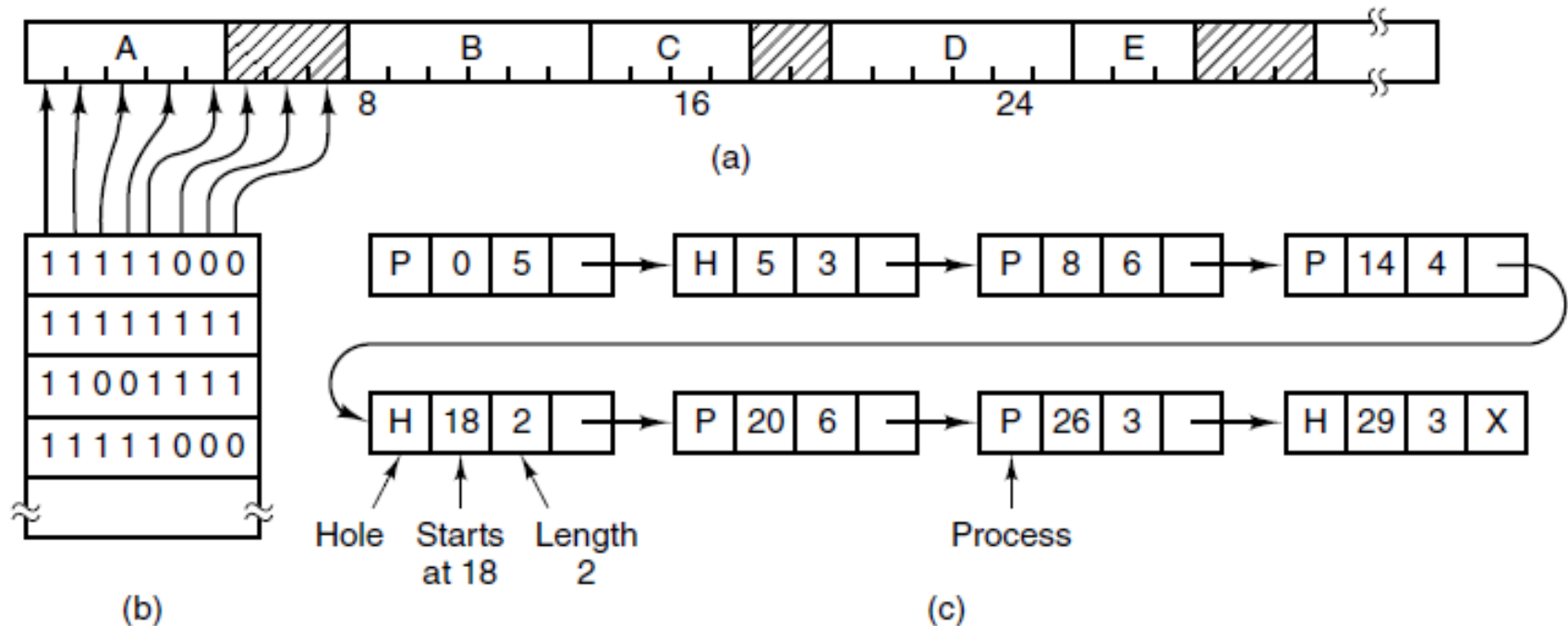
Swapping (2)



(a) Allocating space for a growing data segment.

(b) Allocating space for a growing stack and a growing data segment.

Memory Management with Bitmaps and Linked Lists



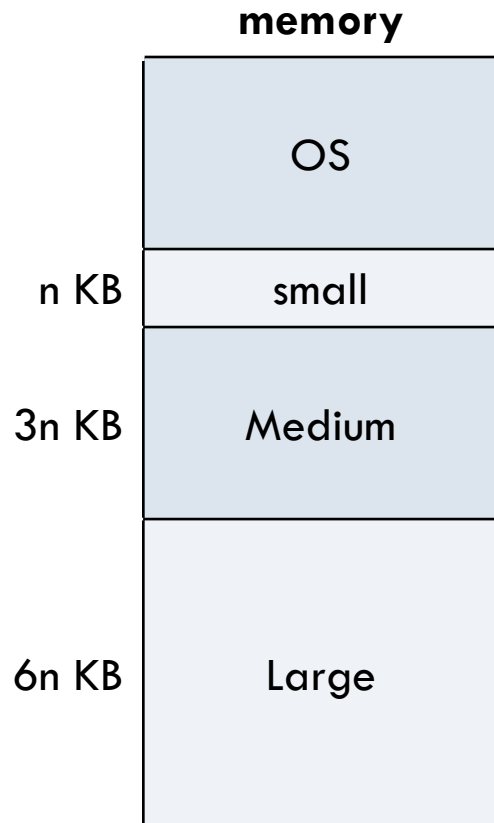
(a) A part of memory with five processes and three holes. The tickmarks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Memory Management with Linked Lists



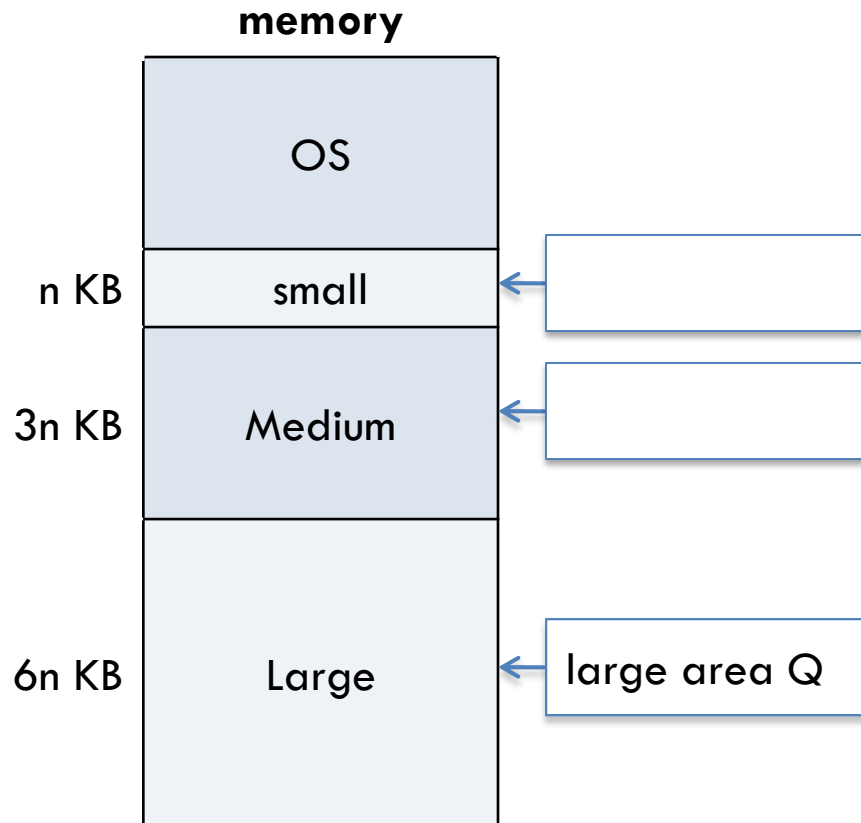
Four neighbor combinations for the terminating process, X.

Fixed Partitioning



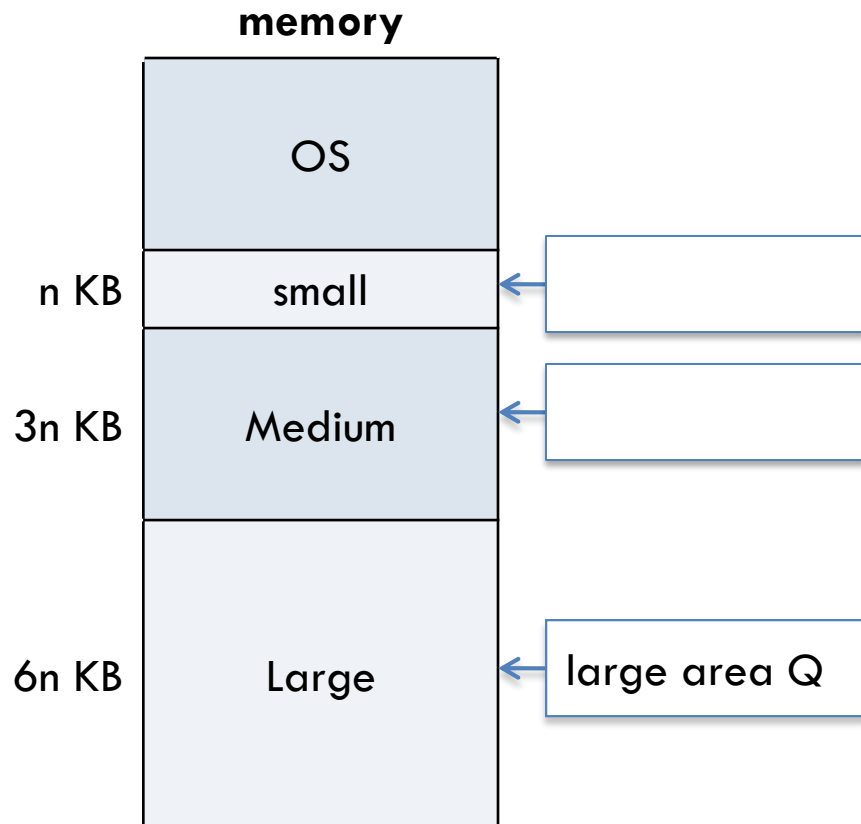
- In this method, memory is divided into partitions whose sizes are fixed.
- OS is placed into the lowest bytes of memory.
- Relocation of processes is not needed.

Fixed Partitioning



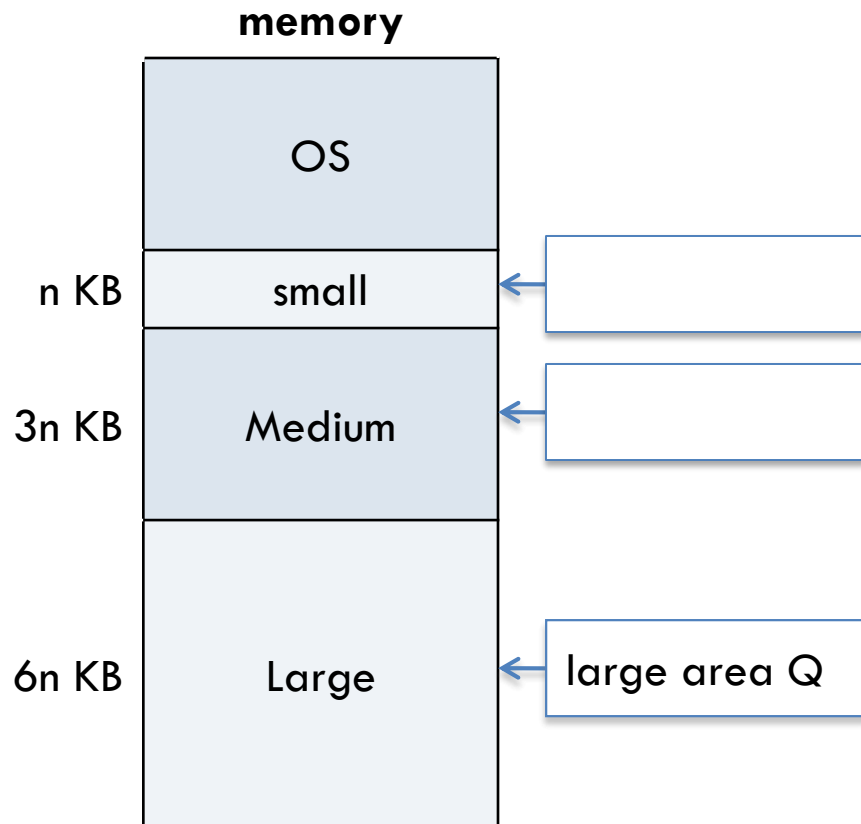
- On entry to the system, processes are classified according to their memory requirements.
- We need one *Process Queue (PQ)* for each class of process.

Fixed Partitioning



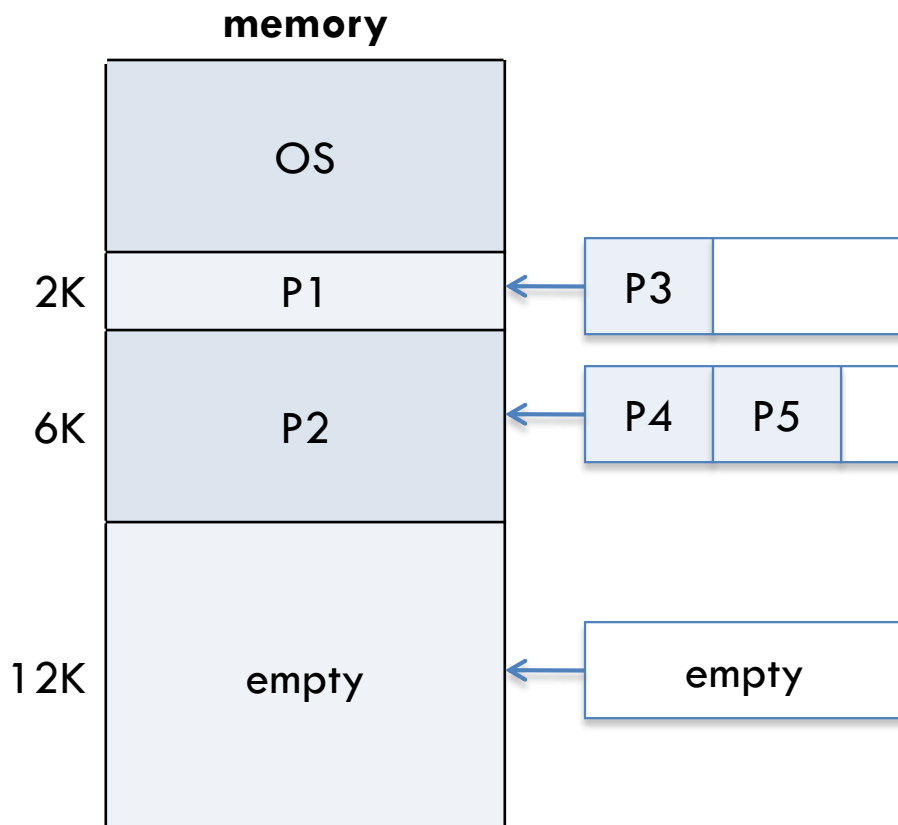
- If a process is selected and is allocated memory, it starts residing in memory and can compete for the processor.
- The number of fixed partitions gives the degree of multiprogramming.
- Since each queue has its own memory region, there is no competition between queues for the memory.

Fixed Partitioning



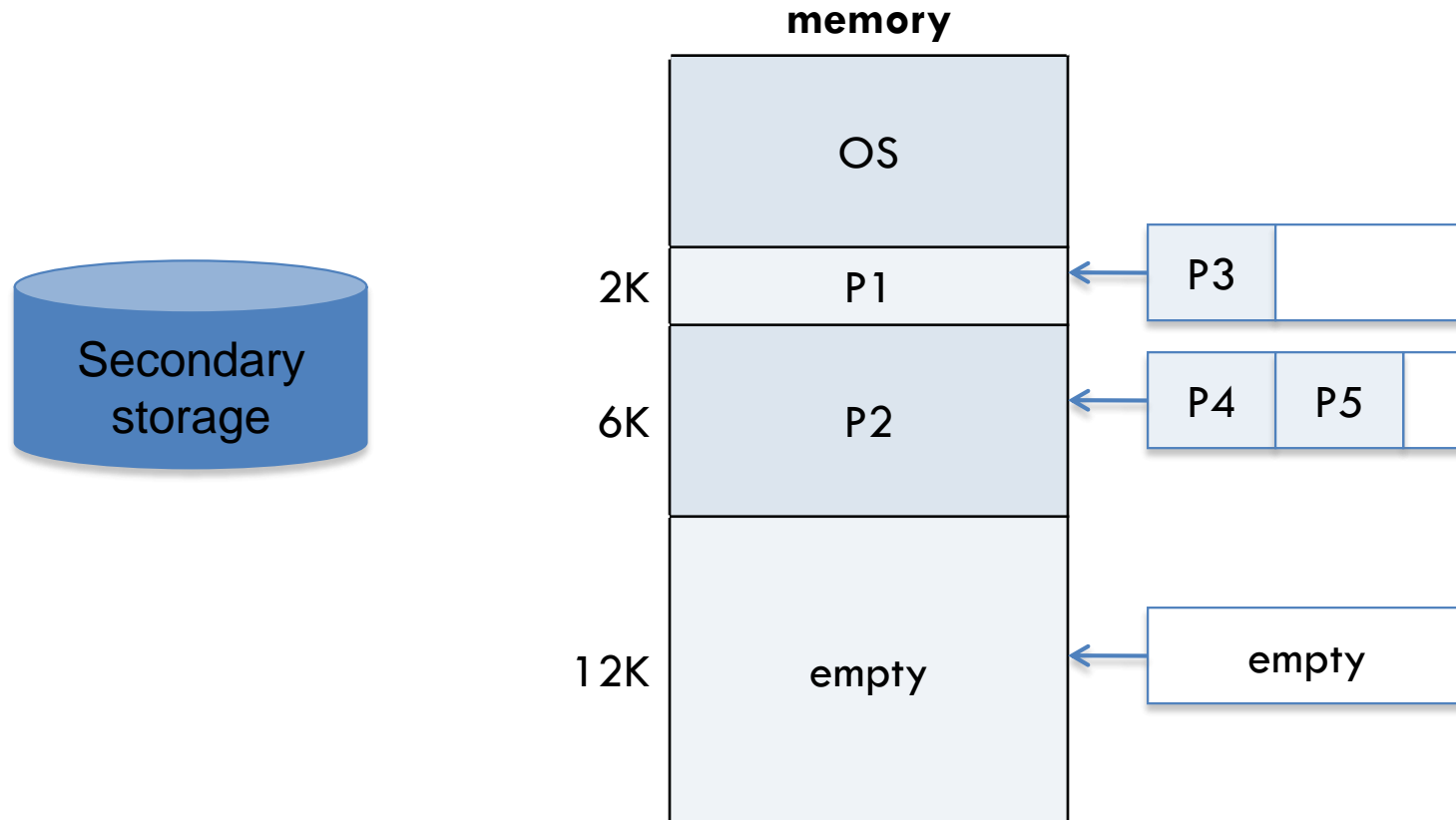
- The main problems with fixed partitioning method are
 - how to determine the number of partitions,
 - how to determine partition sizes.

Fixed Partitioning with Swapping

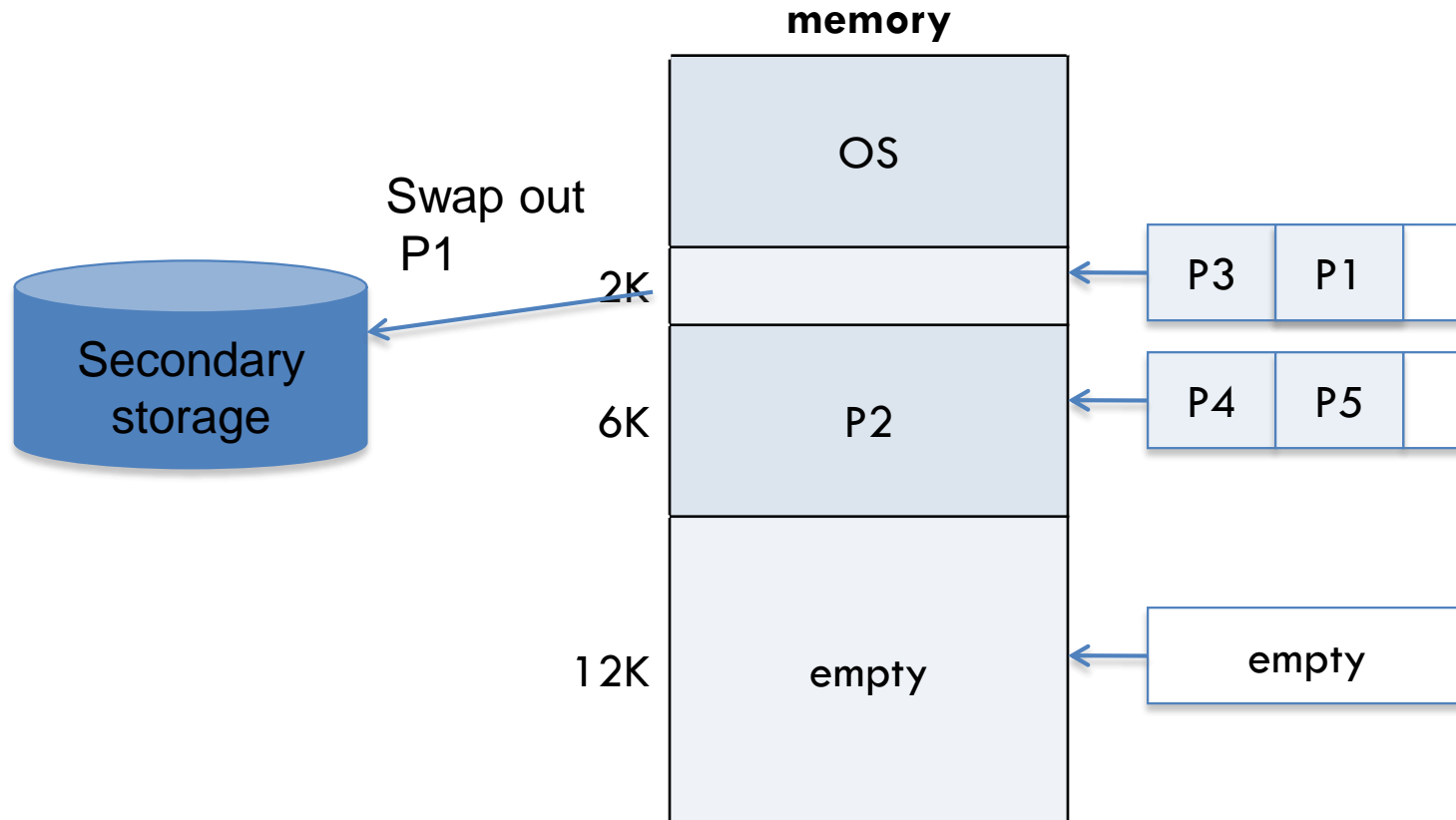


- This is a version of fixed partitioning that uses RRS with some time quantum.
- When time quantum for a process expires, it is swapped out of memory to disk and the next process in the corresponding queue is swapped into the memory.

