Module - 5

Storage Management

5.1 Main Memory

Background

An executing program first fetches an instruction from memory, which is then decoded & the operands are fetched from memory. After the result is received it is stored into the memory.

The memory unit is only concerned with the memory addresses being generated & not by how the program is generating them.

Thus we are only interested in the memory addresses being generated by the program.

1) Basic Hardware:
The CPU can only directly access the main memory and the built-in registers (in the CPU) and not the disk

addresses. Thus all data that are to be used by a program during execution will need to be loaded onto the main memory before execution.

However, the CPU can access data from the in-built registers in 1 CPU clock, & can perform 1 or many simple operations in register contents within one CPU clock.

*Note: CPU clock is the unit of time used by CPUs; ~~with one clock being the time needed for the most basic instruction to be executed (addition of two operands)~~

However, for accessing the main memory, several CPU clocks are required because the memory buses are needed to access the main memory.
In this case, the processor stalls, waiting for an instruction from the main memory.

To avoid this, a faster memory is added between the CPU & main memory: the cache.
This cache is added onto the CPU chip for fast access. This hardware solution speeds up memory access & without any operating system control.

We also need to ensure that the data is correct. For proper system operation we must ensure that the user processes do not access the OS.

This is because the OS also manages the system resources, intervention from the user processes could modify the OS code & make the system unstable. This protection must be provided by the hardware.

We also need to ensure that user processes (in a multiprocessor system) do not interact & update their memory spaces

Thus, we need to ensure that each user program has a separate memory space (in the main memory).
The user process space 1 cannot be accessed by user process 2.

For this memory space, 2 registers are used. The base register & the limit register.

The base register basically holds the lower bound address & the limit register holds the size of the range that can be used. Thus, a base register of 100 and limit of 50 would mean that the

user process can take the memory address from 100 till 150 (of the main memory):

[100 - 150]

Whenever a memory address is generated by the user process, the hardware checks it with the value of base and limit. If the address is in the OS occupied or other process' occupied memory, then the attempt is a fatal error.

The base and limit registers can only be loaded by the OS, which uses the privileged instructions.

The OS also has unrestricted access to all memory spaces, which it needs for loading or unloading processes, performing I/O, etc.

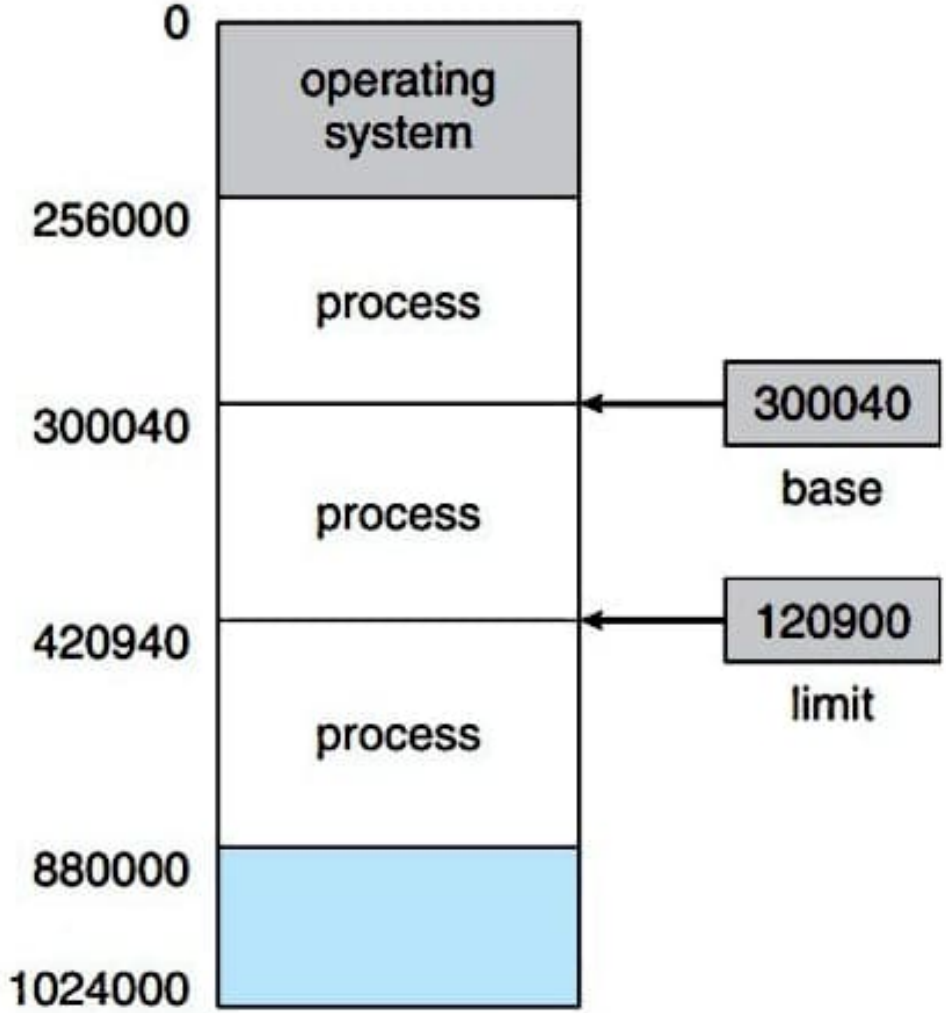


Figure 8.1 A base and a limit register define a logical address space.

2) Address Binding:

Processes are loaded onto the main memory from the disk for execution. The processes in the disk waiting to be loaded onto the

main memory for an 'input queue'. They are different from ready queue, as the ready queue forms in the main memory for CPU time.

One of the processes in input queue is loaded onto main memory and after it is done executing, the process terminates and frees up the space in the memory.

A User program goes through many steps before execution. One of these is deciding the address in the memory.
The address for the user program can be different at every step.

- In the source code, the referenced addresses are often symbolic
  Eg: Count = 80
  count = count + 12

- The compiler binds these symbolic addresses to relocatable addresses, in above example it will be '12 bytes from the beginning.'

- The 'linkage editor' or loader then in turn binds the relocatable address to the absolute address (eg: 10012)

The binding of instruction & data into

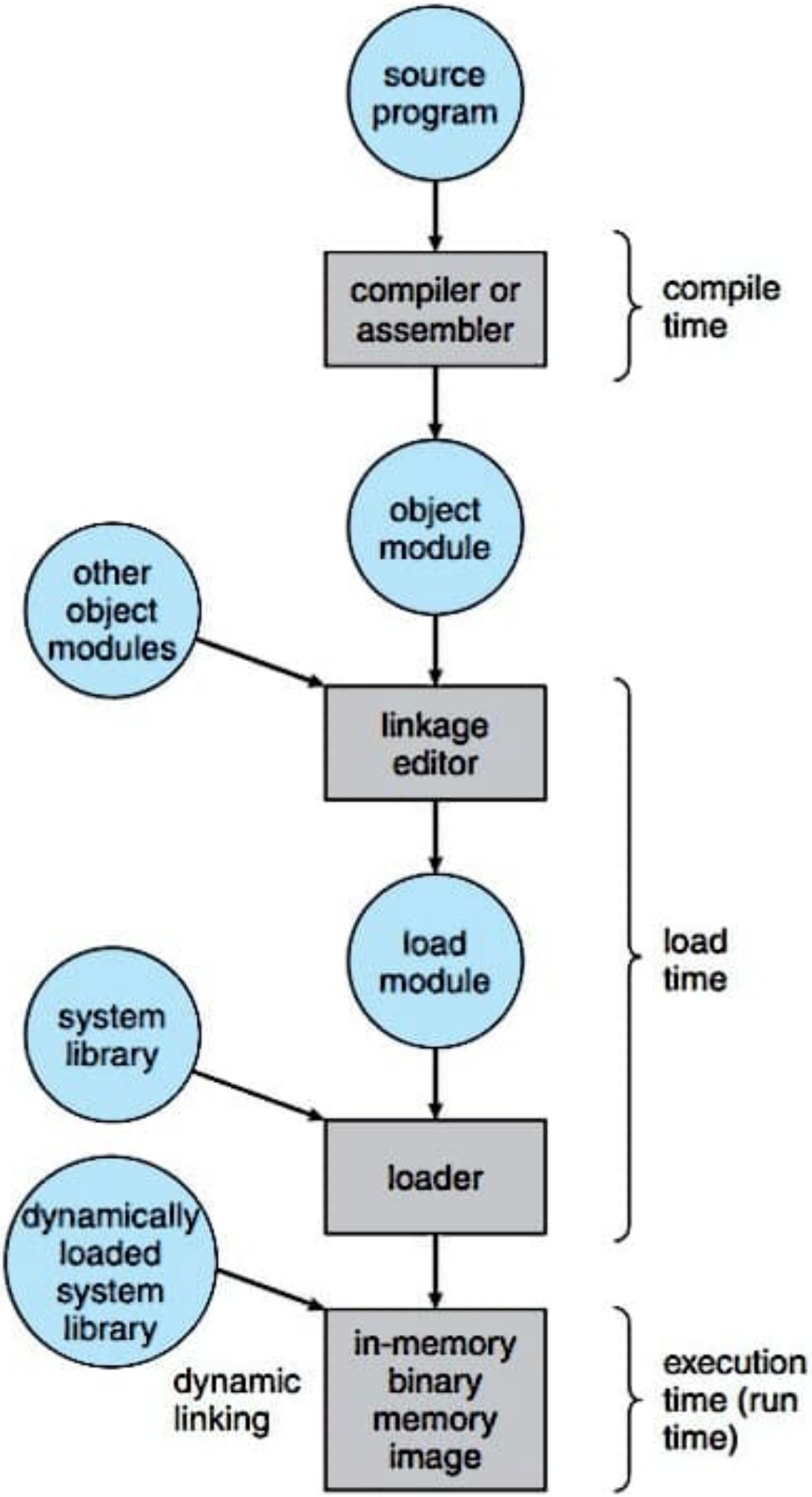


Figure 8.3 Multistep processing of a user program.

memory addresses can be done at any step:

a) Compile Time: If you know where the process will reside at compile time, then the absolute code can be generated at compile time by the compiler.

If the starting location changes, then the code will have to be recompiled. Used in MS-DOS.

b) Load Time: If the address is not known at compile time, then the compiler needs to generate a relocatable code.

The final binding is delayed till load time, and any change with starting location will require reloading of user code.

c) Execution Time: If the process needs to be moved during its execution, then binding has to be done in execution time. Special hardware is needed for execution time binding. Most used.

3) Logical vs Physical Address Space:

The address generated by the CPU is called the logical Address.

The address that is loaded onto the memory-address register is a physical address.

In compile time and Load time ~~load~~ binding, the ~~address~~ logical address and the physical address are identical. In execution time binding, however, the 2 addresses differ.

- logical address space: All logical addresses generated by the program.
- physical address space: All physical addresses generated by the process.
- MMU (Memory - Management Unit): Does the runtime mapping of logical to physical Addresses are done by this hardware device.

In MMU, a relocation register is used. The value of the relocation register is added to the address that the user program generates. Eg: If user program generates the address '0' and the relocation register is 500, then the physical address will be 500 + 0 = 500.

Keep in mind that the above addresses are for the disk, not main memory.

The user program only interacts using the logical address, and the memory-mapping hardware maps the corresponding physical address.

Logical Address:

Range = 0 to max

Physical Address:

Range = R+0 to R+max

R = relocation register

4) Dynamic Loading:

We have only executed processes in main memory fully till now, limiting the size of an executable process.

We can use dynamic loading for better memory utilization.

In dynamic loading, a routine (or we can understand it as a function) is not loaded onto the main memory. The routines stay on the disk in relocatable load format (will relocate with loading).
The main routine executes in the main memory in the beginning. It then checks

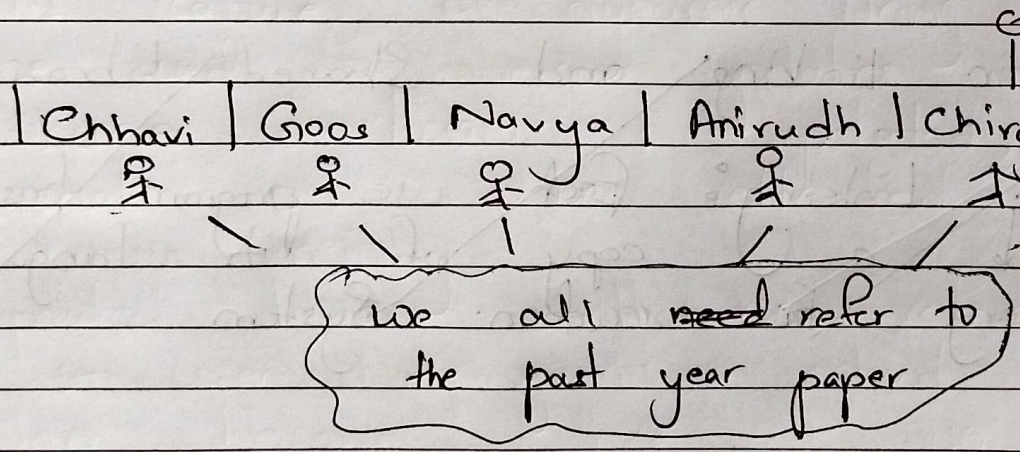
if the next routine needed is in the main memory, if not, that routine is loaded, the program's address table is updated and the control is passed to the newly loaded routine.

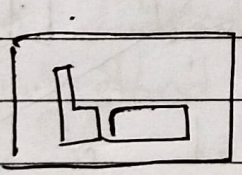
Dynamic loading is useful because, even if the total size of the program may be large, the portion loaded can be small.

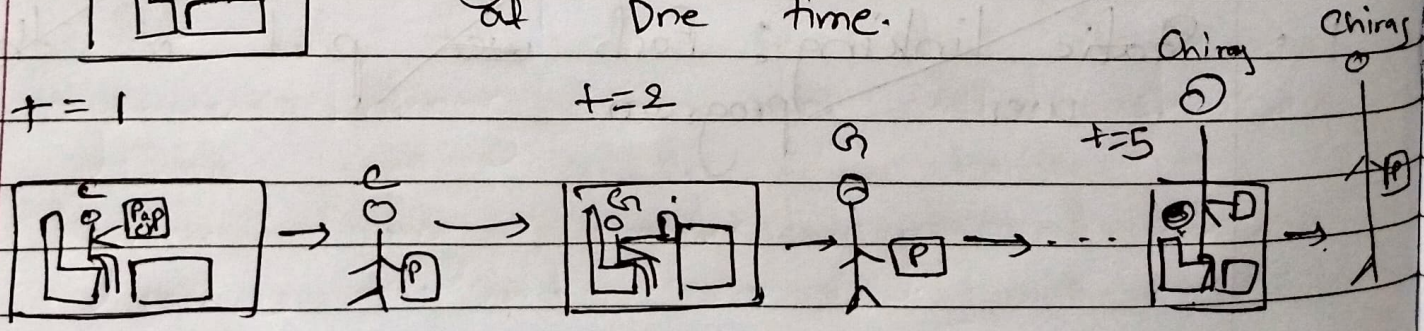
5) Dynamic linking & Shared Libraries:

In dynamic loading, user program is brought into memory in parts.
Every user program has certain libraries or routines that they use.
When the parts of a user program is brought into the main memory, each part will require the same libraries or routines, & each part will load ~~its own~~ these common libraries onto the main memory individually. This loading and unloading of common libraries is redundant and ~~coo~~ costly.

Thus, dynamic linking is used. The above scenario would play in static linking.




→ Only one person can be in room at one time.



Instead of every person ~~pr~~ bringing their own paper, studying and going, we can ask the first person (Chhavi) to leave the paper in room & go so that no one else has to bring their paper.

The remaining people ~~can~~ just need to know where the paper was kept by chhavi, & they can take the paper from that place.

This is what happens in dynamic linking. The first part of the user program brings the routine / library into the main memory while execution, & then it is removed, but the library is kept in the main memory.

~~The next part of the program~~

Now, everyone (goos, Navya, Aniruddh & Chirag) will need to have some way of knowing where chhavi kept the paper. ~~The~~ Thus, they all ~~to~~ carry a small phone with them, ~~it~~ which they can use to contact chhavi & ask for the address / location of the paper.

Similarly, each part of the user program carries a small code called

the 'stub'. The stub indicates whether or not all the libraries needed by the part of user program is there in the memory, if they are, the stub replaces itself with the address of the library / routine.

* Note: It is possible that some parts of the user program require a different system library.

If the library needed is not present in the memory, it is loaded into the memory, after which the stub replaces itself.

This feature of dynamic linking can also be used for version updates in library. Whenever a library is updated, the update will be reflected in the main memory updated library will be referenced by the user programs instead of the older library, thus all user programs need not be relinked after the library is updated.

Dynamic Linking is also called shared library.

Dynamic linking, unlike dynamic loading, requires support from OS.

Swapping

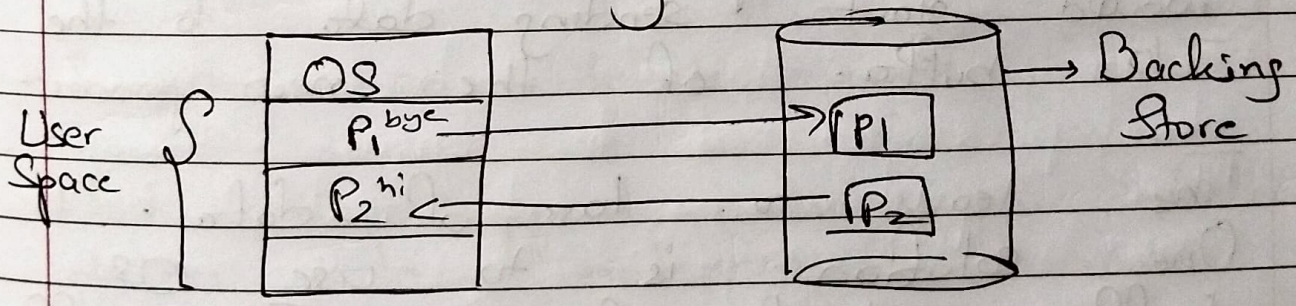
Swapping is used to make it possible to increase the total physical memory available in the system, thus increasing the degree of multiprogra-mming.

The processes are swapped out of main memory to something called a 'backing store'.

Backing store is a disk but it is faster in speed (of transfer, etc.).

1) Standard Swapping:

Standard swapping involves moving processes between main memory and the backing store.



The ready queue maintained by the system contains all the processes from the memory and the backing store.
The memory image of the process that was switched is stored as well in the

backing store.

When a process needs to start execution, the dispatcher checks whether the process is there in main memory or not. If not, the dispatcher loads it from the backing store, & swaps out one of the processes in main memory.

This context switch is extremely costly.

Another issue with swapping arises when a process is waiting for an I/O operation. The I/O buffer is getting sent the data from the I/O device. I/O buffers reside in the memory used by a use process.
Now if this process is swapped out for another process, the I/O device would start sending data to the I/O buffer of the new process.

This leads to loss of data.
One solution is to use OS buffers instead of the I/O buffers. Even if the process is swapped out, the data in the OS buffer stays consistent. But this method has considerable overhead, due to the communication needed between kernel memory & user memory.

Another method is to not swap out processes which are accessing I/O devices altogether.

Modified version of swapping is used in Os like UNIX, Linux, Windows, etc. Two of many modifications are:

- Only swapping processes when the remaining space in main memory is less ~~that~~ than a threshold value
- Swapping portions of the user process, instead of the entire process, as the swap time is directly proportional to the size of the program to be swapped.

Standard swapping is not used in modern OS.

2) Swapping on Mobile Systems:

Most PCs support swapping, but mobile devices do not use swapping, as these devices use flash memory instead of disks for storage. Flash memories have more constraint space, ~~thus~~ and they also can not tolerate too many (frequent) writes.

Apple's IOS: Instead of swapping, IOS

asks applications to reliquish ~~contrat~~ some of its allocated memory. Read only data are removed from the memory, but data that are being modified (like the stack) are kept in memory. However, if any application fails to free up enough space, it may be terminated by OS.

Android: Instead of swapping, it adopts a similar method to iOS, but before an application is to be terminated, android writes its application state to flash memory. This helps the application to get restarted quicker.

* Application State: The current condition or configuration of a running application.

Because of these restrictions, mobile developers need to ensure that their applications do not use too much energy.

Contiguous Memory Allocation:

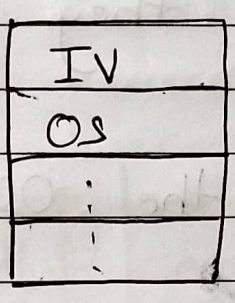
The main memory is partitioned into multiple partitions. One partition contains the OS, other contains the interrupt Vector, and the rest contain user programs.

The interrupt vector is a table used to map interrupt number to its corresponding interrupt handler.

It is important to keep the interrupt vector and the OS close to each other in the main memory so that swift action can be taken by the OS in case of an interrupt.

Both usually reside in the lower part of the main memory.

0



* Contiguous Memory Allocation: In this memory management technique a process is allocated a single contiguous block for its execution.

1) Memory Protection:

As discussed before, it is important to protect a user program's ~~to~~ be memory to be accessed by a different user program, and the OS also needs to be protected by this memory access.

Thus it is important to ensure that the

addresses generated by the user program are all within the range of the relocation register and the limit register.

The relocation register scheme allows for the OS's size to change shape dynamically. This flexibility is desirable, as the OS contains codes that are not commonly used, thus they can be removed and extra space can be saved.

Such codes are called transient (imperma-nent) OS codes as they come and go as needed.

Thus, the size of the OS can change during program execution.

2) Memory Allocation:

a) Fixed Size Partitioning:
Memory is divided into several fixed size partitioning. Each partition will contain only 1 process, thus the degree of multiprogramming is bound by the total no. of partitions.

When a process is terminated, that partition is now free and can be used by a different user program in the input queue.
This method was used by IBM OS/360.

b) Variable Partition Scheme:

Initially all memory blocks are empty and are considered as one big block of available memory, called as a hole.

The operating system selects a program from the input queue and puts it in the main memory, reducing the size of the available hole. It keeps allocating user programs from the input queue to the main memory, till its holes are too small to accomodate a new program.

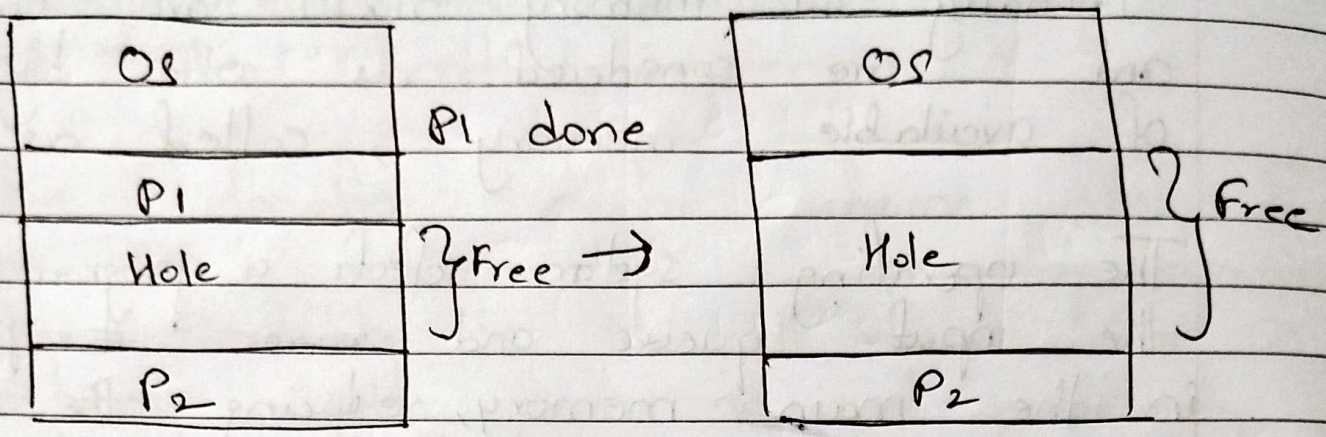
When a user program terminates, the operating system looks for a program, that can fit into the newly created hole, from the input queue.

The hole freed after the user program was terminated is merged with surrounding holes to create a bigger hole.

Notice that the partitions here are created due to the allocation of user programs into memory, & these partitions have a variable size, depending purely on the behaviour of programs.

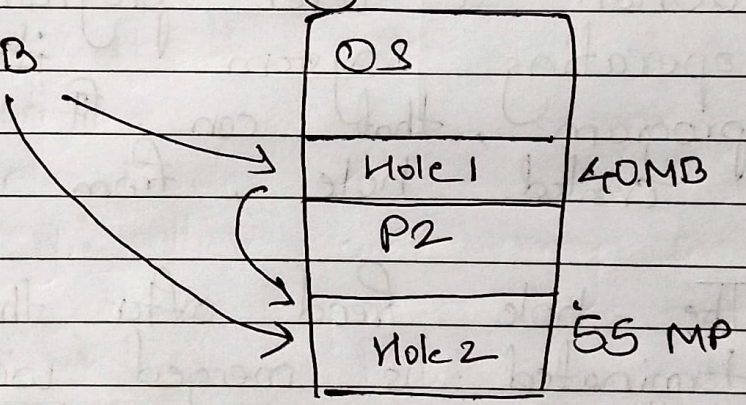
This problem is called dynamic storage -allocation problem & there are many solutions to this problem: First fit, Best fit,

worst fit.

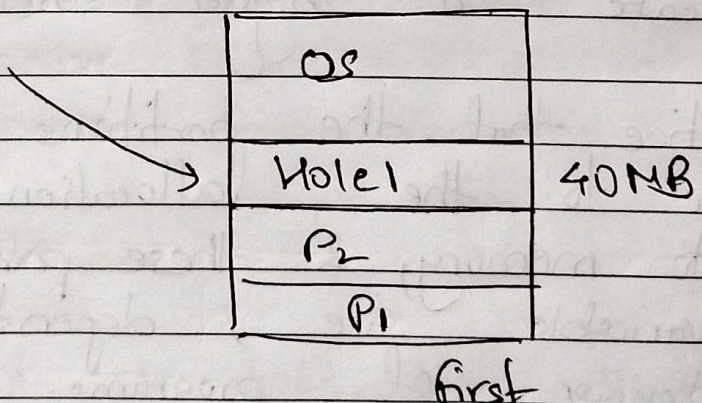


i) First Fit : Allocate the first hole that is big enough to hold the process. With every process to be allocated, start the search from the first hole in main memory always.

→ $P_1 = 50$ MB



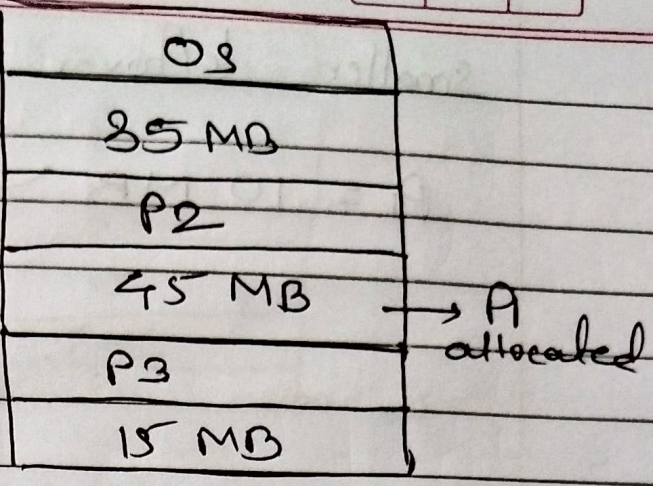
→ $P_3 = 15$ MB



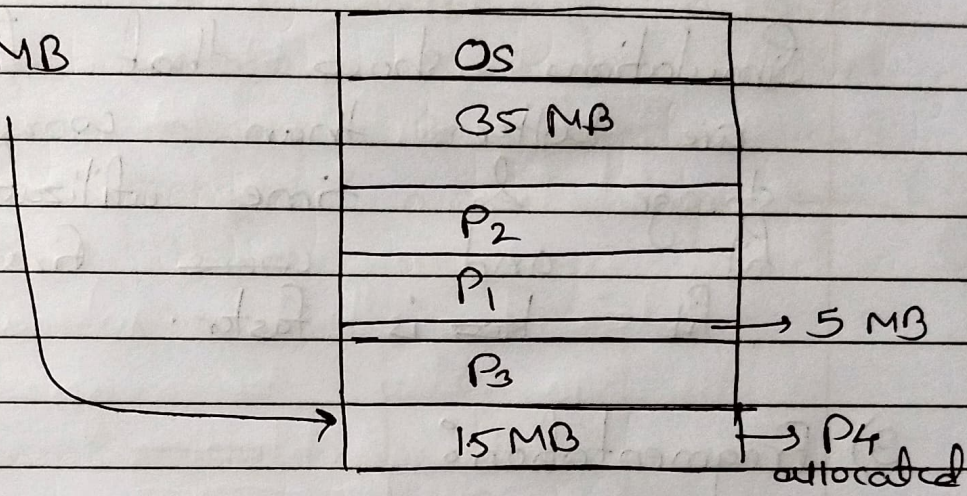
first

ii) Next Fit : Allocate the hole big enough to hold the process, starting from the hole where the previous next fit search ended.

→ P_1 = 40MB

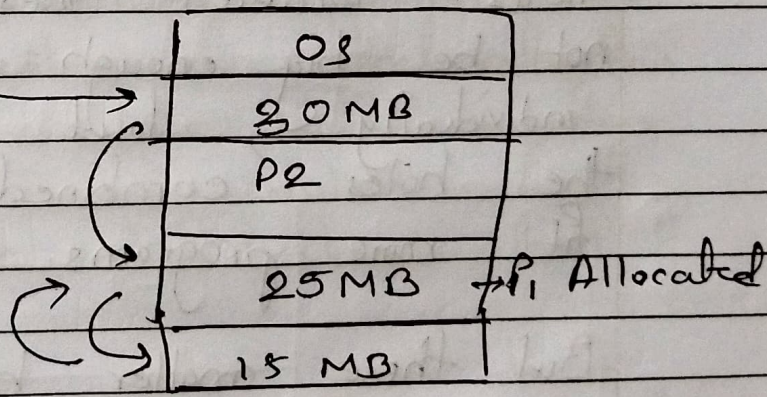


→ P_2 = 15MB



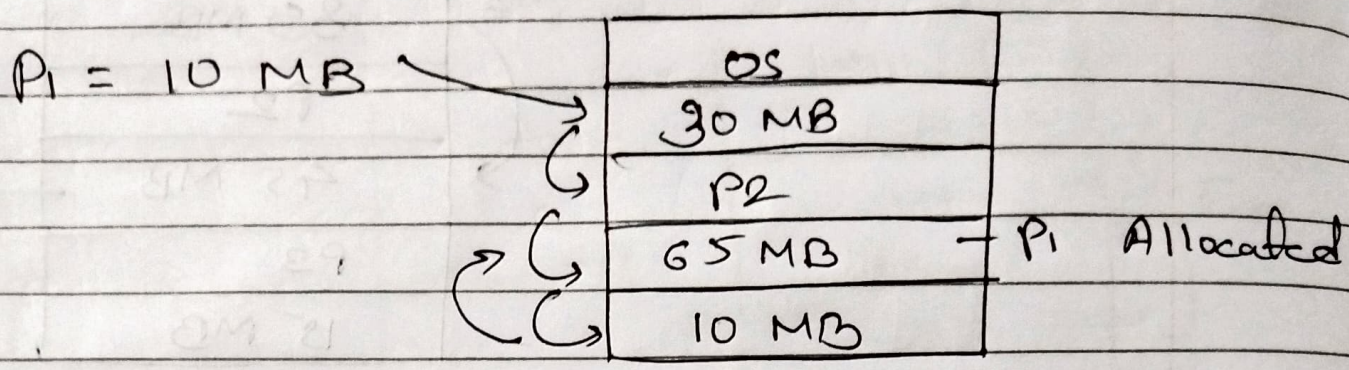
iii) Best fit: Allocate the smallest hole that is big enough, the entire list of holes has to be searched

P_1 = 20MB



iv) Worst fit: Allocate the largest hole. The entire list has to be searched for worst fit too. The leftover hole is the largest, which can be better than many

smaller leftover holes.



Simulations show that first & best fit are better than worst fit for storage & time utilization. Among Best fit and ~~worst~~ first fit, first fit ~~but~~ is faster.

3) Fragmentation:

- External Fragmentation:
As programs are loaded & removed from the main memory, there are many smaller holes formed. These holes might not be big enough to hold processes individually, but ~~aft~~ the size of all the holes combined may be enough to fit more programs.

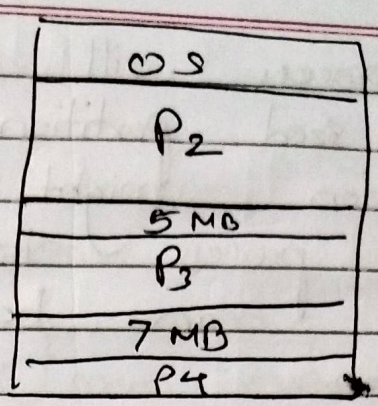
But these smaller holes cannot be used because the process has to be allocated contiguously, & cannot be divided into parts that are stored separately.

One solution is ~~to~~ Compaction.

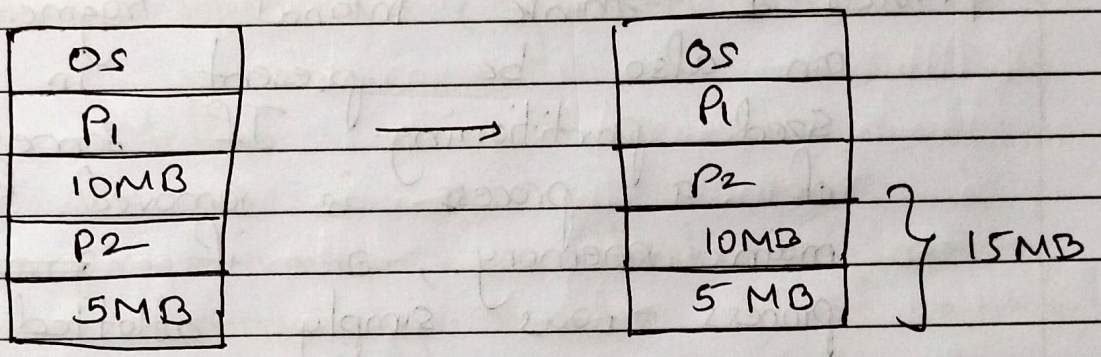
→ P1 = 10MB

↓

Can't be allotted



❖ Compaction: One solution is to shuffle all the memory contents so that the free memory are together and the user programs are kept separately from the free memory.



This reshuffling isn't always possible. It's only possible if the relocation is dynamic & done at execution time, and it's not possible if the relocation is static & done at load time.

Compaction is also extremely costly & has great overhead.

- Internal fragmentation: The problem of internal fragmentation arises in fixed size partitioning.

A process will be assigned to a fixed sized partition, but the size of this partition might be ~~too~~ bigger than the process itself; but no other process can be fitted into the partition.

The difference between the size of the ~~portion~~ partition & the size of the process ~~might~~ is the Internal fragmentation.

#Note: I think internal fragmentation can also be present in variable sized partitioning. If some part of a process is removed from the main memory, or if ~~some~~ the process was simply allotted ~~a~~ ~~lote~~ ~~par~~ more memory than it needed, then internal fragmentation can happen in variable sized partitioning too. I could be wrong.

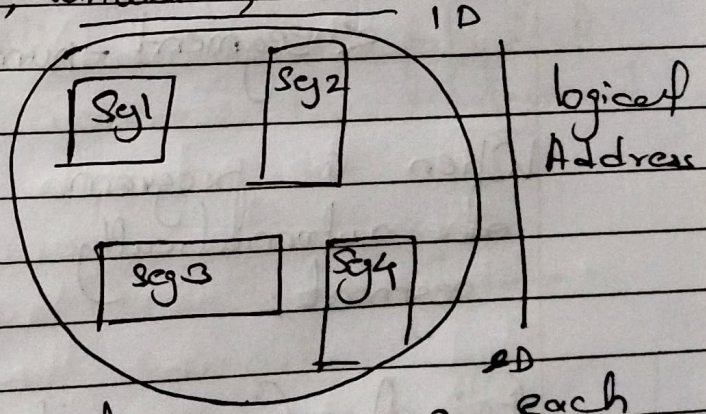
Segmentation

Segmentation was introduced because the system would have an easier way to access memory and the programmer would have a more natural environment while programming.

Segmentation makes it possible for the system & the programmer to access data from memory without needing to know the exact location of that data in the memory (eg: 400300)

1) Basic Method

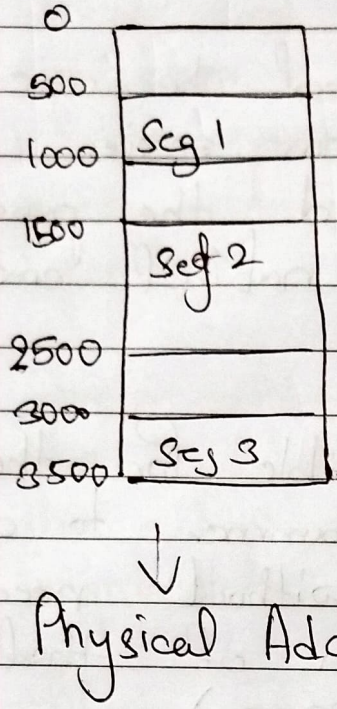
While programming, we think of memory items as containing variable sized segments, these segments could be arrays, stacks, libraries, variables, functions and so on.



What we think:

They are just existing in memory, with no particular order among each other. We don't think about where in memory they are placed. But this is not how memory is actually arraged. Instead of 2 dimensional arrangement of segments, the memory is

1 dimensional (linear) array of bytes:



But this realistic ~~vei~~ view of the memory is not needed by the programmer.

Segmentation is a memory management scheme that supports this programmer's view of memory.

↓

Physical Address

* Attributes of a segment:

Each segment has a name / ID and a length. The length is used to ~~find~~define the range of the segment.

Thus, the logical address consists of 2 tples

< segment-number, offset >

When a program is compiled, the compiler ~~at~~ automatically starts constructing segments.

Eg: A C compiler can create the following segments:

a) ~~Co~~ The Code
b) Gbbal Variables
c) The heap
d) The stacks used by each thread
e) The standard C library

Libraries linked at compile time and at ~~eex~~ execution time are separately segmented.

The loader takes all the segments & assigns them ~~a~~ ~~ss~~ segment numbers.

e) Segmentation Hardware:

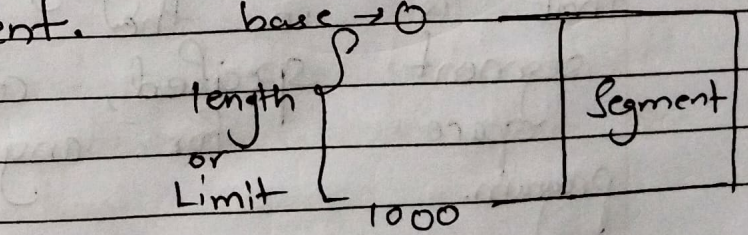
The programmers refers to the memory ~~using~~ using 2-dimensional addressing (name, length), thus we need a way to map these 2 dimensional addresses to the actual 1 dimensional memory.

This mapping is done using a Segment Table.

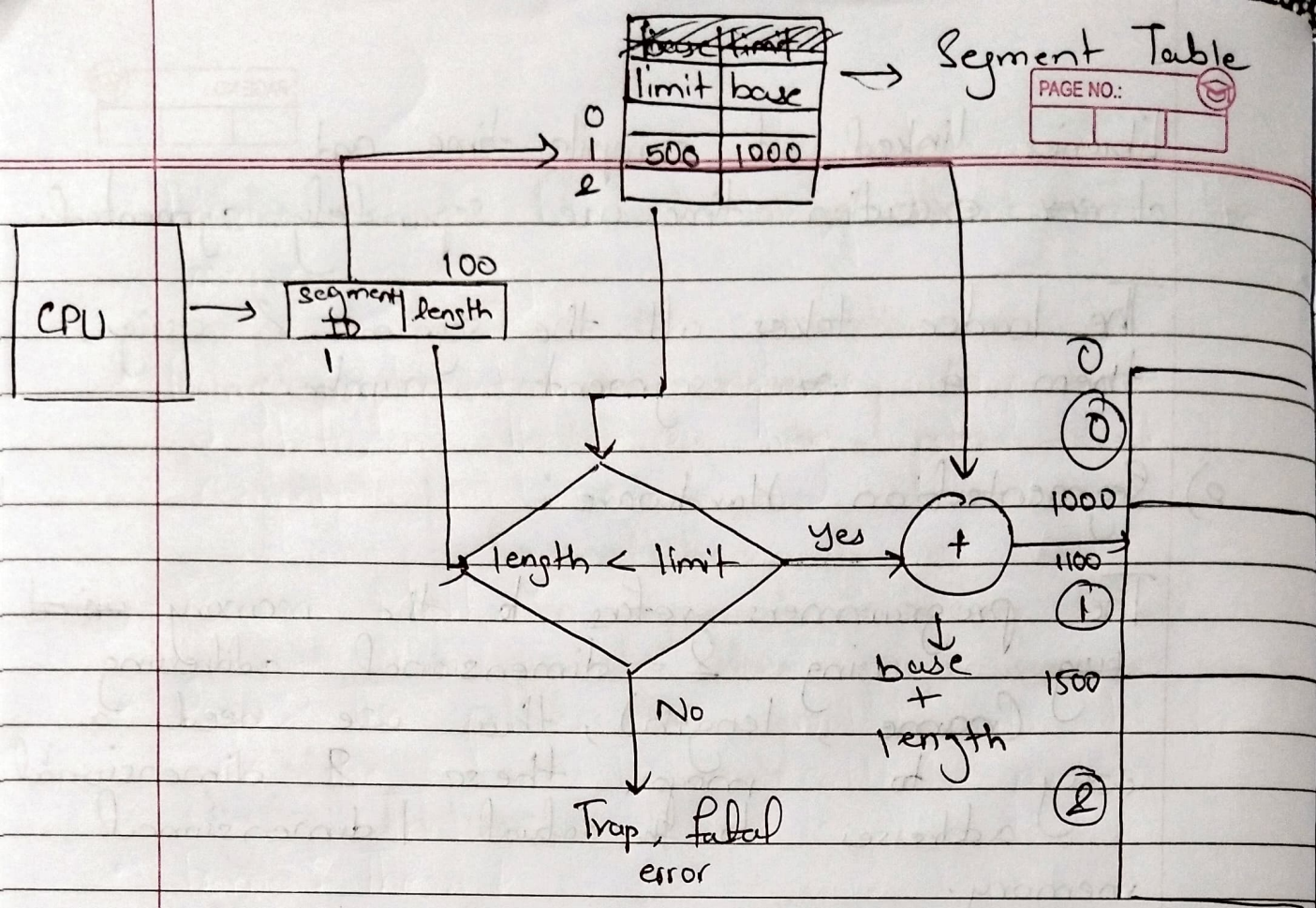
The segment table contains 2 columns, segment base and segment limit.

Segment Base: Defines the starting physical address from where the segment starts.

Segment Limit: Defines the length of the segment.



For example, let's say that a ~~us~~ ~~prgrm~~ programmer tries to access the 100th



element of segement ID 1. Thus the input from CPU is <1,100>. The segment ID is searched for in the segment table. The limit for the corresponding segment ID is compared with the length given by the user (100, for accessing the 100th element).

If the length is smaller than the limit, then the access attempt is safe, meaning the user is trying to access an element that is in the bounds of the segment specified, and isn't the memory space for any other segment or program.

If the length > limit, the access is unsafe, & a trap is sent to the

system, giving a fatal error.

For the example, the length (100) is smaller than the limit (< 500), thus the flow proceeds.

The value of length is added to the base to access the desired element.

If the length was 800, 800 > 500, resulting in the trap.

Figure 8.9 Example of Segmentation

Segmentation allows for non contiguous arrangement of processes in the physical memory. The process can now be divided into parts.

Paging:

- Paging allows for non-contiguous arrangement of processes. a process, like segmentation does.
- Paging also gets rid of the problem of external fragmentation, & thus the need for compaction.
- Paging also solves the problem of fragmentation issues in the backing store, as the

backing store has similar problems as the main memory.

- Paging implementation requires cooperation between the operating system and the computer hardware.