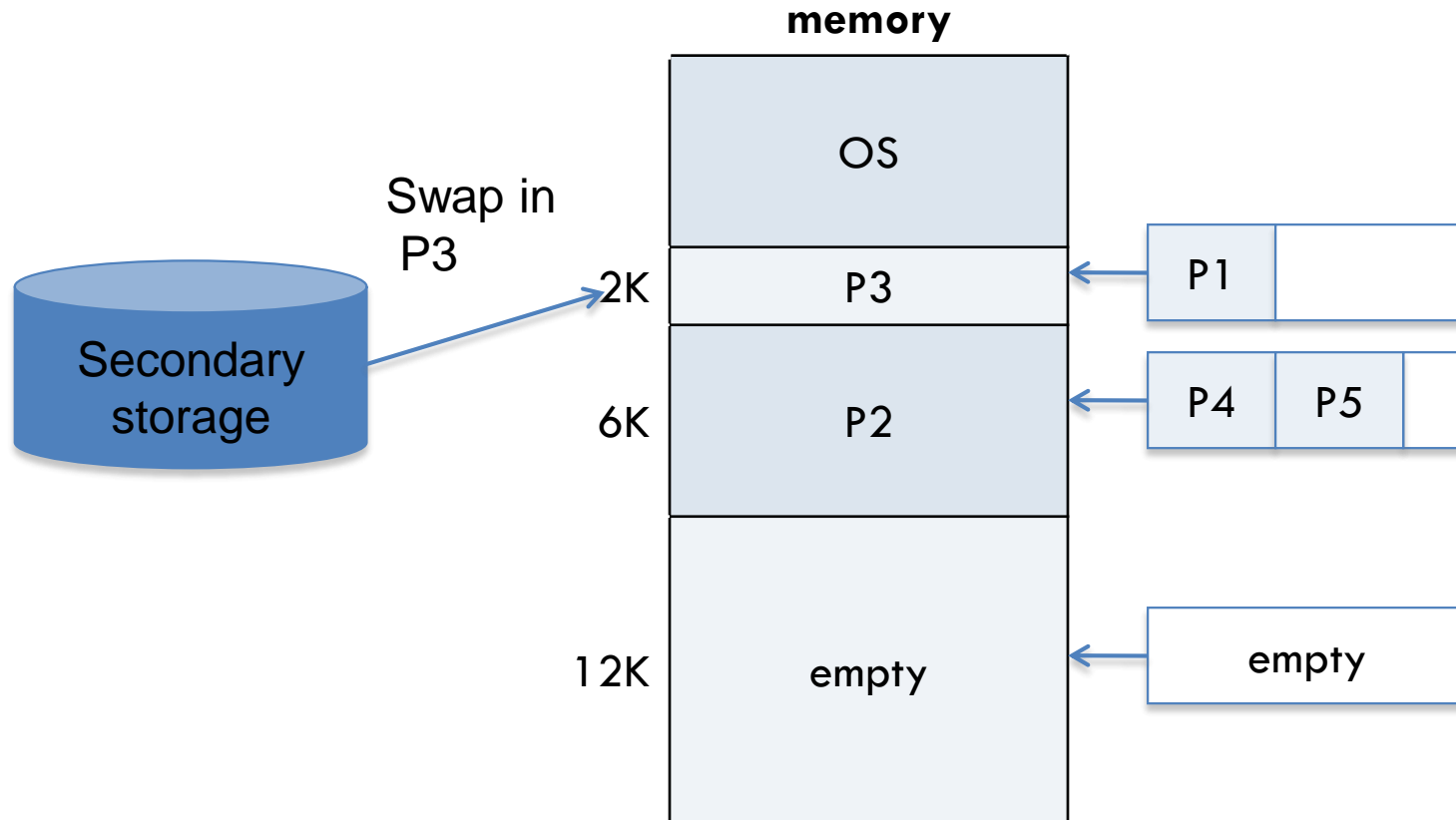
Fixed Partitioning with Swapping



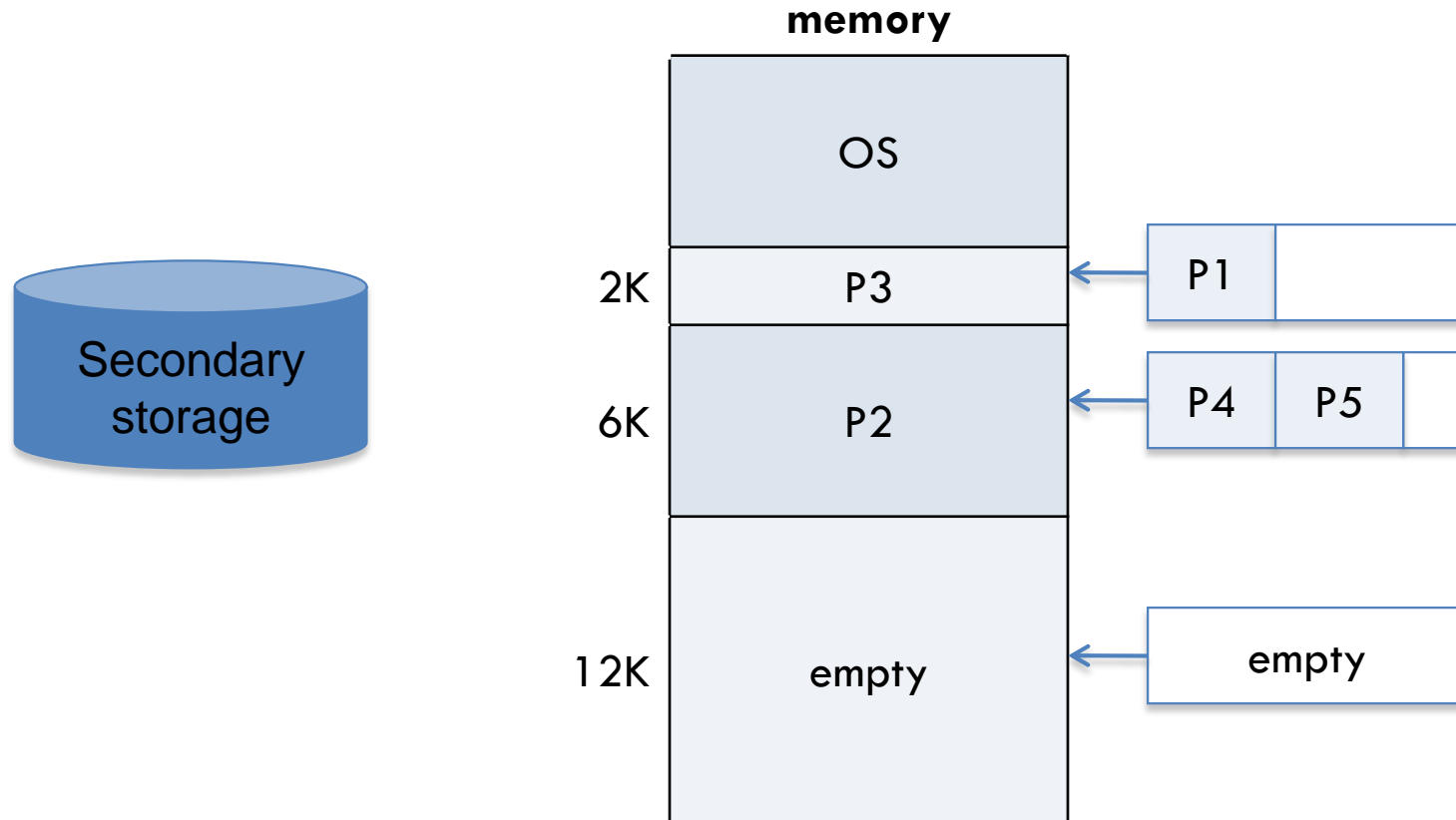
Fixed Partitioning with Swapping



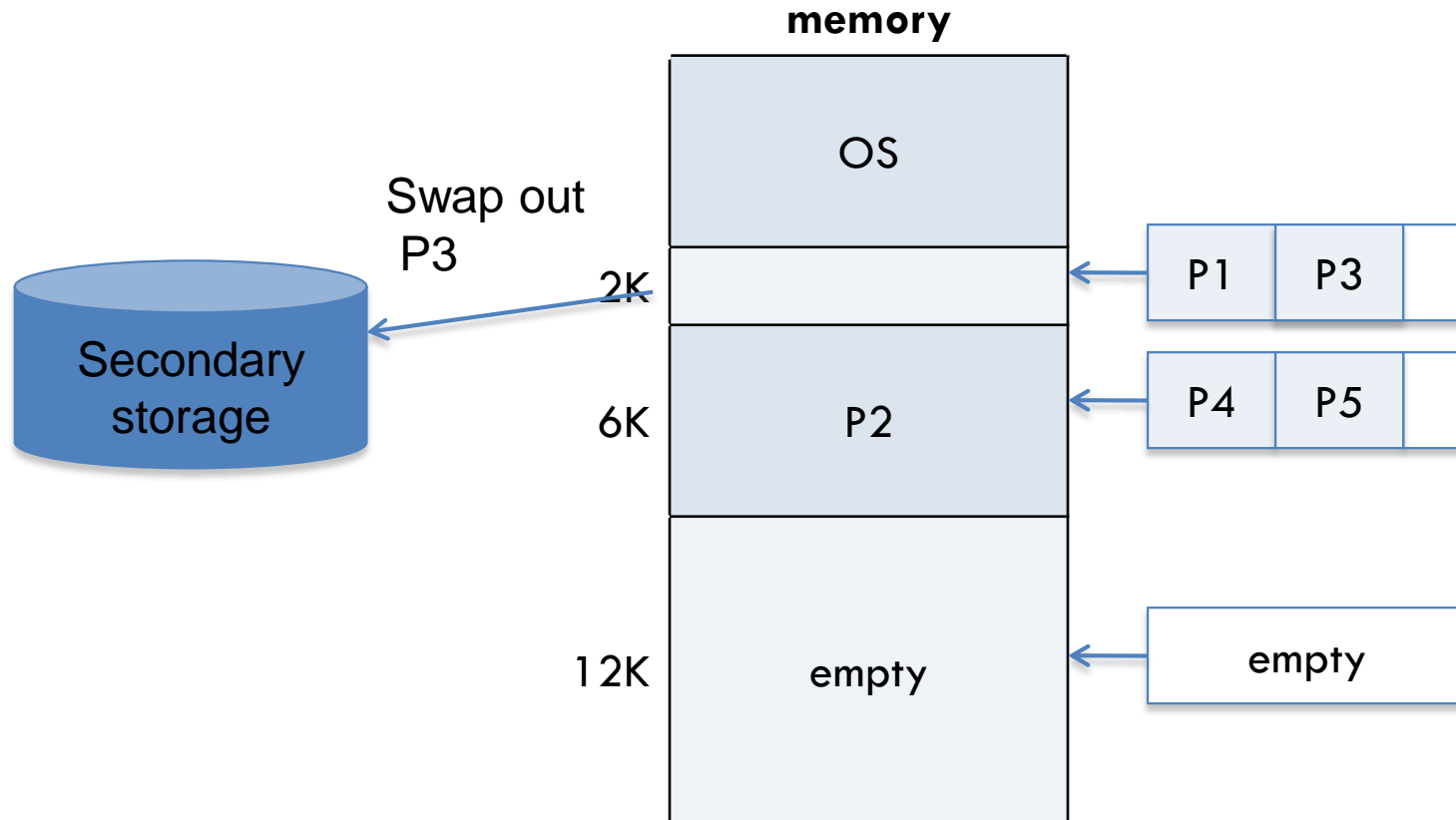
Fixed Partitioning with Swapping



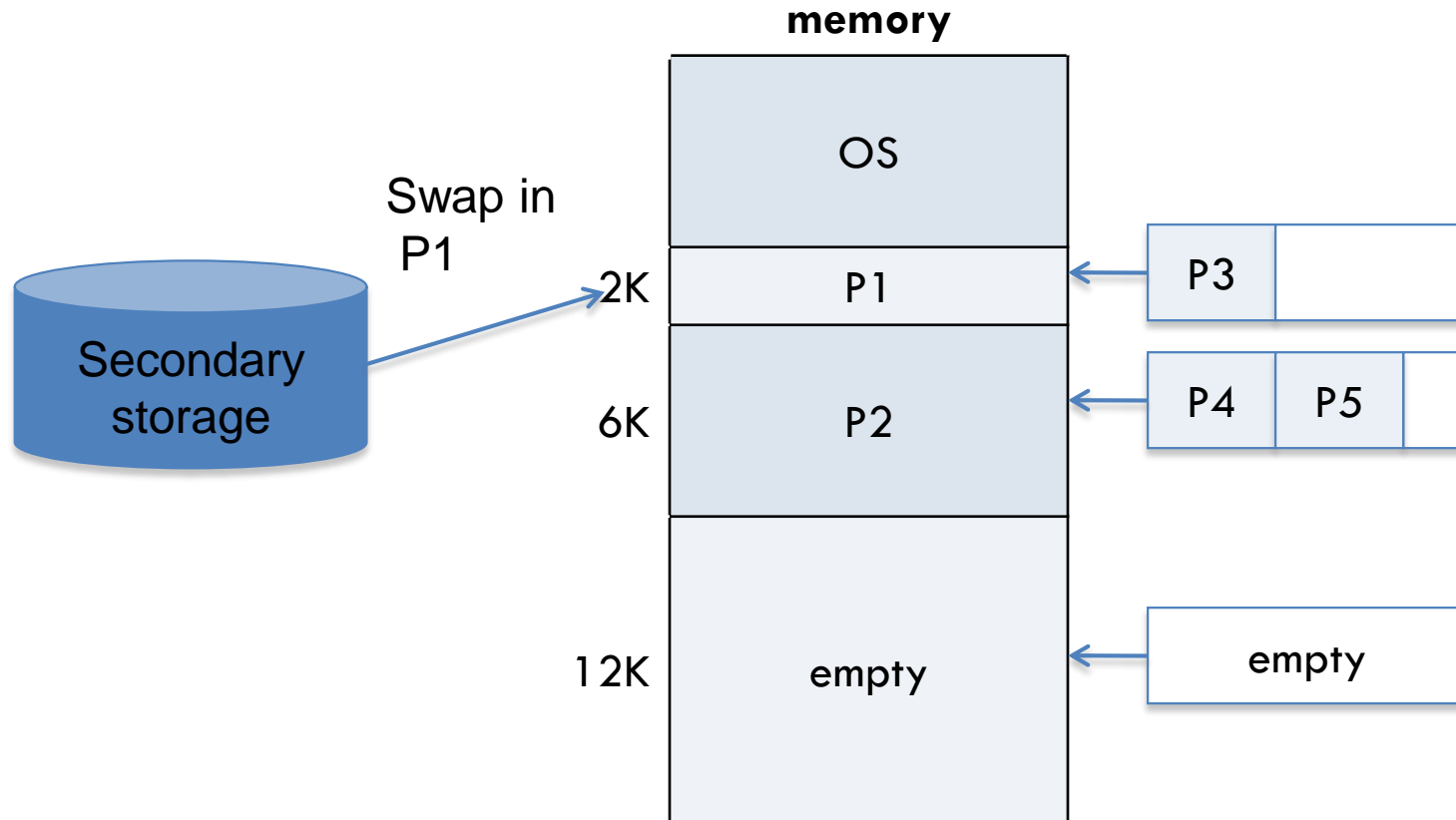
Fixed Partitioning with Swapping



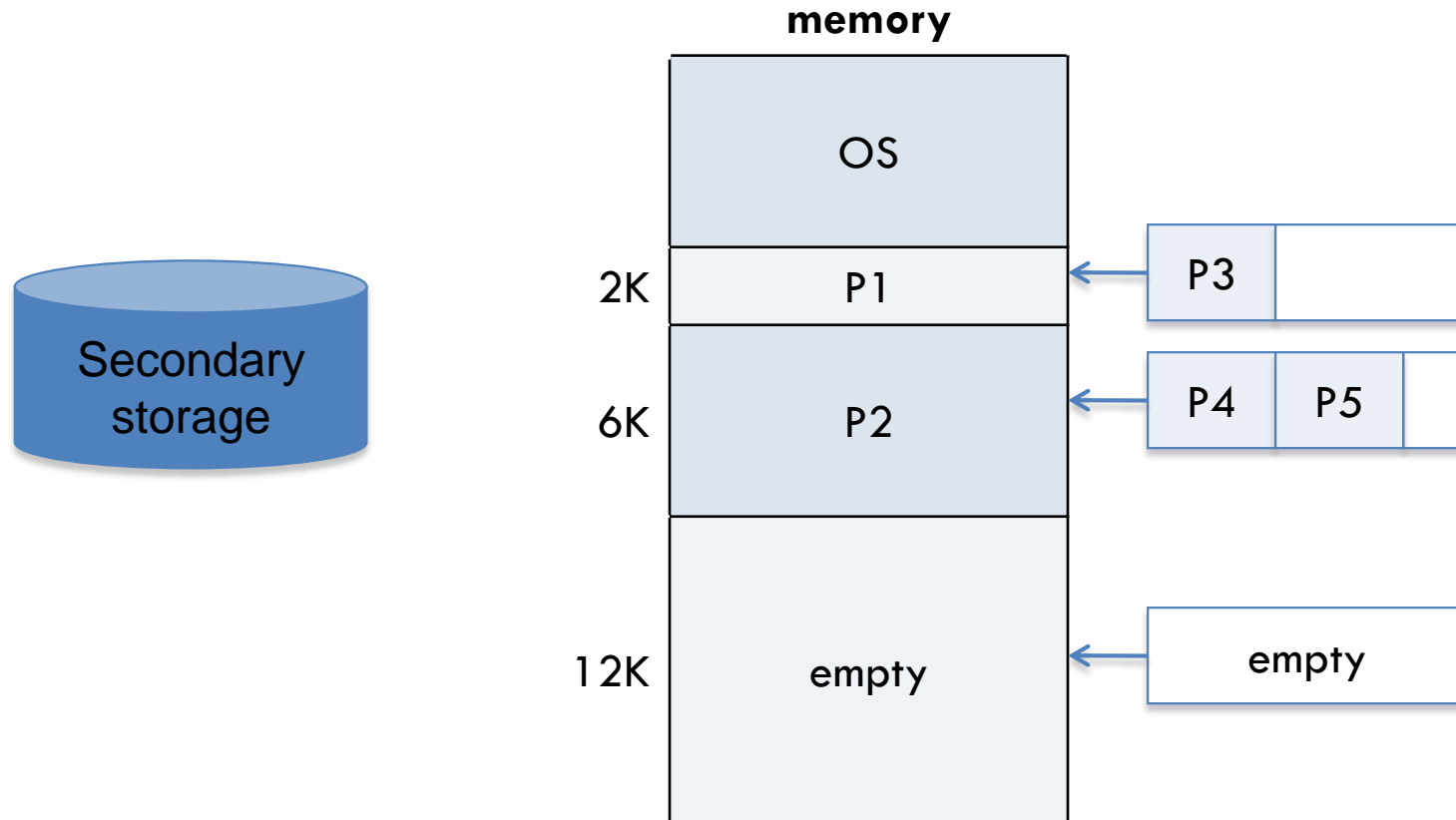
Fixed Partitioning with Swapping



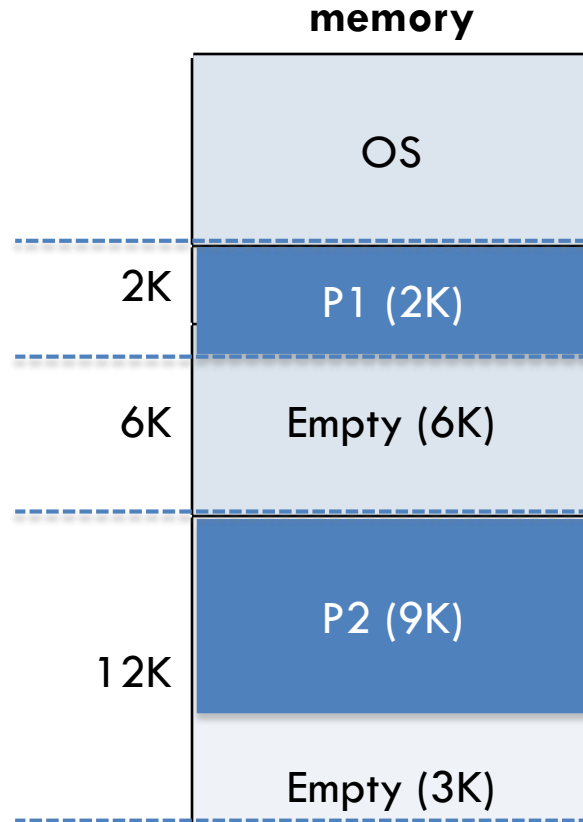
Fixed Partitioning with Swapping



Fixed Partitioning with Swapping



Fragmentation



If a whole partition is currently not being used, then it is called an *external fragmentation*.

If a partition is being used by a process requiring some memory smaller than the partition size, then it is called an *internal fragmentation*.

Variable Partitioning

- With fixed partitions we have to deal with the problem of determining the number and sizes of partitions to minimize internal and external fragmentation.
- If we use variable partitioning instead, then partition sizes may vary dynamically.
- In the variable partitioning method, we keep a table (linked list) indicating used/free areas in memory.

Variable Partitioning

- Initially, the whole memory is free and it is considered as one large block.
- When a new process arrives, the OS searches for a block of free memory large enough for that process.
- After process is loaded into memory, we keep the rest available (free) for the future processes.
- If a block becomes free, then the OS tries to merge it with its neighbors if they are also free.

Memory Management Algorithms for Variable Partitioning

- First fit
- Next fit
- Best fit
- Worst fit
- Quick fit

First fit

First Fit : Allocate the first free block that is large enough for the new process.

This is a fast algorithm.

First fit

Initial memory
mapping

| |
|--------------|
| OS |
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

First fit

P4 of 3KB
arrives

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

First fit

P4 of 3KB
loaded here by
FIRST FIT

| |
|--------------|
| OS |
| P1 12 KB |
| P4 3 KB |
| <FREE> 7 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

First fit

P5 of 15KB
arrives

| OS |
|--------------|
| P1 12 KB |
| P4 3 KB |
| <FREE> 7 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

First fit

P5 of 15 KB
loaded here by
FIRST FIT

| |
|-------------|
| OS |
| P1 12 KB |
| P4 3 KB |
| <FREE> 7 KB |
| P2 20 KB |
| P5 15 KB |
| <FREE> 1 KB |
| P3 6 KB |
| <FREE> 4 KB |

Best fit

- Best Fit :
 - Allocate the smallest block among those that are large enough for the new process.
- In this method, OS has to search the entire list, or it can keep it sorted and stop when it hits an entry, which has a size larger than the size of new process.
- This algorithm produces the smallest left over block.
- However, it requires more time for searching all the list or sorting it
- If sorting is used, merging the area released when a process terminates to neighboring free blocks, becomes complicated.

Best fit

Initial memory
mapping

| |
|--------------|
| OS |
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

Best fit

P4 of 3KB
arrives

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

Best fit

P4 of 3KB
loaded here by
BEST FIT

| |
|--------------|
| OS |
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| P4 3 KB |
| <FREE> 1 KB |

Best fit

P5 of 15KB
arrives

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| P4 3 KB |
| <FREE> 1 KB |

Best fit

P5 of 15 KB
loaded here by
BEST FIT

| |
|--------------|
| OS |
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| P5 15 KB |
| <FREE> 1 KB |
| P3 6 KB |
| P4 3 KB |
| <FREE> 1 KB |

Worst fit

- Worst Fit : Allocate the largest block among those that are large enough for the new process.
- Again a search of the entire list or sorting it is needed.
- This algorithm produces the largest over block.

Worst fit

Initial memory
mapping

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

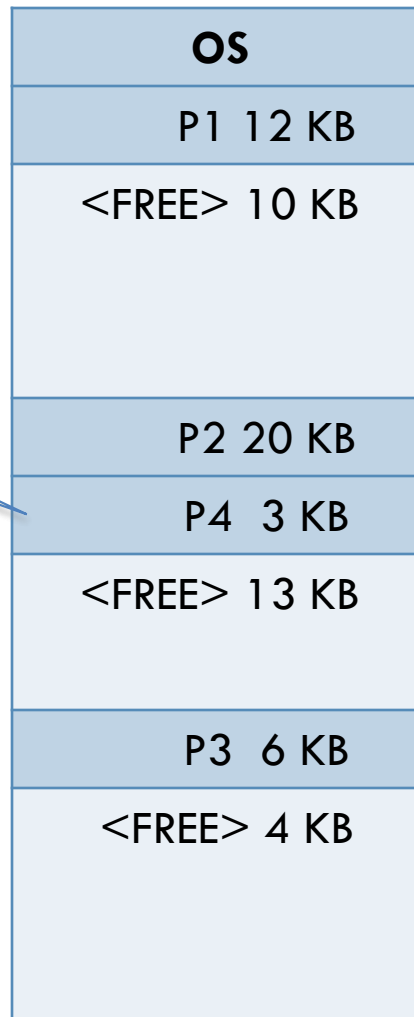
Worst fit

P4 of 3KB
arrives

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| <FREE> 16 KB |
| P3 6 KB |
| <FREE> 4 KB |

Worst fit

P4 of 3KB
Loaded here by
WORST FIT



Worst fit

No place to load
P5 of 15K

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| P4 3 KB |
| <FREE> 13 KB |
| P3 6 KB |
| <FREE> 4 KB |

Worst fit

No place to load
P5 of 15K

Compaction is
needed !!

| OS |
|--------------|
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| P4 3 KB |
| <FREE> 13 KB |
| P3 6 KB |
| <FREE> 4 KB |

Compaction

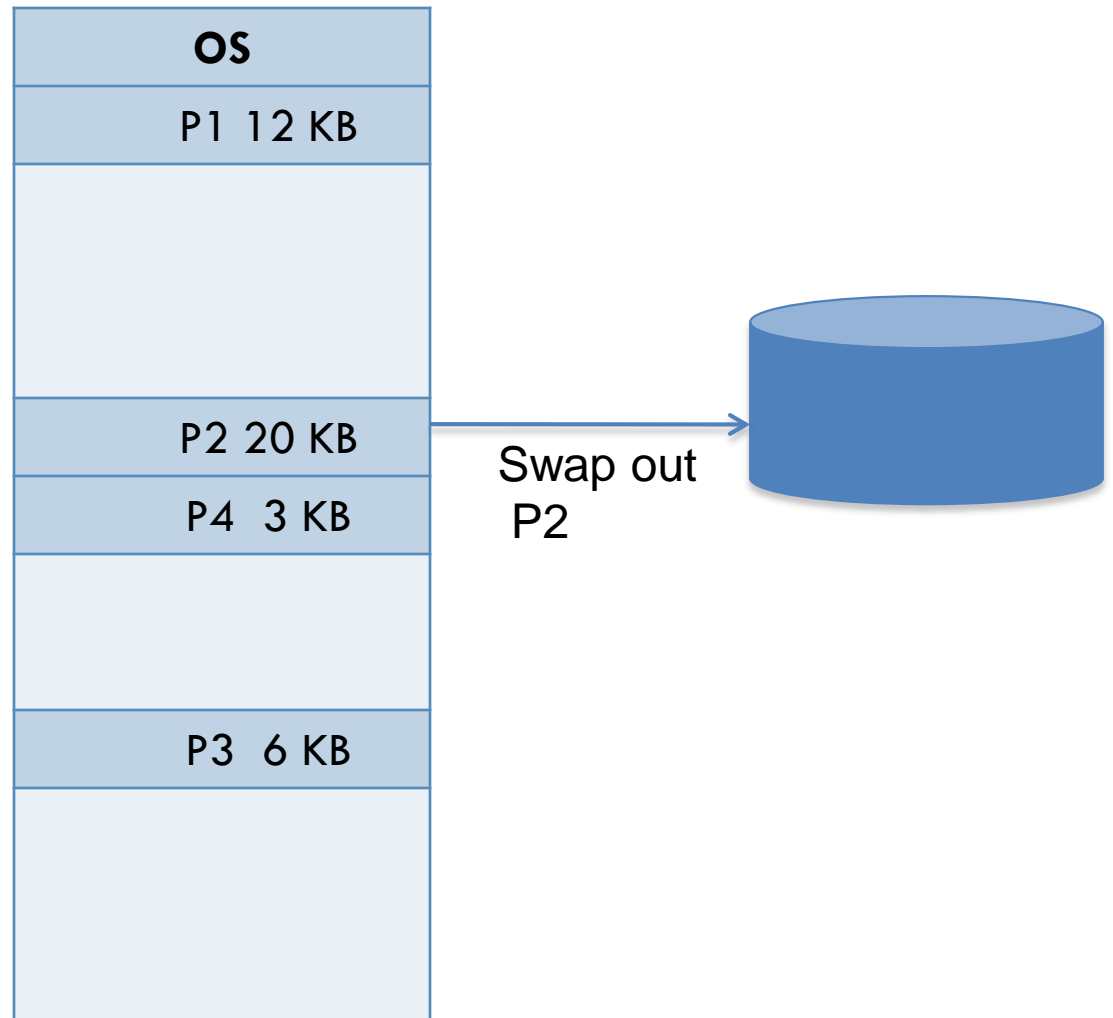
- Compaction is a method to overcome the external fragmentation problem.
- All free blocks are brought together as one large block of free space.
- Compaction requires dynamic relocation.
- Certainly, compaction has a cost and selection of an optimal compaction strategy is difficult.
- One method for compaction is swapping out those processes that are to be moved within the memory, and swapping them into different memory locations

Compaction

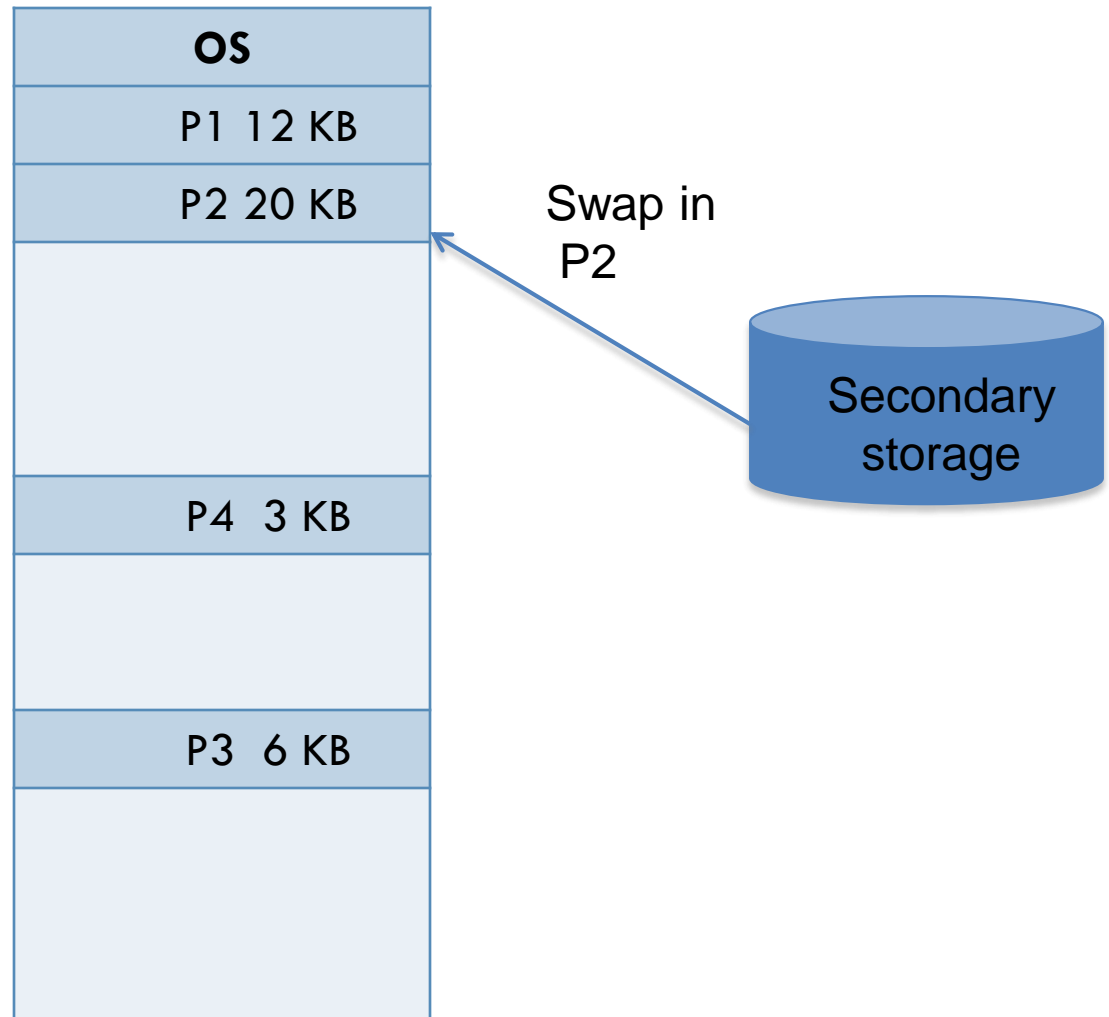
Memory mapping
before
compaction

| |
|--------------|
| OS |
| P1 12 KB |
| <FREE> 10 KB |
| P2 20 KB |
| P4 3 KB |
| <FREE> 13 KB |
| P3 6 KB |
| <FREE> 4 KB |