1) Basic Method:

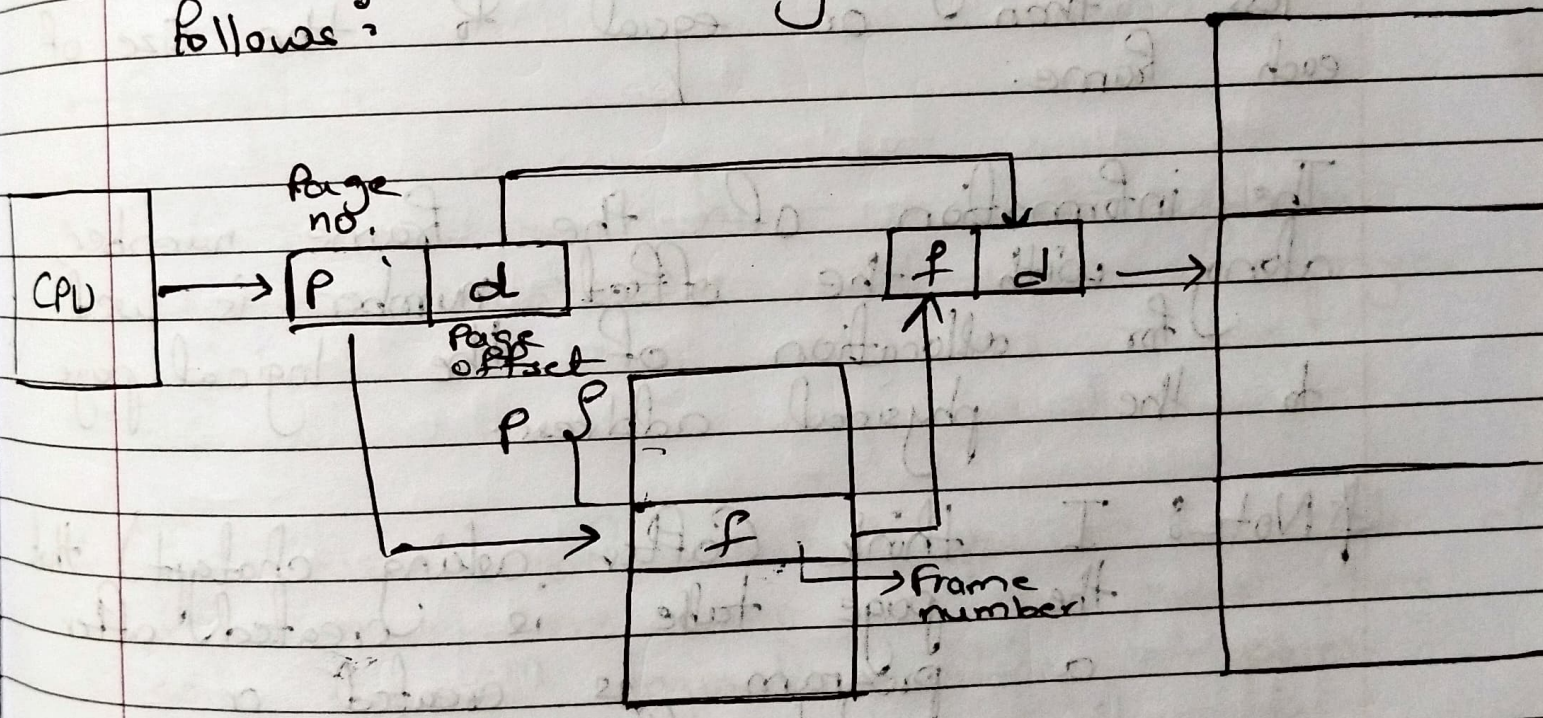
- ~~Pages: The process is divided into blocks of the same size called pages.~~
  ~~This~~
- ~~Frames: The~~

- Pages: A process in the logical memory is divided into small blocks of same size called pages.
- Frames: A process in the physical memory is divided into small blocks of same size that is equal to the size of the pages, called frames.
- A page from the logical memory is mapped to a frame from the physical memory. This allows for complete separation of the logical memory space and the physical memory space, meaning easier access to the physical memory by the system.

- The backing store is also divided into fixed size frames, which are either the same size as the frames in physical memory, or the size of clusters of frames in physical memory. Thus, a process can be bigger in size

- All addresses generated by the CPU have 2 parts: the page number and the page offset.

The hardware design for this is as follows:



Pages for a process is created along with a page table for that process. The CPU generates a page number for the page along with a page offset. The page no. generated is looked up in the process's page table.

* Page Table: The page table is created for every process in the logical memory. The page table has 1 row for each page created for that process and a frame number is given for these page numbers, which points to the frame in physical memory where the page can be mapped.

The page offset gives the length of the page, this length should be less than or equal to the size of each frame.

The information of the frame number, along with the offset number is used for allocation of the logical page to the physical address.

* Note: I think (After asking chatgpt) that the page table is created after a program is created or is ready for execution.

The page table is initially empty. As the process starts execution, & tries to fetch memory, the page no. are allotted a frame number by the OS.

Meaning the ~~data~~ page number in which the data was residing is given a frame in the physical memory, populating the ~~phy~~ page table 1 by 1 by doing so.

- ~~Example:~~
- ~~If each page has n bit address space, then the total no. of pages~~
- ~~If~~
- The size of each page is an order of 2. Given there are m pages, there will be 2^m logical address spaces. Given there are n bits in addressing for each entry in a page, the total size of the page is 2^n (because ~~th~~ for n bits, there are 2^n possible addresses, because these bits will either be 0 or 1).

- Example: There are 4 logical pages with 4 entries each. There are 8 physical frames with 4 entries each.

For the first logical page (at index 0) the frame number given in the page table is 5. Thus, the 1st 0th page will be mapped to the 5th frame.

Let's say we want to access the 3rd element in page 0.

The page table says that the data is stored in frame 5.
Now each frame till page 5 will have 4 bytes of storage (as each page in the logical memory also has 4 bytes of storage). Thus, to reach frame 5, the byte no. will be 5 × 4, 5 because there are 5 frames before frame 5 (0, 1, 2, 3, 4) and 4 because each frame is 4 bytes long.

Now, the 3rd entry of page 0 (or 2nd index of page 0) will be at an offset of 2 from the first address of frame 5 (20). Thus, 2 will be added to the

first address : $5 \times 4 + 2 = 22^{nd}$

- Paging is similar to dynamic relocation as it also bounding a logical address to a physical address using a table.
- Paging gets rid of external fragmentation as any free frame can be utilized by any other page.
- Paging isn't free from internal fragmentation. A process doesn't usually require all pages & frames with full utilization of each page / frame.

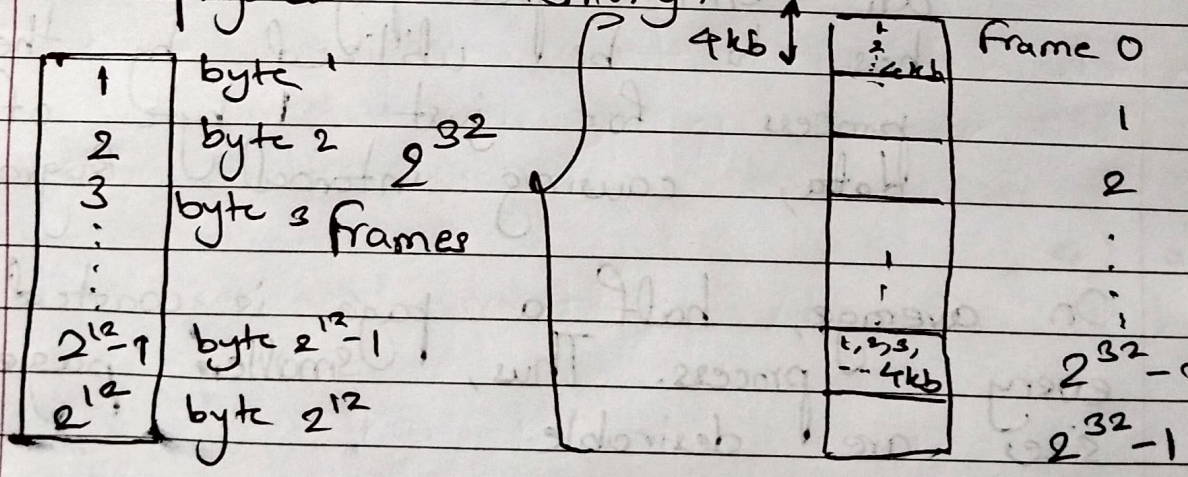
  Eg: If a process's size is $(n+1)$ where n bytes is the size of each page, then an entire part page & frame has to be utilized by the process for just 1 byte of data, causing internal fragmentation.
- On average, half a page is wasted for every process. Thus, smaller page size are desirable.
- With increase in the size of main memory and data processes, the size of pages have also increase. Some operating systems also support variable size pages.

Eg: Solaris uses pages of sizes 4kb & 8kb.

- For a 32-bit CPU, with ~~page table~~ entry 4 bytes long, the total no. of physical frames that can be pointed will be 2^{32} (each of the 32 bits will either be 0 or 1).

If the frame size is 4kb = 2^{12}, then the system can address $2^{12} \times 2^{32} = 2^{44}$ bytes or $2^{44}/2^{40}$ $= 2^4 =$ ~~32~~ 16 TB of physical memory.

Note: We multiplied the total number of frames with the ~~total~~ size of each frame to find the value of total addressable physical memory.



∴ Total Addressable memory

$= 2^{12} \times 2^{32}$

$= 2^{44}$

- When a process enters the system, the pages of that process are loaded onto the frames 1 by 1. If there are n pages in the process, then there should be at least n frames, ~~for~~ & in the memory.

The page table is updated as each page is allotted a frame.

- The logical address & the physical address are completely separated out in ~~from~~ paging, as there is no way for a process to access frames that are outside its page table, meaning the programmer has no way to access the physical memory directly & hence doesn't have any chance of retreiving data ~~from~~ frames outside the page table.

- Frame Table: The OS manages a table to keep track of which frames are allocated & which frames are available. Also the total no. of frames in memory.

  There is 1 entry in the frame table for each process, storing information about if it's free or allocated. If allocated, to which page of which process.

- In paging, the OS manages the page table entries, but the hardware has to do the address translation from logical address to its corresponding physical address.

- However, the OS also keeps a copy of the page table for each process. This is done so that whenever needed, the OS can manually do the translation of the logical address to the physical address without needing hardware support.
  Eg: When a process performs I/O & generates a new address to store necessary data related to the I/O.

  This copy is also used by the CPU dispatcher to define the hardware page table when the process is about to execute.
  Thus, the ~~cote~~ context switch time is increased in paging because of this extra data structure.

2) Hardware Support:

- Every OS has its own methods for storing the page table. Some OS allocate a page table for each process & some others use only one or only a few page tables.

- In the process control block, a pointer to the page table of the process is stored with the other registers.

Before execution, the dispatcher reloads this PCB values & defines the correct hardware page table for the process.

- One example of the hardware implement implementation to support page table is using a high-speed register to store the page table.

These registers are required to be fast so that the address translation is fast. This is extremely important as each memory access goes through the paging table.

- Eg: In DEC PDP-11 architecture, there are 16 bits in the address & each page is 8kb in size (8kb = 2^{13} Bytes)

∴ Total no. of page table entries required = $\frac{2^{16}}{2^{13}} = \frac{\text{(all possible address)}}{\text{(size of 1 page)}}$

$= 2^3 = 8$ page entries.

Thus, DEC PDP-11 uses high speed registers for memory access.

- Hoeever the use of high speed registers is only satisfactory when the page table is relatively small.

Modern computers have page tables with millions of entries though. Thus, fast registers are not feasible in these systems.

- Page Table Base Register (PTBR):
  The page table for some systems are kept in the main memory.
  The PTBR for each process points to its page table in main memory.

The context switch time required for this register is ~~extra~~ small.

* The Problem with PTBR:

Now, access each byte of main memory requires 2 main memory accesses. 1 to find the page table entry for that page & the 2nd to find / locate the frame number given in the page table.

Thus a memory access for each by byte is increased by a factor of Two!

Translation Look-Aside Buffer (TLB):
TLB is a high-speed, associative memory (meaning that the search in this memory is done by searching for the data, not by direct addresses).

Each entry has a key & value: An item presented to the TLB will be compared with the key values of each entry & if a match is found, the value will be returned.

This TLB lookup step is done in the instruction pipeline itself, thus it needs to be small in size. Often time TLB is divided & multiple levels of TLB is used.

✡Working: TLB is like cache memory it contains some page table entries. The 'item' discussed above is the page. When the logical address is generated, its page number is presented to the TLB.

If the page number is found in the TLB, the frame number is immediately returned. (TLB hit)

If the page number is not found, a memory reference to the page table is

is made either automatically by the hardware or using an interrupt to the OS. The frame no. is obtained from there. (TLB miss)

This new entry will also be added in the TLB. If the TLB is already full, it will use replacement algorithms like LRU, depending on the CPU.

Some entries in the TLB are wired down (cannot be replaced).
Eg: Kernel code parts

- Address Space Identifiers (ASIDs):
The ASID uniquely identifies each process. ASID is stored in each TLB entry.

Whenever the TLB tries to find the page, it makes sure that the ASID of the currently running process is same as the ASID associated with the page.

ASID provides protected access and also allows the TLB to store entries for different processes. Without ASID the TLB would have to be flushed to avoid data corruption.

- TLB lookup is extremely fast as page table lookups (which are stored in the main memory) is ~~also~~ ~~too~~ high. TLB is a part of the instruction pipeline itself.

- Hit Ratio:
The percentage of times that a page number of interest is found in the TLB is called the hit ratio.

If desired page is found in the TLB then only one memory access is needed, let's assume that this memory access takes 100 nano seconds.

If the page is not found in the TLB then ~~first~~ first the page table in the main memory will be accessed & the frame number retrieved will be ~~compared with~~ accessed in the main memory, thus requiring 2 main memory accesses, 200 nanoseconds.

* Let's assume 80% hit ratio. Meaning for 100 page nos, 80 page numbers are found in the TLB & the other 20 are ~~lost~~ not found, needing double memory access.

∴ Effective memory access time

nanos ↑ ns ↑

$= 100 \times 0.80 + 200 \times 0.20$

$= 80 + 40$

$= 120$ ns to access memory

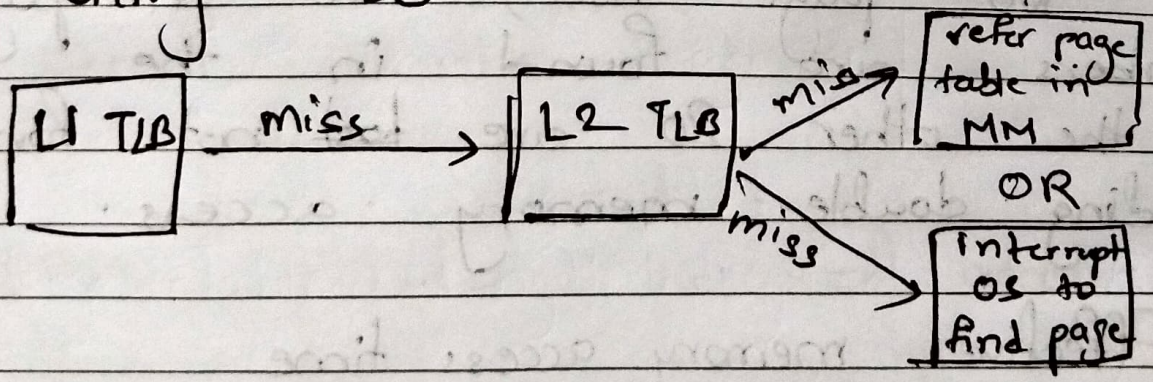
→ In a more realistic scenario the hit ratio would be 99%

∴ effective memory access time
= 0.01 × 200 + 0.99 × 100
= 2 + 99
= 101 ns to access memory.

- This calculation of memory access time would not be simple for modern CPU, where multiple levels of TLB are used.

Thus a complete analysis of such a system would require hit ratio for all different TLB levels.

Intel Core I7 CPU has a 128-entry L1 instruction TLB & a 64-entry L1 data TLB, along with L2 512 entry TLB.



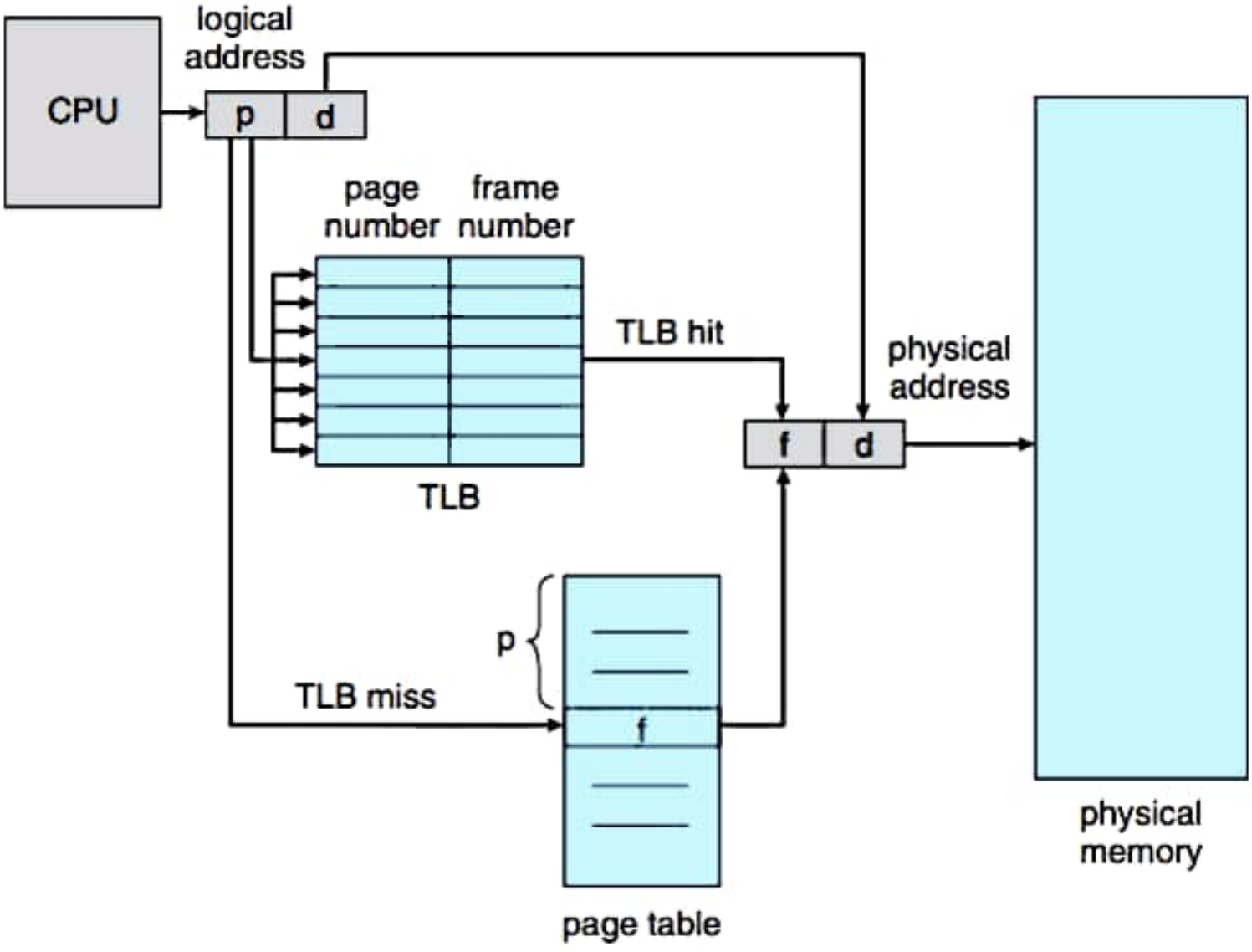


Figure 8.14 Paging hardware with TLB.

3) Protection:

- Memory Protection can be achieved using protection bits which are kept in the page table.

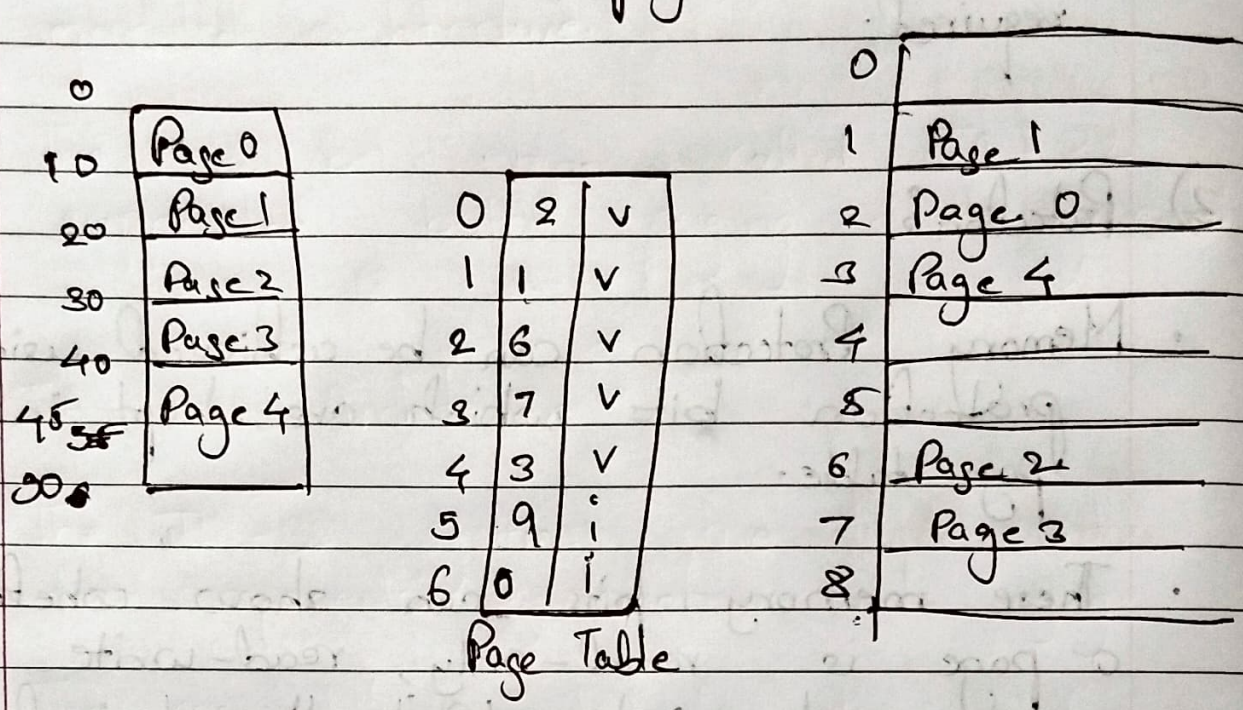
- These memory bits can show whether a page is read-only, read-write or execute-only. While the logical to physical mapping is being done, it can be checked if no writes are being done to read-only page.

- Another additional bit is attached to the page table entries called the valid-invalid bit. This bit can take up 0 for invalid & 1 for valid for each page.

- For a process with address range 0 to 45 ~~55~~ kb, with each page of size 10kb, there would be 5 pages required: Page 0, 1, 2, 3 and 4.

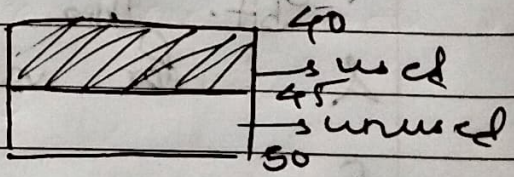
Note that the 4th page will only be half utilized.