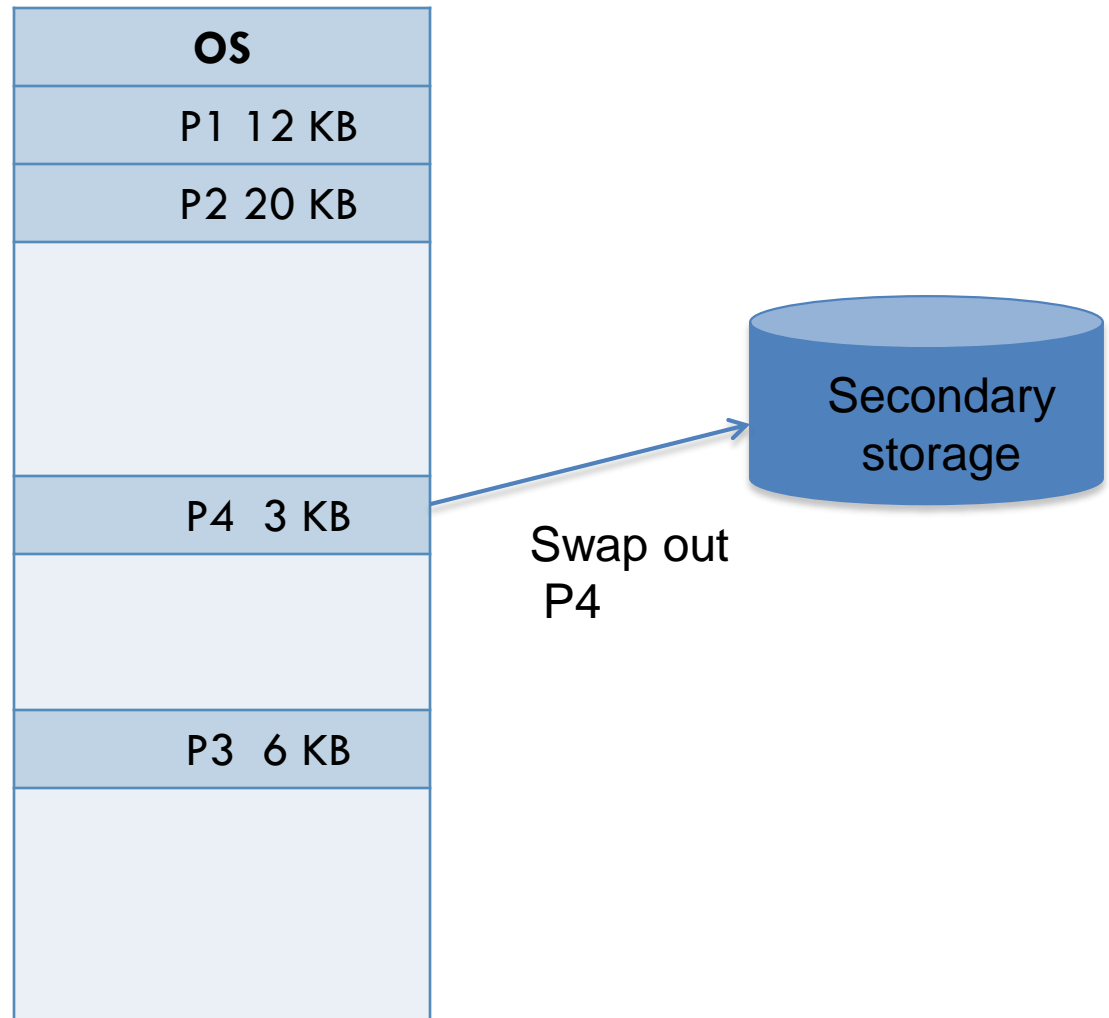
Compaction



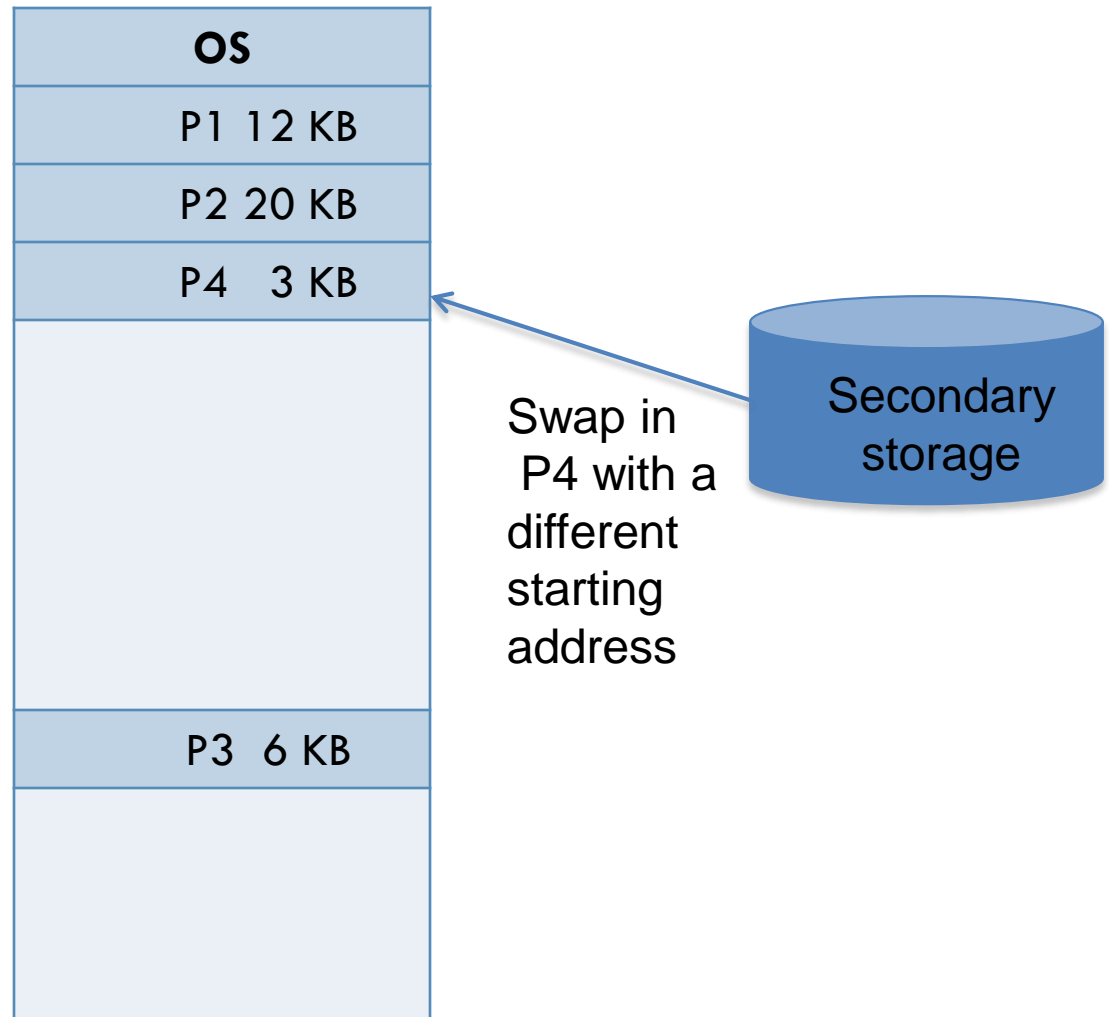
Compaction



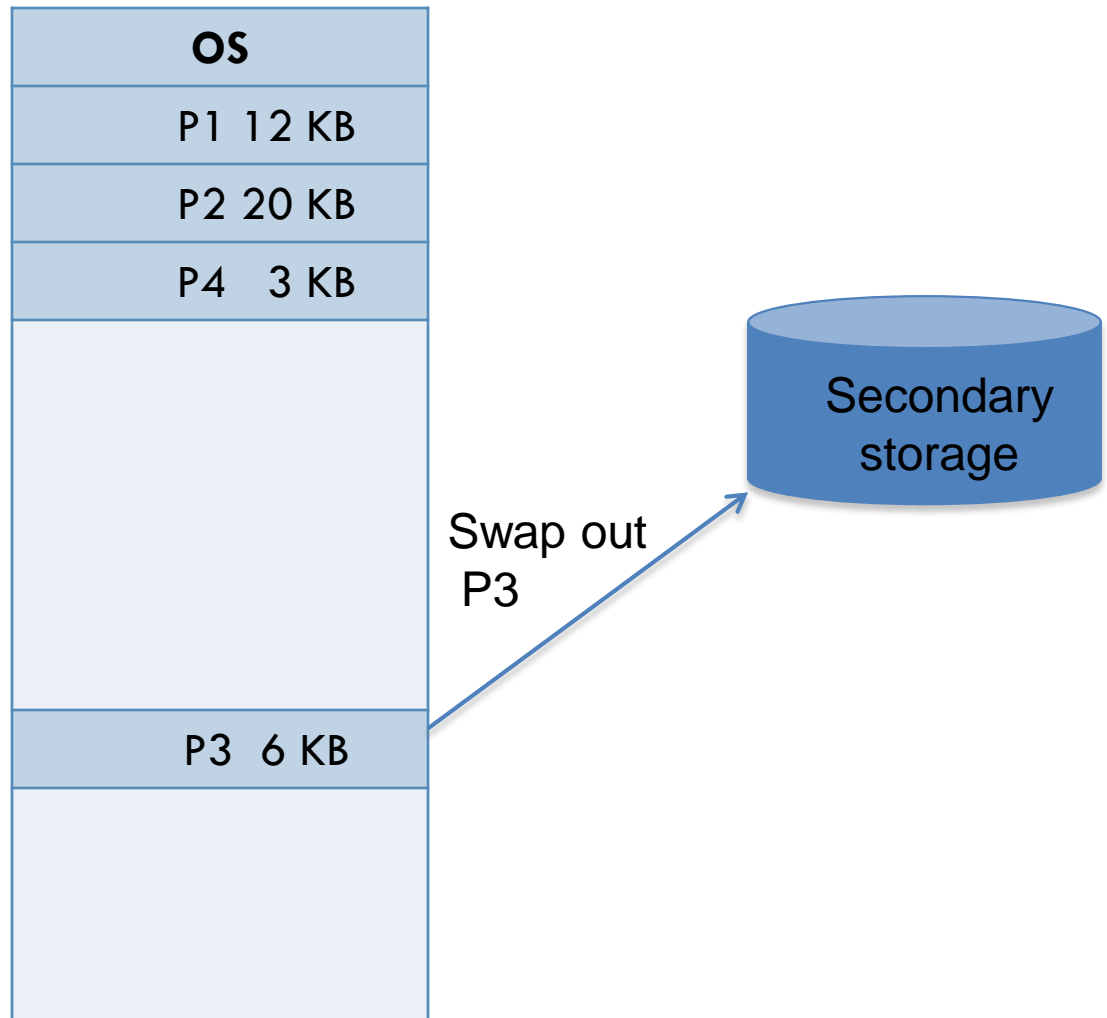
Compaction



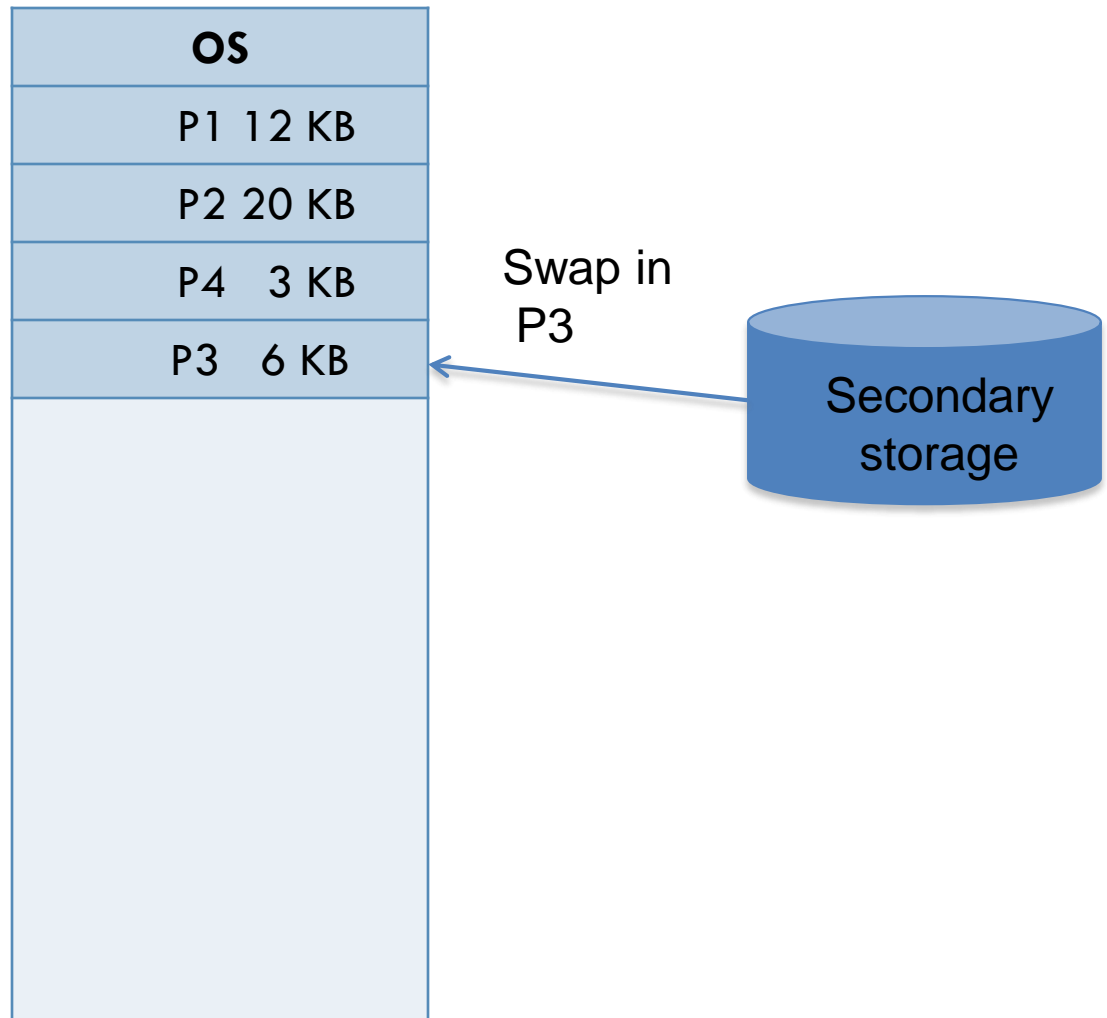
Compaction



Compaction



Compaction



Compaction

Memory mapping
after compaction

| OS |
|--------------|
| P1 12 KB |
| P2 20 KB |
| P4 3 KB |
| P3 6 KB |
| <FREE> 27 KB |

Now P5 of 15KB
can be loaded
here

Compaction

| OS |
|--------------|
| P1 12 KB |
| P2 20 KB |
| P4 3 KB |
| P3 6 KB |
| P5 12 KB |
| <FREE> 12 KB |

P5 of 15KB is loaded

Relocation

- **Static relocation:** A process may be loaded into memory, each time possibly having a different starting address
 - Necessary for variable partitioning
- **Dynamic relocation:** In addition to static relocation, the starting address of the process may change while it is already loaded in memory
 - Necessary for compaction

Virtual Memory

- All the memory management policies we have discussed so far, try to keep a number of processes in the memory simultaneously to allow multiprogramming.
- But they require an entire process to be loaded in the memory before it can execute.

Virtual Memory

- With virtual memory technique, we can execute a process, which is only partially loaded in memory.
- Thus, the logical address space may be larger than physical memory.
- More processes can reside in memory to execute concurrently, hence a greater degree of multiprogramming can be achieved.

Virtual Memory

- The need to run programs that are too large to fit in memory exists since the early days of computing.
- Solution adopted in 1960s, split programs into little pieces, called **overlays**
 - programs kept on the disk, swapped in and out of memory by the overlay manager.
- Virtual memory: each program has its own address space, broken up into chunks called pages

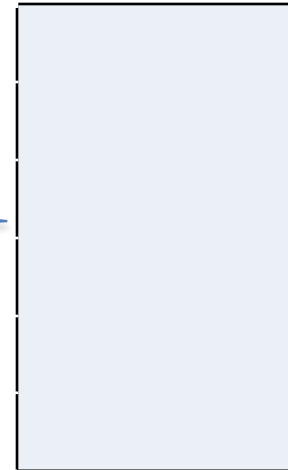
Virtual Memory

- Each page is a contiguous range of addresses.
- Pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time.
 - When a page in physical memory is addressed, HW performs the mapping on the fly.
 - When the addressed page is not in physical memory, OS is alerted to get the missing page and re-execute the instruction that failed.

Paging

In paging, the OS divide the physical memory into frames which are blocks of small and fixed size

Physical
memory



Paging

In paging, the OS divide the physical memory into frames which are blocks of small and fixed size

Physical
memory



Paging

Logical
memory



OS divides also the
logical memory
(program) into
pages which are
blocks of size equal
to frame size.

Physical
memory



Paging

Logical
memory

| |
|----|
| P0 |
| P1 |
| P2 |
| P3 |

OS divides also the logical memory (program) into pages which are blocks of size equal to frame size.

Physical
memory

| | |
|--|----|
| | f0 |
| | f1 |
| | f2 |
| | f3 |
| | f4 |
| | f5 |

Paging

Logical
memory

| |
|----|
| P0 |
| P1 |
| P2 |
| P3 |

PAGE TABLE

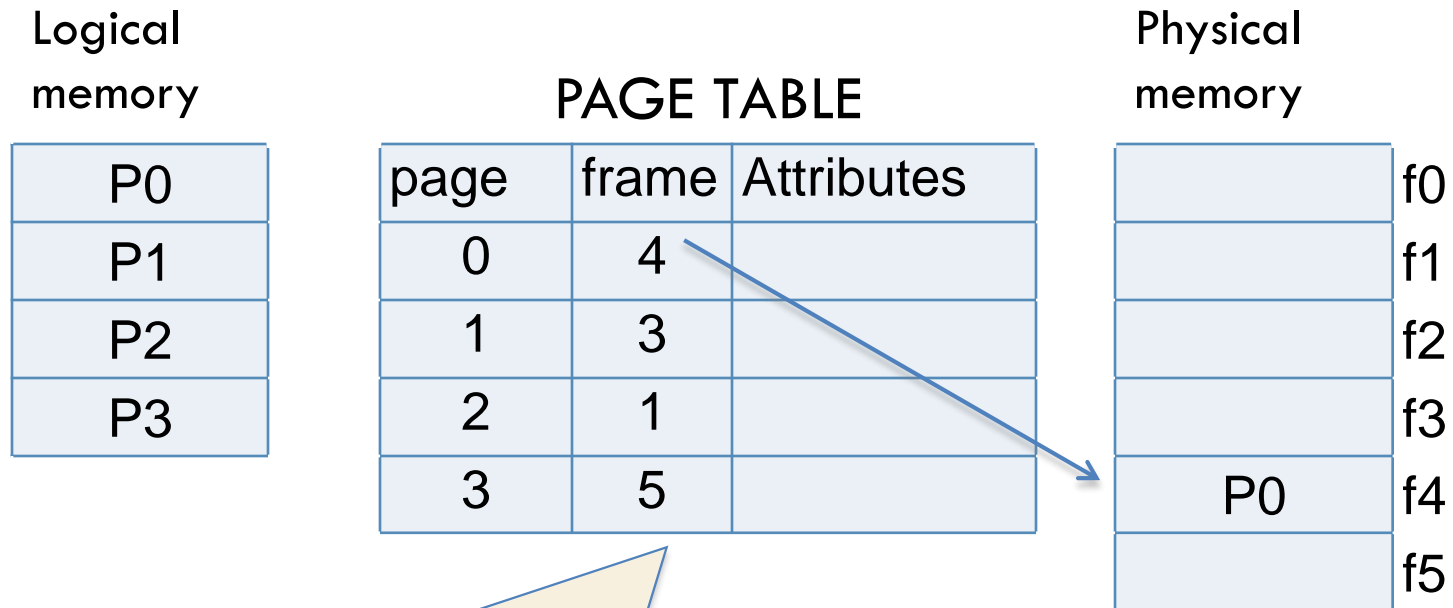
| page | frame | Attributes |
|------|-------|------------|
| 0 | 4 | |
| 1 | 3 | |
| 2 | 1 | |
| 3 | 5 | |

Physical
memory

| | |
|--|----|
| | f0 |
| | f1 |
| | f2 |
| | f3 |
| | f4 |
| | f5 |

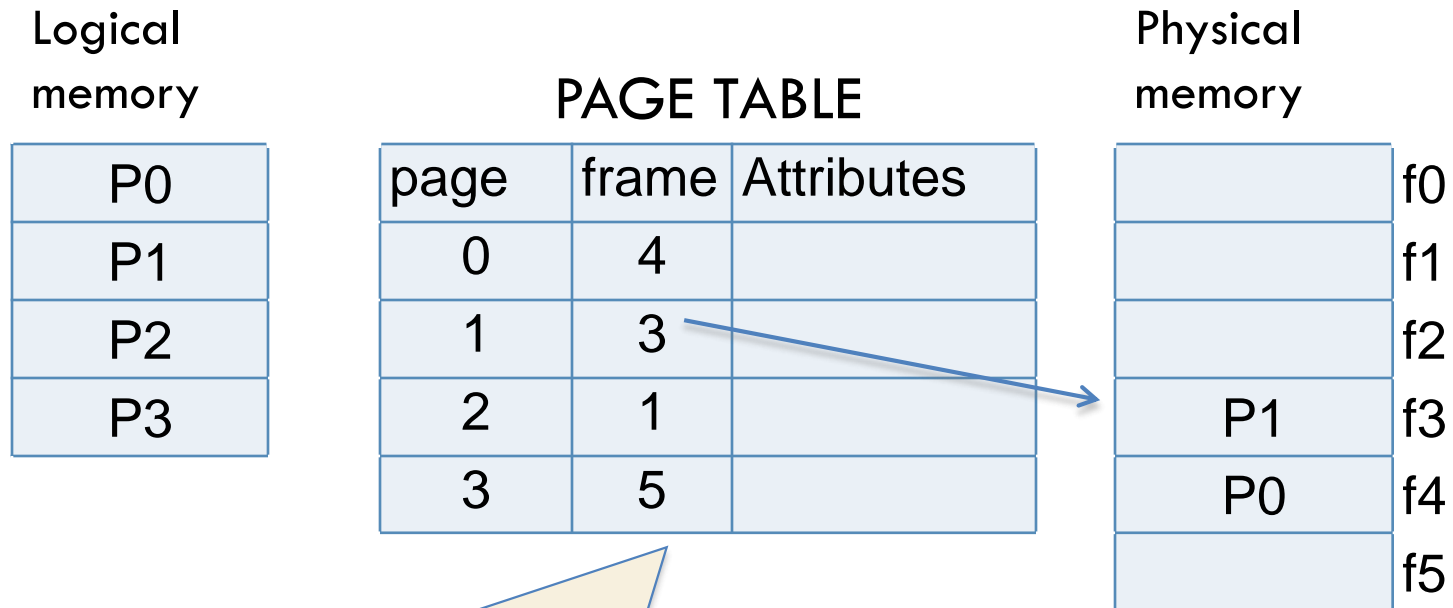
The OS uses a *page table* to map program pages to memory frames.

Paging



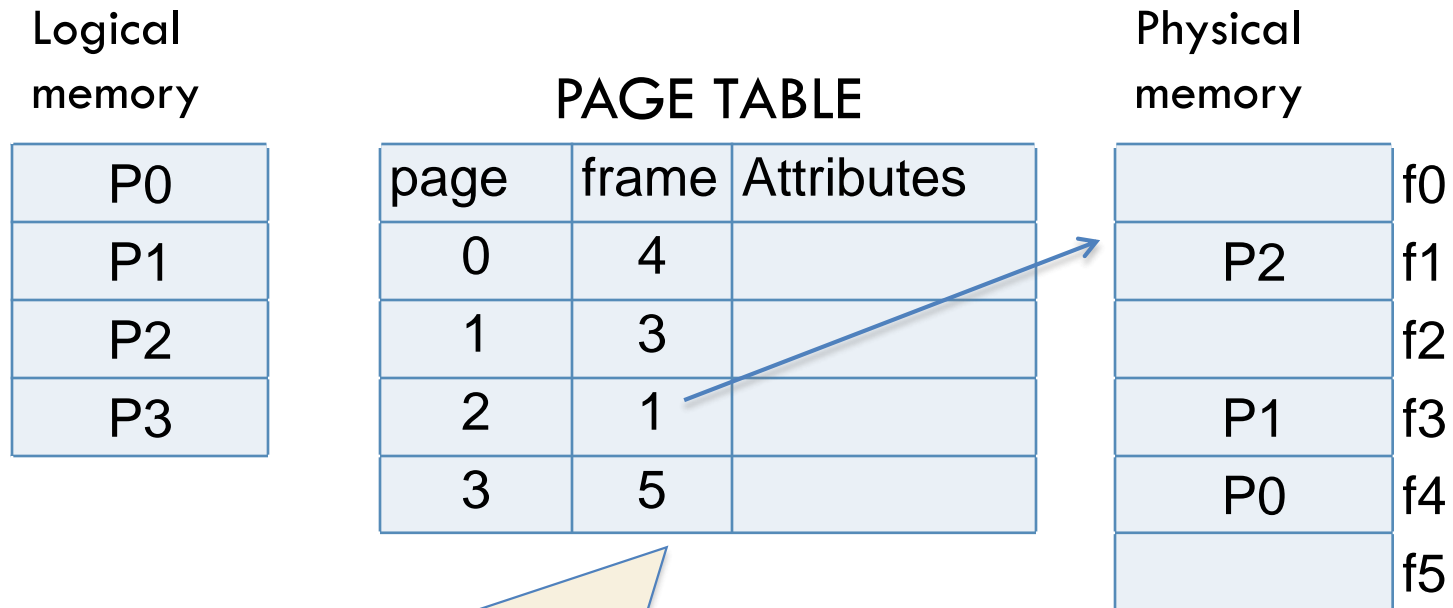
The OS uses a *page table* to map program pages to memory frames.

Paging



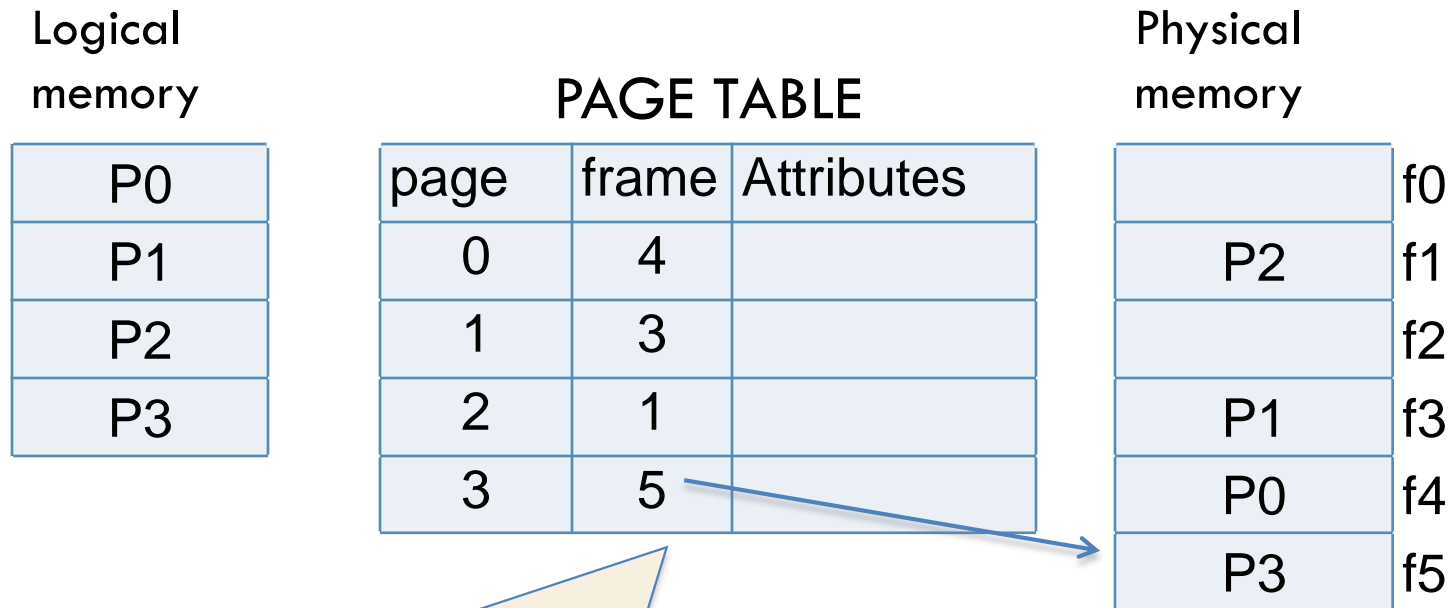
The OS uses a *page table* to map program pages to memory frames.

Paging



The OS uses a *page table* to map program pages to memory frames.

Paging



The OS uses a *page table* to map program pages to memory frames.

Paging

Logical
memory

| |
|----|
| P0 |
| P1 |
| P2 |
| P3 |

PAGE TABLE

| page | frame | Attributes |
|------|-------|------------|
| 0 | 4 | |
| 1 | 3 | |
| 2 | 1 | |
| 3 | 5 | |

Physical
memory

| | |
|----|----|
| | f0 |
| P2 | f1 |
| | f2 |
| P1 | f3 |
| P0 | f4 |
| P3 | f5 |

Paging permits a program to allocate noncontiguous blocks of memory

Paging

- Page size (S) is defined by the hardware.
 - Generally page size is chosen as a power of 2 such as 512 words/page or 4096 words/page etc.
- With this arrangement, the words in the program have an address called as *logical address*.
- Every logical address is formed of $\langle p, d \rangle$ pair.

Paging

- Logical address: $\langle p, d \rangle$
 - *p is page number*
 $p = \text{logical address} \div S$
 - *d is displacement (offset)*
 $d = \text{logical address} \bmod S$

Paging

- When a logical address $\langle p, d \rangle$ is generated by the processor,
 - frame number f corresponding to page p is determined by using the page table
 - physical address is calculated as $(f * S + d)$ and the memory is accessed.

Demand Paging

- *Demand paging* is the most common virtual memory management system.
- It is based on the locality model of program execution.
- *Locality* is defined as a set of pages actively used together.
- As a program executes, it moves from locality to locality.
- A program is composed of several different localities that may overlap:
 - for example, a small procedure when called, defines a new locality.
 - a while-do loop when being executed defines a new locality.

Demand Paging

- In demand paging, programs reside on a swapping device (typically a disk) commonly known as the backing store.
- When OS decides to start a new process, it swaps only a small part of this new process (maybe a few pages) into memory.
- The page table of this new process is prepared and loaded into memory, and the valid/invalid bits of all pages that are brought into memory are set to “valid”.
- All the other valid/invalid bits are set to “invalid” showing that those pages are not brought into memory.

Demand Paging

- If the process currently executing tries to access a page, which is not in the memory, a ***page fault (PF)*** occurs.
- In case of a PF, OS finds the desired page on the backing store.
- If there exists no free frame in memory, OS chooses a suitable page (***victim page***), and swaps it out to the backing store.
- OS changes the valid bit of the victim page in page table to invalid indicating that it is no longer in memory.
- It swaps the desired page into the newly freed frame, modifies frame table and sets the valid/invalid bit of the new page to valid.
- Then executing process is allowed to go on.

Demand Paging

- 'Page Fault Service' operations can be summarized as follows:
 - Suspend the process and do context switch (fast)
 - Check the address and find a free frame or a victim page (fast)
 - Swap out victim page to secondary storage (slow)
 - Swap in the page from secondary storage (slow)
 - Context switch for the suspended process and resume its execution (fast)
- In servicing a PF, time is spent mainly for swap-out and swap-in operations. Time spent for others are negligible in comparison.

Demand Paging

- Virtual memory can be implemented in the form of:
 - paging systems
 - paged segmentation systems
 - segmentation systems (variable size pages)
(However, segment replacement is much more sophisticated than page replacement since segments are of variable size)

Performance Calculation of Demand Paging

- If there is no page fault, effective access time is effective memory access time (emat)

$$\text{eat}_{\text{NO-PF}} = \text{emat}$$

- If there is a page fault, we have

$$\text{eat}_{\text{PF}} = \text{pfst} + \text{emat} \cong \text{pfst}$$

where pfst is page fault service time.

- In general, emat is very small compared to pfst hence it can be ignored.

Performance Calculation of Demand Paging

- Let p be the probability for a page fault ($0 \leq p \leq 1$) to occur.
- Then, the effective access time can be calculated as follows:

$$\begin{aligned} \text{eat} &= p * \text{eat}_{\text{PF}} + (1 - p) * \text{eat}_{\text{NO-PF}} \\ &\cong p * \text{pfst} + (1 - p) * \text{emat} \end{aligned}$$

Performance Calculation of Demand Paging

Example:

Effective memory access time for a system is given as 1 microseconds, and the average page fault service time is given as 10 milliseconds. Let $p=0.001$. Then, effective access time is

$$\begin{aligned} \text{eat} &= 0.001 * 10 \text{ msec} + (1 - 0.001) * 1 \mu\text{sec} \\ &= 0.001 * 10\,000 \mu\text{sec} + 0.999 * 1 \mu\text{sec} \\ &\cong 10 \mu\text{sec} + 1 \mu\text{sec} = 11 \mu\text{sec} \end{aligned}$$

Dirty bit

- In order to reduce PF service time, a special bit called the **dirty bit** can be associated with each page.
- The dirty bit is set to "1" by the hardware whenever the page is modified (written into).
- When we select a victim by using a page replacement algorithm, we examine its dirty bit.
- If it is set, that means the page has been modified since it was swapped in.
- In this case we have to write that page into the backing store.

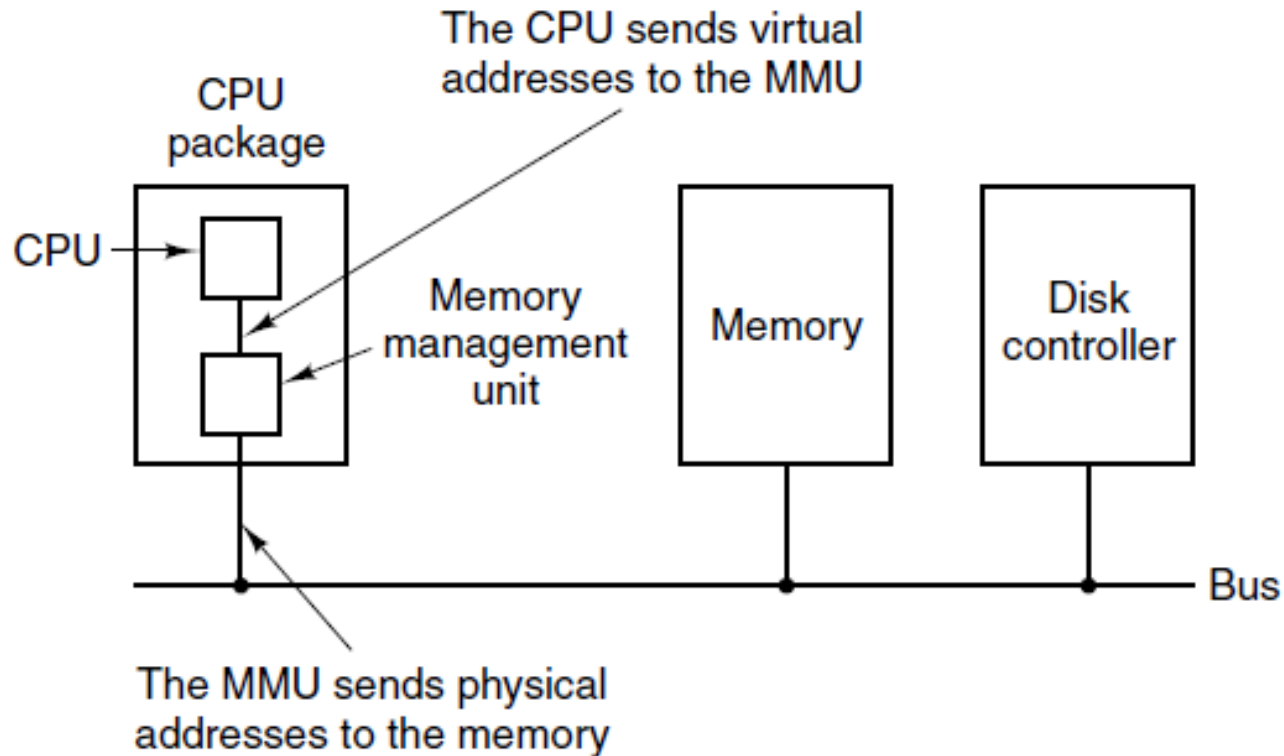
Dirty bit

- However if the dirty bit is reset, that means the page has not been modified since it was swapped in, so we don't have to write it into the backing store. The copy in the backing store is valid.
- Let the probability of a page being dirty be d .

In this case effective access time becomes:

$$eat = p * [(1-d) * swap_in + d * (swap_in + swap_out)] + (1 - p) * emat$$

Paging (1)

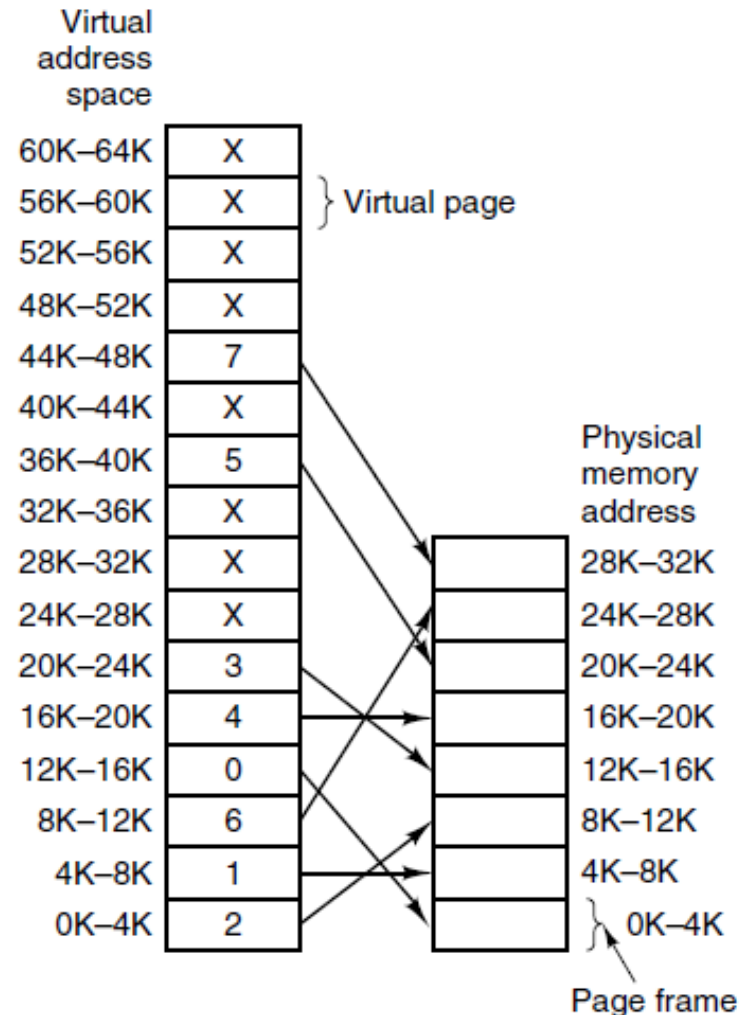


The position and function of the MMU.

- Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays.
- However, logically it could be a separate chip and was so years ago.