- Thus the valid-invalid bit will be set to valid for pages 0, 1, 2, 3 & 4 and invalid for pages 5 onwards.



- But the 5th page (or page 4) is only halfway in the valid address range, but it still has be considered fully valid.

- This problem occurs due to internal fragmentation



- Page Table Length Register (PLTR):

This register ~~indicated~~ indicates the size of the page table. This size can be used to check whether a logical address is in the legal range for the process.

4) Shared Pages:

- In paging it is possible to share common code.

- In a time sharing environment, there are 40 users using the text editor. The text editor has 50 kb of data space (space that can be changed by the user) and 150 kb editor code, which is same for every text editor.

- The total memory requirement for all 40 users could be $40 \times 50 + 40 \times 150 = 8000$ kb

- The text editor's code (150 kb) however is ~~reentrant~~ a pure code, which can only be read. Thus, this code can be shared. Thus, all 40 users can use only one copy of this text editor code

- ∴ Total space taken up $= 40 \times 50 + 1 \times 150 = 2150$ kb, which is much smaller than the previous 8000 kb.

- Other heavily used programs can also be shared: Compilers, Database systems & so on.

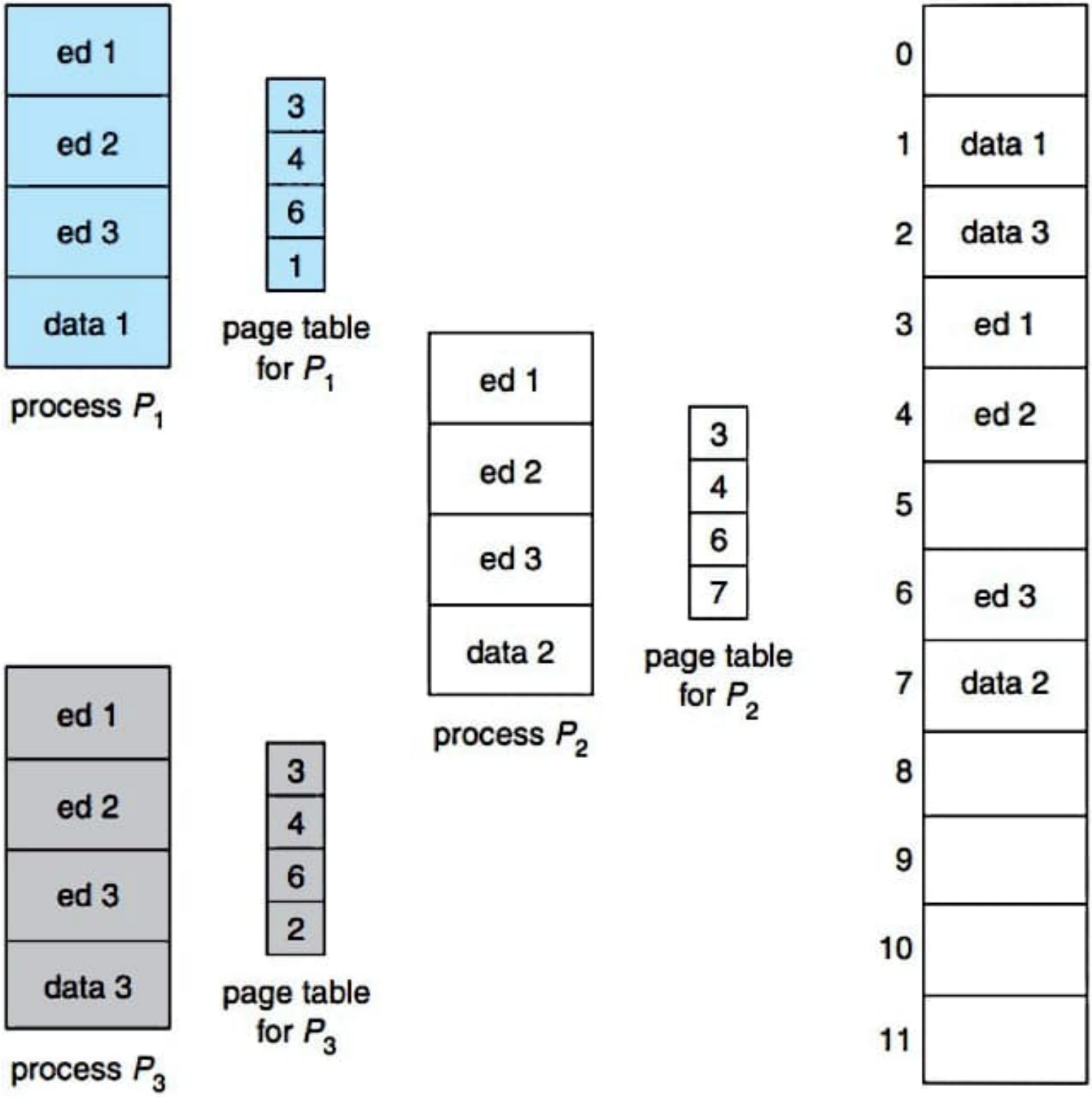


Figure 8.16 Sharing of code in a paging environment.

5) Structure of the Page Table

a) Hierachical Paging

- In modern computer systems, the size of the page table ~~does~~ not is too large to be stored directly in the main memory, as they can contain upto millions of entries.

- One solution is to use the two level paging algorithm, in which the page table itself is also paged.

Eg: A 32-bit logical address & a page size of 4kb, the address is divided like so:

Page No.	Page offset
20 bits	12 bits

If the page table is also paged, the division can be:

P_1	P_2	Page offset
10	10	12

P_1: Index to outer table
P_2: Index of inner table

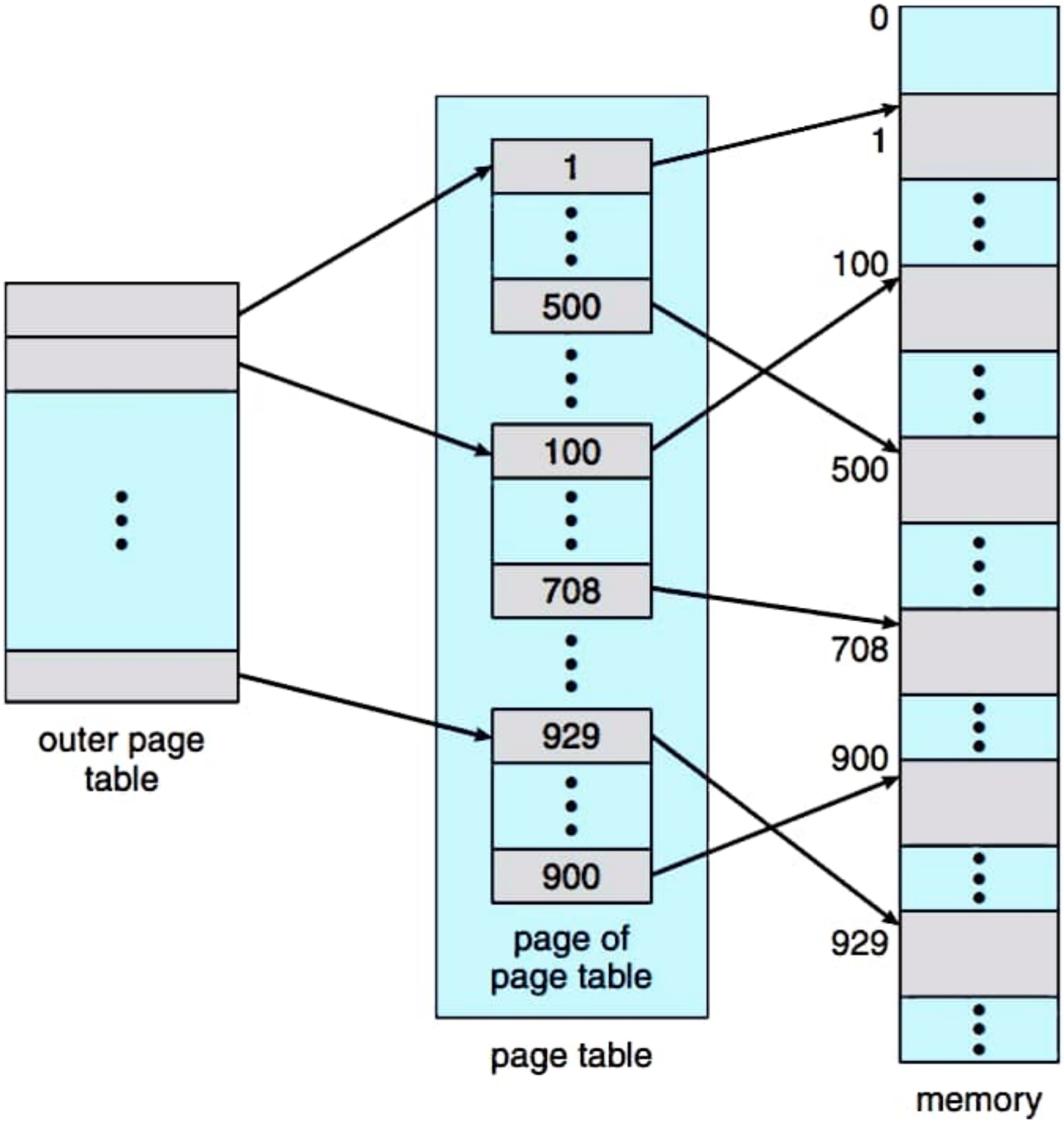


Figure 8.17 A two-level page-table scheme.

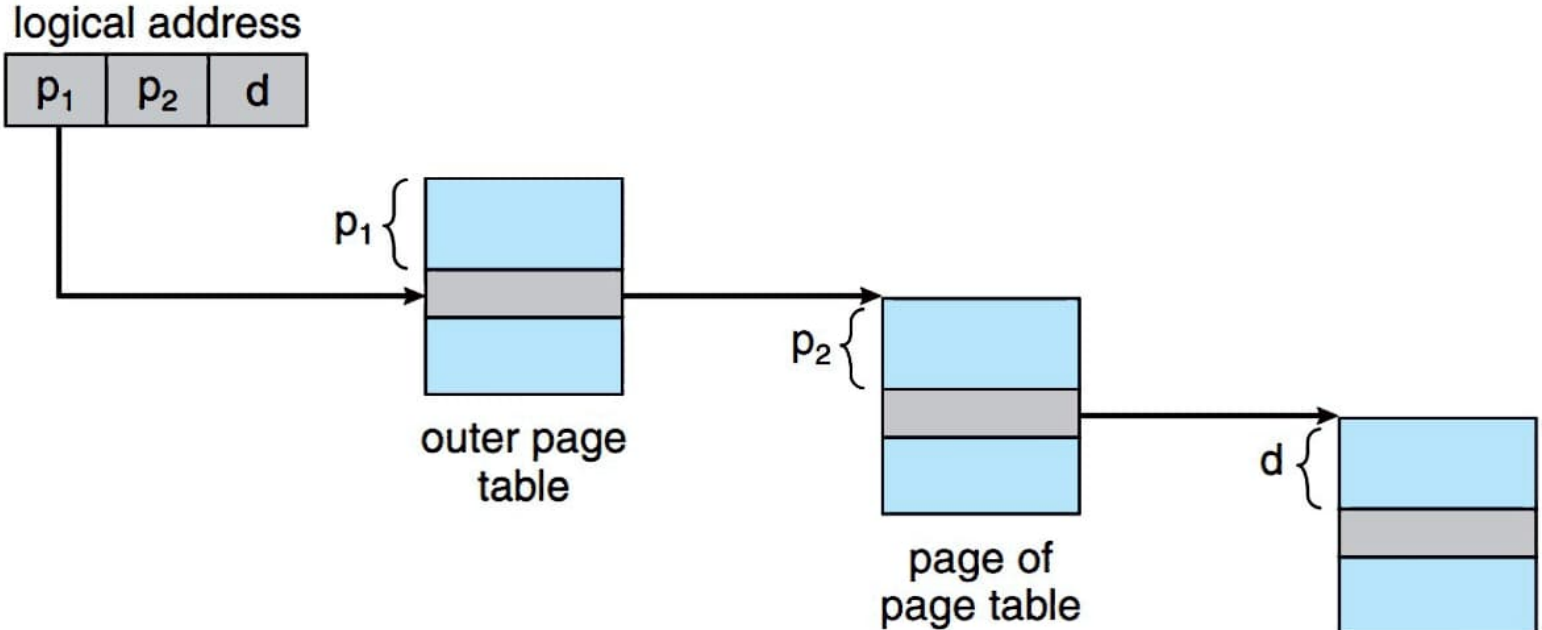


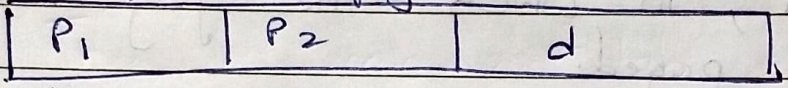
Figure 8.18 Address translation for a two-level 32-bit paging architecture.

This scheme is called "forward-mapped page table" because address translation works from outer page table to inwards.

- Although hierarchical page scheme is good for 32 bit addressing, but not for 64-bit addressing systems.

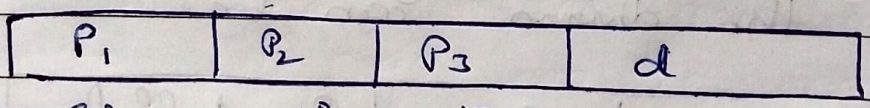
- If 2 level page scheme was used for 64-bit addressing, the division of address will be:

outer page | inner page | offset



The outer table consists of 2^{42} entries! Table too large & defeats the purpose of dividing.

- One solution is to ~~avoid~~ divide the page table into more layers. The address division would look like:



The page table size of the outermost table is still too high.

- Adding more & more layers will just add more overhead. Thus, hierarchical paging is not appropriate for 64-bit addressing.

b) Hashed Page Tables:

- Linked lists are used in this architecture. The virtual address of a page is hashed. The pages with the same hashed value form one linked list.

- A Hash table is used, in which all hashed values listed out. Some or all of the rows in the hash table can contain the above mentioned linked list.

- Each element (and node in the linked list) have the following fields:

Virtual page number	Value of mapped page frame	a pointer to the next element in LL

- Working:

→ A virtual address obtained is hashed and the row in the hash table with this hash value is searched.

→ If found, the first element in the linked list is checked. ~~If it doesn't match~~ first element, the virtual page number, is compared with the given virtual page number.

→ If they match, then the frame number and offset are used to find the frame data.

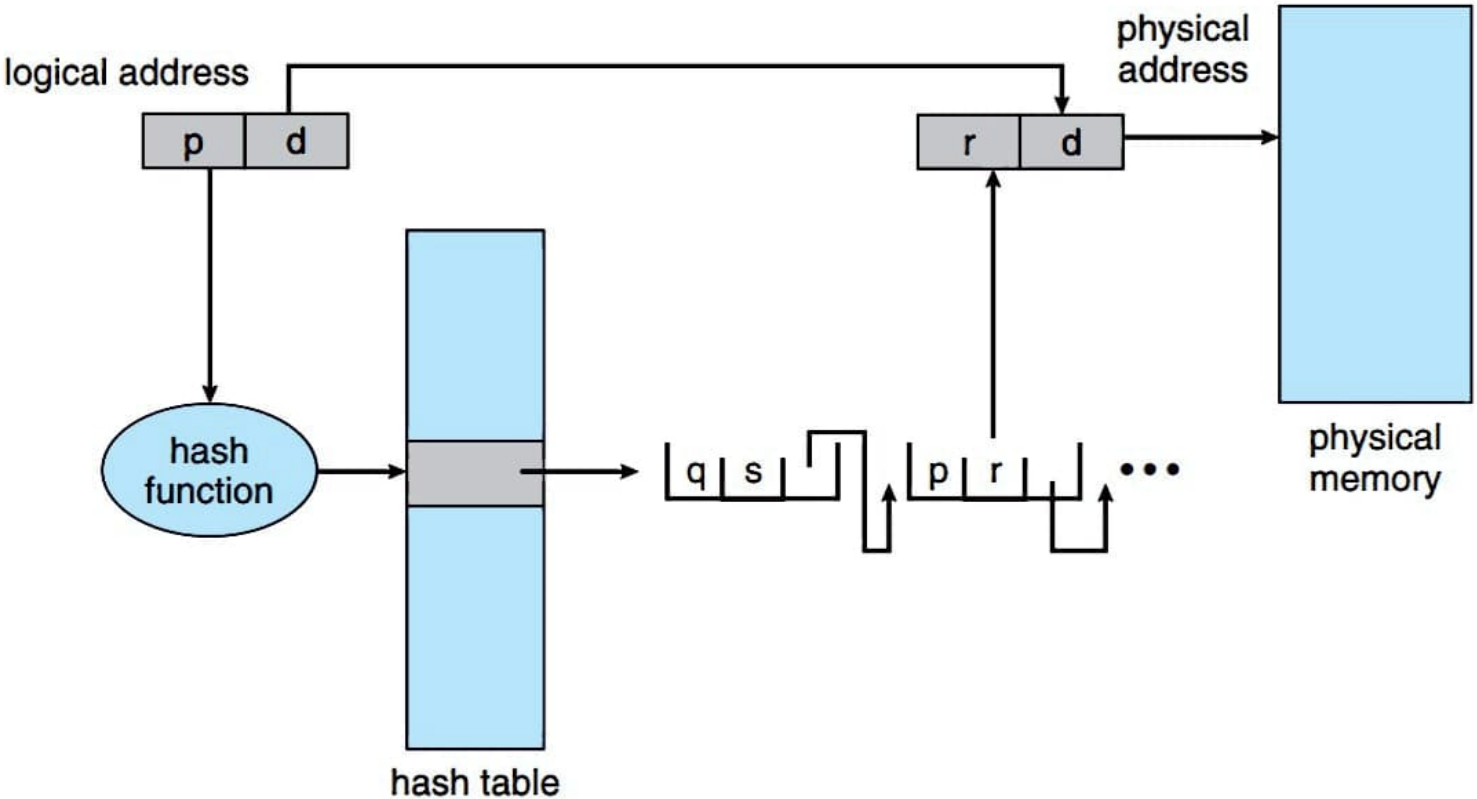


Figure 8.19 Hashed page table.

If not, then the next element in linked list is searched in the same way.

- Clustered Page Tables: Each entry in the hash table refers to several pages, not just one.

  Thus, a single page table entry can store mappings for several physical page frames. This variation is thus most used for 64-bit addressing.
  Most useful for sparse address spaces, where memory references are noncontiguous & scattered.

Note:
In hierarchical paging, the outermost page table is stored in the main memory completely. As pages from the inner page table are getting requested, the page table dynamically adds entries (to map a page to the frame) and is added in the main memory. Thus, only the demanded page table entries are called in main memory. Note that this is different from demand paging, as the pages here are all loaded into the main memory, only the page table is getting dynamically updated.

c) Inverted page tables

- Usually, each process has its own page table that it references to find its frames, but this method has a drawback.

- Each page table may be millions of entries big. The page table needs to cover the entire virtual address space, because searching in page table is reliant on the fact that the entries in page tables are sorted.

- Thus, the page table must contain an entry for all page, even if that page isn't in main memory (you will see how this is possible in demand paging environments).

- In Inverted Page Tables:

→ Only pages that have been mapped to a frame in main memory are added in the page table.

→ Each page table entry contains the process that the page belongs to.

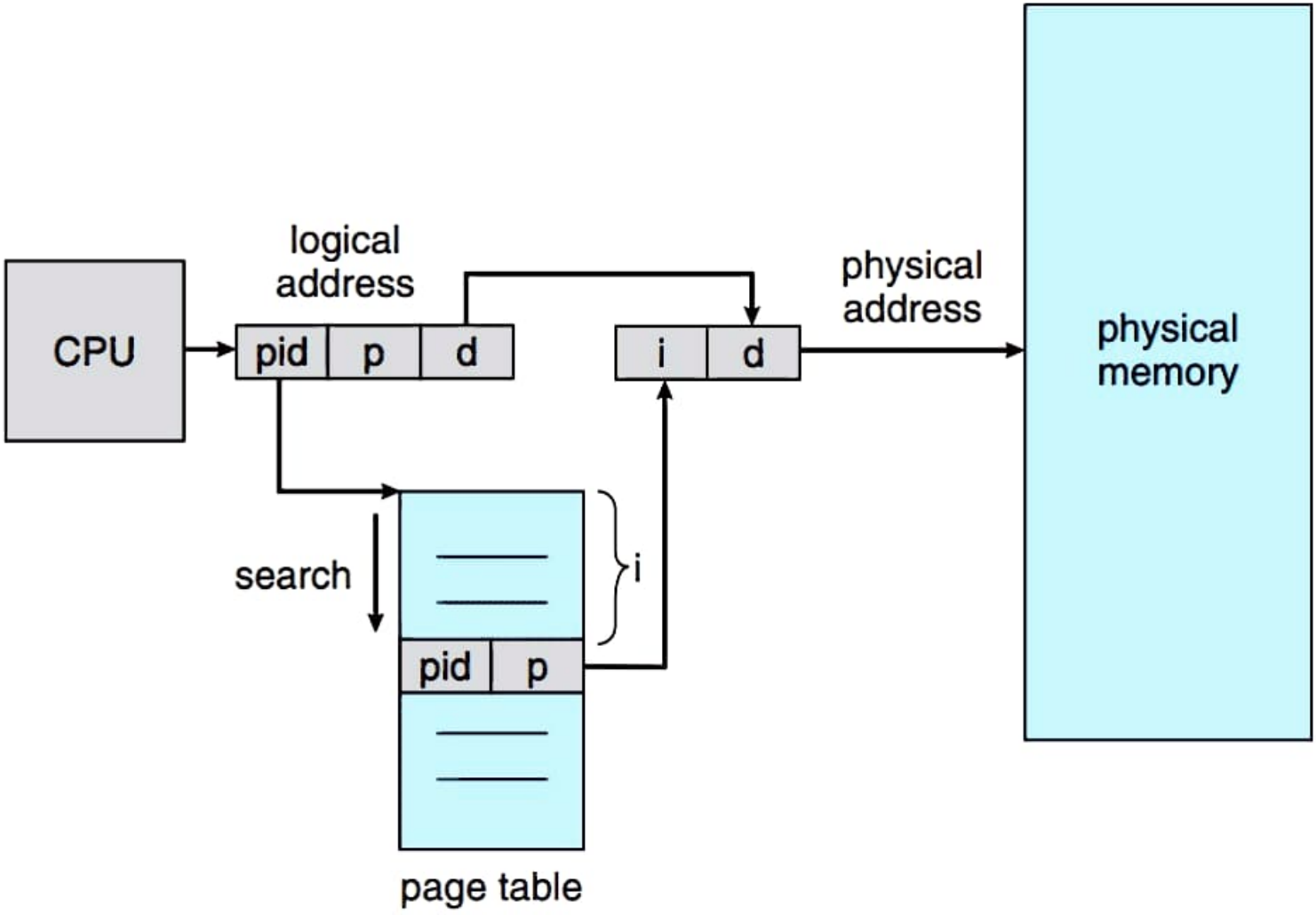


Figure 8.20 Inverted page table.

→ There is only one page table in the entire system.

→ The virtual address is in the form

<process_id, page-number, offset>

* When memory reference occurs the part of virtual Address <process-id, page-number> is given to the system.
* The inverted page table is searched with <process-id, page-number>
* ~~If found, then th at lets say i;~~
* If found, at lets say entry i, the physical address is calculated using <i, offset>

- Drawback is that the time needed to search the table has increased.
- To solve this issue, the hash table can be used, which can slightly speed up the searching.
- Another drawback is that shared memory ~~tables~~ cannot be easily implemented, as each <process-id, pag-number> can ~~access~~ only be mapped to a unique physical address. (entry i is different for all)

32 and 64 bit architecture examples

- Intel 8086 and 8088 used segmented architecture.
- The 32-bit chips, called IA-32, supported both paging and segmentation.

1) IA-32 Architecture

- The IA-32 systems are divided into two components: segmentation and paging.
- The segmentation unit takes the logical address from the CPU and converts it into a linear address (similar to what actual physical memory is like) by creating segments of the given address.
- The paging unit turns the linear address to actual physical memory.
- Thus the segmentation & paging units create what MMU is, effectively.

{{{ figure 8.21 logical to physical }}}

a) IA-32 Segmentation:

- The segment size in IA-32 architecture can

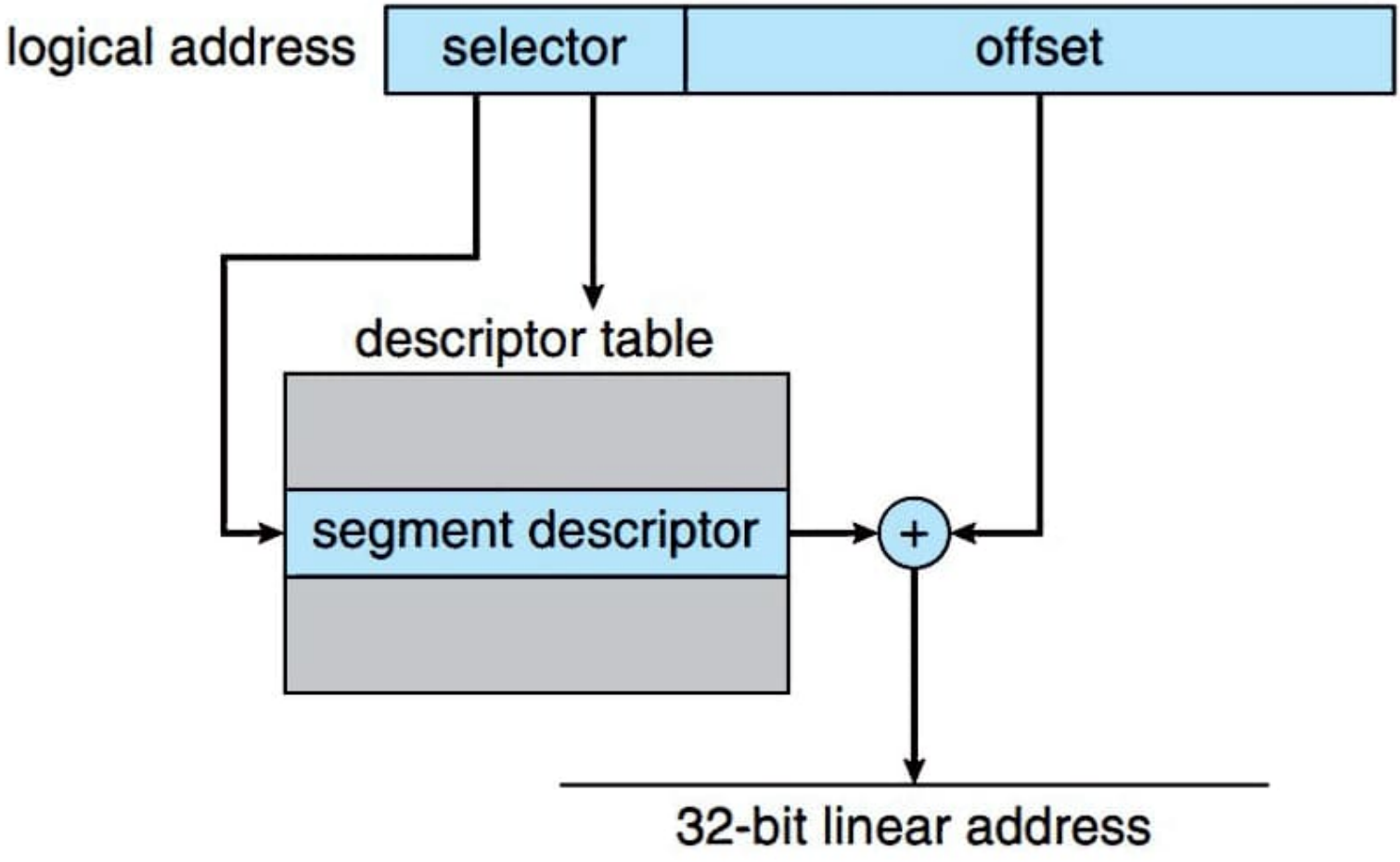


Figure 8.22 IA-32 segmentation.

be the size upto size 4GB and the maximum number of segments per process is 16k.

- The logical address part of the address is divided into two partitions, each of 8k maximum segments.
- The first partition is private to the process while the 2nd partition is shared among all the processes.
- Information about the first segment is stored in Local Descriptor Table (LDT)
- Information about the second segment is stored in Global Descriptor Table (GDT).
- Each of the segments in the LDT & GDT table have 8 byte segment descriptors. This descriptor gives information about the segment like the location & limit of that register.
- The logical address is divided into components: The selector & the offset.

The selector tells which segment we are in & the offset tells the location of the byte

within the segment.

- The selector is a 16-bit number with
  - → first 13 bits for designating the segment number.
  - → Next 1 bit to tell whether the segment is in LDT or GDT
  - → Last 2 bits to deal with protection

Segment	Table	Protection
13	1	2

- IA-32 has 6 segment registers, allowing for 6 segments to be accessed by a process at one time.
- A segment register includes all the necessary information needed to retrieve all data related to a segment.
- The linear address formed by the segmentation is 32-bit long.
- Working:
  - → The segmentation register points to the appropriate entry in GDT or LDT.

→ The base & limit information found in the table is used to form the 32 bit address.

→ First the limit is checked for address validity. If it's not valid, then error is thrown.

→ If the limit is valid, the the base & offset are added to create the linear address.

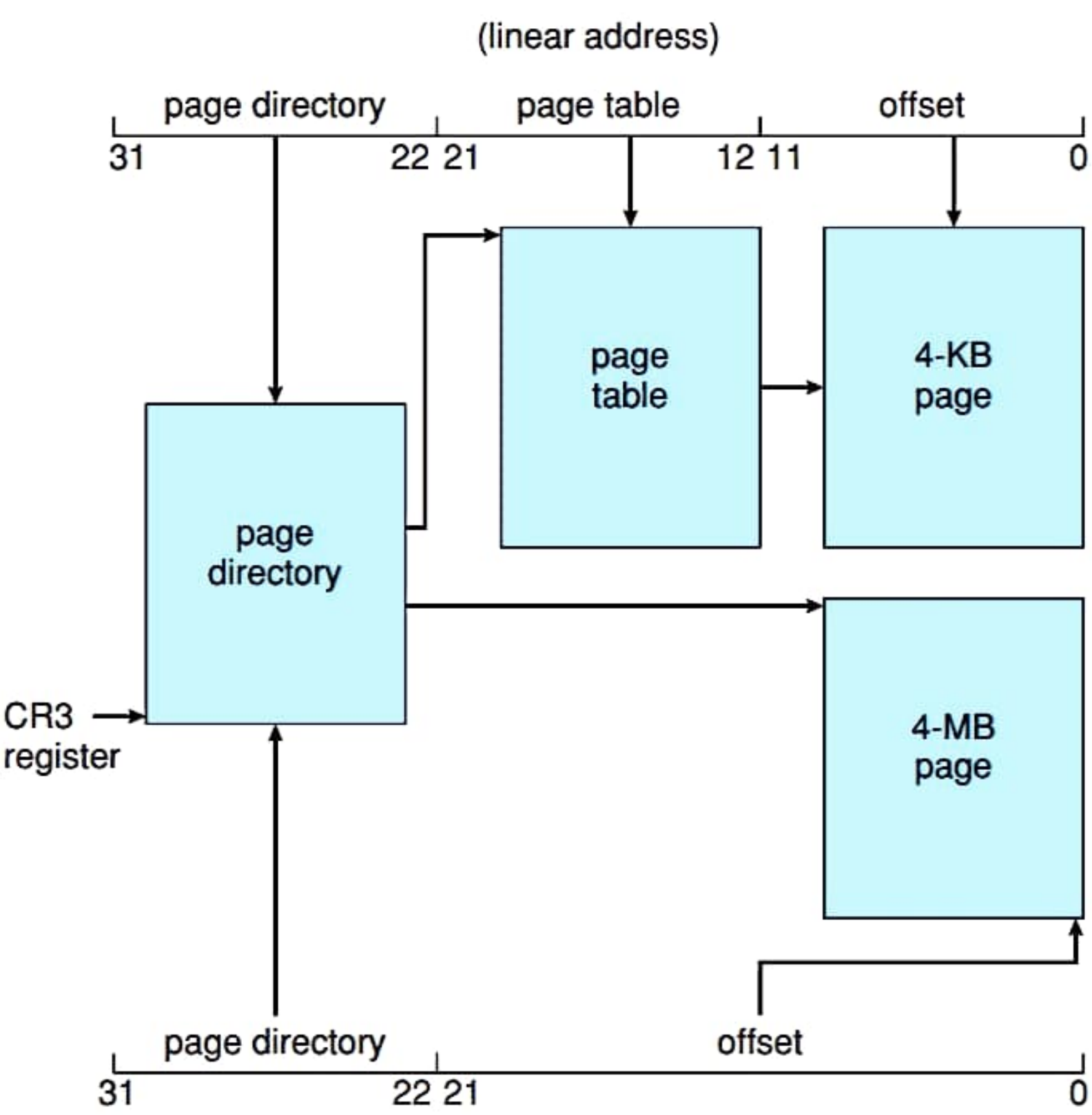


Figure 8.23 Paging in the IA-32 architecture.

- PAE increased the base address of page tables & page frames to extend from 20 bits to 24 bits. Combining with the 12 bit offset, the address space increased from 32 bits to 24 + 12 = 36 bits.

This increase allowed for up to 64 GB of physical memory.

- Before PAE, each segment could be of maximum size of 4GB. With PAE, since the overall physical size has increased, the size of each segment also increases.

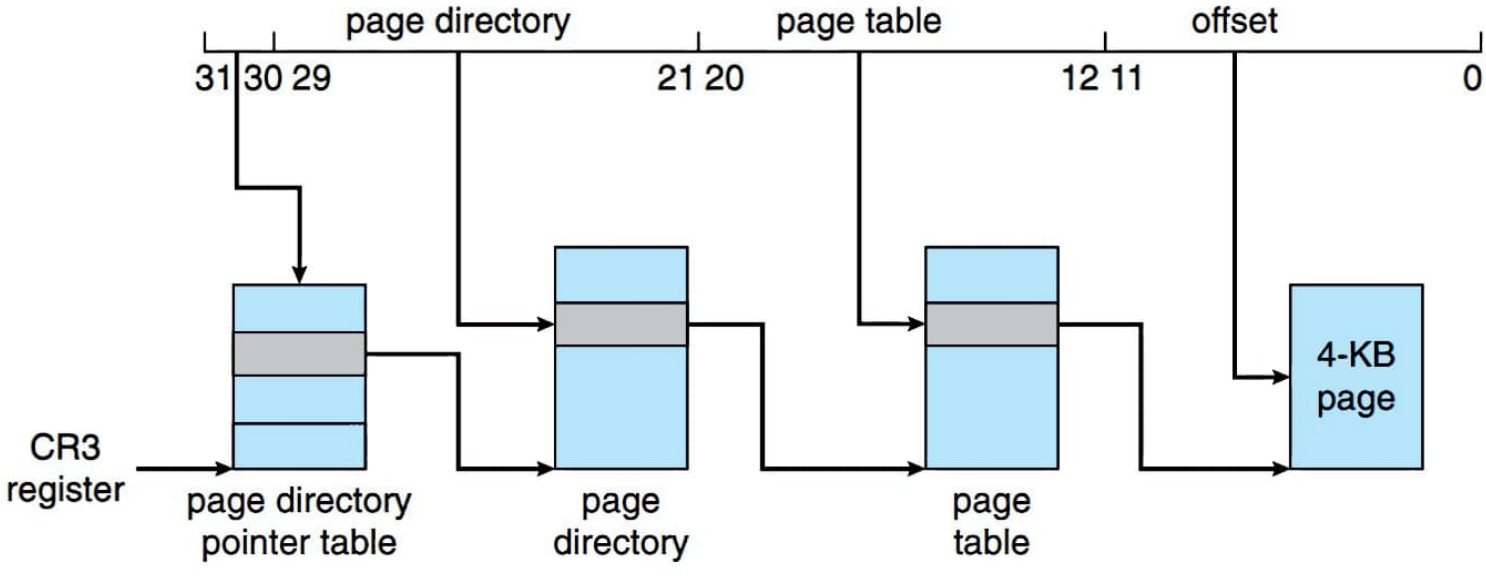


Figure 8.24 Page address extensions.

2) x86-64:

- Supports 64 bit address space, & thus total 2^{64} bytes of addressable memory. = (> 16 quintillion)
- However, a number much smaller than 2^{64} is used for addressing.
- Currently it provides a 48-bit virtual address & supports page sizes 4kb, 2mb and 1GB.
- There are 4 levels of paging hierarchy used.
- Since PAE can be used in ~~x86~~ x86-64, the addressable memory can be increased from 48 bits to 56 bits physical address.

5.2

Background

- As we have seen, it is required for the whole process to be main memory for execution, but this restriction (even though important) is unfortunate.
- That is because processes contain a lot of pages, data structures, error conditions, etc. that they usually do not need, and only exist for special cases.
- Thus a process can be rid of these pages, etc. before execution while it is in main memory. Thus, only partially in main memory.
- Thus, with virtual memory, user can create a process with a large virtual memory requirement without needing to worry about fitting it in the main memory.
- More programs can also be kept in the main memory if partial processes ~~or~~ are kept in main memory. Thus reducing context switch times overall.

Thus, both the system & the user would benefit from this system.

- Virtual memory thus allows for a further separation between programmer's view and the actual physical memory.
- A programmer can be given vast virtual memory, even if the physical memory is small.
- Virtual Address Space of a process: Refers to how the logical address of a process is stored in memory.
- The Memory Management Unit (MMU) maps the given virtual address to the physical address.

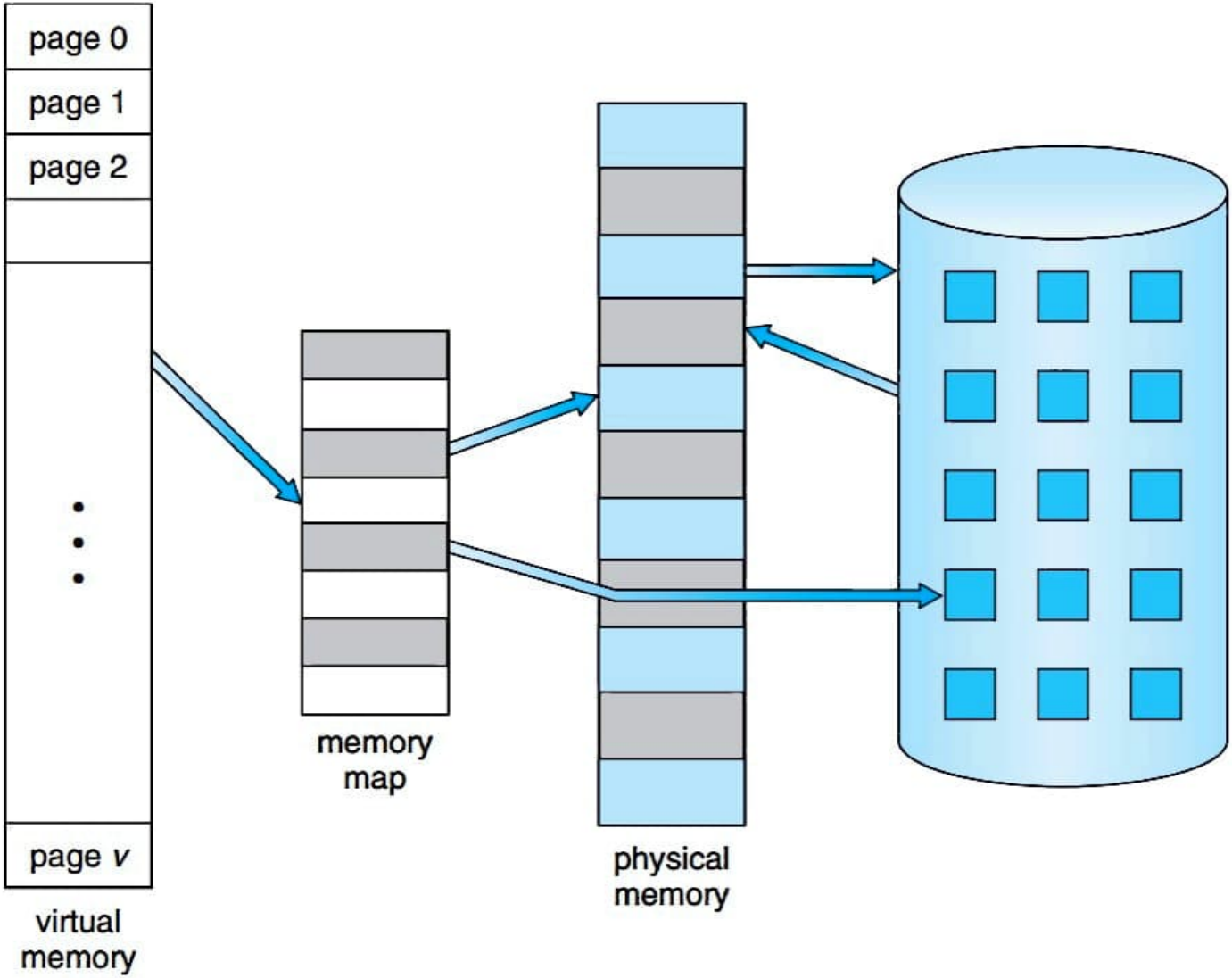


Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

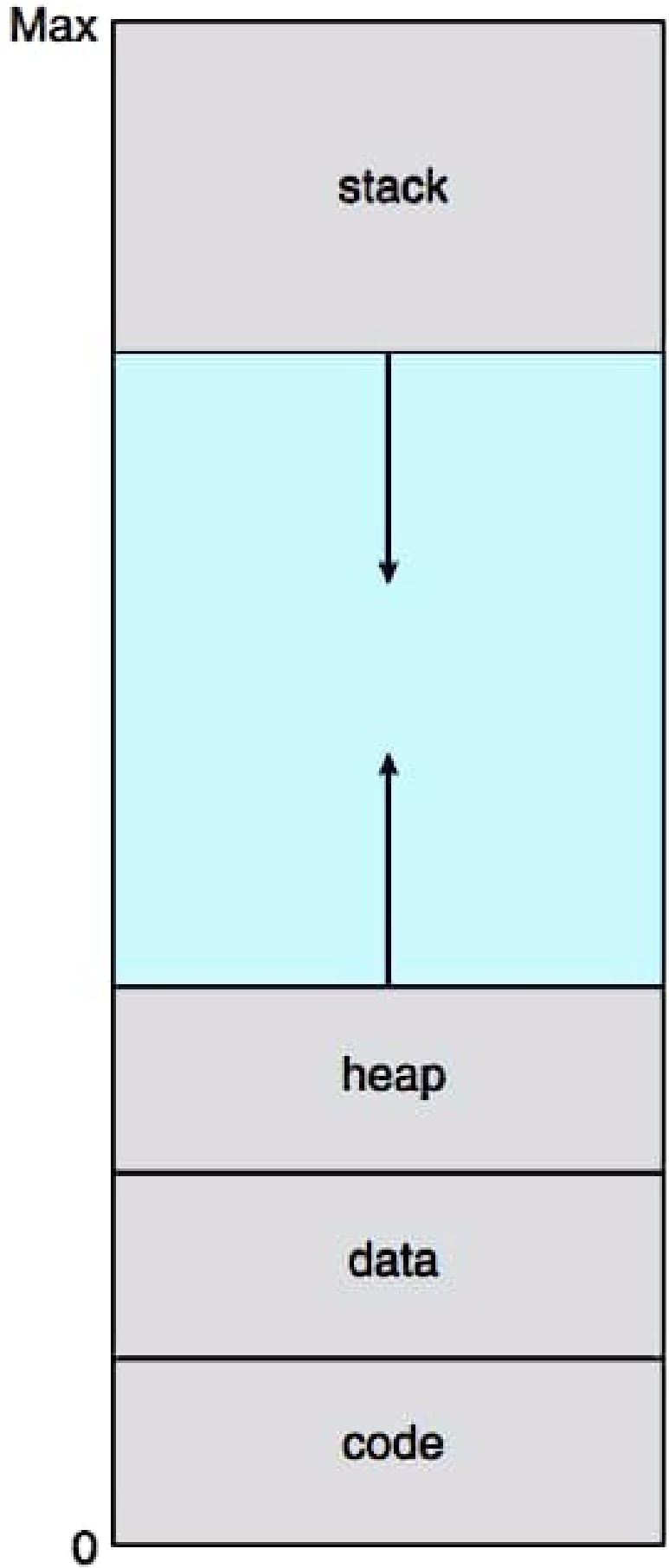


Figure 9.2 Virtual address space.

- The architecture above allows for the heap to grow if for dynamic allocation and the stack to grow in case function calls are done.
- The heap grows upwards and the stack

grows downwards.

- The gap in between is the virtual address space. It need mapping to physical only if the heap or stack grow.

- ~~Virtual memory address space is beneficial~~

- The process has an illusion of contiguous blocks of memory, but not all parts of this block might be mapped to a physical memory. This is the gap in the memory layout block. This part is not yet mapped to a physical address & acts are free space for the stack & heap to grow.

- Virtual address # spaces that ~~require~~ include holes are known as sparse address spaces.

- Virtual memory also allows for processes to share files & memory. Each process thinks that they have unique file they are accessing in the virtual memory, but this virtual memory for each process can be mapped to a single physical.

address, enabling sharing of files & memory among processes.