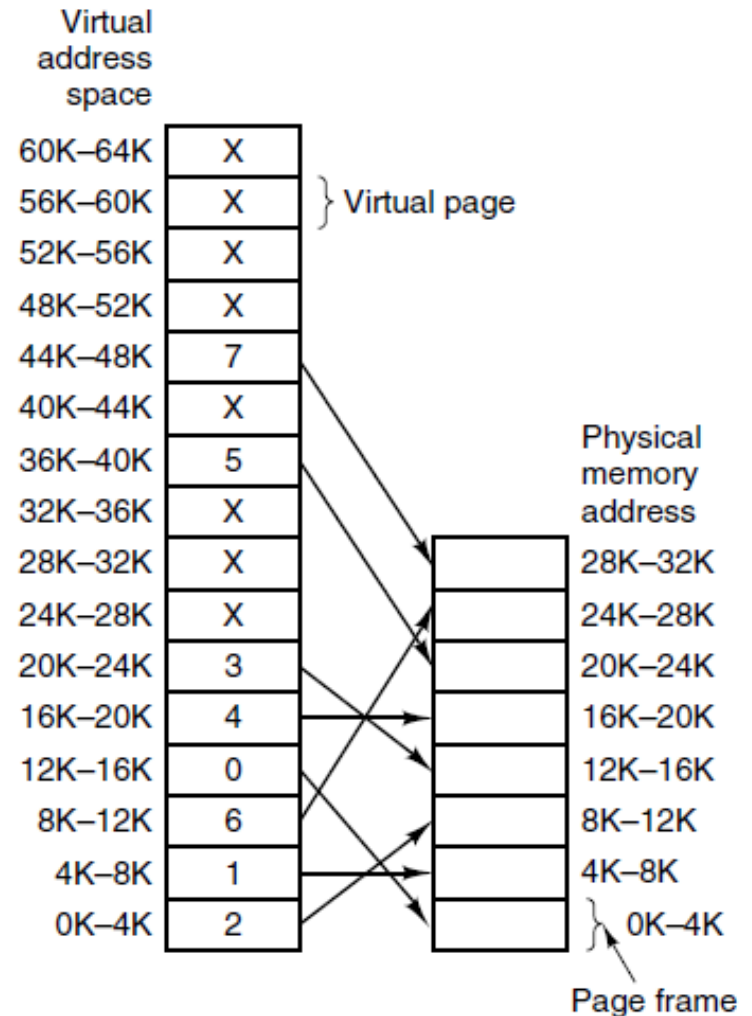
Paging (2)

- The relation between virtual addresses and physical memory addresses is given by the page table.
- Every page begins on a multiple of 4096 and ends 4095 addresses higher
- 4K–8K means 4096–8191
- 8K to 12K means 8192–12287



Paging (3)

- `MOV REG,0` (virtual page 0)
becomes `MOV REG,8192` (physical page 2)
- `MOV REG, 8192`(virtual page 2)
becomes `MOV REG, 24576` (physical page 6)
- `MOV REG, 32780` (byte 12 in virtual page 8)
→ PAGE FAULT
→ SWAP frame 1 and page 8
- `MOV REG, 32780` (byte 12 in virtual page 8)
becomes `MOV REG, 4108` ($4096+12$)



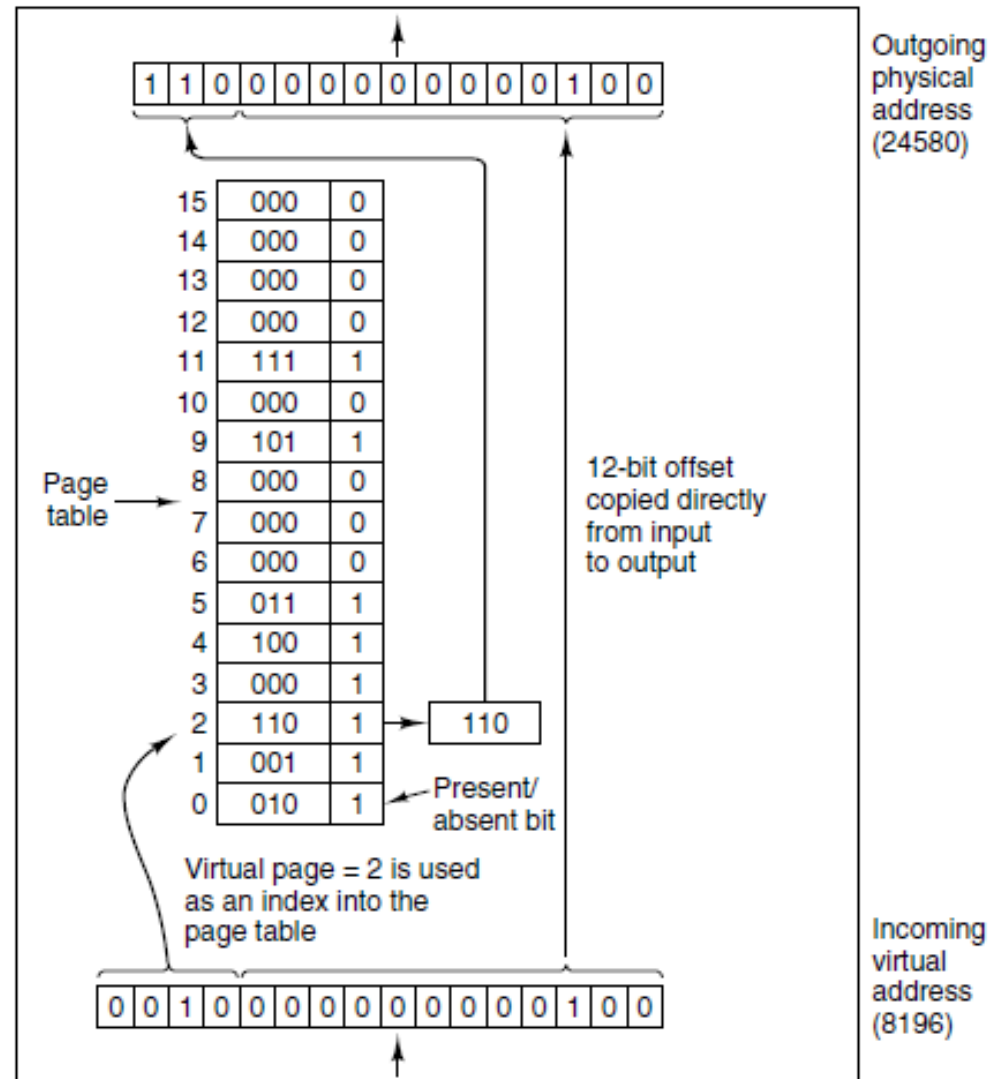
Paging (4)

The page number is used as an index into the **page table**.

If the *Present/absent* bit is 0, a trap to the operating system is caused.

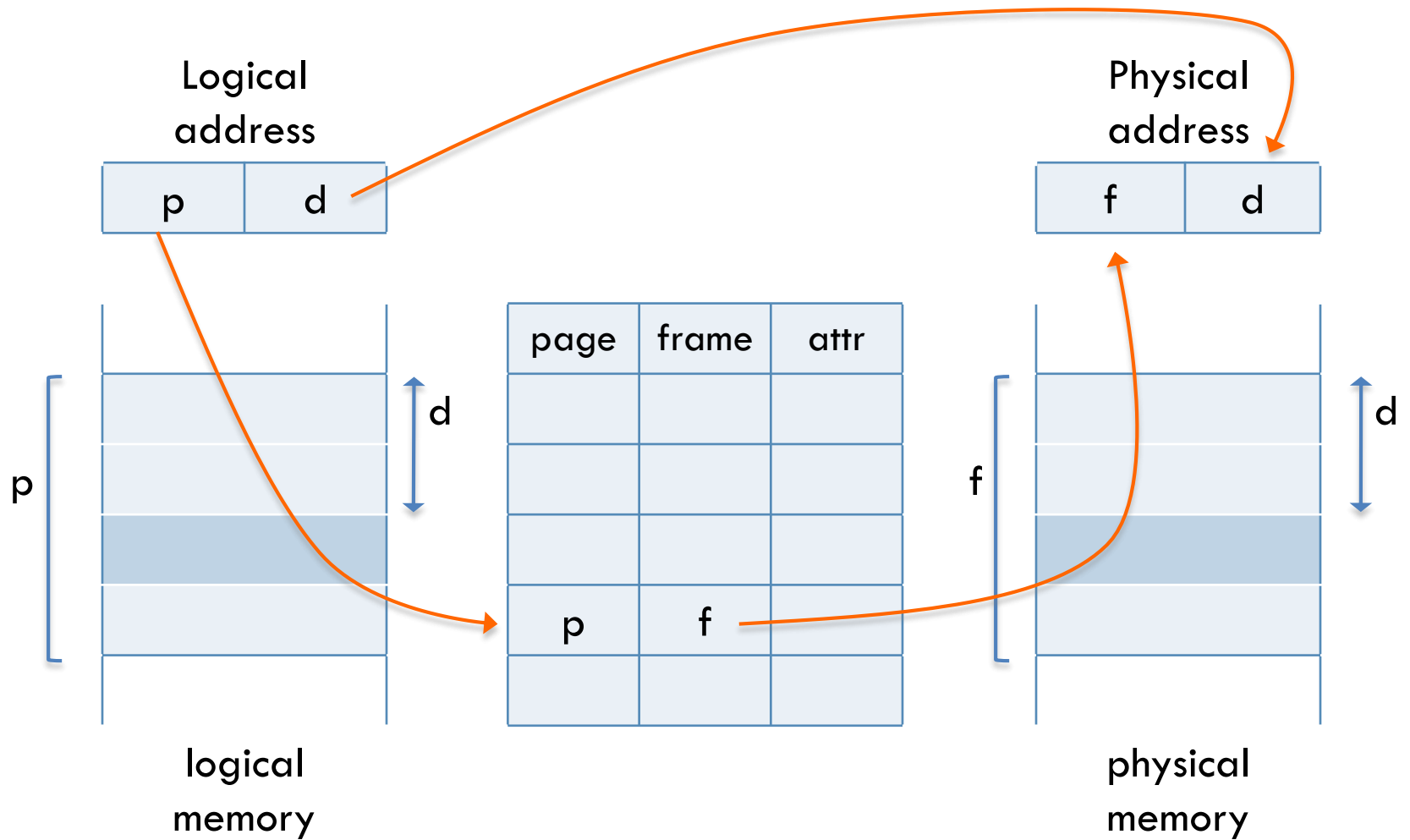
If the bit is 1, the page frame number is copied to the high-order 3 bits of the output register, along with the 12-bit offset

Together they form a 15-bit physical address.



The internal operation of the MMU with 16 4-KB pages.

Paging



Paging

Example

- Consider the following to form a physical memory map.
- Page Size = 8 words \rightarrow d : 3 bits
- Physical Memory Size = 128 words
 $= 128/8=16$ frames \rightarrow f : 4 bits
- Assume maximum program size is 4 pages \rightarrow p : 2 bits
- A program of 3 pages where P0 \rightarrow f3; P1 \rightarrow f6; P2 \rightarrow f4

Logical memory

| | |
|---------|----------------|
| Word 0 | Page 0 (P0) |
| Word 1 | |
| ... | |
| Word 7 | |
| Word 8 | Page 1 (P1) |
| Word 9 | |
| ... | |
| Word 15 | |
| Word 16 | Page 2 (P2) |
| Word 17 | |
| ... | |
| Word 23 | |

PAGE TABLE

| Page | Frame |
|------|-------|
| 0 | 3 |
| 1 | 6 |
| 2 | 4 |

Physical memory

| | |
|---------|-----------------|
| ... | ... |
| Word 0 | Frame 3 (f3) |
| Word 1 | |
| ... | |
| Word 7 | Frame 4 (f4) |
| Word 16 | |
| Word 17 | |
| ... | |
| Word 23 | Frame 6 (f6) |
| | |
| ... | |
| | |
| Word 8 | Frame 6 (f6) |
| Word 9 | |
| ... | |
| Word 15 | |
| ... | ... |

Paging

| Program Line |
|-----------------|
| Word 0 |
| Word 1 |
| ... |
| Word 7 |
| Word 8 |
| Word 9 |
| ... |
| Word 15 |
| Word 16 |
| Word 17 |
| ... |
| Word 23 |

Paging

| Program Line | Logical Address |
|--------------|-----------------|
| Word 0 | 00 000 |
| Word 1 | 00 001 |
| ... | ... |
| Word 7 | 00 111 |
| Word 8 | 01 000 |
| Word 9 | 01 001 |
| ... | ... |
| Word 15 | 01 111 |
| Word 16 | 10 000 |
| Word 17 | 10 001 |
| ... | ... |
| Word 23 | 10 111 |

Paging

| Program Line | Logical Address | Offset |
|--------------|-----------------|--------|
| Word 0 | 00 000 | 000 |
| Word 1 | 00 001 | 001 |
| ... | ... | ... |
| Word 7 | 00 111 | 111 |
| Word 8 | 01 000 | 000 |
| Word 9 | 01 001 | 001 |
| ... | ... | ... |
| Word 15 | 01 111 | 111 |
| Word 16 | 10 000 | 000 |
| Word 17 | 10 001 | 001 |
| ... | ... | ... |
| Word 23 | 10 111 | 111 |

Paging

| Program Line | Logical Address | Offset | Page Number |
|--------------|-----------------|--------|-------------|
| Word 0 | 00 000 | 000 | 00 |
| Word 1 | 00 001 | 001 | 00 |
| ... | ... | ... | ... |
| Word 7 | 00 111 | 111 | 00 |
| Word 8 | 01 000 | 000 | 01 |
| Word 9 | 01 001 | 001 | 01 |
| ... | ... | ... | ... |
| Word 15 | 01 111 | 111 | 01 |
| Word 16 | 10 000 | 000 | 10 |
| Word 17 | 10 001 | 001 | 10 |
| ... | ... | ... | ... |
| Word 23 | 10 111 | 111 | 10 |

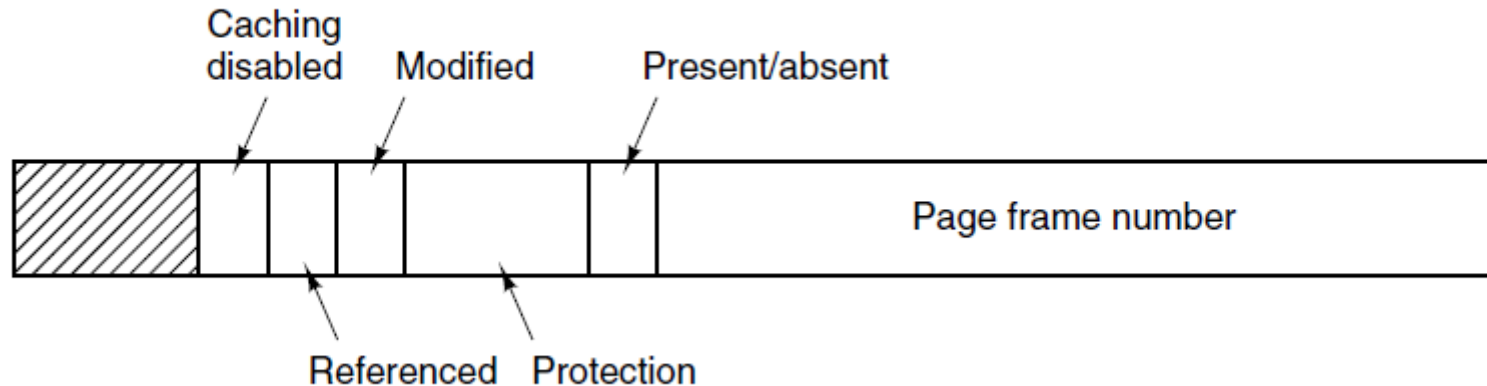
Paging

| Program Line | Logical Address | Offset | Page Number | Frame Number |
|--------------|-----------------|--------|-------------|--------------|
| Word 0 | 00 000 | 000 | 00 | 0011 |
| Word 1 | 00 001 | 001 | 00 | 0011 |
| ... | ... | ... | ... | ... |
| Word 7 | 00 111 | 111 | 00 | 0011 |
| Word 8 | 01 000 | 000 | 01 | 0010 |
| Word 9 | 01 001 | 001 | 01 | 0010 |
| ... | ... | ... | ... | ... |
| Word 15 | 01 111 | 111 | 01 | 0010 |
| Word 16 | 10 000 | 000 | 10 | 0100 |
| Word 17 | 10 001 | 001 | 10 | 0100 |
| ... | ... | ... | ... | ... |
| Word 23 | 10 111 | 111 | 10 | 0100 |

Paging

| Program Line | Logical Address | Offset | Page Number | Frame Number | Physical Address |
|--------------|-----------------|--------|-------------|--------------|------------------|
| Word 0 | 00 000 | 000 | 00 | 0011 | 0011 000 |
| Word 1 | 00 001 | 001 | 00 | 0011 | 0011 001 |
| ... | ... | ... | ... | ... | ... |
| Word 7 | 00 111 | 111 | 00 | 0011 | 0011 111 |
| Word 8 | 01 000 | 000 | 01 | 0110 | 0110 000 |
| Word 9 | 01 001 | 001 | 01 | 0110 | 0110 001 |
| ... | ... | ... | ... | ... | ... |
| Word 15 | 01 111 | 111 | 01 | 0110 | 0110 111 |
| Word 16 | 10 000 | 000 | 10 | 0100 | 0100 000 |
| Word 17 | 10 001 | 001 | 10 | 0100 | 0100 001 |
| ... | ... | ... | ... | ... | ... |
| Word 23 | 10 111 | 111 | 10 | 0100 | 0100 111 |

Structure of a Page Table Entry



A typical page table entry.

- *Protection* bits tell what kinds of access are permitted (for example read/write/execute).
- *Modified* and *Referenced* bits keep track of page usage.
- *Modified* bit is sometimes called the **dirty bit**.
- *Referenced* bit is set whenever a page is referenced, either for reading or for writing.

Virtual Memory Implementation

- A process is an abstraction of the physical processor (CPU).
- Virtual memory (address space) is an abstraction of physical memory
 - Virtual memory can be implemented by breaking the virtual address space up into pages, and
 - mapping each one onto some page frame of physical memory or having it (temporarily) unmapped.

Speeding Up Paging

Major issues faced:


1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

32 bit address space -> 4 GB

4K page size -> 1 M pages ~

64 bit address space -> $\sim 1,8 \times 10^{19}$

4K page size -> $\sim 4,5 \times 10^{15}$ pages



each process needs its own page table

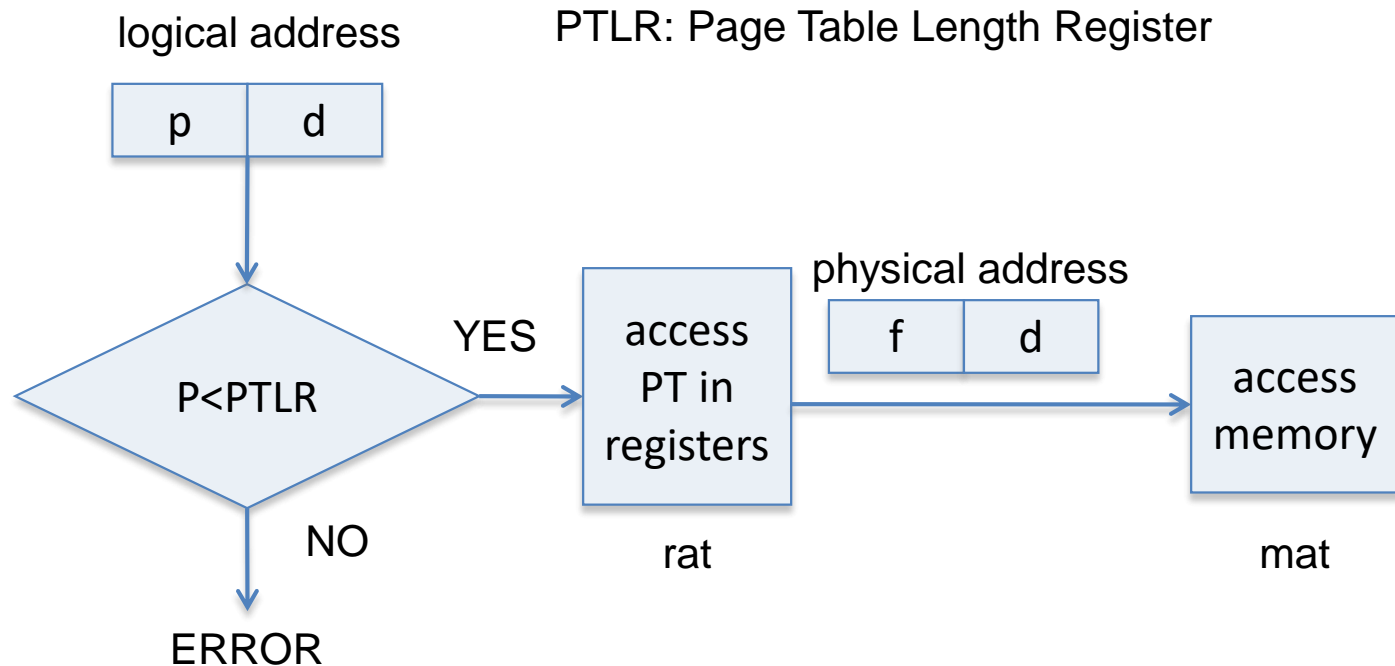
Paging

- Every access to memory should go through the page table.
- Therefore, it must be implemented in an efficient way.
- How to implement the page table?
 - Using fast dedicated registers
 - Keep the page table in main memory
 - Use content-addressable associative registers

Using fast dedicated registers

- A single page table consisting of
 - an array of fast hardware registers,
 - one entry for each virtual page,
 - indexed by virtual page number
- When a process is started up, OS loads the registers with the process' page table from main memory.
- Only the OS is able to modify these registers.
- Advantages
 - requires no memory references during mapping.
- Disadvantage
 - unbearably expensive if the page table is large
 - loading full page table at every context

Using fast dedicated registers

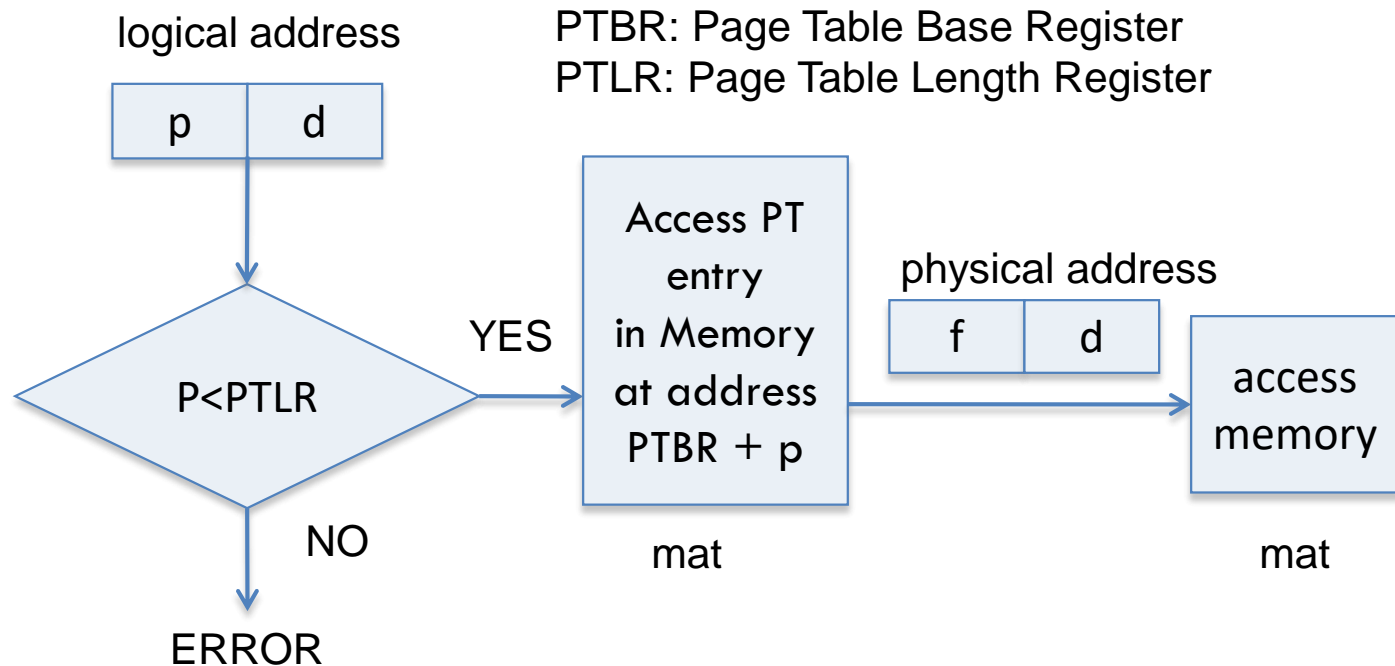


Effective Memory Access Time
 $emat = rat + mat$

Keep the page table in main memory

- In this second method, the OS keeps a page table in the memory, instead of registers.
- For every logical memory reference, two memory accesses are required:
 1. To access the page table in the memory, in order to find the corresponding frame number.
 2. To access the memory word in that frame
- This is cheap but a time consuming method.

Keep the page table in main memory



Effective Memory Access Time:

$$emat = mat + mat = 2mat$$

Translation Lookaside Buffers (TLB)

Associative memory (TLB or content addressable memory):

- a special h/w that can do search in parallel for a content and provides a result of MATCH (HIT) or NO MATCH (MISS).

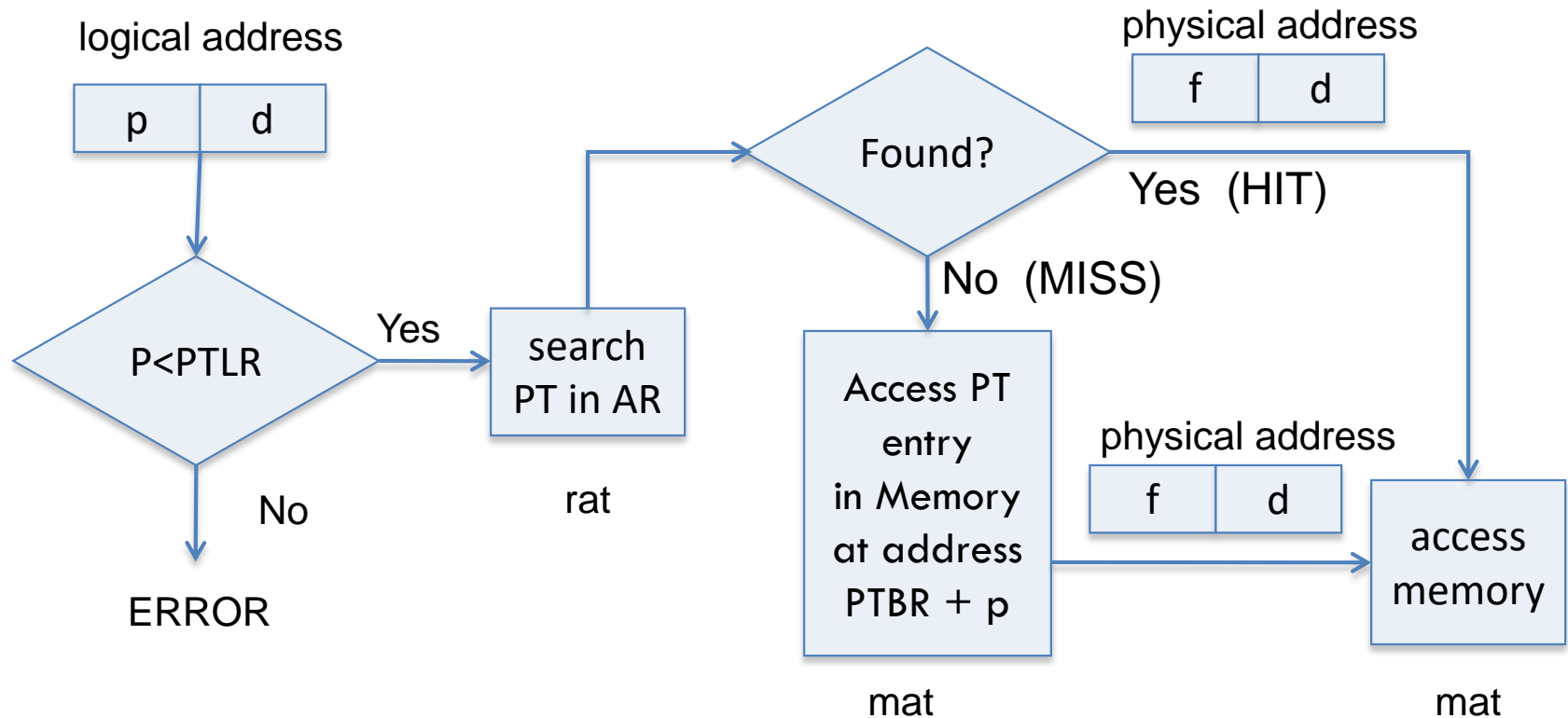
| Valid | Virtual page | Modified | Protection | Page frame |
|-------|--------------|----------|------------|------------|
| 1 | 140 | 1 | RW | 31 |
| 1 | 20 | 0 | R X | 38 |
| 1 | 130 | 1 | RW | 29 |
| 1 | 129 | 1 | RW | 62 |
| 1 | 19 | 0 | R X | 50 |
| 1 | 21 | 0 | R X | 45 |
| 1 | 860 | 1 | RW | 14 |
| 1 | 861 | 1 | RW | 75 |

Contents of an example TLB (associative memory) used to speed up paging.

Use content-addressable associative registers

- This is a mixing of first two methods.
- Associative registers are small, high speed registers built in a special way so that they permit an associative search over their contents.
- That is, all registers may be searched in one machine cycle simultaneously. However, associative registers are quite expensive. So, a small number of them should be used.
- Overall, the method is not-expensive and not-slow.

Use content-addressable associative registers



Effective Memory Access Time:

h: hit ratio

$$E_{mat} = h * e_{mat}_{HIT} + (1-h) * e_{mat}_{MISS} = h(rat+mat) + (1-h)(rat+mat+mat)$$

Use content-addressable associative registers

- Assume we have a paging system that uses associative registers.
- These associative registers have an access time of 30 ns, and the memory access time is 470 ns.
- The system has a hit ratio of 90 %.

- $t_{ar} = 30 \text{ ns}$
- $t_{mat} = 470 \text{ ns}$
- $h = 0.9$

Use content-addressable associative registers

rat=30 ns, mat=470ns, h=0.9

- Now, if the page number is found in one of the associative registers, then the effective access time:
- $\text{emat}_{\text{HIT}} = 30 + 470 = 500 \text{ ns.}$
- Because one access to associative registers and one access to the main memory is sufficient.

Use content-addressable associative registers

rat=30 ns, mat=470ns, h=0.9

- On the other hand, if the page number is not found in associative registers, then the effective access time:
- $\text{emat}_{\text{MISS}} = 30 + (470+470) = 970 \text{ ns.}$
- Since one access to associative registers and two accesses to the main memory are required.

Use content-addressable associative registers

rat=30 ns, mat=470ns, h=0.9

emat_{HIT} = 500 ns, emat_{MISS} = 970 ns.

- Then, the emat is calculated as follows:

$$\begin{aligned}\text{emat} &= h * \text{emat}_{\text{HIT}} + (1-h) * \text{emat}_{\text{MISS}} \\ &= 0.9 * 500 + 0.1 * 970 \\ &= 450 + 97 = 547 \text{ ns}\end{aligned}$$

Multilevel Page Tables

- Avoid keeping all the page tables in memory.
- Consider the case where virtual address space is 32 bits.
Assume pages are 4 KB.

This implies 4M pages per process

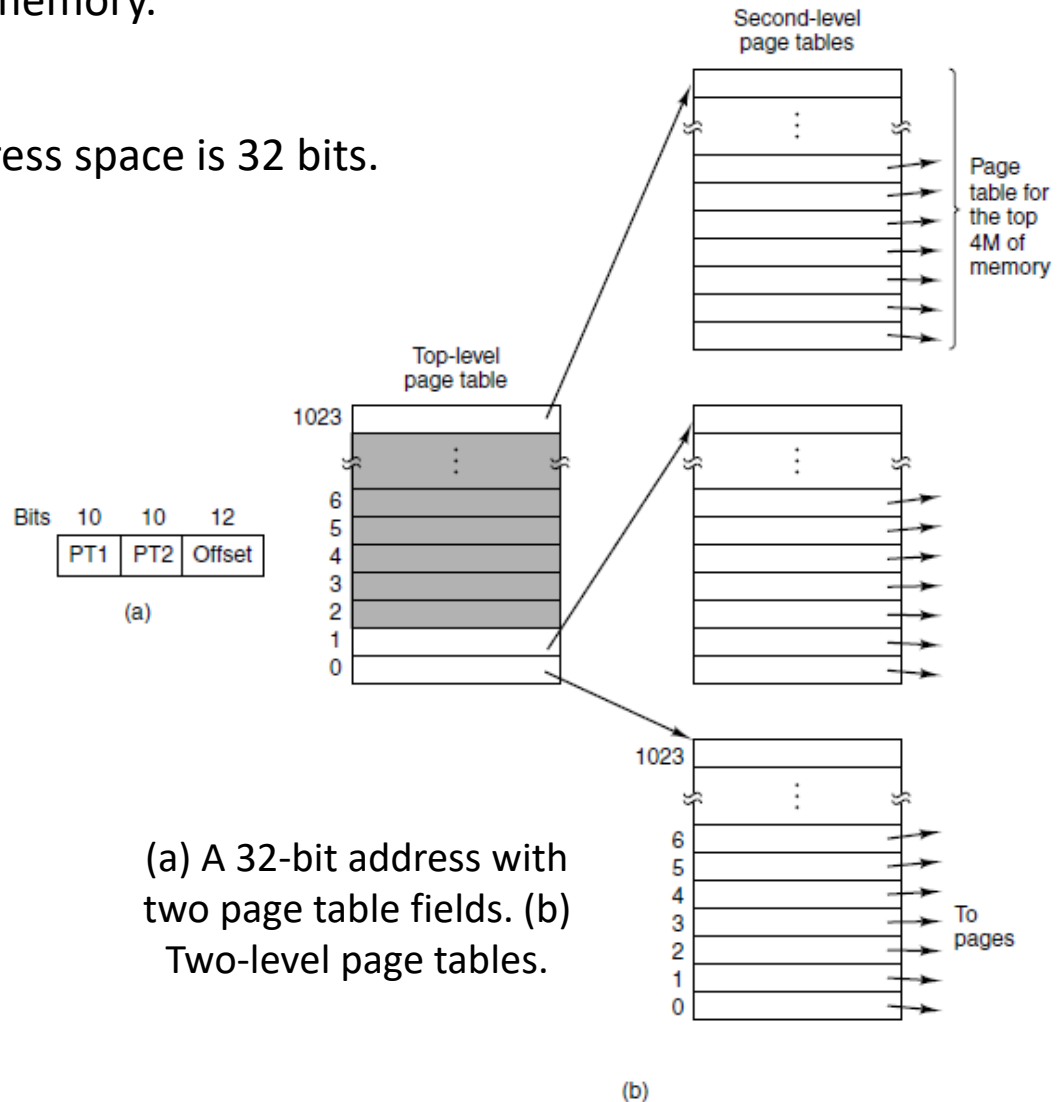
- Using a 2-level page table approach, we need to keep only four tables of size 1024 instead.

- Ex:

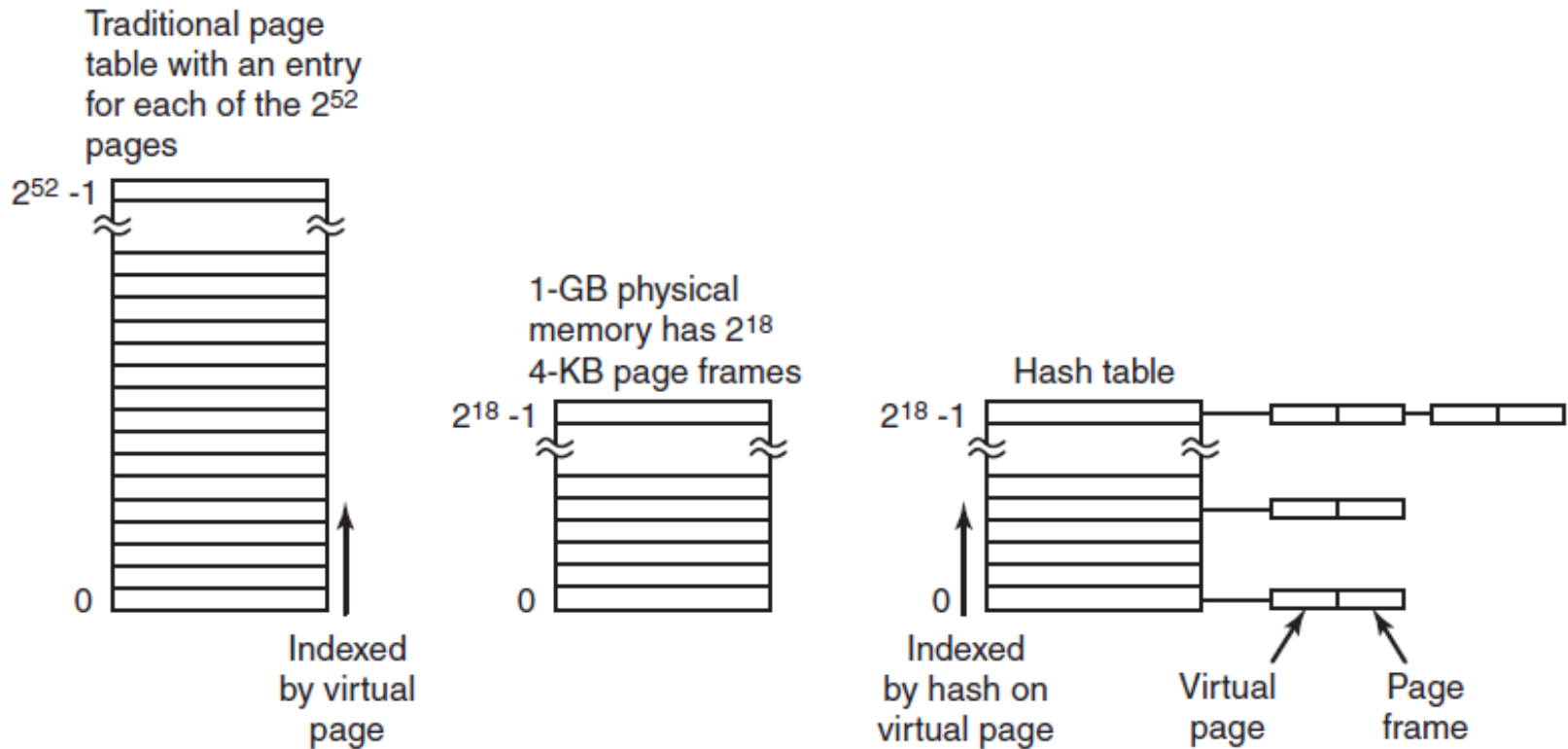
Virtual address=0x00403004

PT1=1, PT2=3, offset=4

- Unused pages are marked in the first level and generates an error if referenced.



Inverted Page Tables



Comparison of a traditional page table
with an inverted page table.

Sharing Pages

- Sharing pages is possible in a paging system, and is an important advantage of paging.
- It is possible to share system procedures or programs, user procedures or programs, and possibly data area.
- Sharing pages is especially advantageous in time-sharing systems.

Sharing Pages

- A re-entrant program (non-self-modifying code = read only) never changes during execution.
- So, more than one process can execute the same code at the same time.
- Each process will have its own data storage and its own copy of registers to hold the data for its own execution of the shared program.

Sharing Pages

Example

- Consider a system having page size=30 MB.
- There are 3 users executing an editor program which is 90 MB (3 pages) in size, with a 30 MB (1 page) data space.
- To support these 3 users, the OS must allocate $3 * (90+30) = 360$ MB space.
- However, if the editor program is reentrant (non-self-modifying code = read only), then it can be shared among the users, and only one copy of the editor program is sufficient.
- Therefore, only $90 + 30 * 3 = 180$ MB of memory space is enough for this case

| | User-1 |
|----|--------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data1 |

| PT-1 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 7 |

| | Physical Memory |
|-----|--------------------|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | e2 |
| f5 | e3 |
| f6 | |
| f7 | data1 |
| f8 | e1 |
| f9 | |
| f10 | |
| f11 | |
| f12 | |
| f13 | |
| f14 | |
| f15 | |

| User-1 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data1 |

| PT-1 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 7 |

| Physical Memory | |
|-----------------|--------|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | e2 |
| f5 | e3 |
| f6 | |
| f7 | data1 |
| f8 | e1 |
| f9 | |
| f10 | |
| f11 | |
| f12 | data 2 |
| f13 | |
| f14 | |
| f15 | |

| User-2 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data2 |

| PT-2 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 12 |

| User-1 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data1 |

| PT-1 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 7 |

| Physical Memory | |
|-----------------|--------|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | e2 |
| f5 | e3 |
| f6 | |
| f7 | data1 |
| f8 | e1 |
| f9 | |
| f10 | data3 |
| f11 | |
| f12 | data 2 |
| f13 | |
| f14 | |
| f15 | |

| User-2 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data2 |

| PT-2 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 12 |

| User-3 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data3 |

| PT-3 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 10 |

| User-1 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data1 |

| PT-1 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 7 |

| Physical Memory | |
|-----------------|-------|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | e2 |
| f5 | e3 |
| f6 | |
| f7 | data1 |
| f8 | e1 |
| f9 | |
| f10 | data3 |
| f11 | |
| f12 | |
| f13 | |
| f14 | |
| f15 | |

User 2 terminates:
Data2 page is removed
from memory, but
editor pages remain..

| User-3 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data3 |

| PT-3 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 10 |

User 1 terminates:
data1 is also removed
from memory.

| User-3 | |
|--------|-------|
| P0 | e1 |
| P1 | e2 |
| P2 | e3 |
| P3 | data3 |

| PT-3 | |
|-------|--------|
| Page# | Frame# |
| 0 | 8 |
| 1 | 4 |
| 2 | 5 |
| 3 | 10 |

| Physical Memory | |
|--------------------|-------|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | e2 |
| f5 | e3 |
| f6 | |
| f7 | |
| f8 | e1 |
| f9 | |
| f10 | data3 |
| f11 | |
| f12 | |
| f13 | |
| f14 | |
| f15 | |

When User 3
terminates:
Data-3 and also editor
segments are removed
from memory.

| Physical Memory | |
|--------------------|----|
| f0 | OS |
| f1 | OS |
| f2 | OS |
| f3 | |
| f4 | |
| f5 | |
| f6 | |
| f7 | |
| f8 | |
| f9 | |
| f10 | |
| f11 | |
| f12 | |
| f13 | |
| f14 | |
| f15 | |

Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm
- Working set algorithm
- WSClock algorithm

Page Replacement Algorithms

- A page replacement algorithm determines how *victim page* (page to be replaced) is selected when a page fault occurs.
 - The aim is to minimize the page fault rate.
- Efficiency is evaluated by running the algorithm on a particular string of memory references and computing the number of page faults.
- Reference strings are either generated randomly, or by tracing the paging behavior of a system and recording the page number for each logical memory reference.

Page Replacement Algorithms

- Consecutive references to the same page may be reduced to a single reference, because they won't cause any page fault except possibly the first one:
 $(1,4,1,6,1,1,1,3) \rightarrow (1,4,1,6,1,3)$.
- We have to know the number of page frames available in order to be able to decide on the page to be replaced.
- Optionally, a frame allocation policy may be followed.

Optimal Page Replacement Algorithm (OPT)

- In this algorithm,
 - the victim is the page, which will not be used for the longest period.
- For a fixed number of frames, OPT has the lowest page fault rate among all page replacement algorithms.
- Problem: OPT is not possible to be implemented in practice since it requires future knowledge.
- It is used for performance comparison.

Optimal Page Replacement Algorithm (OPT)

Example

- Assume we have 3 frames
- Consider the reference string below.

5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

- Show the content of memory after each memory reference if OPT page replacement algorithm is used.
- Find also the number of page faults

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

[illegible]

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

[illegible]

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

[illegible]

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | | | | | | | | |
| f3 | | | 6 | 6 | 6 | | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | | | | | | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | | | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | | | | | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | | | | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | | | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|------|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|------|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | 7 | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | 7 | | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|------|---|------|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | 7 | 7 | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | 7 | same | |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|------|---|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | 7 | 7 | 7 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | 7 | same | same |

Optimal Page Replacement Algorithm (OPT)

The victim is the page, which will not be used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|------|------|---|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 | 7 | 7 | 7 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | same | same | 7 | same | same |

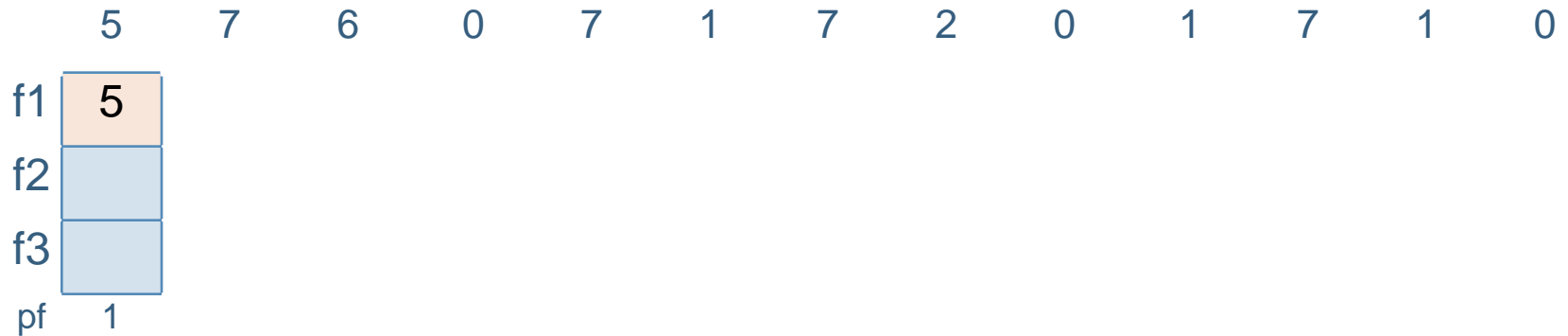
According to the given information, this algorithm generates a page replacement scheme with 7 page faults.

Not Recently Used Algorithm

- At page fault, system inspects pages
- Categories of pages based on the current values of their R and M bits:
 - Class 0: not referenced, not modified.
 - Class 1: not referenced, modified.
 - Class 2: referenced, not modified.
 - Class 3: referenced, modified.
- The NRU (Not Recently Used) algorithm removes a page at random from the lowest-numbered nonempty class.

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.



First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

[illegible]

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | | | | | | | | |
| f3 | | | 6 | 6 | 6 | | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | | | | | | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | | | | | | | |
| f3 | | | 6 | 6 | 6 | 6 | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | | | | | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | | | | | | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | | | | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | | | | | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | | | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | | | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | | | | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | | | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|----|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | 0 | | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | 1 | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | 10 | | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|----|------|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | 1 | 1 | |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | 10 | same | |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|----|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | 10 | same | same |

First-In-First-Out (FIFO)

This is a simple algorithm, and easy to implement. The idea is straight forward: choose the oldest page as the victim.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|---|---|---|---|----|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| f3 | | | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| pf | 1 | 2 | 3 | 4 | same | 5 | 6 | 7 | 8 | 9 | 10 | same | same |

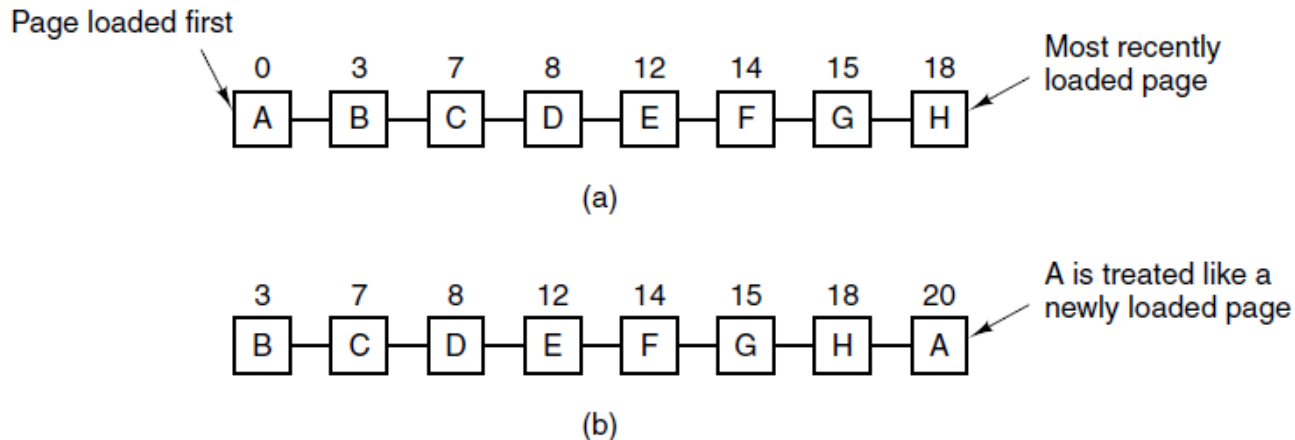
10 page faults.

Belady's Anomaly

- Normally, one would expect that as the total number of frames increases, the number of page faults decreases.
- However, for FIFO, there are cases where this generalization fails. This is called Belady's Anomaly.
- As an exercise consider the reference string below.
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Apply the FIFO method and find the number of page faults considering different number of frames.
- Then, examine whether the replacement suffers Belady's anomaly.

Second-Chance Algorithm

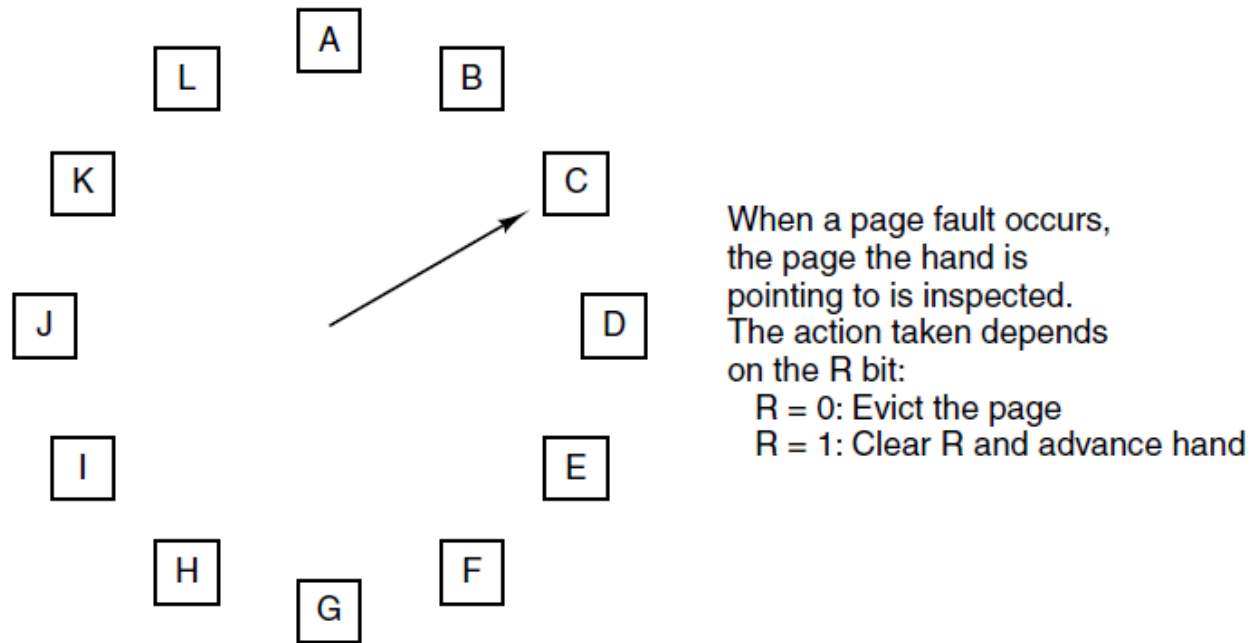
If $R=0$, the page is both old and unused, so it is replaced immediately.
If $R=1$, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory.
Then the search continues.



Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.

Clock Page Replacement Algorithm

A better (more efficient) implementation of the second chance algorithm



The clock page replacement algorithm.

Least Recently Used (LRU)

- In this algorithm,
 - victim is the page that has not been used for the longest period.
- OS using this method, has to associate with each page, the time it was last used, which means extra storage.
- In the simplest way, OS sets reference bit of a page to "1" when it is referenced.
- This bit will not give the order of use but it will simply tell whether the corresponding frame is referenced recently or not.
- The OS resets all reference bits periodically.

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

[illegible]

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

[illegible]

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

[illegible]

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

[illegible]

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | | | | | | | | |
| f3 | | | 6 | 6 | 6 | | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | | | | | | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|---|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | | | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | | | | | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | | | | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | | | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|---|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | 1 | | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | 9 | | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | | | | | | | | | | | | | |
|----|---|---|---|---|------|---|------|---|---|---|---|------|---|
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | 1 | 1 | |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | 9 | same | |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|---|---|---|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | 9 | same | same |

Least Recently Used (LRU)

victim is the page that has not been used for the longest period.

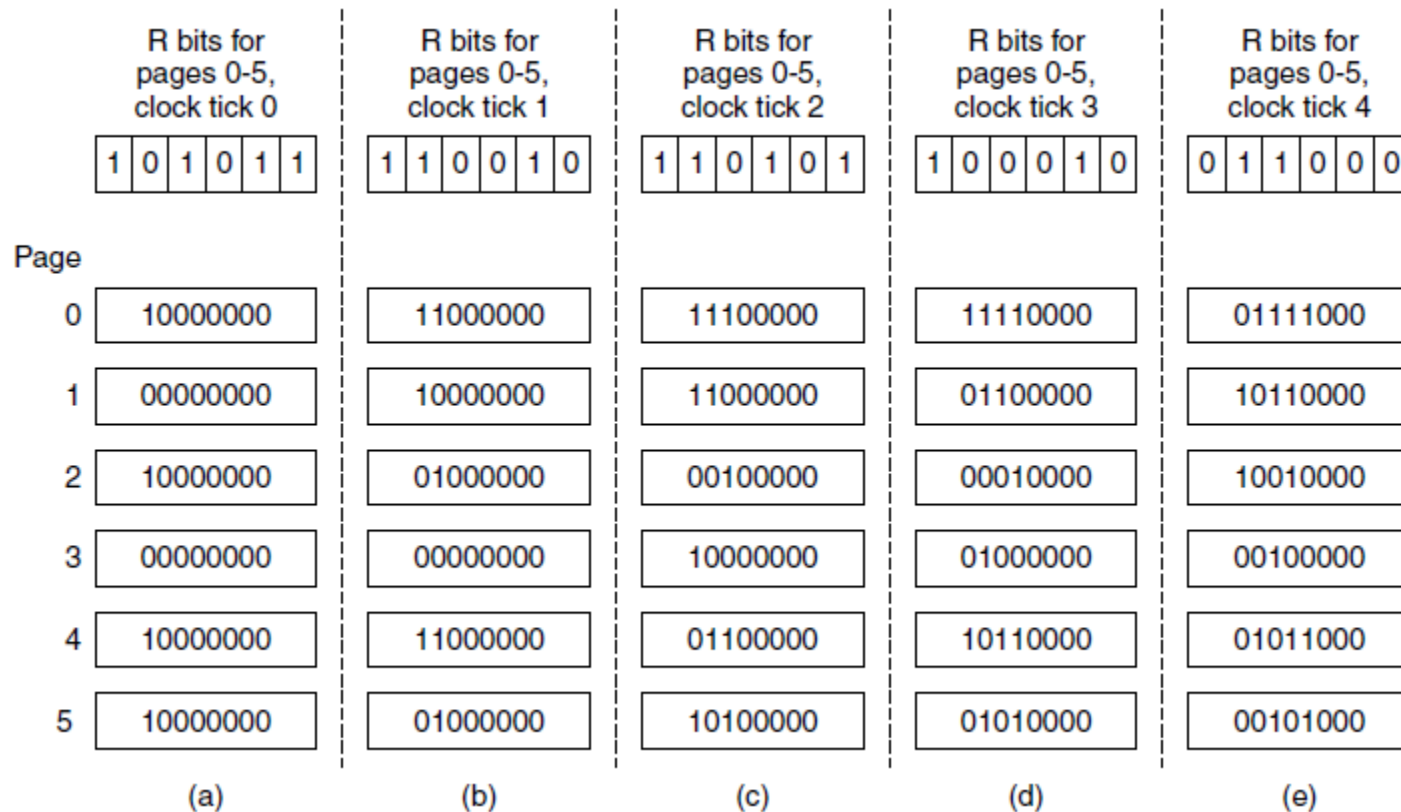
| | 5 | 7 | 6 | 0 | 7 | 1 | 7 | 2 | 0 | 1 | 7 | 1 | 0 |
|----|---|---|---|---|------|---|------|---|---|---|---|------|------|
| f1 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 7 | 7 | 7 |
| f2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 1 | 1 | 1 | 1 |
| f3 | | | 6 | 6 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| pf | 1 | 2 | 3 | 4 | same | 5 | same | 6 | 7 | 8 | 9 | same | same |

9 page faults.

LRU by Hardware

- Although LRU is theoretically realizable, it is costly to implement.
- Need to maintain a linked list of all pages in memory
 - The most recently used page at the front and the least recently used page at the rear.
- Difficulty: the list must be updated on every memory reference.
 - time consuming even if this structure is implemented in hardware
- There are other ways to implement LRU with special hardware.
 - equipping the hardware with a 64-bit counter, C , that is automatically incremented after each instruction
 - each page table has a field large enough to contain value of C
 - after each memory reference, the current value of C is stored in the page table entry for the page just referenced.
 - When a page fault occurs, OS examines all the counters in the page tables to find the lowest one.
 - It is the least recently used.

Simulating LRU in Software



The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Frame Allocation

- In page replacement, the following policies may be followed:
 - Global Replacement:
 - A process can replace any page in the memory.
 - Local Replacement:
 - Each process can replace only from its own reserved set of allocated page frames.

Frame Allocation

- In case of local replacement, OS should determine how many frames should it allocate to each process.
- The number of frames for each process may be adjusted by using two ways:
 - Equal Allocation:
 - If there are n frames and p processes, n/p frames are allocated to each process.
 - Proportional Allocation:
 - Let virtual memory size for a process p be $v(p)$. Let m processes and n frames.
 - Total virtual memory size: $V = \sum v(p)$.
 - Allocate $(v(p) / V) * n$ frames to process p .

Frame Allocation

Example

- Consider a system having 64 frames. Assume 4 processes with the following virtual memory sizes:
 $v(1) = 16$, $v(2) = 128$, $v(3) = 64$ and $v(4) = 48$.
- Equal Allocation:
 - OS allocates $n/p = 64 / 4 = 16$ frames to each process.

Frame Allocation

$v(1) = 16$, $v(2) = 128$, $v(3) = 64$ and $v(4) = 48$.

- Proportional Allocation: $V = 16 + 128 + 64 + 48 = 256$.

OS allocates:

- $(16 / 256) * 64 = 4$ frames to process 1,
- $(128 / 256) * 64 = 32$ frames to process 2,
- $(64 / 256) * 64 = 16$ frames to process 3,
- $(48 / 256) * 64 = 12$ frames to process 4.

Thrashing

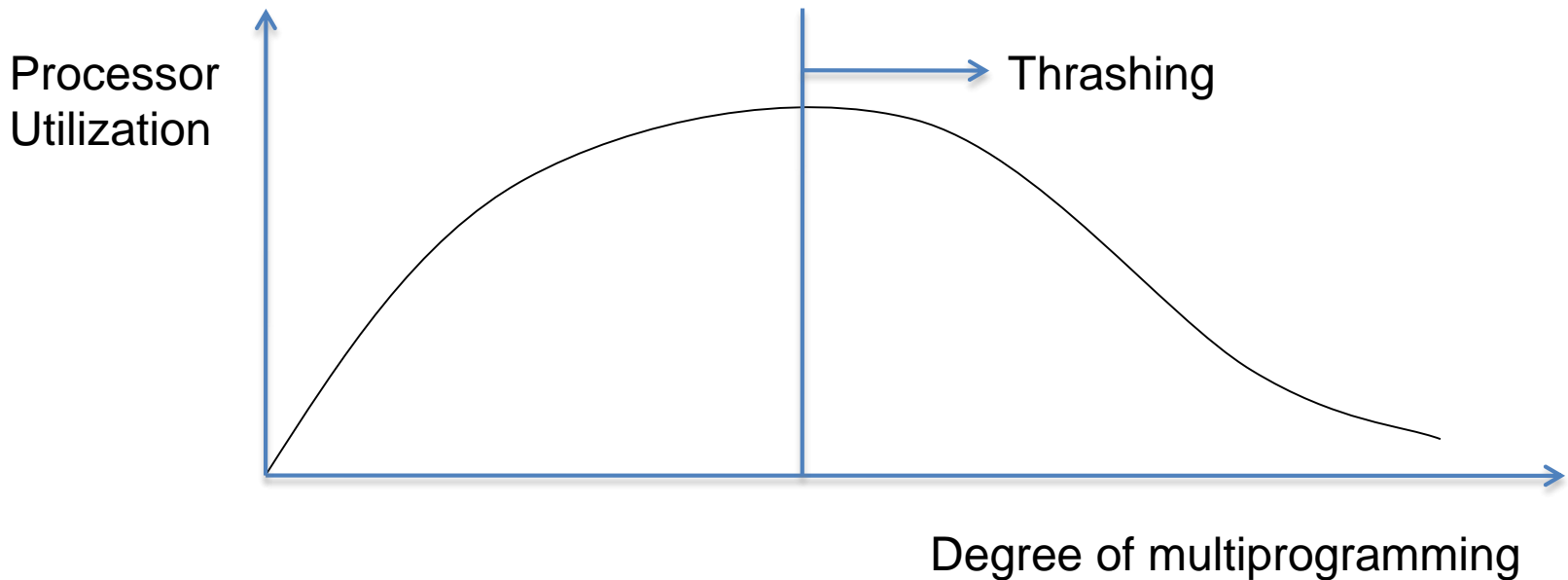
- A process is thrashing if it is spending more time for paging in/out (due to frequent page faults) than executing.
- Thrashing causes considerable degradation in system performance.
- If a process does not have enough number of frames allocated to it, it will issue a page fault.
- A victim page should then be chosen, but all pages may be in active use.
- So, victim page selection and new page replacement will be needed again in a very short time.
- This means that another page fault will be issued shortly, and so on and so forth.

Thrashing

- In case a process thrashes, the best thing to do is to suspend its execution and swap out all of its pages in the memory to the backing store.
- Local replacement algorithms can limit the effects of thrashing.

Thrashing

- If the degree of multiprogramming is increased over a limit, processor utilization falls down considerably because of thrashing.



Working Set Model

- To prevent thrashing, OS should provide a process as many frames as it needs.
- For this, working set model, which depends on the locality model of program execution, can be used.
- We shall use a parameter, Δ , called the ***working set window size*** and examine the last Δ page references.
- The set of pages in the last Δ references is to be called the ***working set of a process***.

Working Set Model

$\Delta=10$, calculate ws and wss at every 5 seconds

2 2 1 5 7 7 7 7 5 1

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1

$t=10$

$WS(t) = \{2, 1, 5, 7\}$ $WSS(t) = 4$

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3

$t=15$

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3

$t=15$

$WS(t) = \{7, 5, 1, 3, 4\}$ $WSS(t) = 5$

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4



Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4 3



Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4 3 4



Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4 3 4 4



Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4 3 4 4 4

t=20

Working Set Model

$\Delta=10$

2 2 1 5 7 7 7 7 5 1 3 4 4 4 3 4 3 4 4 4

$t=20$

$WS(t) = \{3,4\}$ $WSS(t) = 2$

Working Set Model

- Note that in calculating the working sets, we do not reduce consequent references to the same page to a single reference.
- Choice of Δ is crucial.
 - If Δ is too small, it will not cover the entire working set.
 - If it is too large, several localities of a process may overlap.
- It is suggested to be about 10.000 references.