- Pages from parent process can be shared by the child process after fork(), increasing the speed of process creation.

Demand Paging

- As discussed before, the pages that the process never uses are also loaded onto the main memory & these entries are updated in the page table.

- Demand Paging: Pages are loaded onto the main memory only as they are needed by the process.
  Thus, pages that are never accessed are not loaded onto the main memory.

- Demand paging is similar to using the backing store; pages are loaded only when needed from the secondary memory to the primary memory, ~~But~~ instead of the whole process getting loaded.

- This partial swapping of a process from disk to main memory is

done by a swapper called 'lazy swapper" or "pager".

1) Basic Concepts:

- The pager guesses which pages will be needed by the process during execution, & only loads these pages.
- With demand paging, we need hardware support for determining which pages of a process are in memory and which are in the disk.
- For determing this, the valid-invalid bit discussed earlier (in 5.1) is used.
  The bit is set to
  → valid: when the page is legal as well as in the memory
  → invalid: when the page is illegal or not in memory.
- If all the pages that the process needs are in the main memory, then the process will run as expected.
  If the page required is not found, then the invalid bit causes a page fault, causing a trap to the operating system. The procedure for handling this

error is as follows:

a) We determine whether the reference to the address was actually valid or not.

b) If the reference was invalid, the process is terminated.

c) If the reference was valid but the page was not loaded in memory, then the page is brought to the main memory.

d) A free frame is found & the desired page is read into this frame.

e) When the page is loaded, we update the page table.

f) The instruction that was trapped can now be restarted & the process can continue execution as though nothing happened.

- Pure Demand Paging: A process' pages can also be not loaded entirely, initially. As the process & requests for the pages, the pages are loaded.

- Programs have "locality of reference" which help

the pager to load the most used pages into the main memory initially, reducing the performance of demand paging.

- This locality of reference is important because a process may try to access several new pages from the main memory at once.

- Hardware Requirement for demand paging:

→ Page Table: Contains frames & valid-ivalid bits.

→ Secondary Memory: This is a high speed disk called the "swap device". It holds those pages that are not in main memory.

- One important requirement for demand paging is to restart the trapped instruction exactly from where it was trapped.

- This can be difficult cause a lot of computation to be counted.

Eg: Adding A and B and storing the sum C. Instruction is:

1. Fetch & decode the instruction (ADD).
2. Fetch A
3. Fetch B

4. Add A and B
5. Store the sum in C

If the fault was in step 5 & the page for C wasn't found, then after the page is loaded in memory, then the fetch & addition will have to be performed again.

- This can also cause problems like some instruction, that updates data in multiple pages, getting trapped in between.
  After the page is loaded, the pages are already updated but the instruction has to restart.

2) Performance of Demand Paging:

- Demand paging can significantly affect the performance of a computer segment system. Let's compute the effective access time for memory with demand paging implemented.

- The memory access time (m) ranges from 10 to 200 ns.
  Without page demand paging the memory access time is same as the effective memory access time. But differs in the system with demand paging implemented.

- If a page fault occurs for the page not being there in memory then a sequence of steps are needed to be followed by the system which require a considerable amount of time (8 milisecond total).

- To calculate the effective memory access time, we assume the p to be the probability of a desired page is not being in the main memory.

- Thus, the effective memory access time

$= (1-p) \times m + p \times$ (page fault algo time required)

$= (1-p) \times 200 + p \times 8 \times 1000\,000$ ns

$= 200 + 7999800\,p$

∴ effective memory access time $\propto p$

∴ Thus we need the value of p to be extremely small:

$220 > 200 + 7999800p$

$\Rightarrow p < 0.0000025$

- Disk accesses to the swap device is much faster than accessing file systems. Thus, paging speed can be increased if

the file image is copied to the swap device from where pages will be loaded.

- Another way is to demand required pages from file system at load time itself & then swap them out to the swap device as their parts are done.
- The space in a swap device that is used for storing pages is called the swap space.
- Some systems save the swap space used by using demand paging of read only binary files.
- The read-only binary files can be loaded from the file system to the memory, but they do not need to be swapped out. Instead they can simply be overwritten by some other page.
- This can be done because these files are read-only & do not need to be stored in the swap space. If needed, they can simply be demanded from the file system again.
- Thus, the file system acts like a backing store in this context.

- Anonymous Memory: pages that are not associated with a file, eg: stack & heap for a process.

These pages have to be kept in the swap space even if the binary read-only files aren't kept.

3) Copy-On-Write

- A discussed before in 5.1, the child process created after fork() can have some shared pages between itself & the parent, speeding up the process creation.

- Copy-on-Write: Initially when the child process is created, it shares all the pages that the parent process has. These pages are marked as "copy-on-write" (except for the pages that are read only).

When the child or the parent are about to write something on page 0, a copy of page 0 is created, where the process (child or parent) can write the modifications.

This copy cannot be accessed by the other process.

the address of this copy will be updated in the page table of the process that did the modifications.

- For the pages needed for copying the modified page onto, many OS provide a pool of free pages.
- These free pages are allocated when the stack or heap for a process expands, or when copy-on-write has to be executed.
- Zero-fill-on-demand: A technique used by many OS in which pages are zeroed out or erased before being allocated.
- Virtual memory fork (vfork()):

The child process is created and the parent process is suspended. There is no copy-on-write used, thus, the child process can potentially change the page contents in the parent's address space.

Thus, we need to be careful that the child process does not modify anything. Vfork() is used when child process executes exec() right after being formed. Because no copying of pages takes place, vfork() is very efficient.

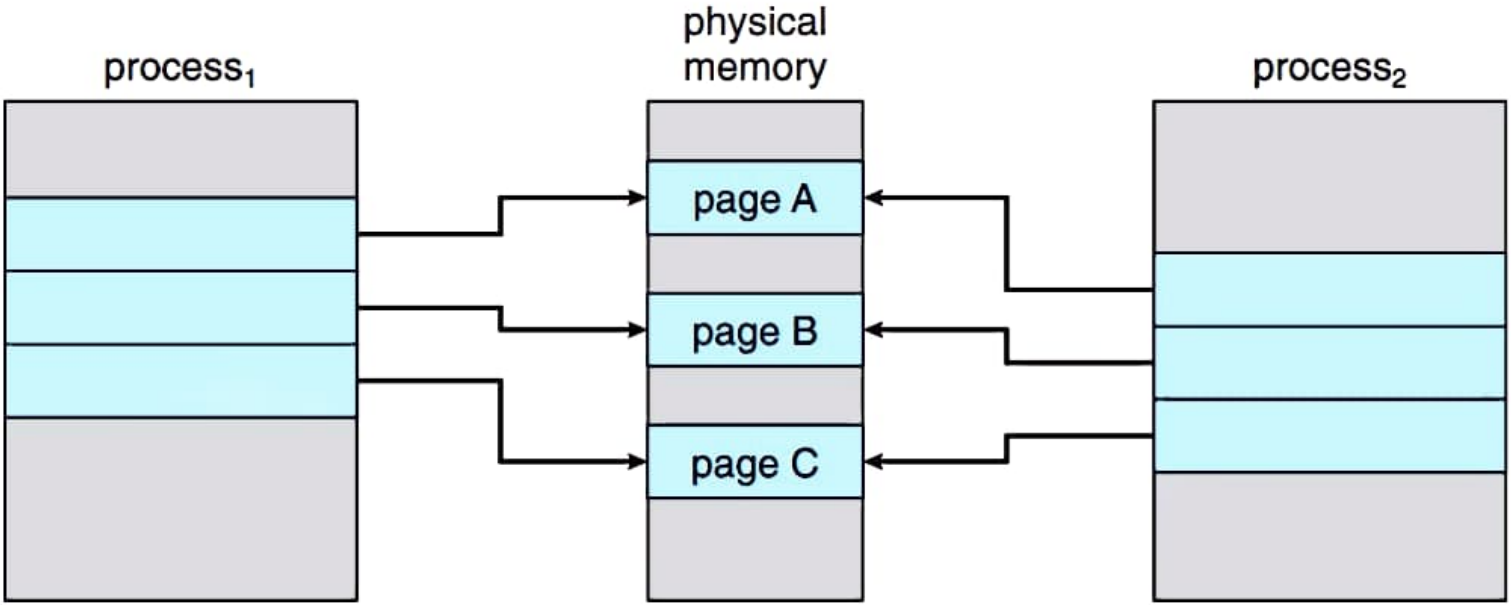


Figure 9.7 Before process 1 modifies page C.

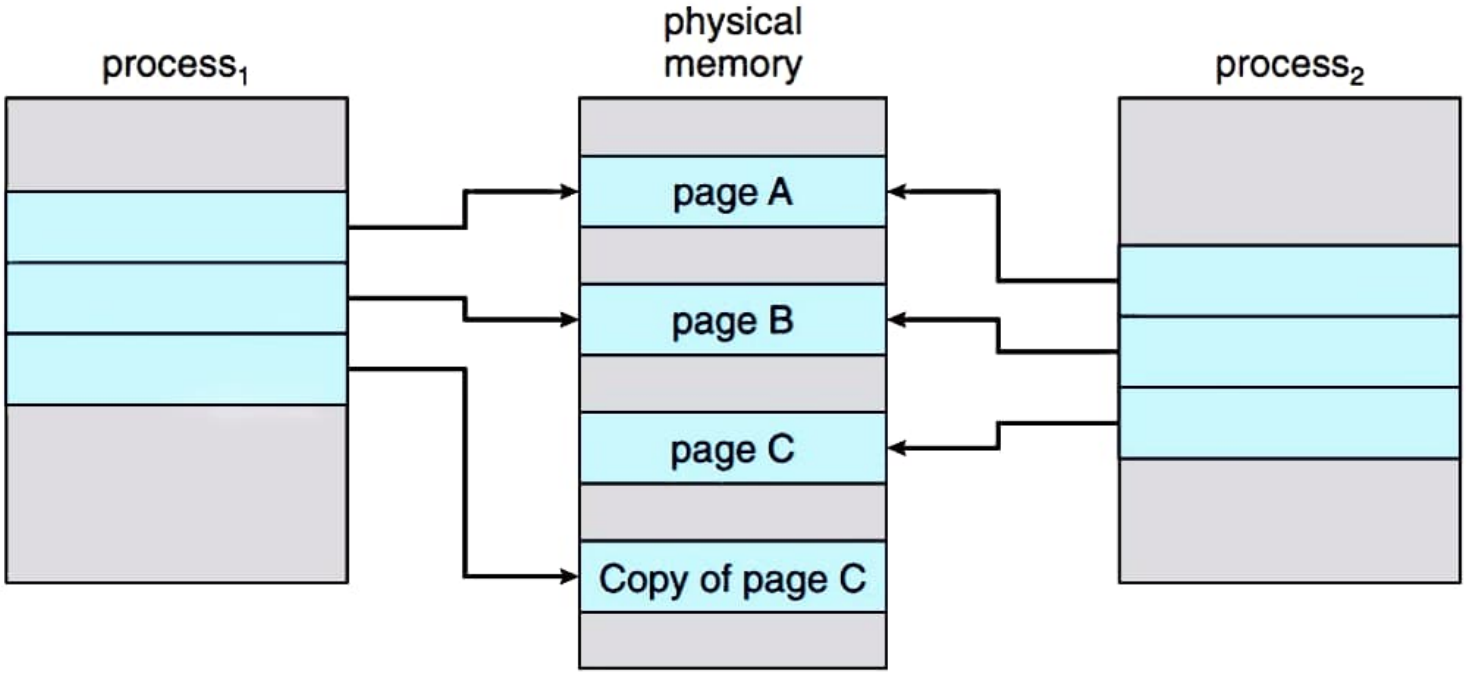


Figure 9.8 After process 1 modifies page C.

Page Replacement

- By trying to increase the degree of multip-rogramming (adding lots of processes in the main memory), we are also overallocat-ing memory.

- If there are 6 processes, and each process has 10 pages total, out of which 5 pages are in the main memory each.

- let's assume that the main memory can hold 40 frames maximum, holding 5 pages for 6 processes consumes 35 frames.

- If, however, all 6 processes need all 10 of their pages in main memory, sudde-nly, then we need 60 frames ⇒ 20 extra that we do not have.

- In these cases, the OS could swap out a process from the main memory to the disk, freeing the frames it occupied & reducing the level of multiprogramming.

- Another solution is to swap out pages of a process, we discuss some page replacement algorithms.

1) Basic Page Replacement

- If no frame is free, then we find a frame that is not currently being used & free it. The content of the page is written to the swap space and the page table for that page is modified to reflect that the page is no longer in memory.

{{ figure 9.10 + algorithm }}

- In some systems, each frame has a modify bit (or dirty bit) which is set when the frame is modified (written into).

- If the modify bit is 1 of a frame that is to be swapped out, the content of frame has to be

- If the modify bit is set to 0 when a victim frame is selected, then that frame is discarded without writing its content back to the swap space. This is because that frame has not been modified & the original page which was copied onto the frame already holds the data.

- If the modify bit is set to 1, then

the frame has to be swapped out, writing its content on the swap space.

Note that the initial state of the modify bit is 0.

- Using modify bit reduces the overhead if the frame was not modified, because now we're not needlessly swapping out frames that had no modifications, as these pages can be demanded from the swap space whenever needed without swapping them out.

- Thus two problems need to be addressed in demand paging:

  → Frame-allocation algorithm: How many frames to allocate to the processes.

  → Page Replacement Algorithm: Which frames to replace.

- Reference String: String of addresses produced by a CPU in one second is recorded. These addresses are in the millions in number.

  This string is called the reference string.

  However, these million addresses

are narrowed down based on the following factors:

a) for an address generated, we only need the page number from that address.

b) If a page is already accessed once, the references to that same page that immediately follow the first refrence have no way of creating a page fault, as the first reference implies that the page is already in memory.

- Eg: reference string recorded is:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610

if the page size is 100 bytes long then 0100 is in page 1, 0432 is in page 4 and so on.

Thus using the above considerations, the reference string is reduced to:

1, 4, 1, 6, 1, 6, 1, 6

- As the no. of frames increase, the number of page faults decrease as more pages can now be allocated

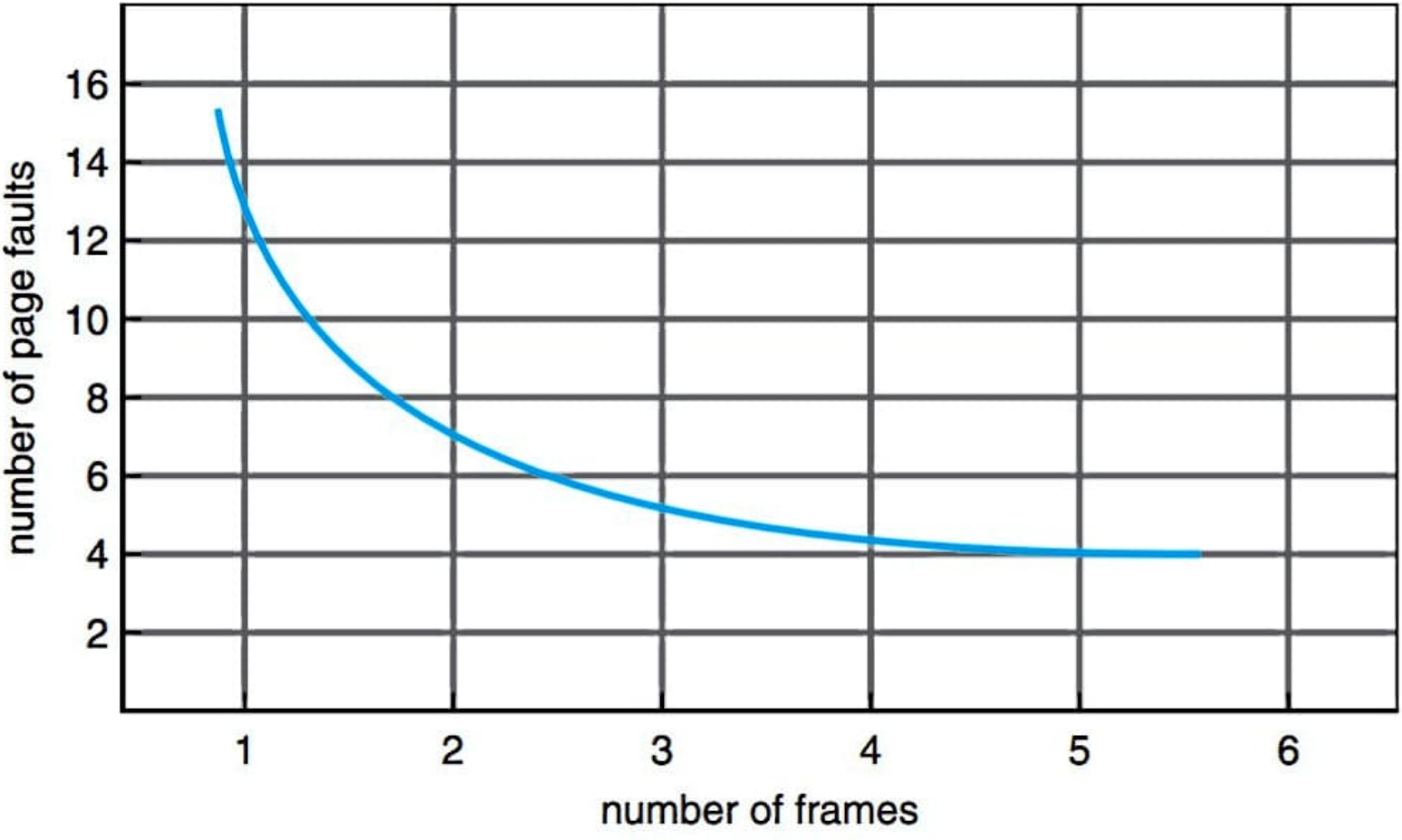


Figure 9.11 Graph of page faults versus number of frames.

to frames.

{{{ Figure 9.11 }}}

- We observe this page fault behaviour for different algorithms. We are using the reference string mentioned above for the calculations.

2) FIFO Replacement :

- The oldest page that was there in the queue is chosen as the victim.
- The FIFO ~~en~~ algorithm is easy to understand & program.
- The problem with FIFO could be that a page containing a variable ~~could be~~ getting highly referenced might become the victim.

There won't be any error with the execution of the process, as the pages referencing it will immediately raise page fault when that page is swapped out & the page will be swapped back in (again using FIFO).

But the above scenario ~~can~~ still causes multiple page faults & overhead.

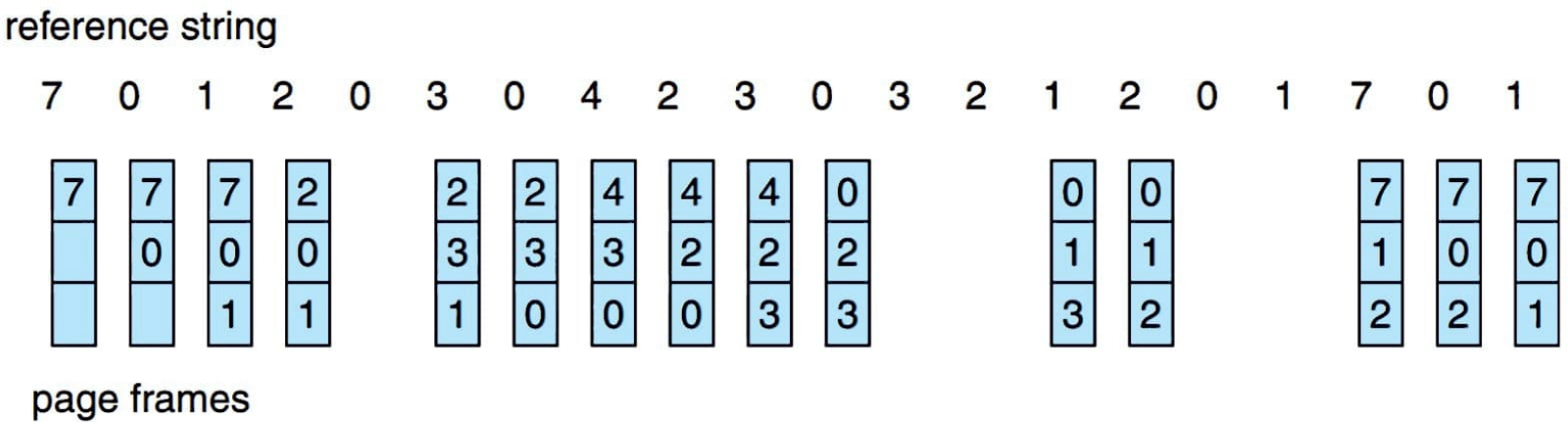


Figure 9.12 FIFO page-replacement algorithm.

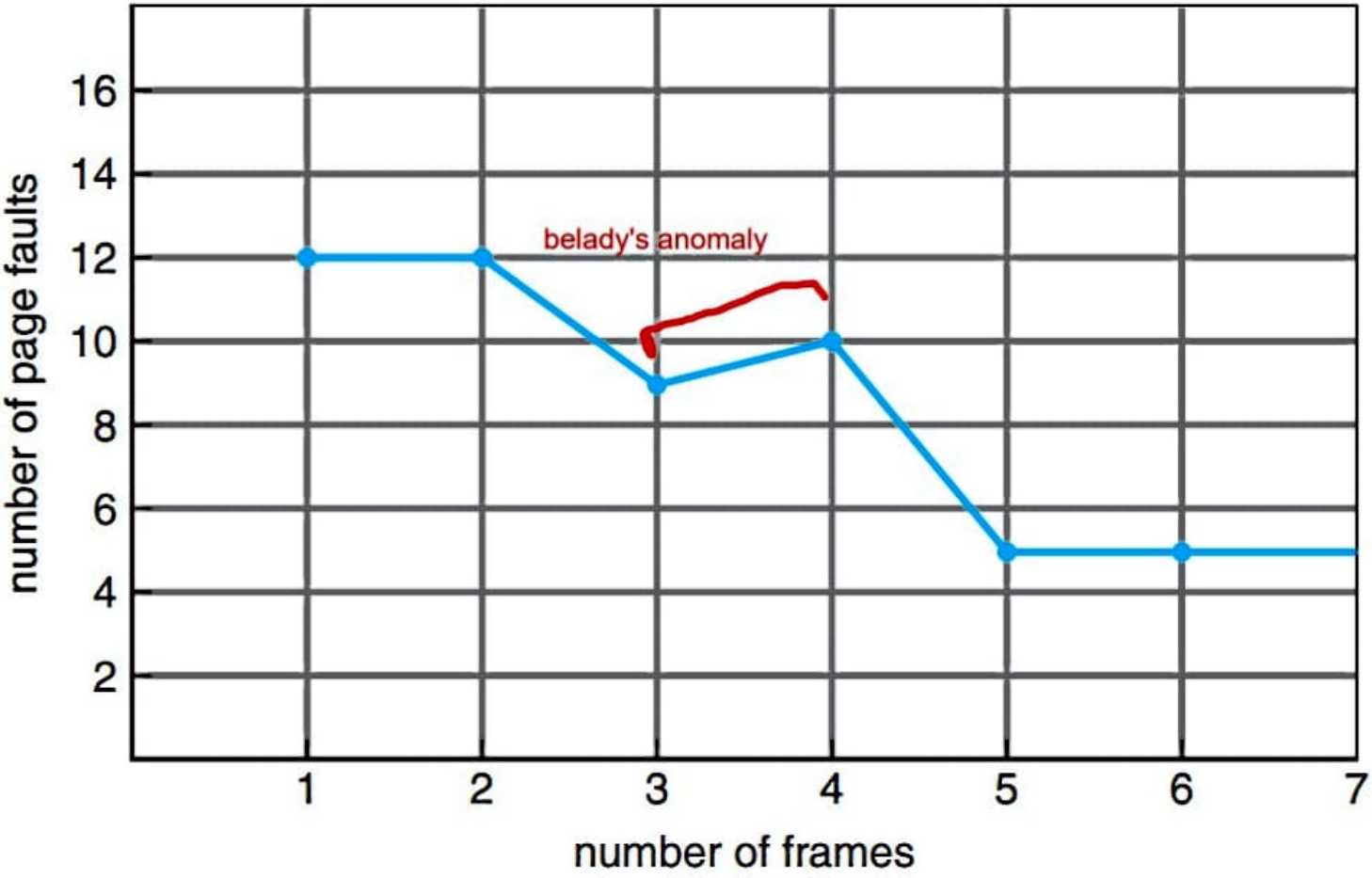


Figure 9.13 Page-fault curve for FIFO replacement on a reference string.

{{Figure 9.12}}

PAGE NO.:

Belady's

- This scenario can cause the ~~Bayesian~~ Anamoly: An increase in page fault with increase in frames.

{{{Figure 9.13}}}

3) Optimal Page Replacement: (OPT)

- OPT should cause the least page faults. It Replaces the page that will not be used for the longest time.
- However, the OPT algorithm is difficult to implement as it requires future knowledge of the reference string.
- The OPT algorithm is mainly used for comparison between algorithms.

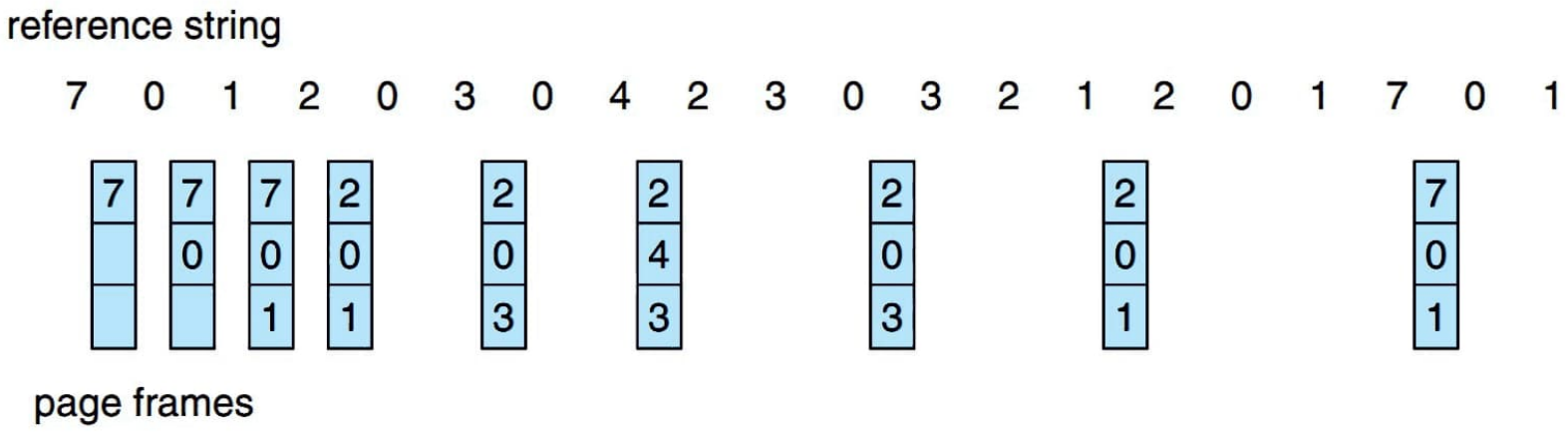


Figure 9.14 Optimal page-replacement algorithm.

4) LRU Page Replacement.

- The Least Recently Used (LRU) algorithm looks at the recent history to predict the future usage of page.
- The page that has not been used for the longest period of

time is swap the victim.

- In LRU, each page is associated with the time of that page's last use.

{{{Figure 9.15}}}

{{{Figure

- The problem is how to implement LRU. Hardware support is required, since we need to keep track of when the page was used last.

There are 2 possible implementations:

a) Counters: A time-of-use is associated with every entry in the page table. The CPU has a logical clock or counter now, that increments everytime a memory reference happens in the entire system.

Whenever a reference to a page occurs, the contents of the counter is copied onto the time-of-use entry for that page.

We just replace with the smallest time-of-use as it was used the longest time ago.

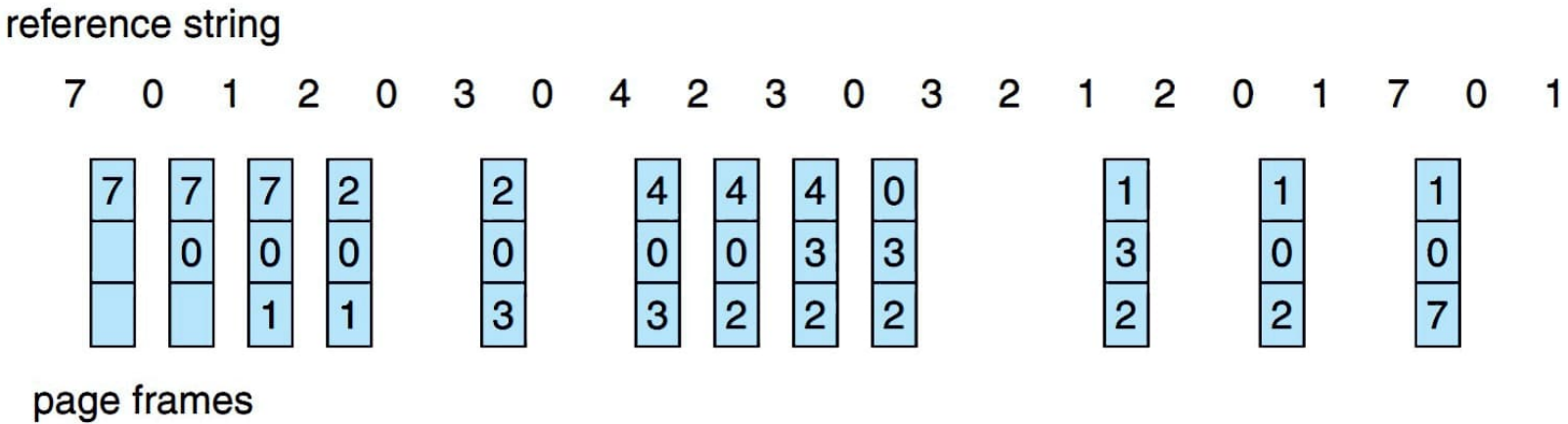


Figure 9.15 LRU page-replacement algorithm.

Problems:

→ Requires the time-of-use to be updated at every memory access.

→ The clock may overflow.

b) Stack: A stack of pages is maintained, the page that gets used is taken & placed at the top of the stack.

Thus, the recently used page is on the top, and the least used page is at the bottom.

Best implemented using double ended linked list.

Problem:

→ May require 6 pointer changes to put a page on the top of the stack at worst.

- Stack Algorithms: Algorithms that do not exhibit Belady's anomaly.
  Eg: OPT, LRU

- LRU requires update in couter / pointers at every memory reference. Thus, hardware support is needed, as giving interrupts for every memory

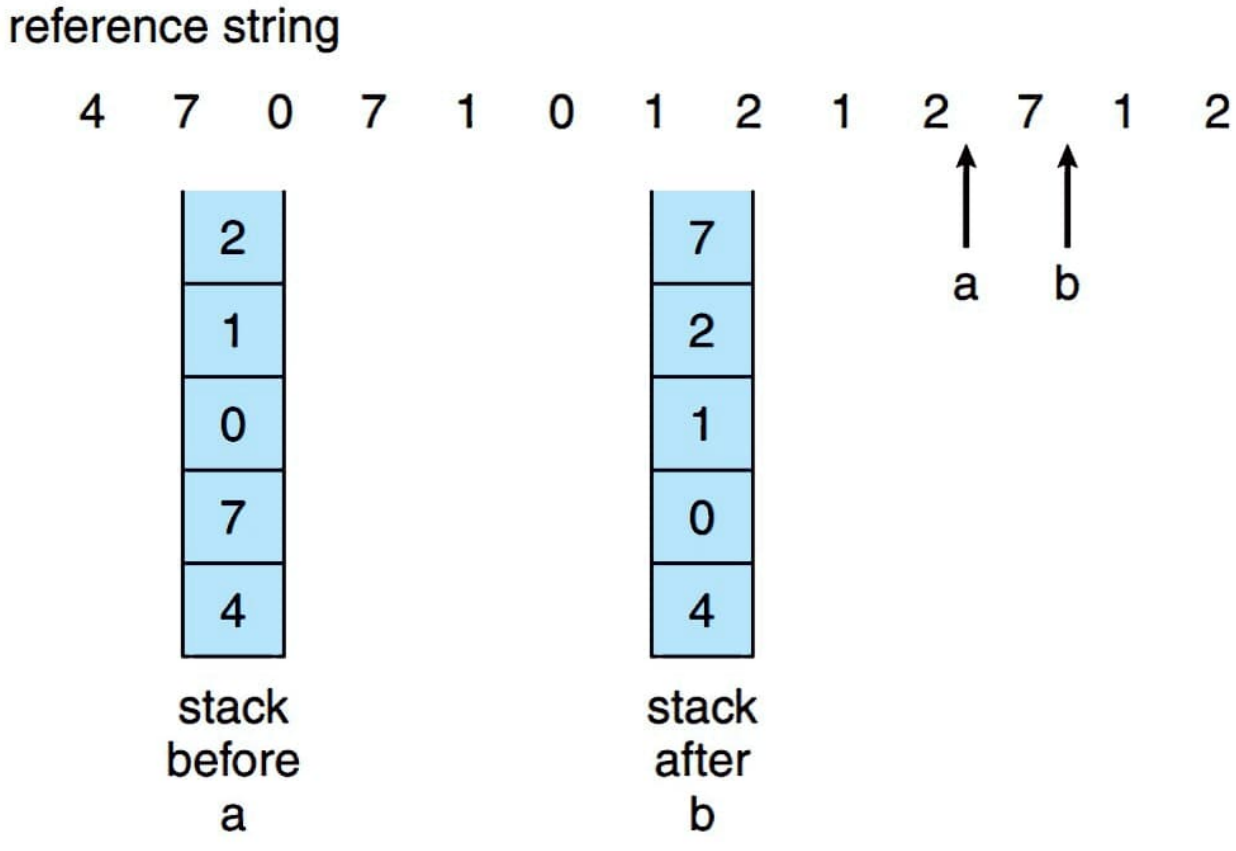


Figure 9.16 Use of a stack to record the most recent page references.

access would have great overhead.

{{ Figure 9.16 }}

* It's not an actual stack lol.

5) LRU Approximation Page Replacement

- Not many hardware support LRU. Many systems do provide support in the form of reference bits however.

- The reference bit is set to 0 initially. When the page is used, its corresponding reference bit is set to 1.

- Thus we can tell which pages have been used & which haven't, but we can't know the ~~or~~ order of use.

a) Additional - Reference - Bits Algorithm:

- Each page has an 8 bit byte for each page table entry.

- ~~Note~~ Initially all bits are set to 0. At regular intervals, the 8 bits are ~~sh~~ shifted to the right. If the page is used in that time interval, the highest order bit is set to 1. If not, then it

is set to 0:

Page is used:

00000000 $\xrightarrow{\text{time}}$ 10000000

Page is not used:

00000000 $\xrightarrow{\text{time}}$ 00000000

- A page with the byte as 11001001 is thus more recently used than a page with a page with by 01001100.

Thus, the ~~ree~~ least recently used page can be selected.

- It is possible that two least recently used pages have the same 8 byte code. In this case, the FIFO is used to select the victim from the two.

b) Second Chance Algorithm:

- Uses FIFO's advanced form. The reference bit (which is 0 at the beginning and is set to 1 if page is ~~accessed~~ used) is used.

- Using the FIFO, when a victim is selected, the reference bit is checked. If it is 0, then the

page is selected as victim. If it is 1, the page is given a second chance.

- The arrival time for this page is set to the current time & the refernce bit is set to 0; essentially sending it back to the end of the queue.
- Second chance algorithm is implemented using a circular queue.
- The pointer of the circular queue thus advances till it finds the next victim.
- Problem: Becomes FIFO if all bits are 1.

c) Enhanced Second - Chance Algorithm:

- The modify bit and the refrence bit are used to create an ordered pair. There are 4 possible classes that can be created:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement

3. (1, 0) recently used but clean—probably will be used again soon
4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

- For page replacement, the scheme is similar to the 2nd chance algorithm, but ~~instead~~ instead of checking for just the reference bit, the class is checked.

- The page with the lowest class is replaced. We may have to scan the entire queue to find this page.
- In enhanced second chance algorithm, preference is given to the modified pages to avoid I/O overhead.

6) Counting Based Page Replacement.

- We can counter of the number of ~~refr~~ references that have been made to each page for the following schemes:

  a) Least Frequently Used (LFU): The page with the smallest count is replaced.
  However, problem arises if the process was refrenced heavily at the start & is no longer used.

  b) Most Frequently used (MFU): MFU page is victim lol. The argument is that a page with least frequent uses might have just been brought in.

- LFU & MFU are not used often.

7) Page Buffering Algorithms

- When a victim is to be replaced by another page, the page is written to

free frame before the victim is written out completely.

- This is because the process doesn't have to wait for the page to be ~~replaced~~ written out before executing the new page.

- Whenever the paging device is idle, the modified page is written to the disk. This page's modify bit is then set to 0.

Thus, the probability of finding a page that's not modified is increased, the next time a page has to be replaced, thus it won't be needed to be written out.

- Another modification is we can keep track of the page that was in the frames in the free frame pool, 'a' the data is written out but isn't yet replaced in the frame.

- This old page can thus be referenced without needing to bring it again. When a page fault occurs, it can be checked whether the page is in the free frame pool or not.

8) Applications of page replacement:

- Some application like some database systems,

do not require the operating system's buffering. These applications understand their disk usage better than the OS.

- Thus, the OS gets of the way for these applications.

Raw Disk: The I/O to this disk bypasses all file system services like demand paging.

Allocation of Frames

- The basic strategy is to allocate any free frame to the user process, using appropriate page replacement algorithms when needed.

- Three free frames can always be kept in the memory for page replacement. The page requested will arrive on this frame & the process will continue, while the victim page is getting written out & a frame is freed.

1) Minimum Number of Frames:

- A need for the mininimum number of frames per process to be defined is there. Since we know that the greater the number of frames, the lesser the page faults.

- We must have enough frames to hold the pages required by an instruction to execute.

- The minimum no. of free frames is defined by the ~~ad~~ architecture.

2) Allocation Algorithms

- Equal Allocation: If there are m frames and n processes, each process gets m/n frames to work with.
  ~~Eqe~~ Eg: $m = 30$, $n = 3$
  each process gets $\frac{30}{3} = 10$ frames.

- Proportional Allocation: We recognize that ~~not~~ the size of each process won't be same and giving them all equal no. of frames won't be needed.

  In proportional allocation, we allocate available memory to each process according to its size.

  → Let there be 3 processes, P_1, P_2, P_3 with virtual memory sizes S_1, S_2, S_3.

  Let $S = \sum_{i=1}^{3} S_i$

If m is the total number of available frames, and a_i frames are allocated to

frame 1, then

$$a_1 = \frac{s_1}{S} \times m$$

$S \downarrow$

$$\frac{\text{Virtual memory of process 1}}{\text{Total memory of all processes}} \times (\text{available frames})$$

$\lceil a_1 \rceil$ frames are allocated to process 1.

- Allocation varies for different multiprogramming levels. Lesser frames will be allocated to every process if the ~~frames~~ degree of multiprogramming is high and vice versa.

- In both equal & proportional allocation, all processes are treated equally.

If priority is to be considered, then we can use an allocation where the number of frames also depends on the priority.

3) Global vs Local Allocation.

Another important consideration is how a frame ~~page~~ is selected for replacement:

a) Local Replacement: Each process selects only from its own allocated frames to replace.

b) Global Replacement:
A process can select any frame from its own frames or any other frame outside its frames.

Eg: A high priority process can choose a frame from the lower priority process to replace.

- ~~Problem with Global Replacement is that it does not need~~

- The problem with Global replacement is that the process cannot control its own page fault value. There are potential external factors like a higher priority process affecting which pages are ~~im~~ in memory & which aren't.

Thus, the process can execute 1 execution in 0.5 seconds & the next one in 10.8 seconds.

- The problem with local replacement is that a process cannot choose an empty frame from outside its memory space.

Thus, Global replacement gives better performance and is used more.

4) Non-Uniform Memory Access:

- So far we have assumed that the main

memory access throughout takes equal amount of time to access; a uniform main memory.

- Often times this is not the case, depending on the architecture of the CPU.

- Non-Uniform ~~Access In~~ Memory Access (NUMA): Systems in which memory access time varies significantly.

- In NUMA systems, the location of the frames makes a significant ~~impanc~~ impact on the performance.

- Memory changes should be allocated as close to the CPU as possible.

Thrashing

Thrashing: A process is thrashing if it is spending more time in paging than execution.

1) Causes of Thrashing:

- The OS monitors CPU utilization. If it gets too low, the CPU increases degree of multiprogramming by sending processes in the main memory.

- The Global page replacement is used. Let's say the new process needs frames for execution.

It takes frames from another ~~process~~ process' space.

- Now this process is short on frames & takes frames from some other processes.
- This can go further & further. The CPU comes to know that CPU utilization is still low, but doesn't know its because the processes are sending multiple page faults.
- The CPU sends more processes to increase the CPU utilization, causing more page faults & reducing CPU utilization.

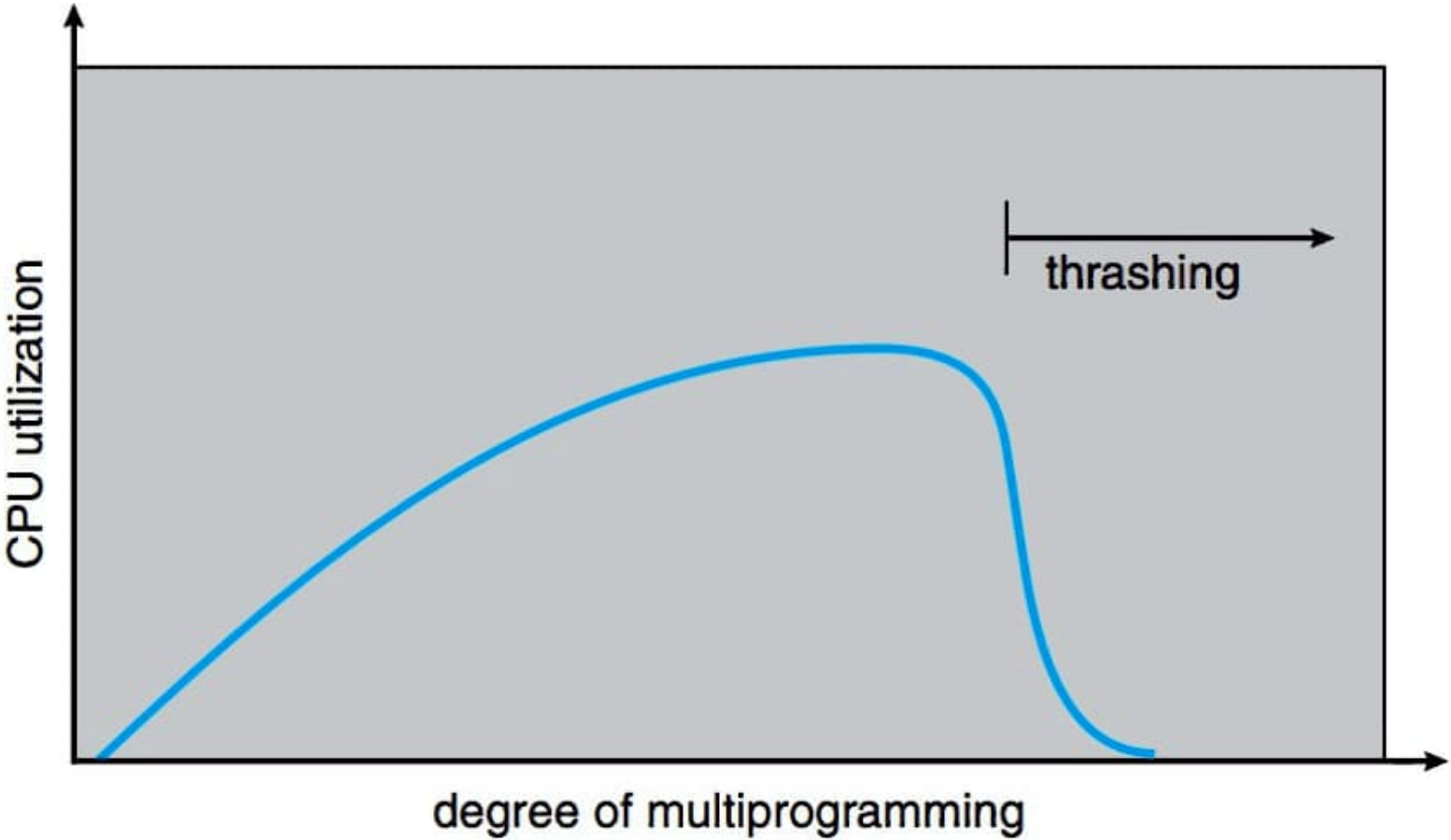


Figure 9.18 Thrashing.

- Thus, the degree of multiprogramming has to be reduced after 1 point, to increase CPU utilization lol.

- Using local replacement algorithm can limit thrashing, since it cannot steal frames from different processes.
However, it can still start thrashing on its own, which also affects CPU utilization.

- To avoid thrashing, we must know exactly how many frames a process needs, & give that many

frames to it.