Working Set Model

- Now, compute the WS size (WSS) for each process, and find the total demand, D of the system at that instance in time, as follows:

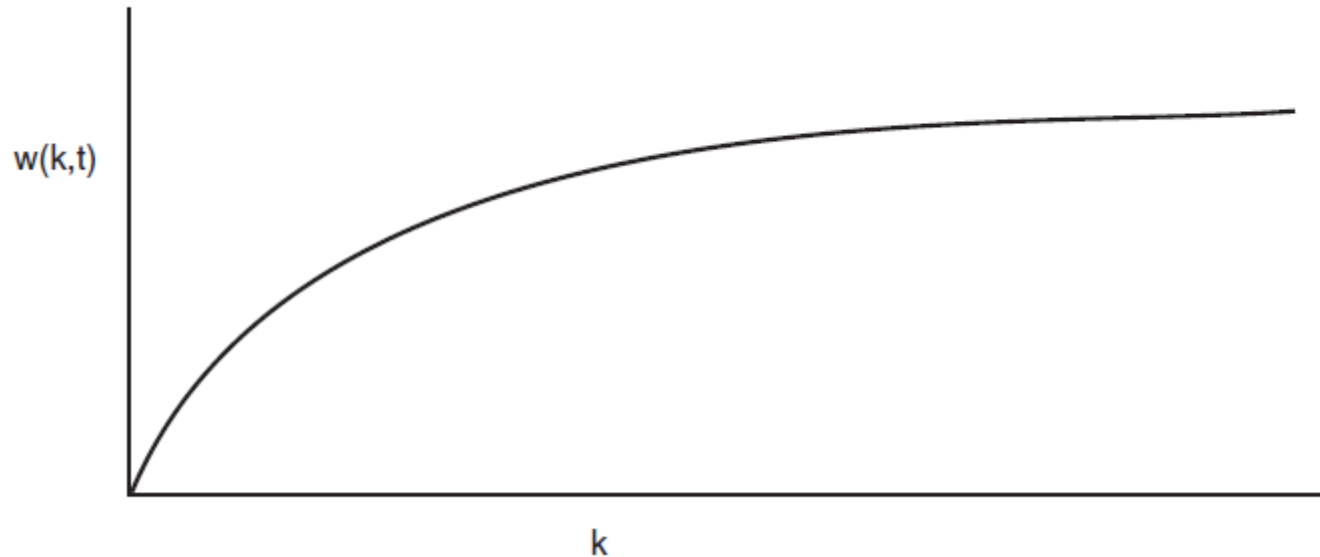
$$D(t_{now}) = \sum_{i=1}^p WSS_i(t_{now})$$

- If the number of frames is n , then
 - If $D > n$, the system is thrashing.
 - If $D < n$, the system is all right, hence degree of multi-programming can possibly be increased.

Working Set Model

- To be able to use the WS model for virtual memory management, OS keeps track of the WS of each process.
- It allocates each process enough frames to provide them with their WSs.
- If at one point $D > n$, OS selects a process to suspend.
- The frames that were used by the selected process are reallocated to other processes.
- PF frequency can also be used to decide on decreasing or increasing the no. of frames allocated to a process.

Working Set Algorithm (1)

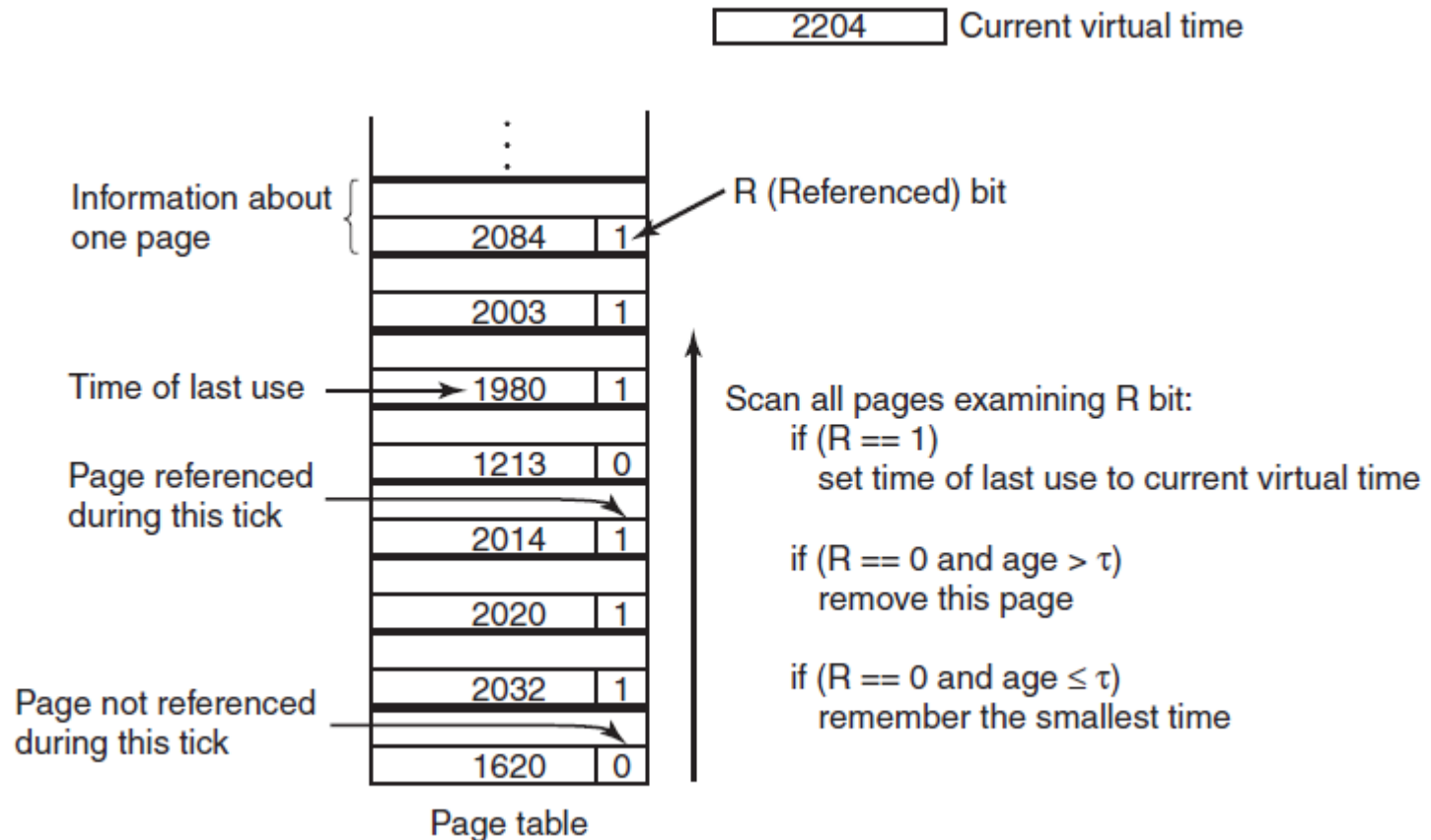


The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t .

Working Set Algorithm (1)

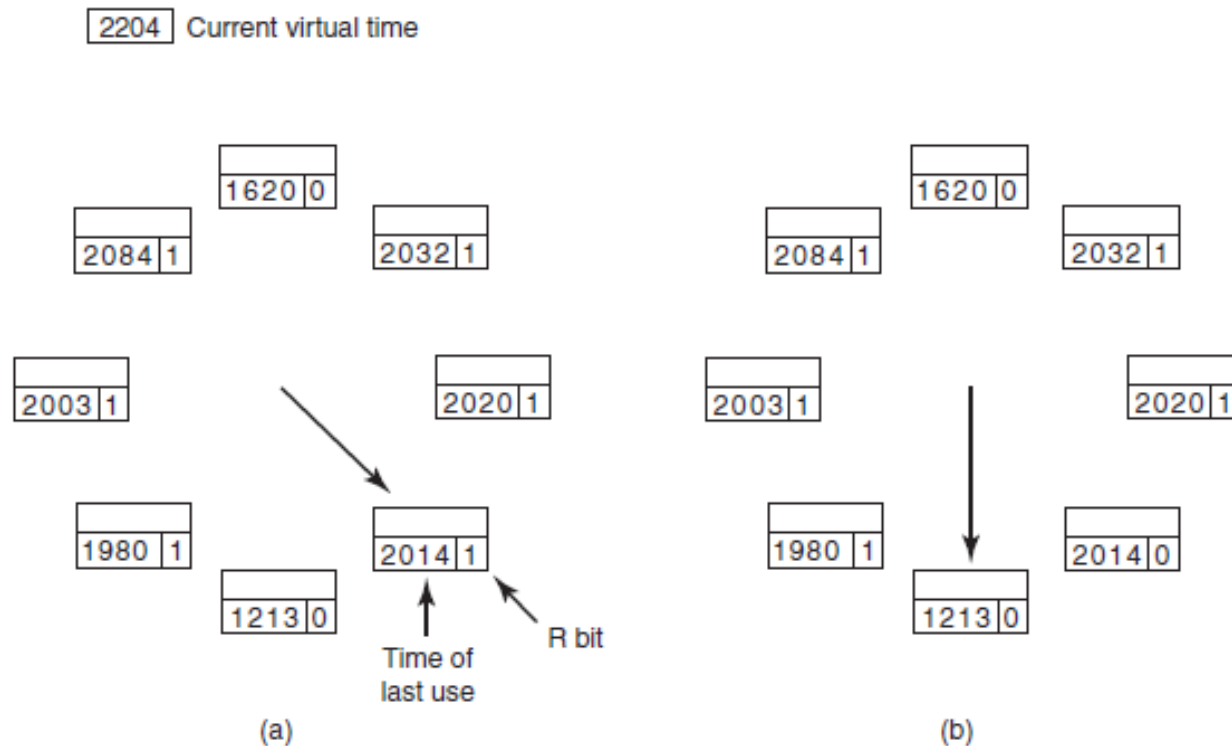
- To implement WS model, it is necessary for OS to keep track of which pages are in WS.
- Algorithm: when a PF occurs, find a page not in the WS and evict it.
- To implement such an algorithm, we need a precise way of determining which pages are in the working set.

Working Set Algorithm (2)



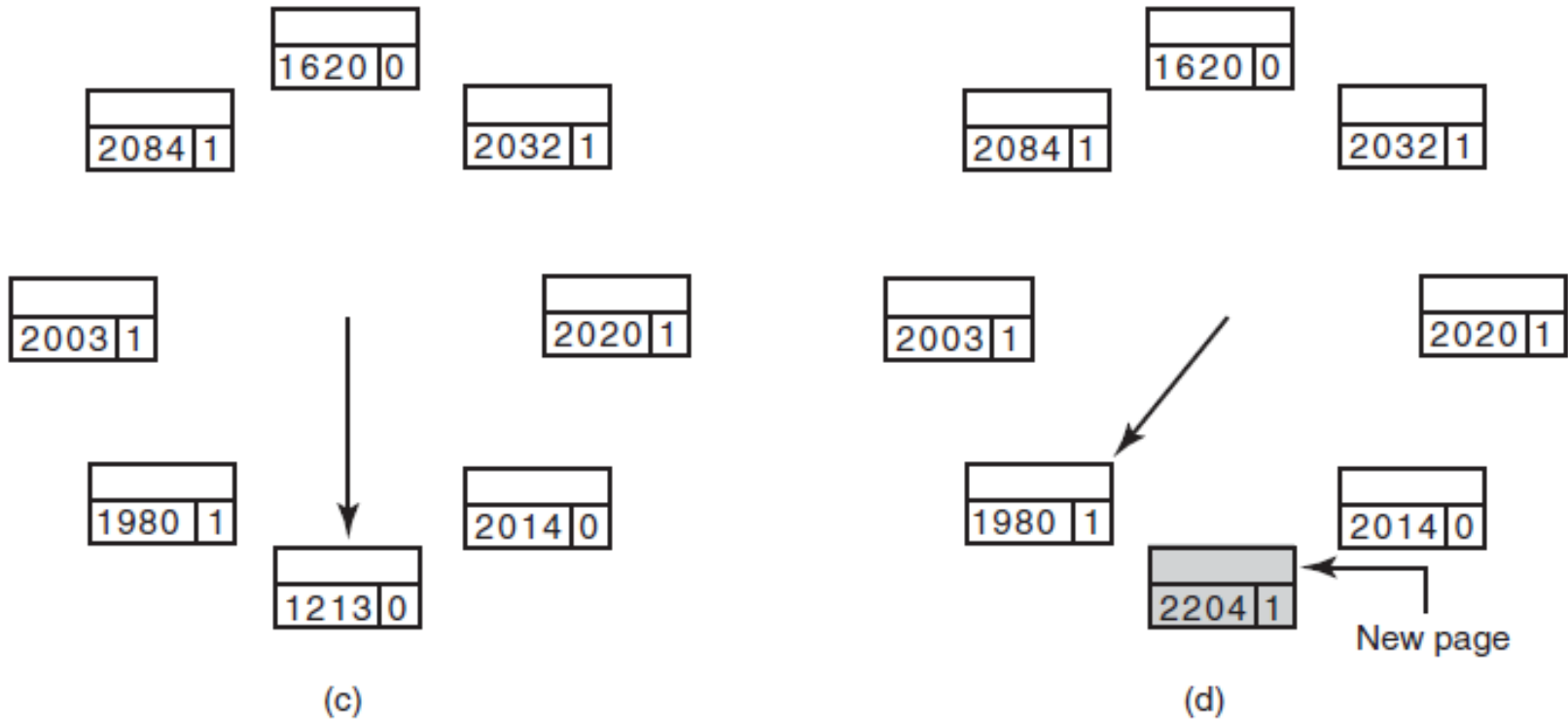
The working set algorithm.

WSClock Algorithm (1)



Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$.

WSClock Algorithm (2)



Operation of the WSClock algorithm.
(c) and (d) give an example of $R = 0$.

Summary of Page Replacement Algorithms

| Algorithm | Comment |
|----------------------------|--|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very crude approximation of LRU |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second, chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

Page replacement algorithms discussed in the text.

Design Issues: Local versus Global Allocation Policies (1)

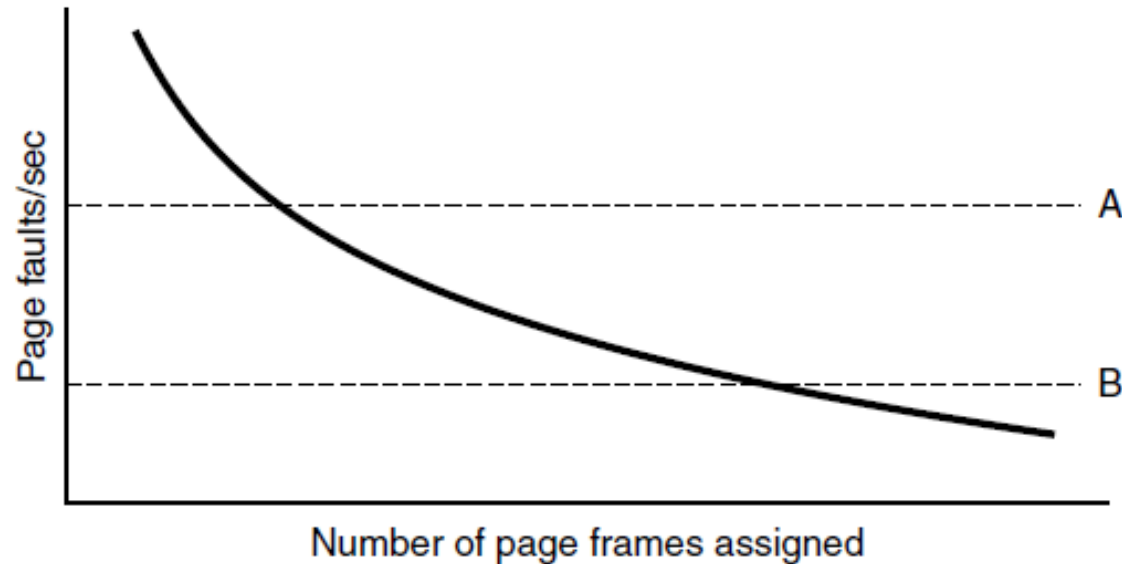
| Age | | | | | |
|-----|----|-----|--|-----|--|
| A0 | 10 | A0 | | A0 | |
| A1 | 7 | A1 | | A1 | |
| A2 | 5 | A2 | | A2 | |
| A3 | 4 | A3 | | A3 | |
| A4 | 6 | A4 | | A4 | |
| A5 | 3 | A6 | | A5 | |
| B0 | 9 | B0 | | B0 | |
| B1 | 4 | B1 | | B1 | |
| B2 | 6 | B2 | | B2 | |
| B3 | 2 | B3 | | A6 | |
| B4 | 5 | B4 | | B4 | |
| B5 | 6 | B5 | | B5 | |
| B6 | 12 | B6 | | B6 | |
| C1 | 3 | C1 | | C1 | |
| C2 | 5 | C2 | | C2 | |
| C3 | 6 | C3 | | C3 | |
| (a) | | (b) | | (c) | |

Local versus global page replacement.

(a) Original configuration. (b) Local page replacement.

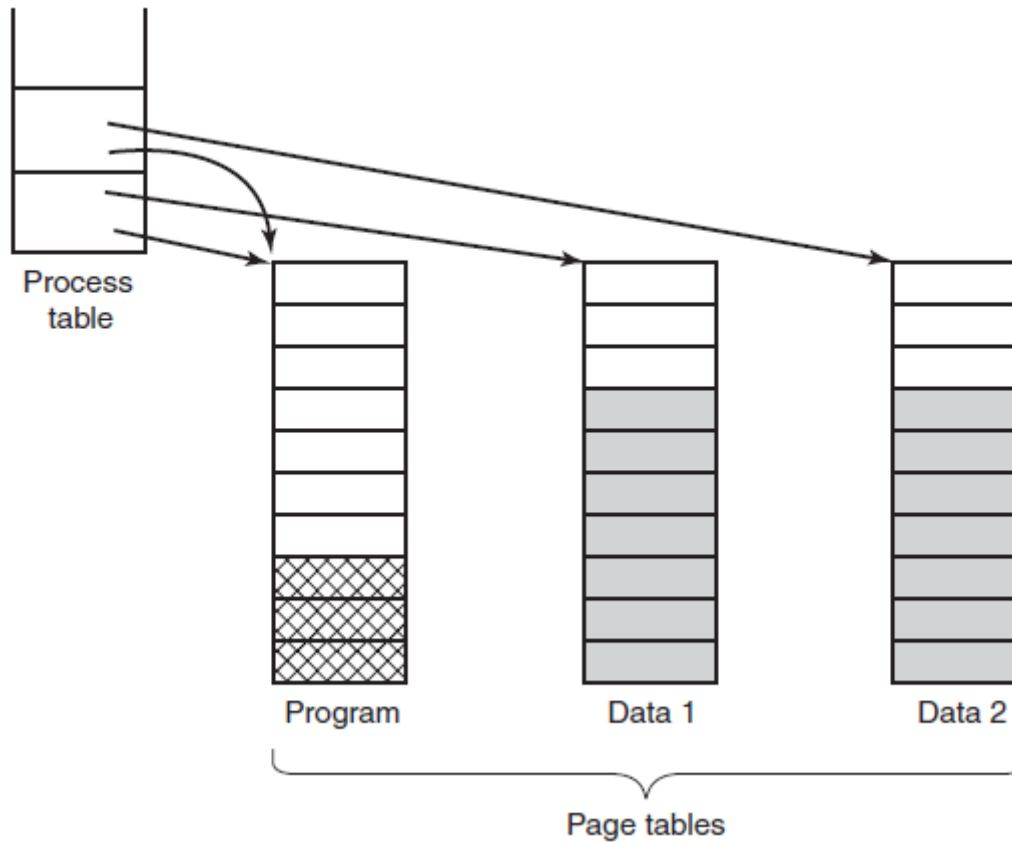
(c) Global page replacement.

Local versus Global Allocation Policies (2)



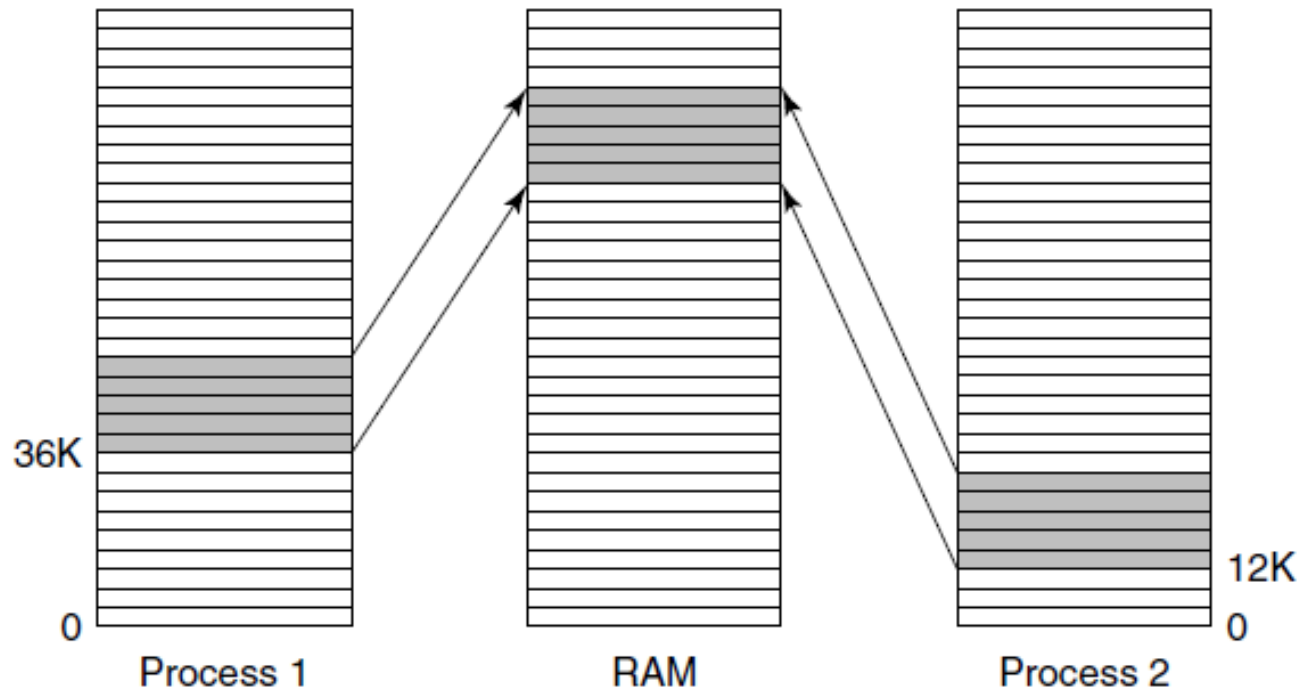
Page fault rate as a function of the number of page frames assigned.

Shared Pages



Two processes sharing the same
program sharing its page table.

Shared Libraries



A shared library being used by two processes.

Page Fault Handling (1)

1. The hardware traps to kernel, saving program counter on stack.
2. Assembly code routine started to save general registers and other volatile info
3. System discovers page fault has occurred, tries to discover which virtual page needed
4. Once virtual address caused fault is known, system checks to see if address valid and the protection consistent with access

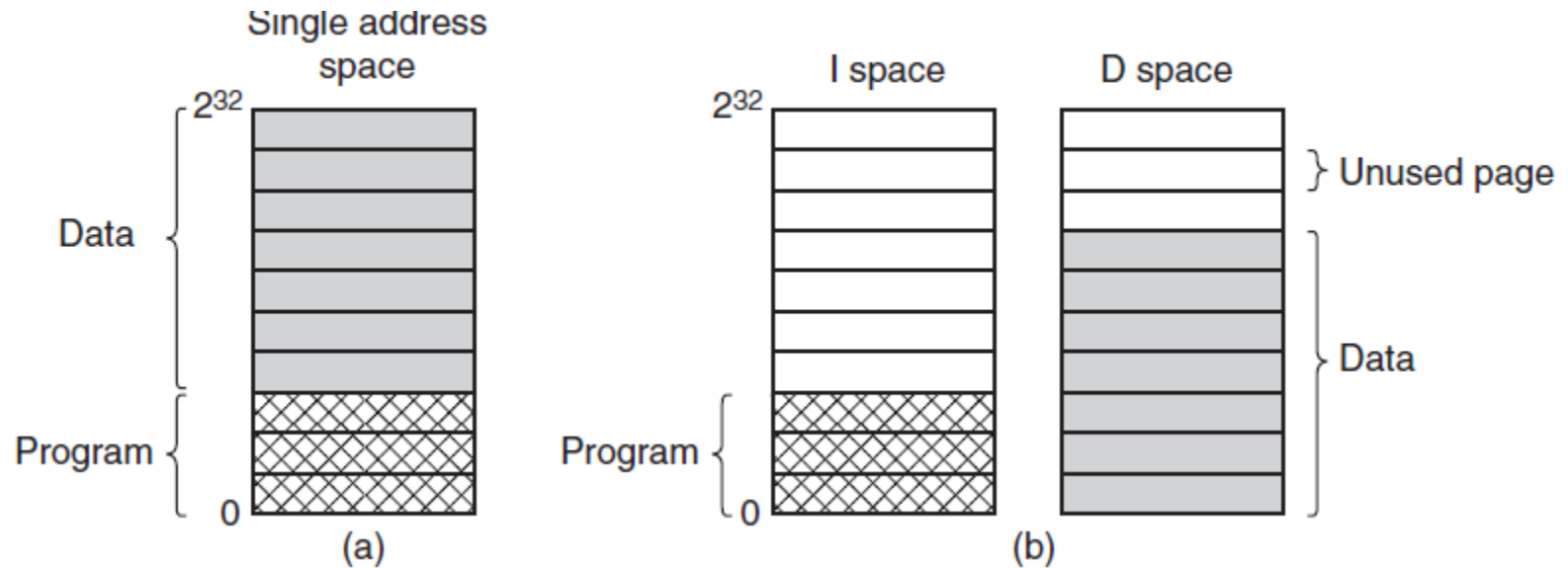
Page Fault Handling (2)

5. If frame selected dirty, page is scheduled for transfer to disk, context switch takes place, suspending faulting process
6. As soon as frame clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
7. When disk interrupt indicates page has arrived, tables updated to reflect position, and frame marked as being in normal state.

Page Fault Handling (3)

8. Faulting instruction backed up to state it had when it began and program counter is reset
9. Faulting process is scheduled, operating system returns to routine that called it.
10. Routine reloads registers and other state information, returns to user space to continue execution

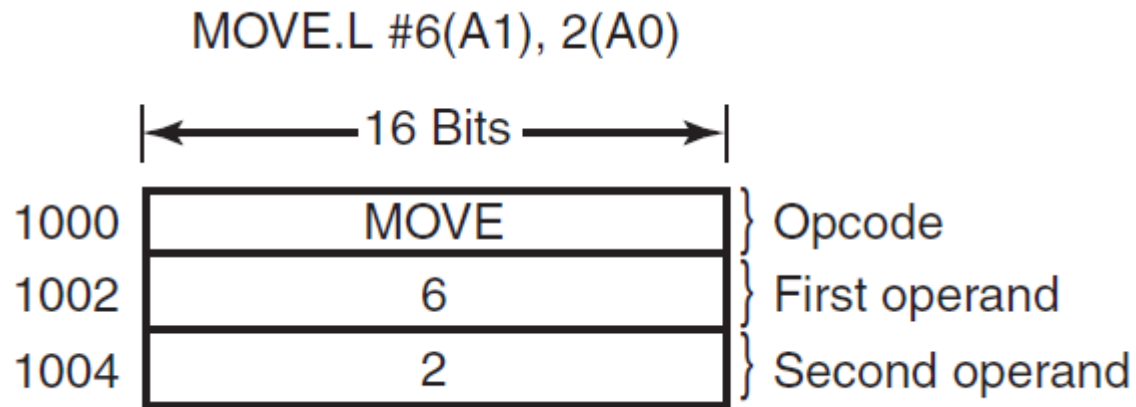
Separate Instruction and Data Spaces



(a) One address space.

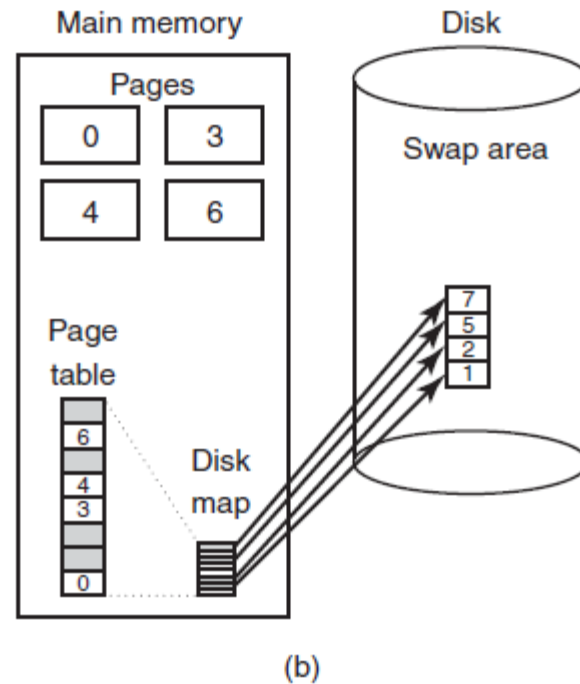
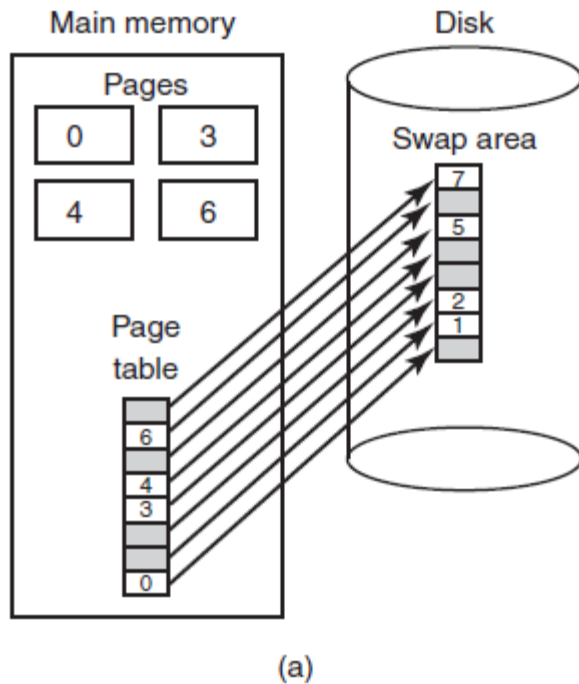
(b) Separate I and D spaces.

Instruction Backup



An instruction causing a page fault.

Backing Store



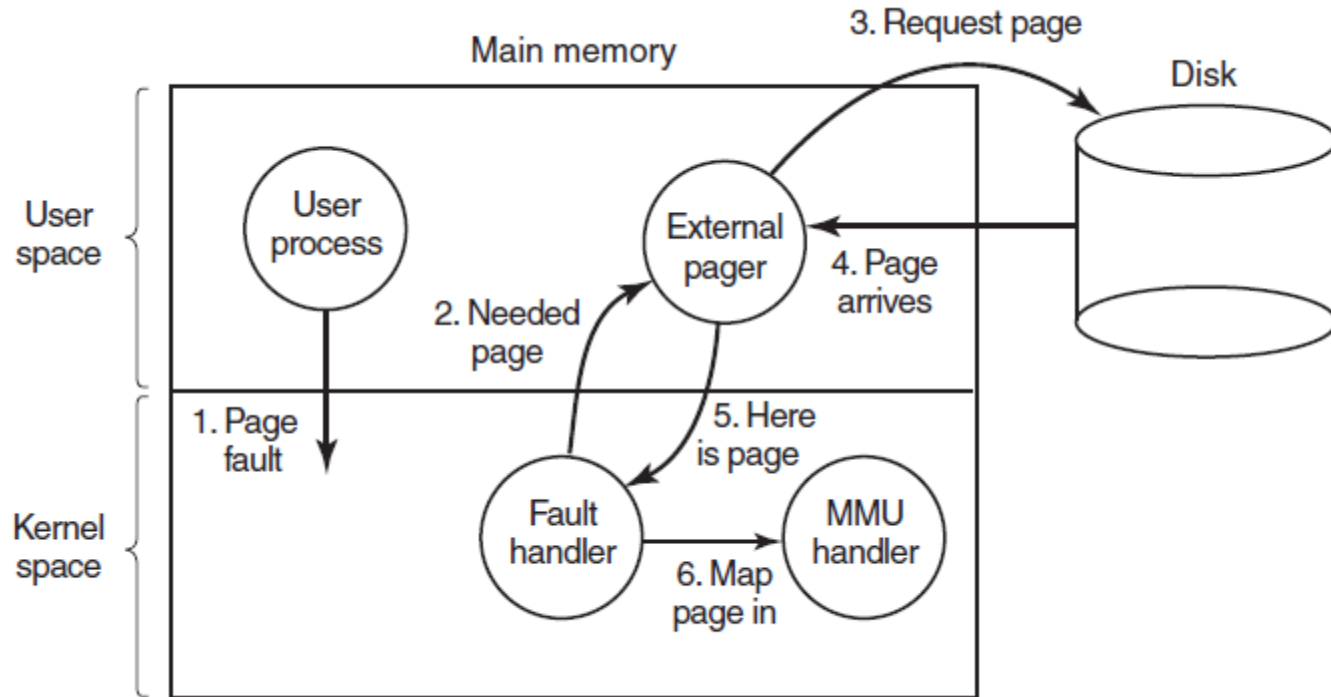
- (a) Paging to a static swap area.
(b) Backing up pages dynamically.

Separation of Policy and Mechanism (1)

Memory management system is divided into three parts

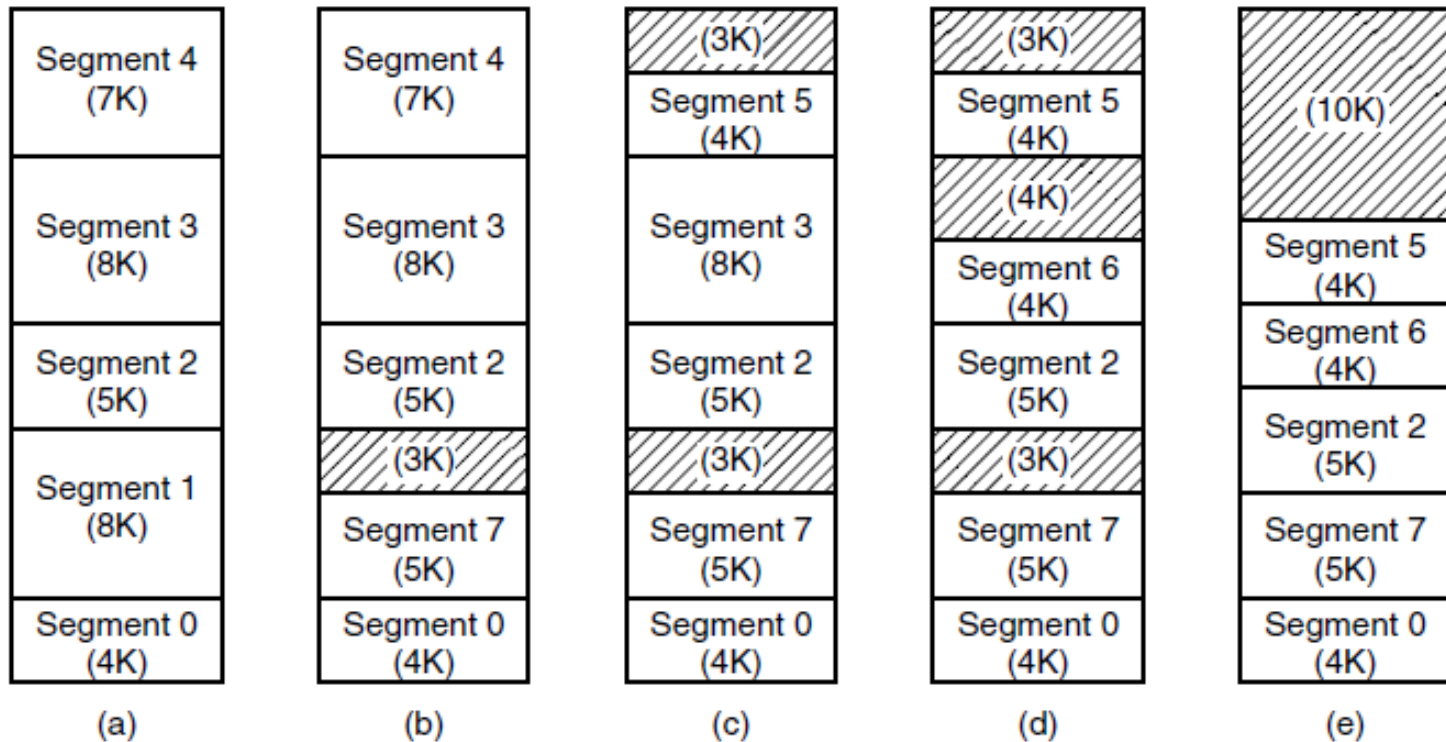
1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

Separation of Policy and Mechanism (2)



Page fault handling with an external pager.

Implementation of Pure Segmentation



(a)-(d) Development of checkerboarding.

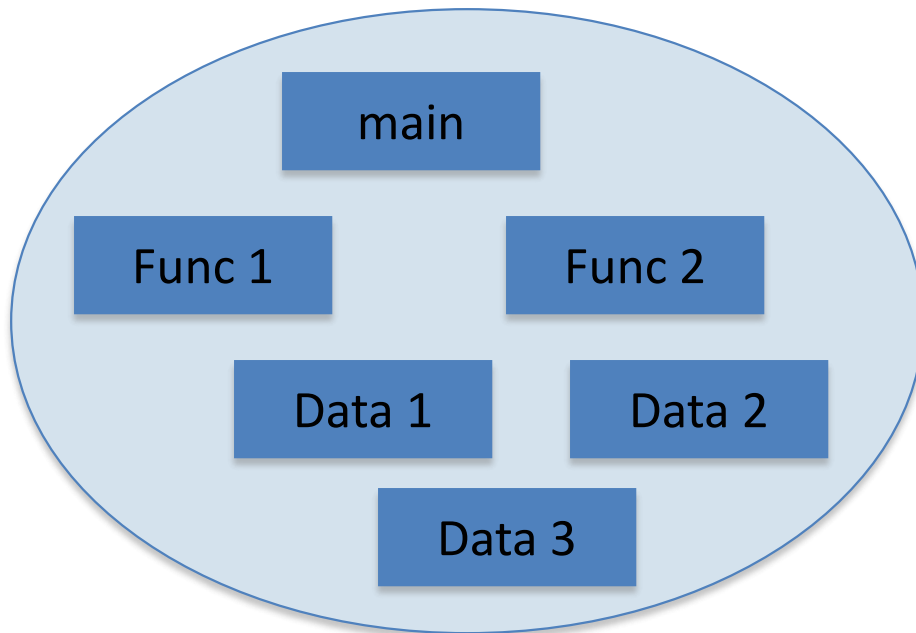
(e) Removal of the checkerboarding by compaction.

Segmentation

- In segmentation, programs are divided into variable size segments, instead of fixed size pages.
- This is similar to variable partitioning, but programs are divided into small parts.
- Every logical address is formed of a segment name and an offset within that segment.
- In practice, segments are numbered.
- Programs are segmented automatically by the compiler or assembler.

Segmentation

- For example, a C compiler may create segments for:
 - the code of each function
 - the local variables for each function
 - the global variables.



Logical memory

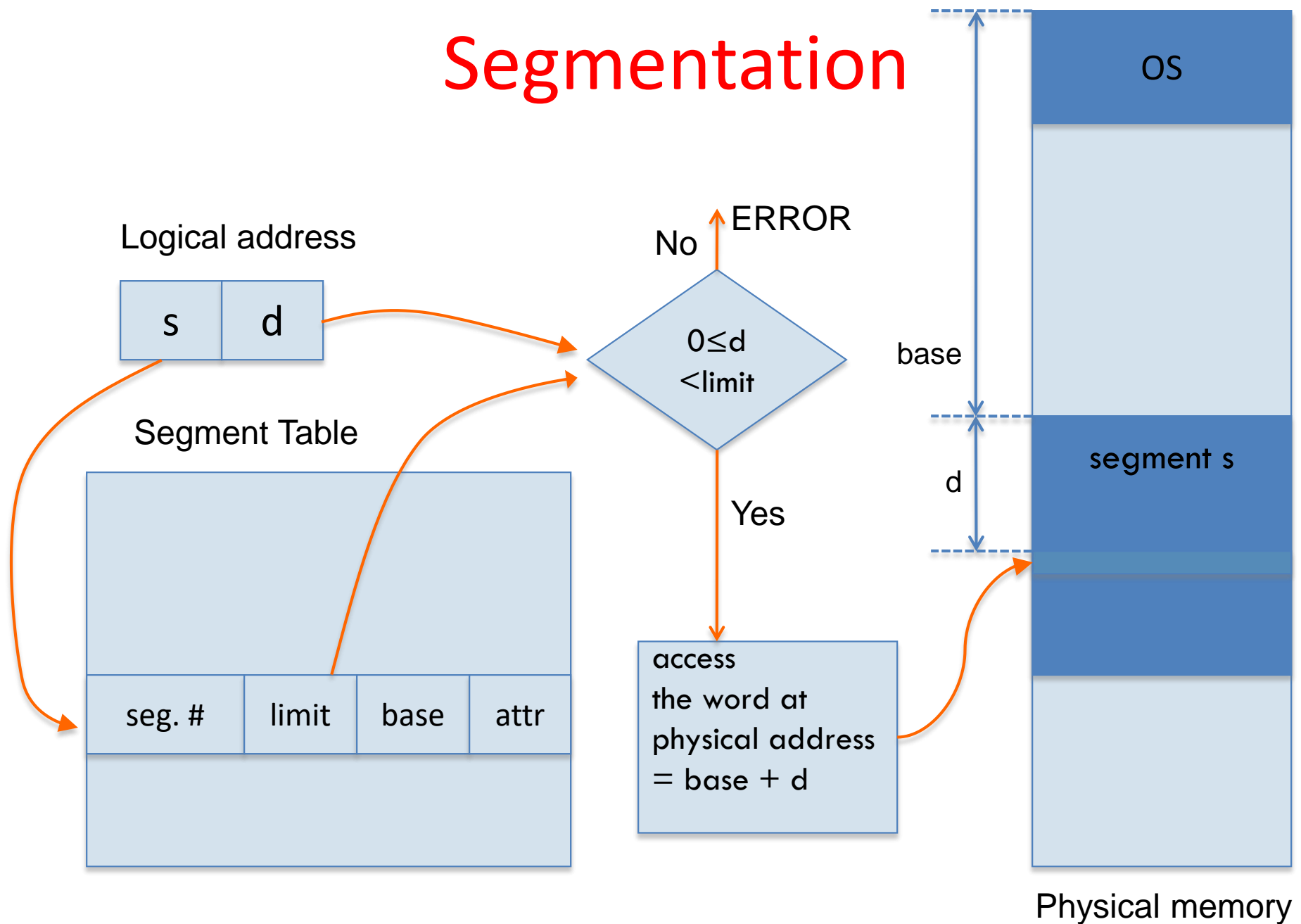


Physical memory

Segmentation

- For logical to physical address mapping, a *segment table* is used. When a logical address $\langle s, d \rangle$ is generated by the processor:
 1. Base and limit values corresponding to segment s are determined using the segment table
 2. The OS checks whether d is in the limit. ($0 \leq d < \text{limit}$)
 3. If so, then the physical address is calculated as $(\text{base} + d)$, and the memory is accessed.

Segmentation



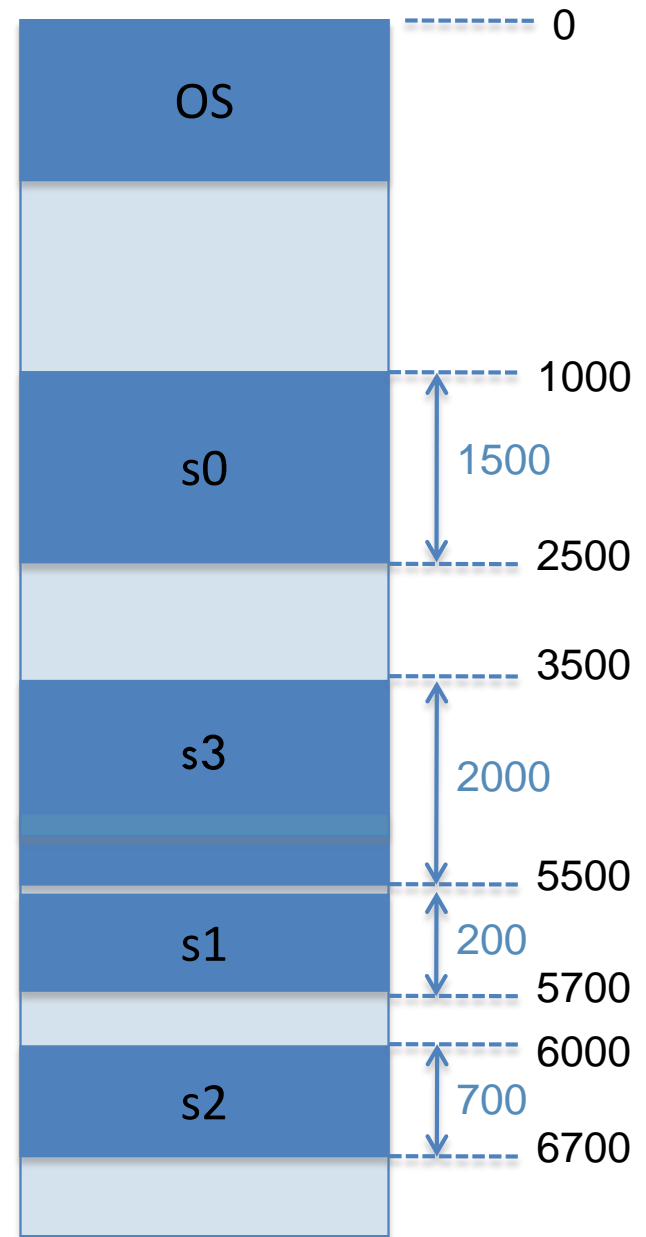
Segmentation

- **Example**
- Generate the memory map according to the given segment table. Assume the generated logical address is $\langle 3, 1123 \rangle$; find the corresponding physical address.

| Segment | Limit | Base |
|---------|-------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |

Segmentation

| Segment | Limit | Base |
|---------|-------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |



Physical memory

Segmentation

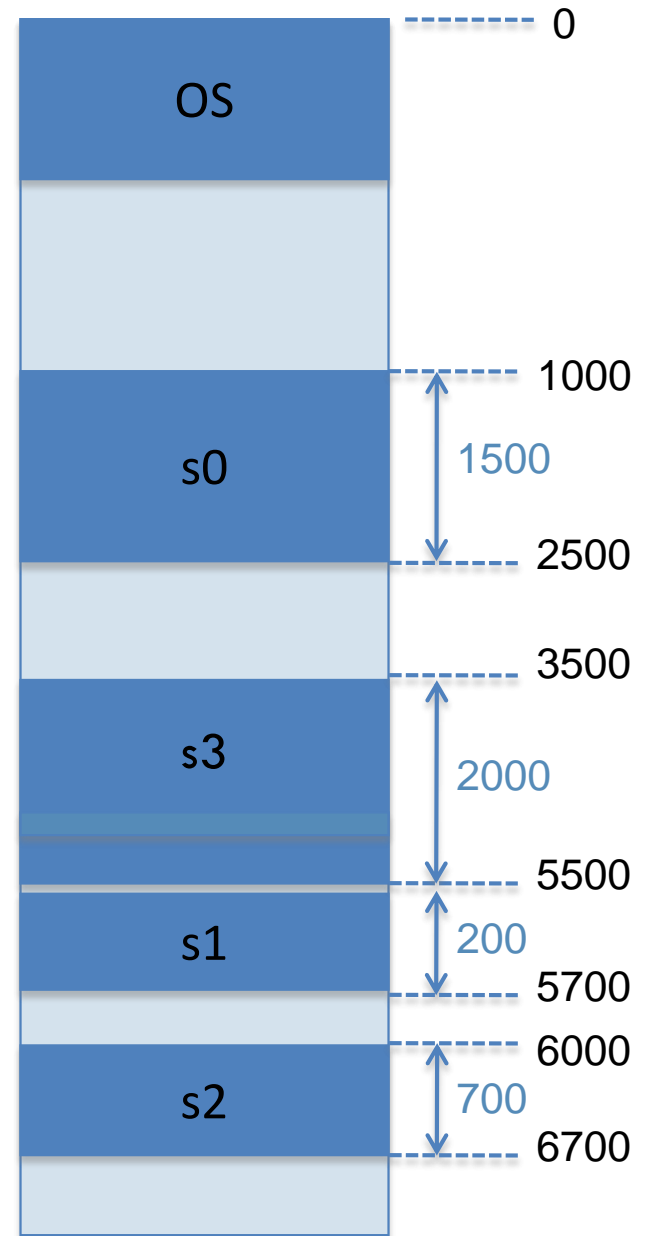
| Segment | Limit | Base |
|---------|-------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |

Logical address: $\langle 3, 1123 \rangle$

$s=3, d=1123$

Check if $d < \text{limit}$? $1123 < 2000$, OK

Physical address = $\text{base} + d = 3500 + 1123 = 4623$



Physical memory

Segmentation

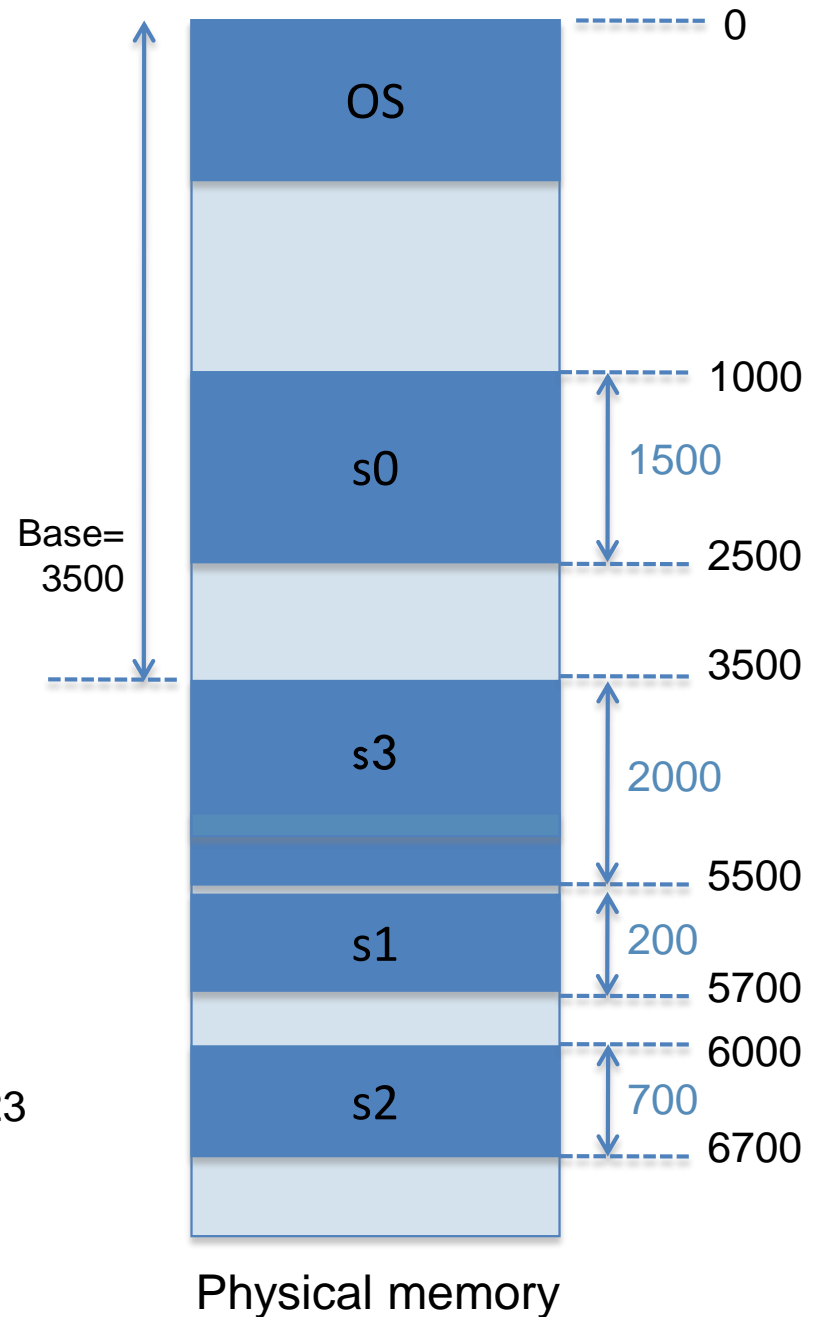
| Segment | Limit | Base |
|---------|-------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |

Logical address: $\langle 3, 1123 \rangle$

$s=3, d=1123$

Check if $d < \text{limit}$? $1123 < 2000$, OK

Physical address = $\text{base} + d = 3500 + 1123 = 4623$



Segmentation

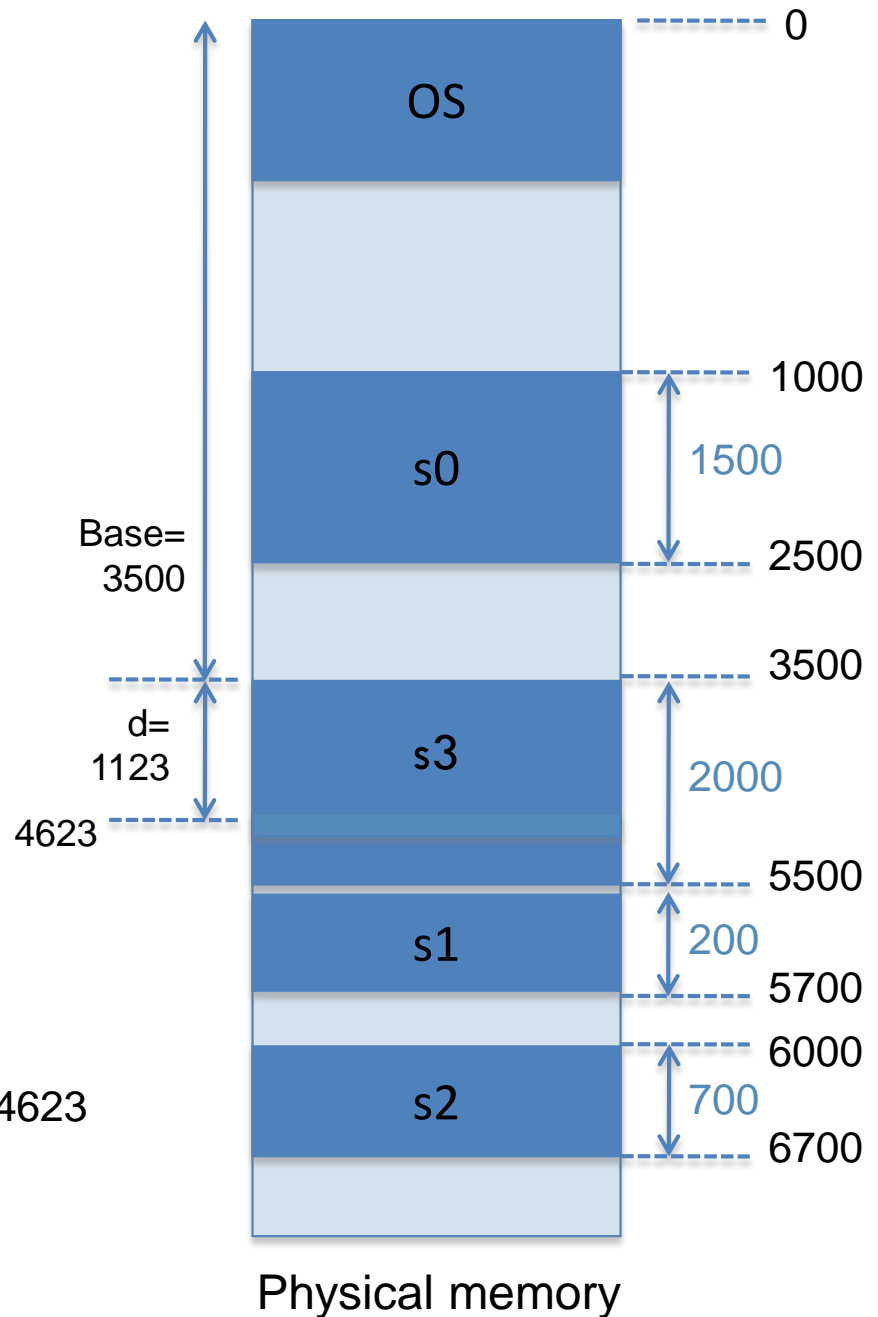
| Segment | Limit | Base |
|---------|-------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |
| 2 | 700 | 6000 |
| 3 | 2000 | 3500 |

Logical address: $\langle 3, 1123 \rangle$

$s=3, d=1123$

Check if $d < \text{limit}$? $1123 < 2000$, OK

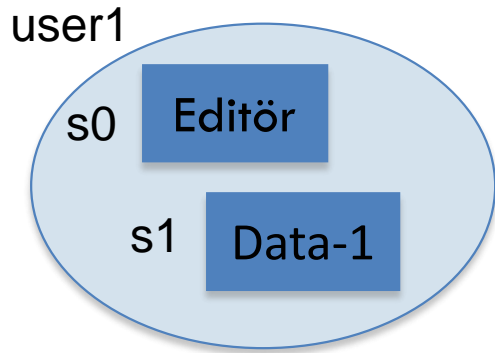
Physical address = $\text{base} + d = 3500 + 1123 = 4623$



Segmentation

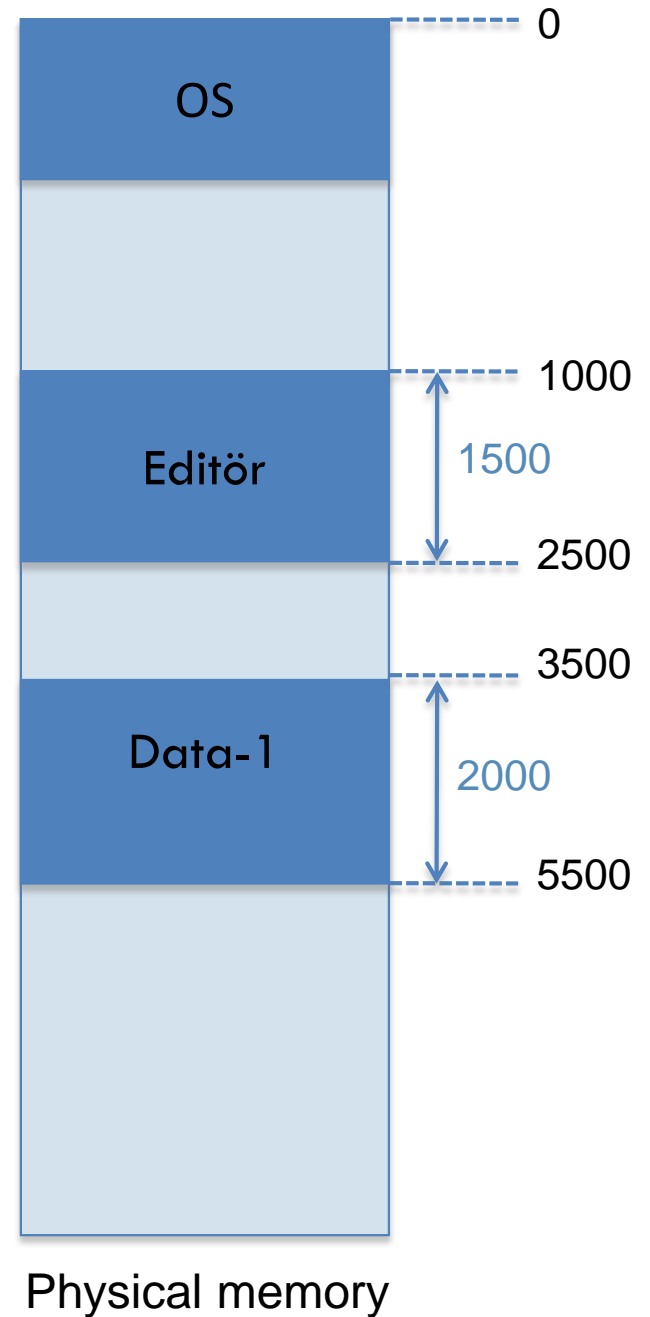
- Segment tables may be implemented in the main memory or in associative registers, in the same way it is done for page tables.
- Also sharing of segments is applicable as in paging. Shared segments should be read only and should be assigned the same segment number.