- One strategy to find the number of frames a process needs is using the locality model.
- Locality Model: A program is divided into different localities and it is said that it moves from one locality to the other.

Eg: A program we wrote will have some functions & variable declaration at the start, this is 1 locality.

It then goes on to execute some functions, this is another locality, & so on.

Localities are defined by the program structure and data structures.

Thus, at every stage of execution of a process, we must allocate it enough frames needed for executing its current locality.

If done, page fault won't occur again until the localities change.

2) Working-Set Model

- The working-set model works on the concept of locality.

- The model uses a parameter, x, to define the working-set window.
- If a page is in recent use, it will be present in x, otherwise it will drop from the working time x.
- Thus, x is used to approximate a program's locality.
- The accuracy of the working set depends on the value of x, if x is too small then the whole locality wouldn't get encompassed. If x is too large, it would include multiple localities.

Thus, the most important factor of the working set is its size.

- Lets say the working set size of i^{th} process is wss_i, then the total demand of frames for all processes is

$$D = \sum_{i=1}^{n} wss_i$$, if there are m available frames, and $D > m$, then thrashing will occur, as some processes will not have enough frames.

- Once x (working set) is selected, the OS allocates the memory to the process.

- Once the locality changes, the frame requirement can reduce or increase.
- If page frame requirement reduced then another process is given these frames.
- If frame requirement increased ~~then the OS select~~ & there isn't enough memory in the main memory, then the OS selects a process to suspend. The frames thus freed are allocated to another process.
  The suspended process can be restarted later.
- The working set strategy avoids thrashing, while keeping CPU utilization high.

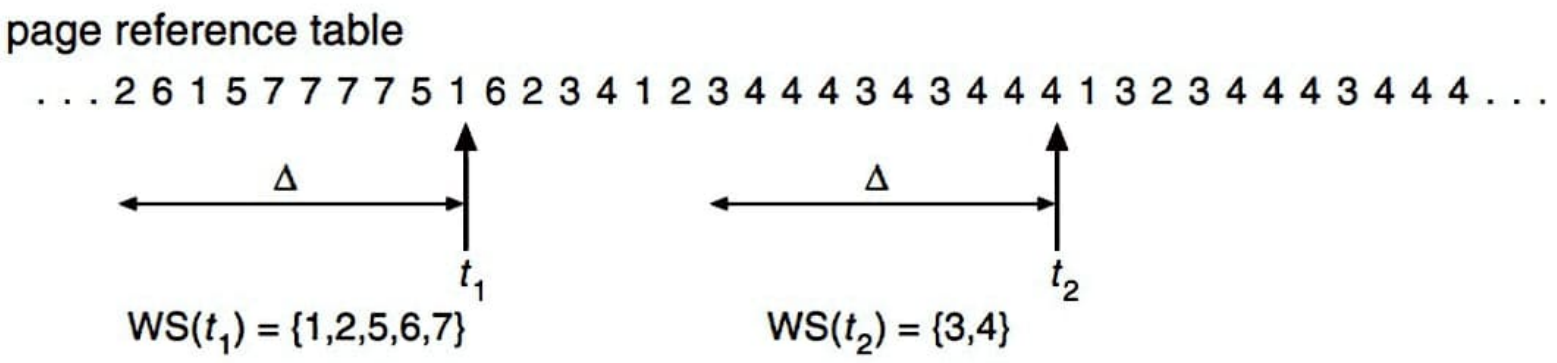


Figure 9.20 Working-set model.

- The working set x is like a sliding window. At every memory reference, a new reference appears at the end and one is dropped from the working set.

3) Page-Fault Frequency

- This strategy uses Page-Fault rate to get a direct approach to solving thrashing.
- When the Page-Fault Rate is high, the process doesn't have enough frames & may

may be thrashing.

- If the page-fault rate is too low, the process may be allocated many extra frames.

- If page fault rate increases, we give the process frames, if the page fault rate decreases, we remove frames from the process.

- In both Working Set and in Page fault rate algorithms, if the number of frames required cannot be satisfied then the whole process will have to be swapped out to the backing store.

Allocating Kernel Memory

- Kernel memory is allocated from a memory pool different from the user process' memory pool. This is because:

a) Kernel needs to use memory conservatively as it has data structures of varying sizes, which can cause lots of internal fragmentation.

b) Kernel code may be needed to be kept contiguously.

There are 2 strategies used for managing the free memory assigned to kernel processes:

1) Buddy System

- The buddy system allocates memory in fixed size segments. Memory is allocated using the power-of-2 allocator, which allocates memory to the kernel in powers of 2 (4, 8, 16 kb, etc.)

- If the size of the kernel's request is not in power of 2 then the closest higher power of 2 is chosen (eg: 16 kb for 11 kb request).

- In the buddy system this power of 2 is maintained by splitting the memory available in halves till a half that is just the right size is selected.

- Eg: Initially the memory segment is 256 kb. The kernel requests for 21 kb.
It is divided in two equal sized segments of buddies each of 128 kb size.
One of
Then the buddies is divided again to form 2 smaller buddies of size 64 kb each.

Again one of the buddies is divided again to form 2 smaller 32 kb buddies, one of which is assigned to the 21 kb request.

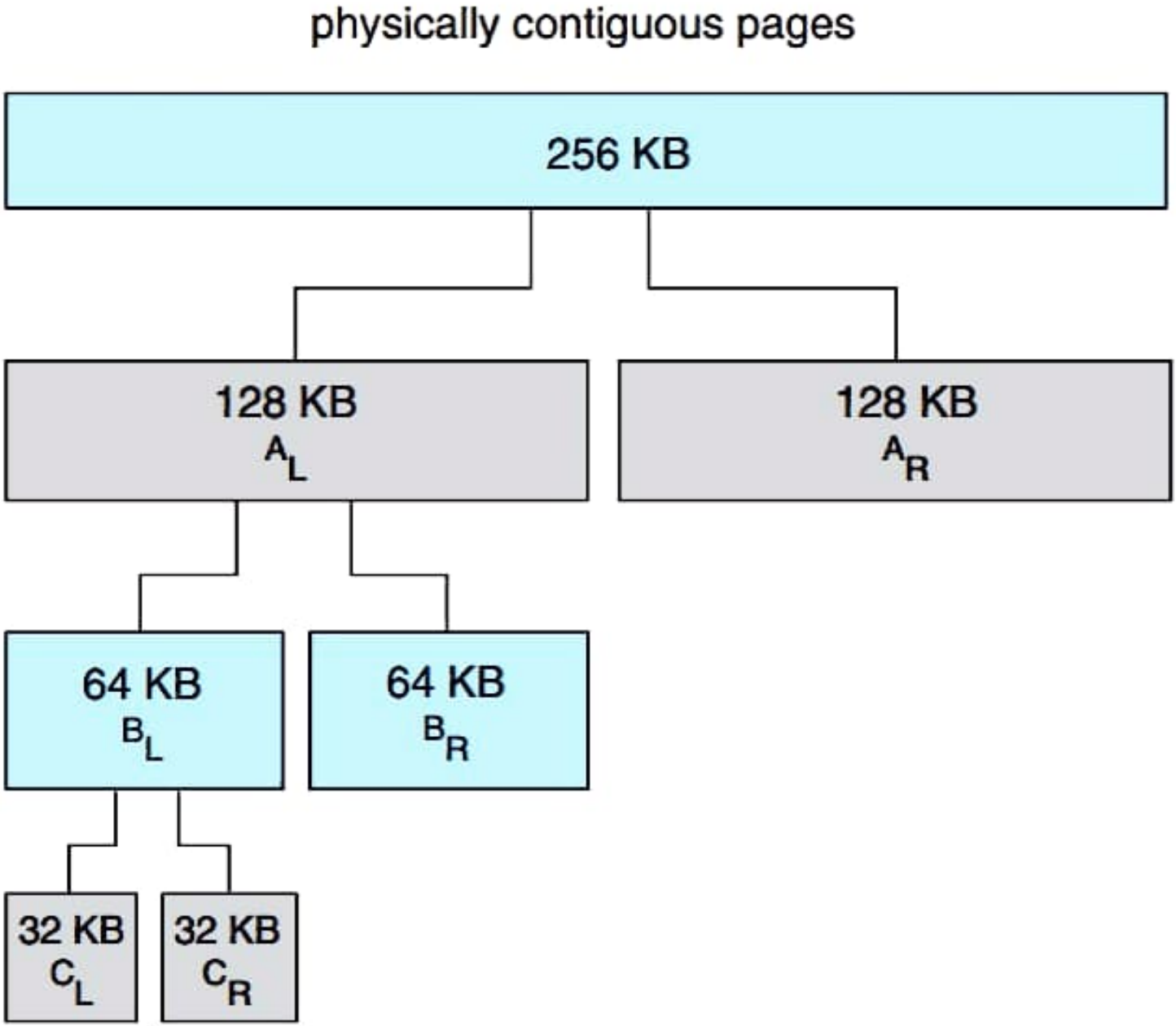


Figure 9.26 Buddy system allocation.

- One advantage of the buddy system is that the smaller buddies can easily combine to create a bigger buddy ☺, this is known as coalescing.

- Drawback is that internal fragmentation could still be high when powers of 2 are used to allocate memory.
  Eg: for a 33 ~~kb~~ kb request, 64 kb will be allocated, wasting 31 kb of memory.

2) Slab Allocation

- Slab: A slab is made up of 1 or more physically contiguous pages.

- Cache: A cache consists of 1 or more slabs

  Each ~~data~~ structure, file objects, etc. of the kernel is an object.

  These objects are stored in the cache, with 1 cache given to each ~~type~~ data structure, file object, etc. €

  {{figure 9.27}}

- The number of objects that the cache can store depends on the slab it points

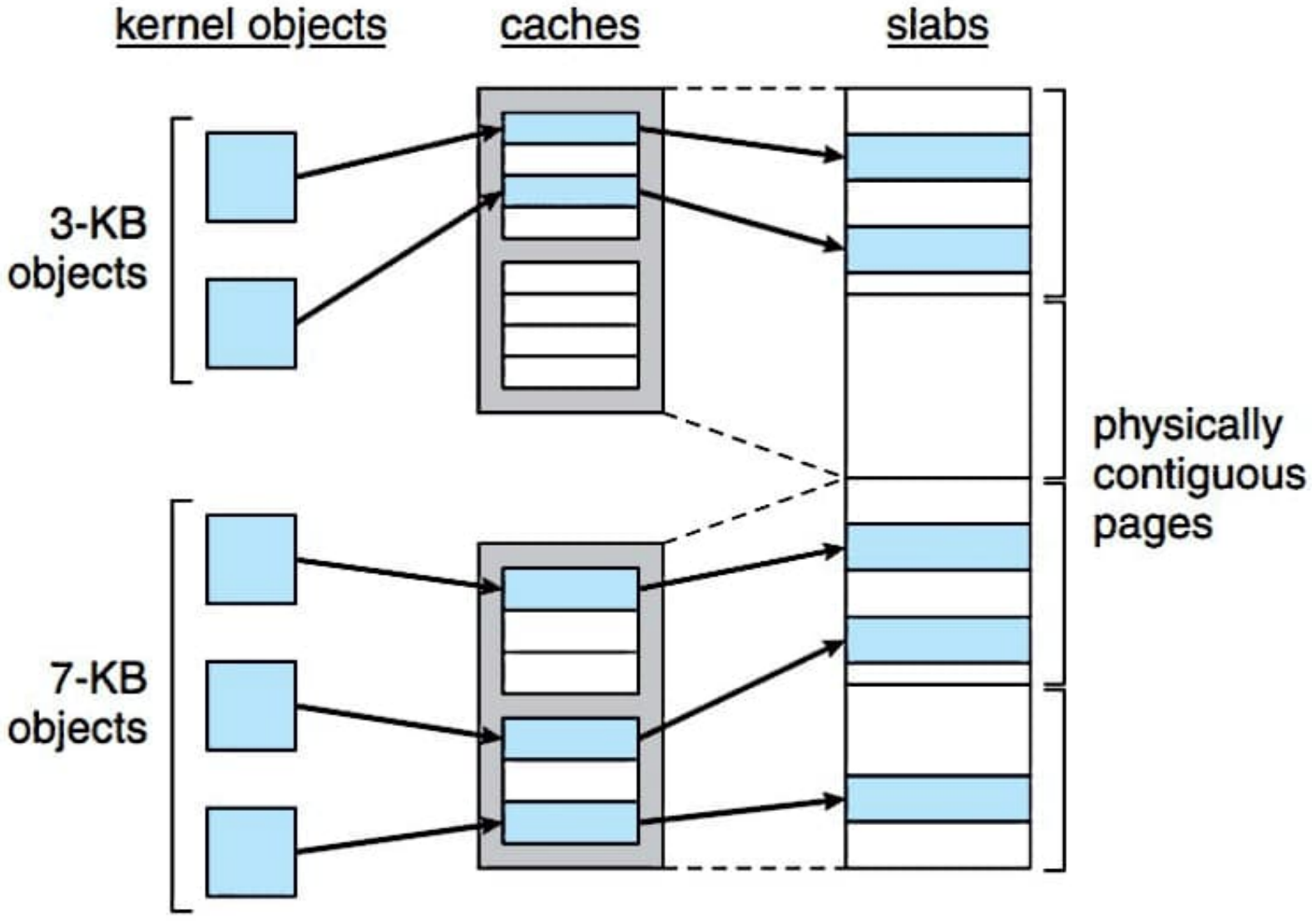


Figure 9.27 Slab allocation.

to. Bigger the slab, greater the no. of objects.

- Initially all objects in the cache are empty or free. Whenever the kernel needs a new object to store a data structure, the object in ~~the~~ cache is filled & marked as used.

- A slab may be in any of these possible states:

a) Full: All objects in the slab are marked as used.
b) Empty: All objects in the slab are marked as free.
c) Partial: The slab consists of both used & free objects.

- For a request to allocate an object, the slab allocator
  ⇒ first checks for empty objects in partial slabs. If not found
  → Then checks for empty slabs objects. If not found then
  → finally creating a new slab in the contiguous physical memory & assigns it a cache, ~~with~~ which has free objects

Advantages of Slab Allocation:
a) No internal fragmentation is observed because

the slab allocator returns the exact amount of memory that was requested.

b) Memory requests can be satisfied quickly. Objects are created in advance and can quickly be allocated from the cache.

- SLOB Allocator: Simple list of Blocks.
  Three lists of objects are created:
  → small: objects less than 256 bytes
  → medium: objects less than 1024 bytes
  → large: objects less than 1024 bytes
  Memory requests are allocated from an object on an appropriately sized list to save memory space.

- SLUB Allocator: Addressed performance issues in the SLAB allocator.

  → It moved the metadata from under the SLAB allocation table to the page structure.

  → It removed per-CPU queues that was maintained in the SLAB allocator for objects in cache.

Operating System Examples

1) Windows:

- Windows uses demand paging with clustering: The page that is demanded is loaded

in along with the several pages following this page.

- When a process is first created, it is assigned a working set minimum & maximum.

- Working-Set minimum: The minimum no. of pages that the process is guaranteed to have in memory.

- Working-Set maximum: The maximum No. of pages that a process can be assigned.

- The Virtual Memory Manager maintains a list of free page frames.

→ If a page fault occurs at a stage when the process is below its ~~maxim~~ working set maximum, ~~of~~ ~~of~~ one of the free ~~pages~~ frames is allocated to the page.

→ If a page fault occurs at a stage when the process is at its maximum working set, a page is replaced ~~out~~ out using Local LRU page replacement.

- Automatic Working-Set Trimming:
When the amount of free memory falls below a threshold, the automatic working set trimming kicks in.

→ If a process has more frames allocated

to it than it work-set minimum, then the frames are taken from it.

→ The A process can get more frames allocated to it when there's free memory available.

Both user mode & system processes use trimming.

2) Solaris

- The kernel needs to keep sufficient amount of memory (or frames) free because in a process, whenever the thread asks for a page, the kernel assigns it a free frame. Threads, being so granular in nature, might raise many page faults & may require many frames.
- lotsfree : A parameter found along with the free frames list to ~~find~~ define the minimum no. of ~~proces~~ frames there can be in the memory.

→ It is usually $\frac{1}{64} \times$ (Size of physical memory)

→ The kernel checks the no. of free frames 4 times in a second and compares them with lotsfree value.

→ If it is less than lotsfree, then a process called pageout starts.

- Pageout: The process is similar to the second chance algorithm, except it uses 2 hands for scanning instead of 1.

→ The front hand scans all the pages & sets all reference bits to 0.

→ At a later stage, the back hand checks the reference bit. If it is still 0, it removes the page from memory.

→ If the page was modified, it is written out, otherwise it is simply discarded.

- Solaris maintains a cache of the list of frames whose pages have been written out, but not overwritten.
- Pages can be accessed in the cache, before they are invalid.
- A free list is there that contains the frames that are invalid.
- Scanrate: The speed of scanning in the pageout algorithm is the scanrate.

→ slowscan: When free memory is above lotsfree, scanning occurs at slowscan.

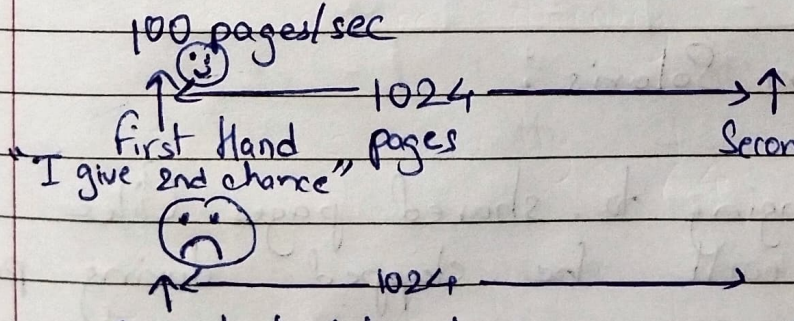
→ fastscan: When free memory falls below lotsfree, scanning occurs at fastscan.

- →Slowscan default value is 100 pages/sec.
  →fastscan default value is $\frac{(\text{total physical pages})}{2}$ pages/sec. Maximum speed is 8192 p/s

- handspread: The distance between the hands of the pageout algorithm.

→ The amount of time between front hand's second chance & the backhand's discarding is dependent on ~~handr~~ handspread & scanrate.

→ eg: scanrate = 100 pages/sec
handspread = 1024 pages



distance = 1024 pages
speed = 100 pages/sec
time = $\frac{1024}{100}$ = 10.24s

∴ the page got 10.24 seconds between the first

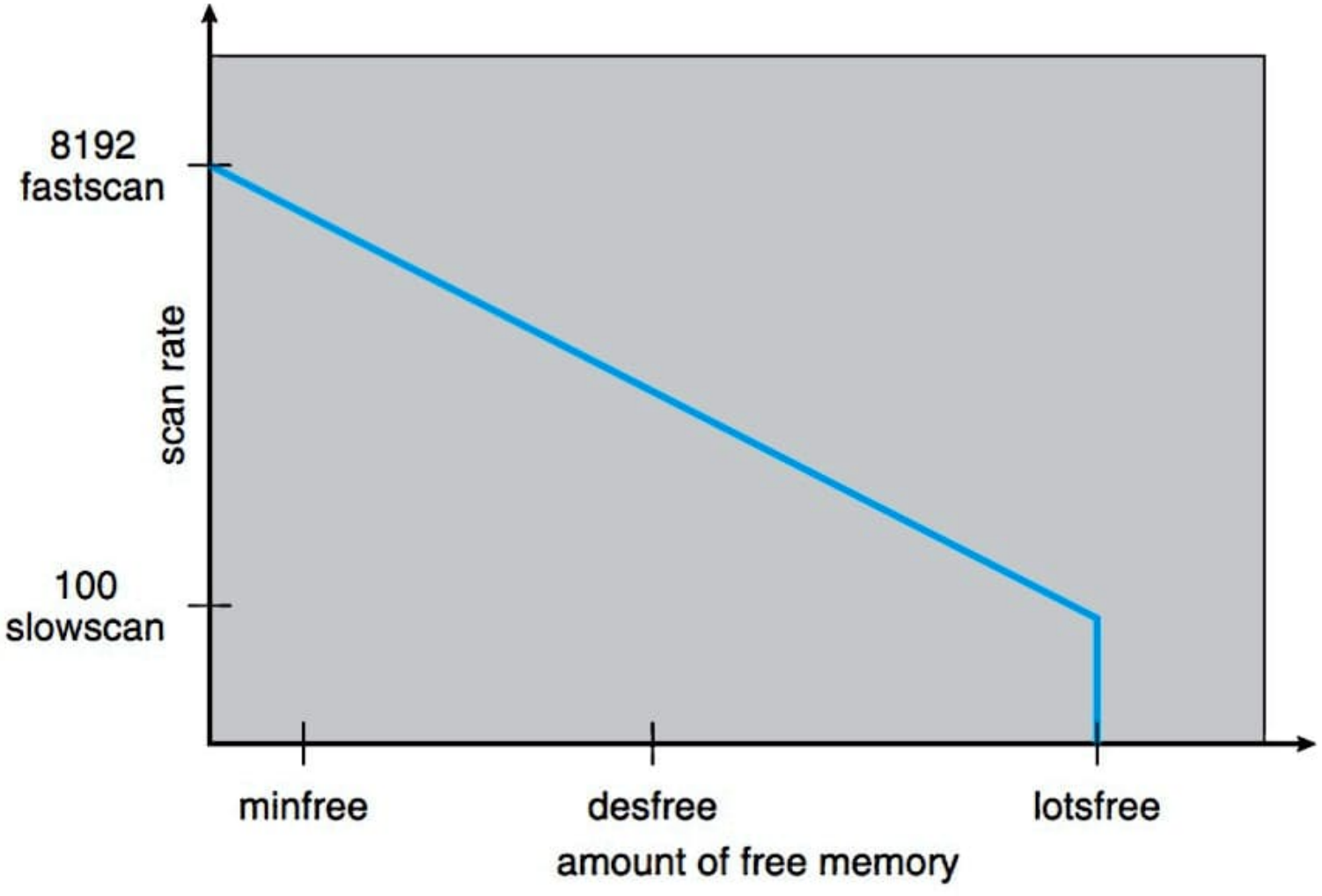


Figure 9.29 Solaris page scanner.

hand & the second hand.

* Scanrate in reality goes to thousands.

{{Figure 9.29}}

- As mentioned before, the kernel checks the free memory 4 times per second.
- However, if the amount of free memory falls below desfree, then it will be checked 100 times per second.
- If pageout is unable to keep the memory at desfree for at least 30 seconds, then the kernel starts swapping & writing the process in the backing store.
- If it drops below minfree, pageout process will occur for every page request.
- Enhancements in Solaris:

→ Pages belonging to shared pages (like libraries) will be skipped during page scanning.

→ Priority Paging: Pages that are given to process & pages allocated to regular files can be distinguished.