Sharing Segments

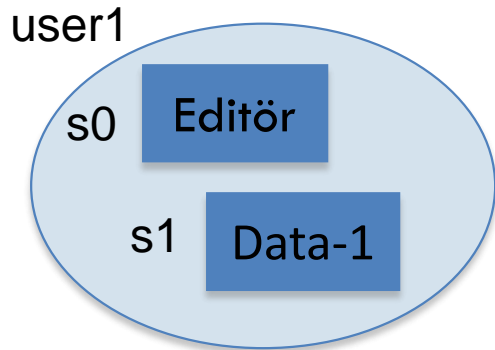


ST1

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 2000 | 3500 |

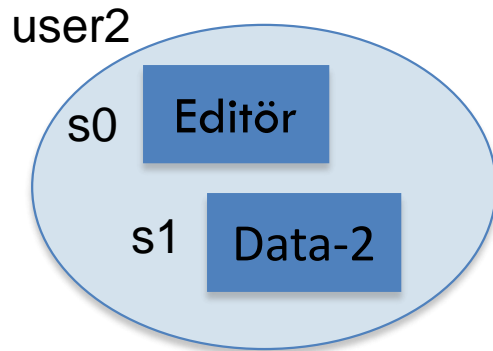


Sharing Segments



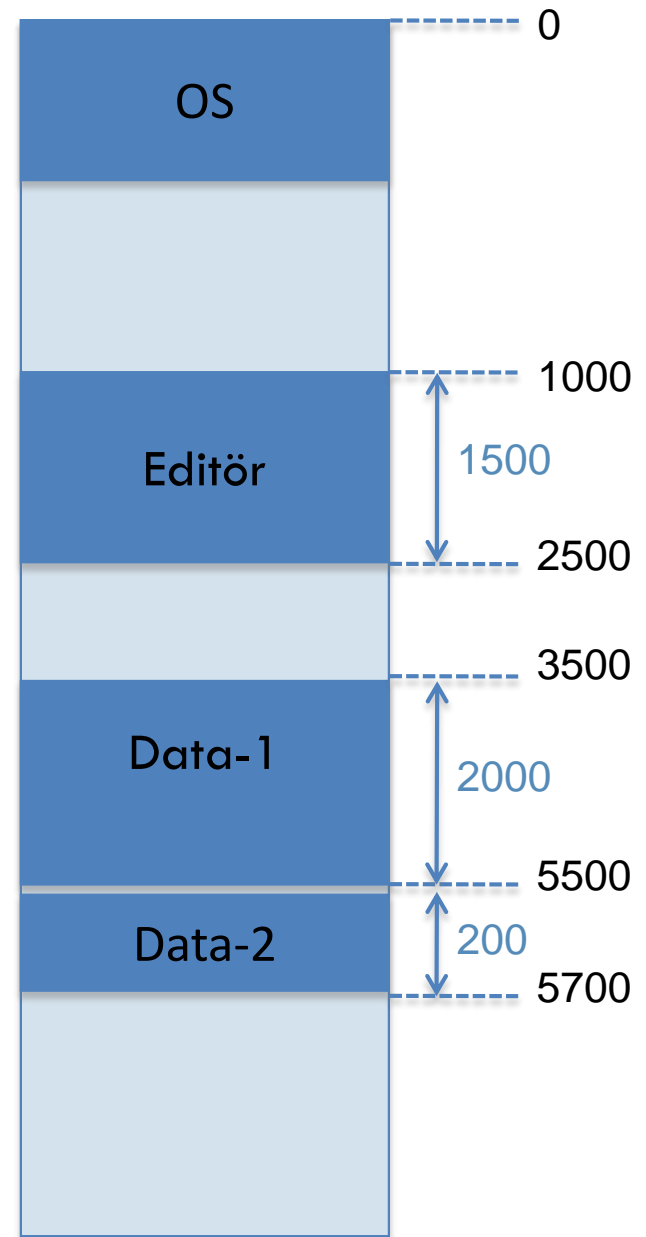
ST1

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 2000 | 3500 |



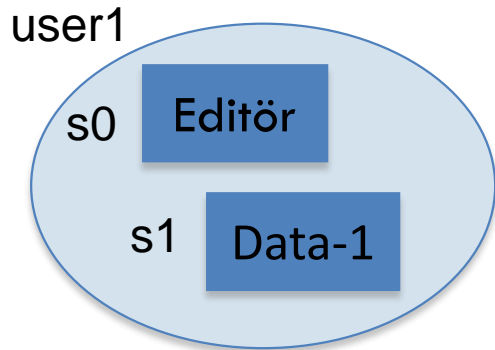
ST2

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |



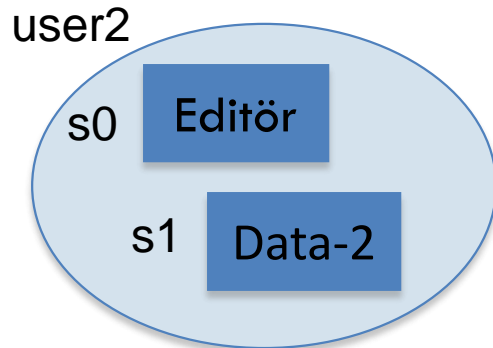
Physical memory

Sharing Segments



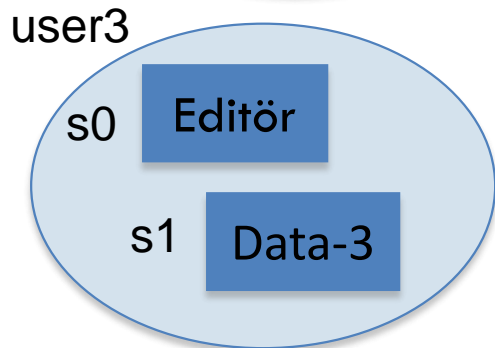
ST1

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 2000 | 3500 |



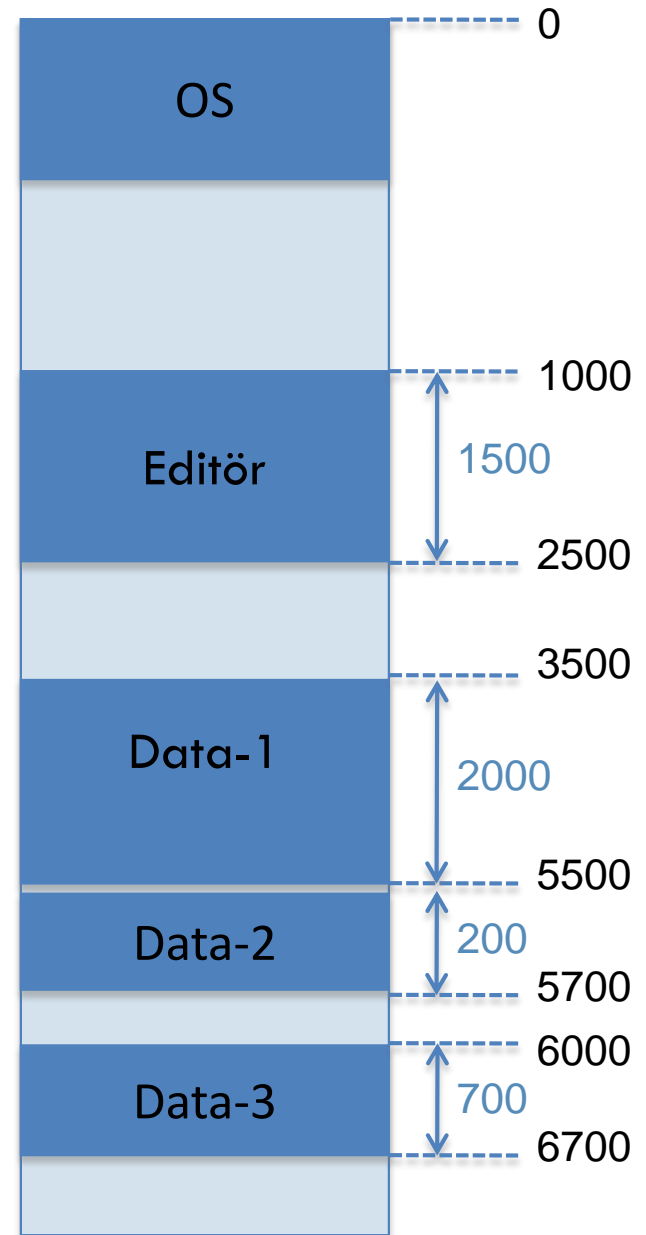
ST2

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 200 | 5500 |



ST3

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 700 | 6000 |



Physical memory

Sharing Segments

user1



ST1

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 1000 |
| 1 | 2000 | 3500 |

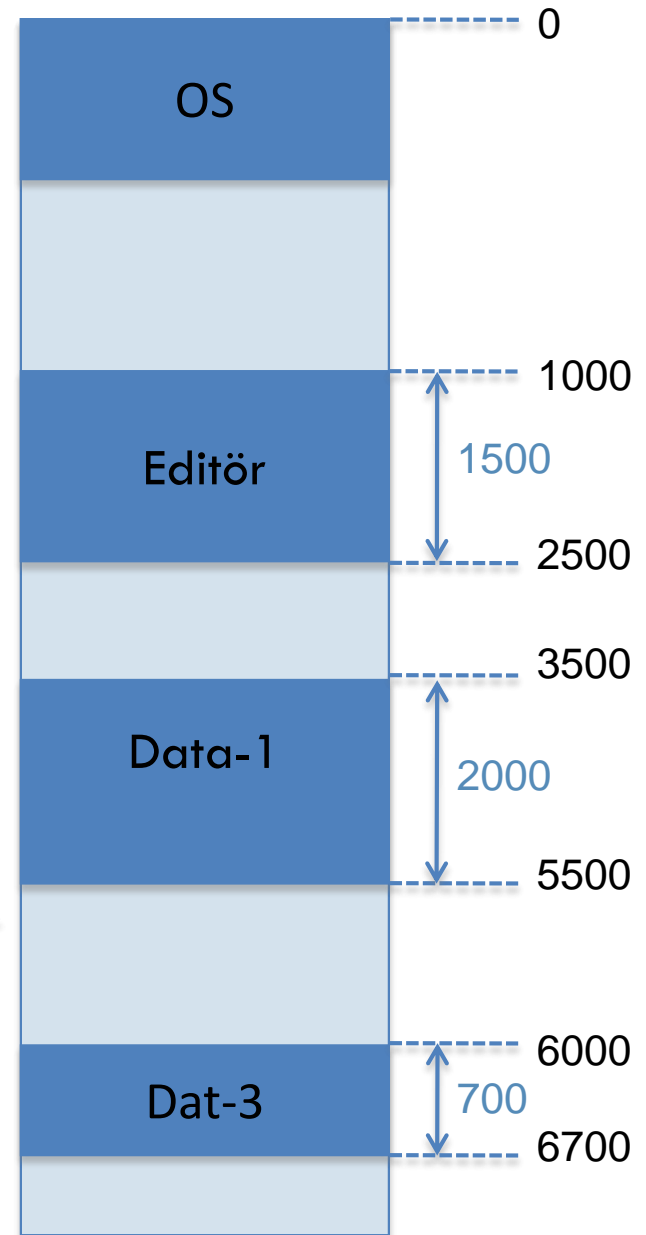
User 2 terminates:
Data-2 removed from
memory, but editor
remains..

user3



ST3

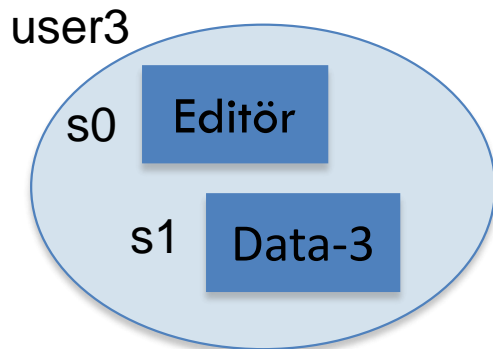
| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 100 |
| 1 | 700 | 6000 |



Physical memory

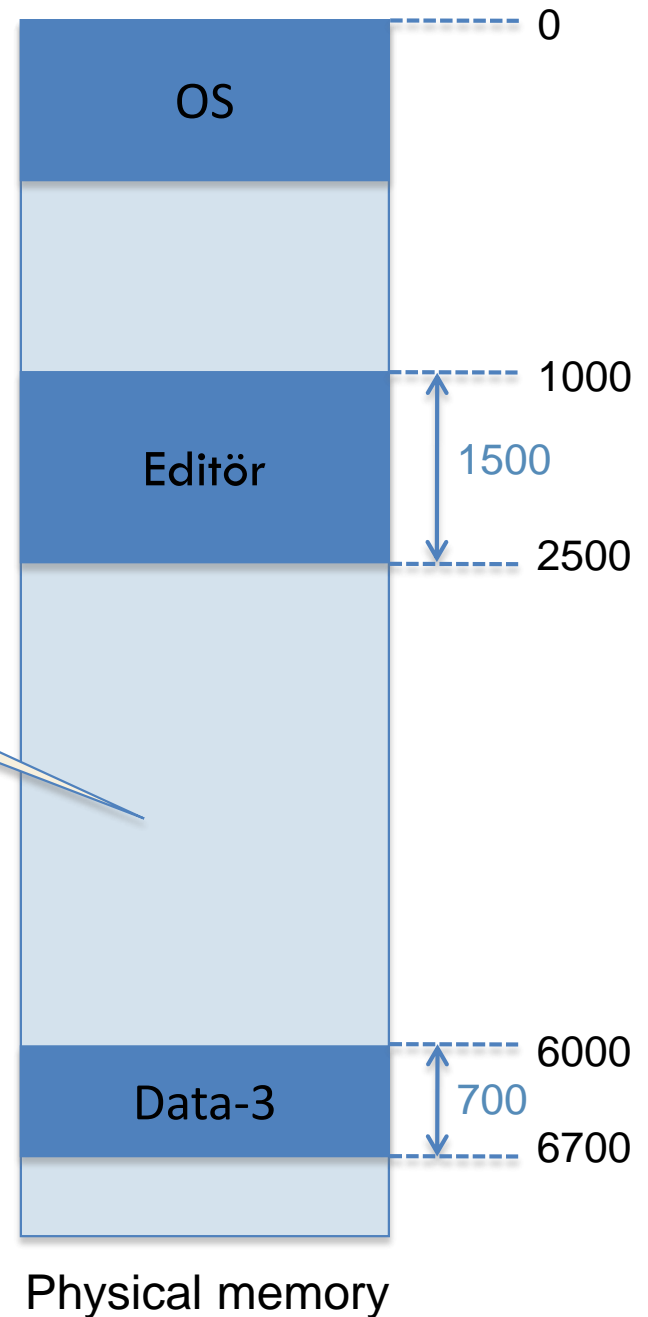
Sharing Segments

User 1 terminates:
Data-1segment is
removed from memory.

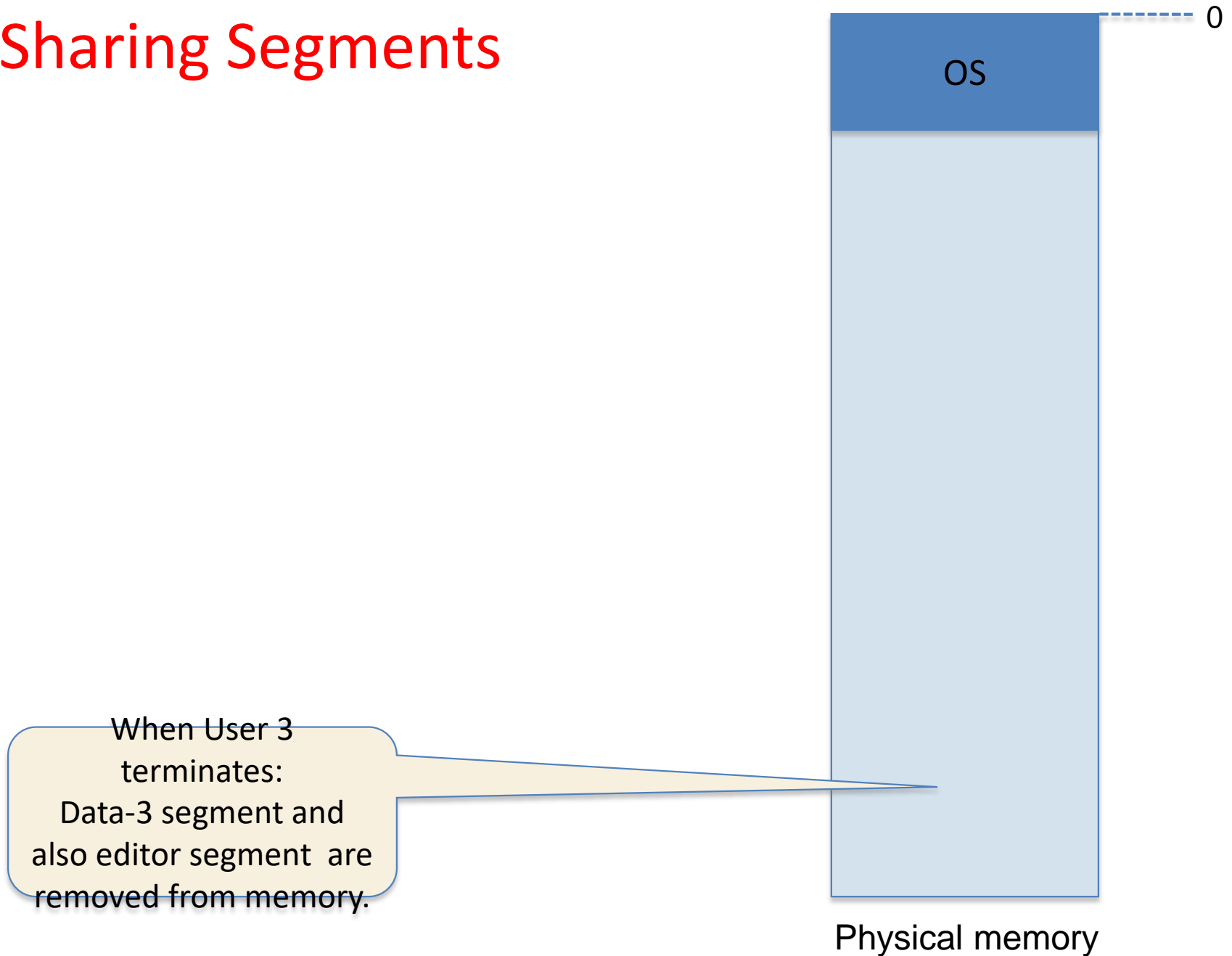


ST3

| seg | lim | base |
|-----|------|------|
| 0 | 1500 | 100 |
| 1 | 700 | 6000 |



Sharing Segments

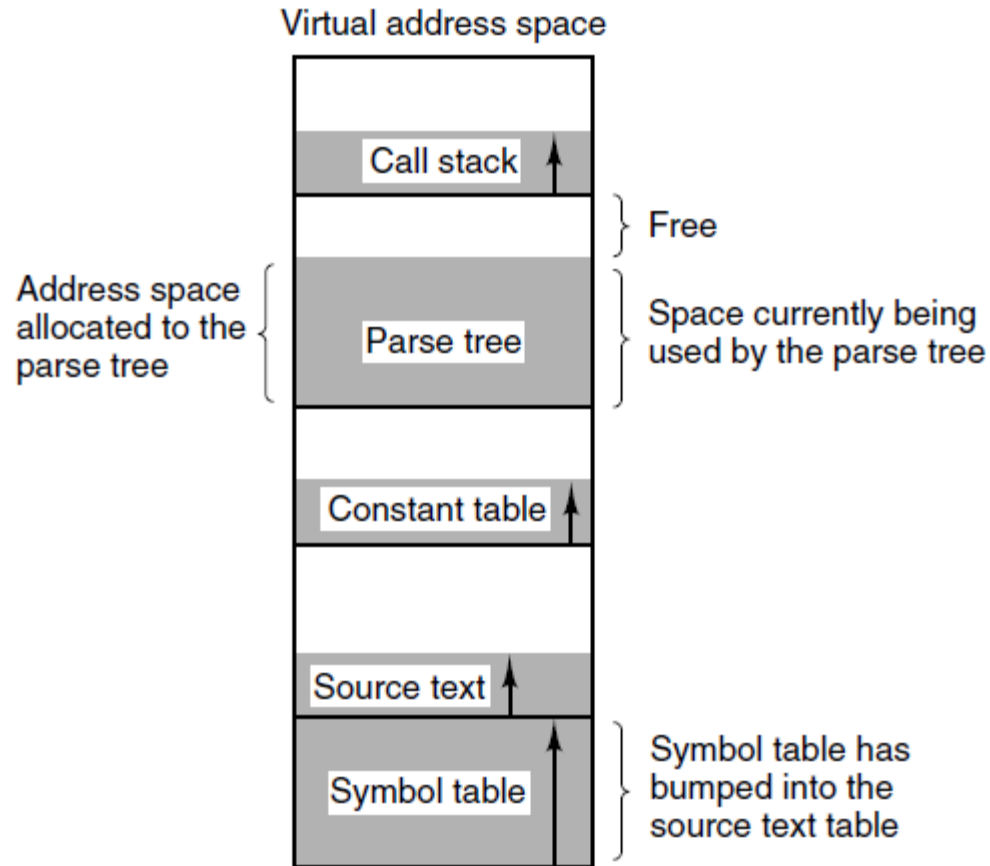


Segmentation (1)

Examples of tables generated by compiler:

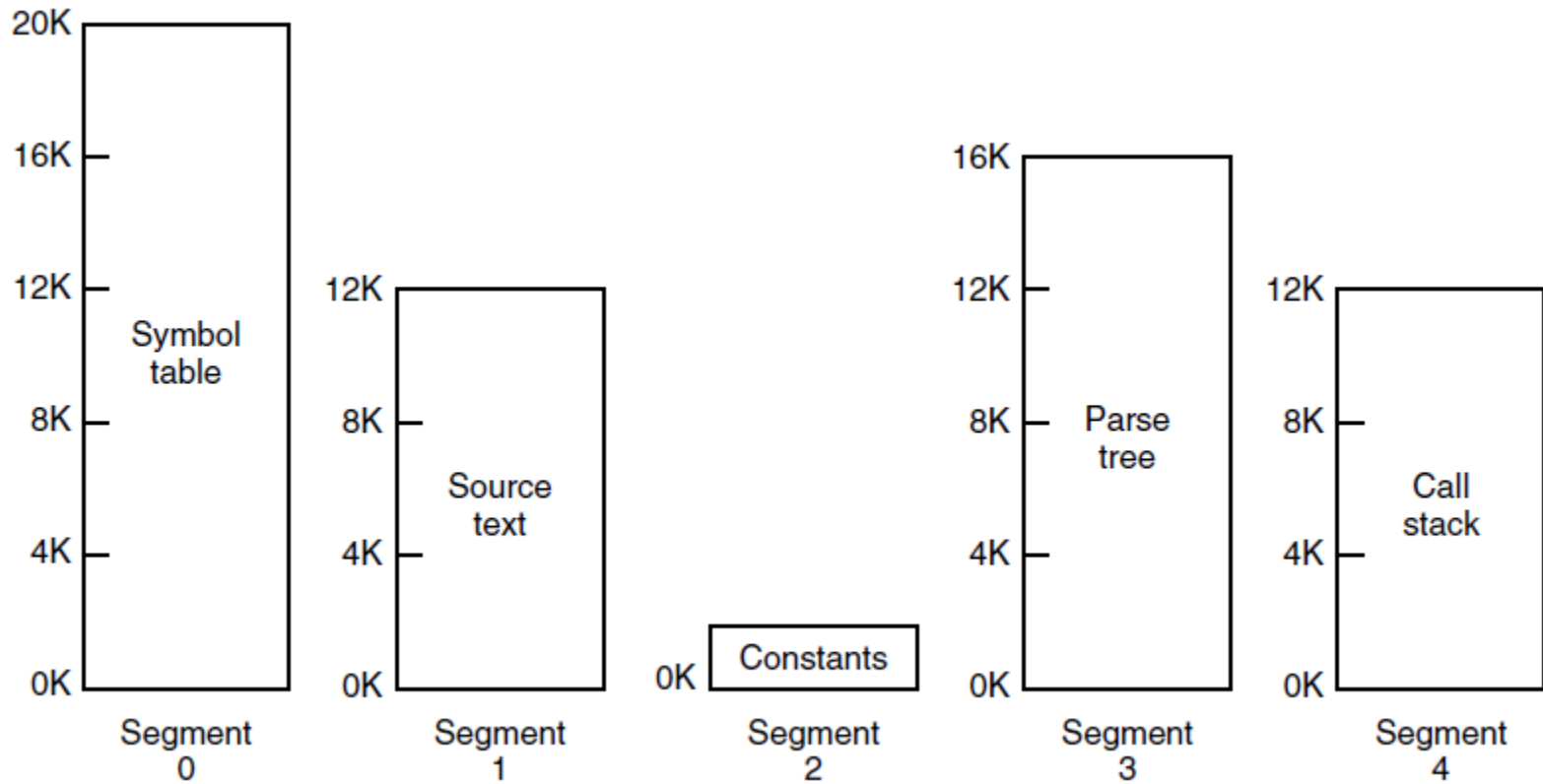
1. The source text being saved for the printed listing
2. The symbol table, names and attributes of variables.
3. The table containing integer and floating-point constants used.
4. The parse tree, syntactic analysis of the program.
5. The stack used for procedure calls within compiler.

Segmentation (2)



In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation (3)



A segmented memory allows each table to grow or shrink independently of the other tables.

Segmentation (4)

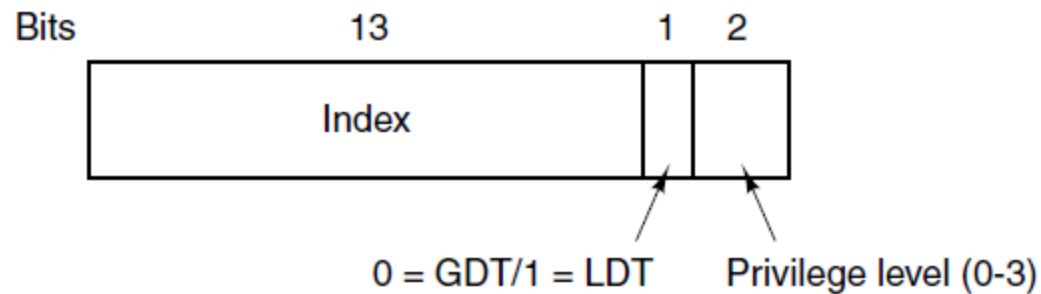
| Consideration | Paging | Segmentation |
|--|--|--|
| Need the programmer be aware that this technique is being used? | No | Yes |
| How many linear address spaces are there? | 1 | Many |
| Can the total address space exceed the size of physical memory? | Yes | Yes |
| Can procedures and data be distinguished and separately protected? | No | Yes |
| Can tables whose size fluctuates be accommodated easily? | No | Yes |
| Is sharing of procedures between users facilitated? | No | Yes |
| Why was this technique invented? | To get a large linear address space without having to buy more physical memory | To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection |

Comparison of paging and segmentation

Reading Assignment

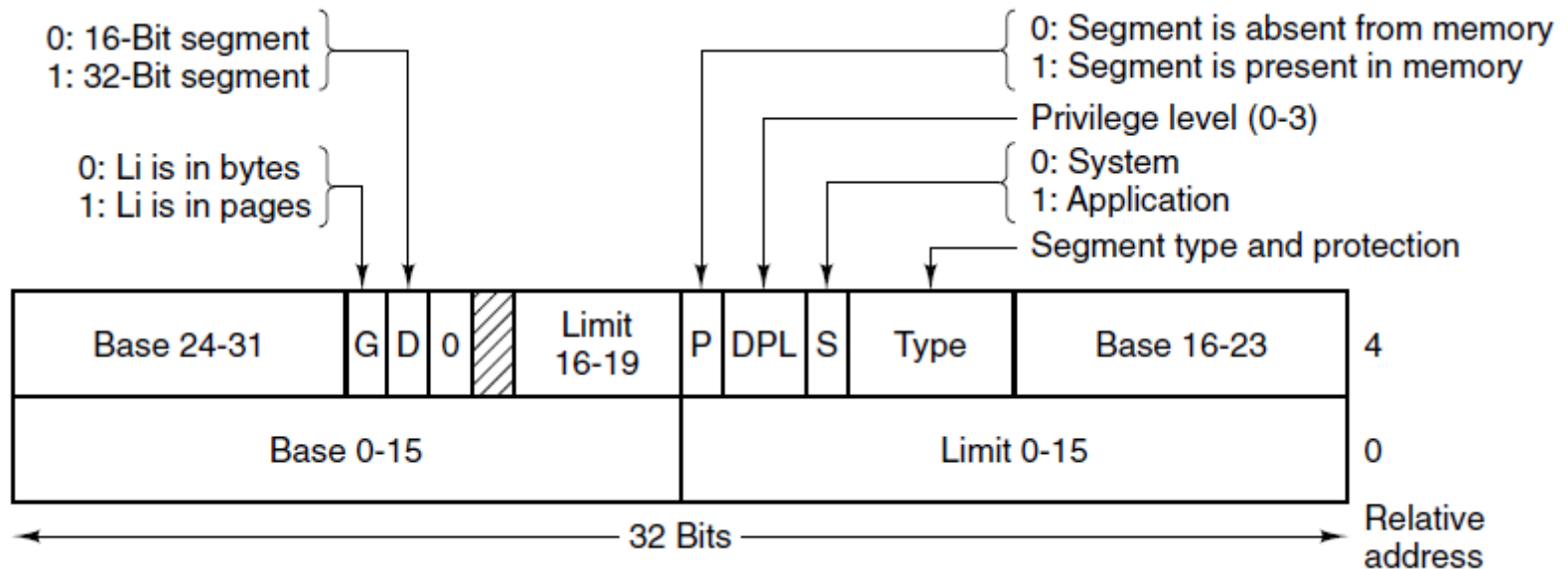
- 3.7.2. excluded
- 3.7.3. reading assignment

Segmentation with Paging: The Intel x86 (1)



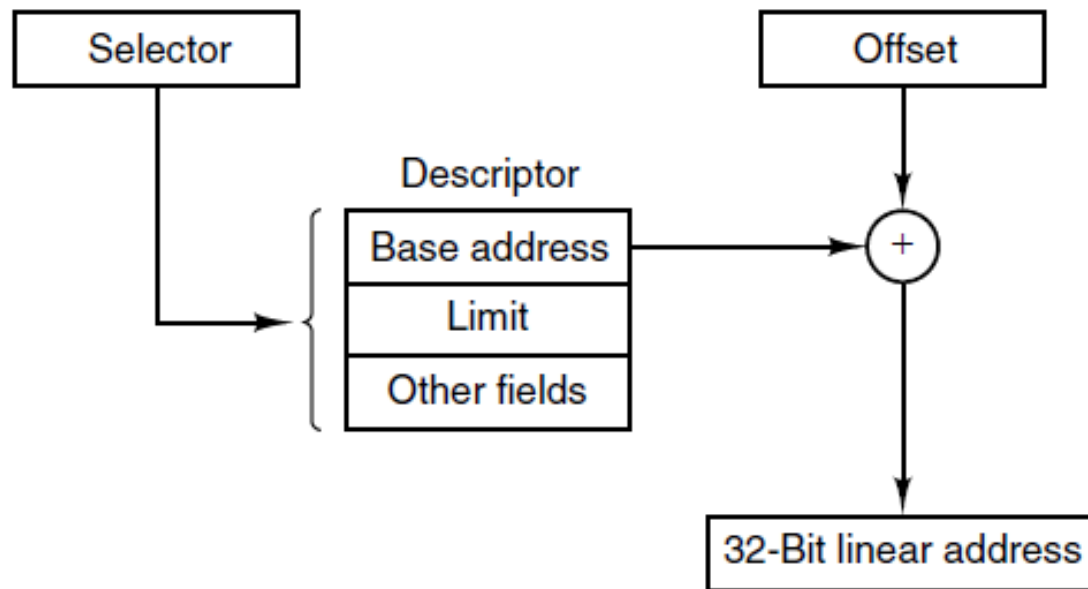
An x86 selector.

Segmentation with Paging: The Intel x86 (2)



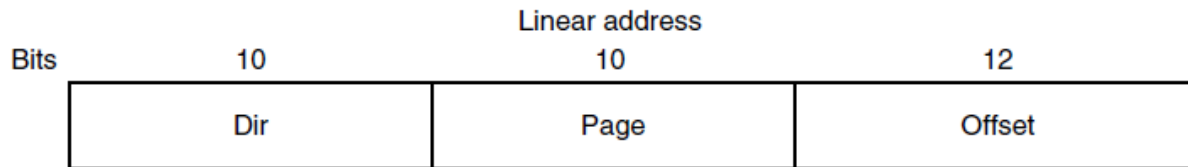
x86 code segment descriptor.
Data segments differ slightly.

Segmentation with Paging: The Intel x86 (3)

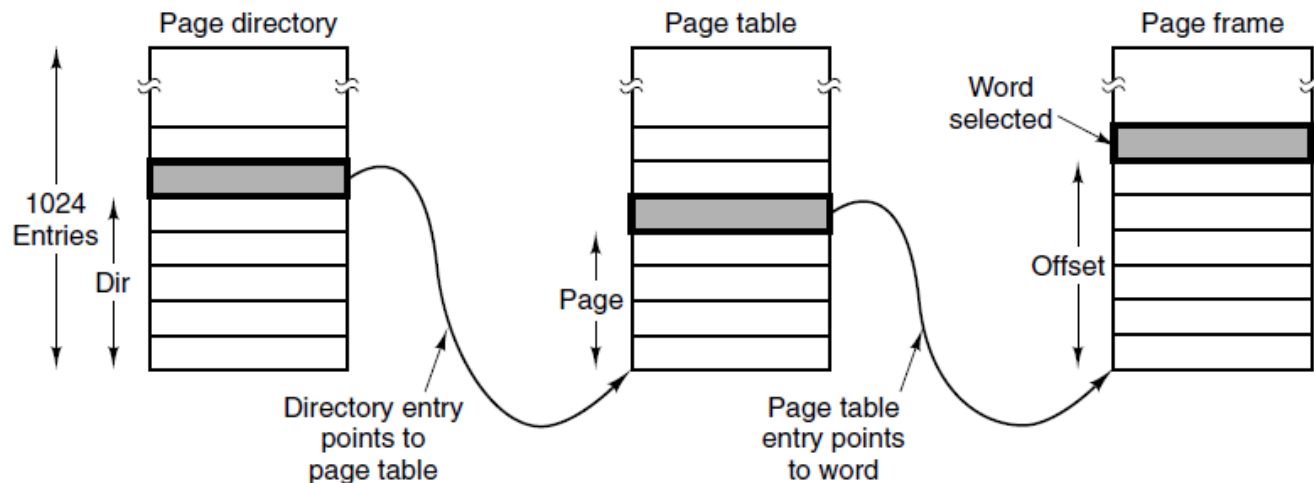


Conversion of a (selector, offset)
pair to a linear address.

Segmentation with Paging: The Intel x86 (4)



(a)



(b)

Mapping of a linear address
onto a physical address.

End

Chapter 3