UNIVERSAL SYLLABUS · OPERATING SYSTEMS

Operating Systems

Addresses, swapping and holes

The notes for the lesson “Addresses, swapping and holes”: chapters 4.1 to 4.3, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

3 chapters | Chapters 4.1 to 4.3

About these notes

These are the notes for the University free n easy video lesson **Addresses, swapping and holes** on Operating Systems: chapters 4.1 to 4.3, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

4.1 Memory management strategies and address spaces

A process never touches physical memory directly. It uses its own logical addresses, numbered from zero, and a small hardware unit translates each one to a physical address after a safety check. This separation lets the OS place a program anywhere in RAM, move it later, and stop one process from reading another's data. Every phone app, UPI payment and IRCTC booking runs behind this translation. This chapter sets up the model: binding time, logical versus physical addresses, and the base-limit memory-management unit.

You need: dual-mode operation and interrupts from an earlier lesson, Foundations; the process and its PCB address space from an earlier lesson, Processes, Threads and CPU Scheduling.

Address binding and two kinds of address

Before the CPU can run an instruction, the instruction and its data must sit at some physical address in RAM. The act of fixing those addresses is called address binding. It can happen at three moments.

Compile time: the compiler knows the exact address and emits absolute code. The program must load at that address, so moving it means recompiling. Small embedded firmware often works this way.

Load time: the compiler emits relocatable code. The loader chooses a start address when the program is first loaded and adjusts addresses once. The process cannot move after that.

Execution time: the process may move while running. The CPU must translate every address at the moment of use, so this needs hardware support. General-purpose operating systems use this.

The address the CPU generates is a logical address; many syllabi call it a virtual address. The address the memory unit sees is a physical address. The set of logical addresses a process uses is its logical address space. A process's logical space starts at 0, independent of where it sits in RAM. The OS keeps the mapping private to each process, which is the base of protection.

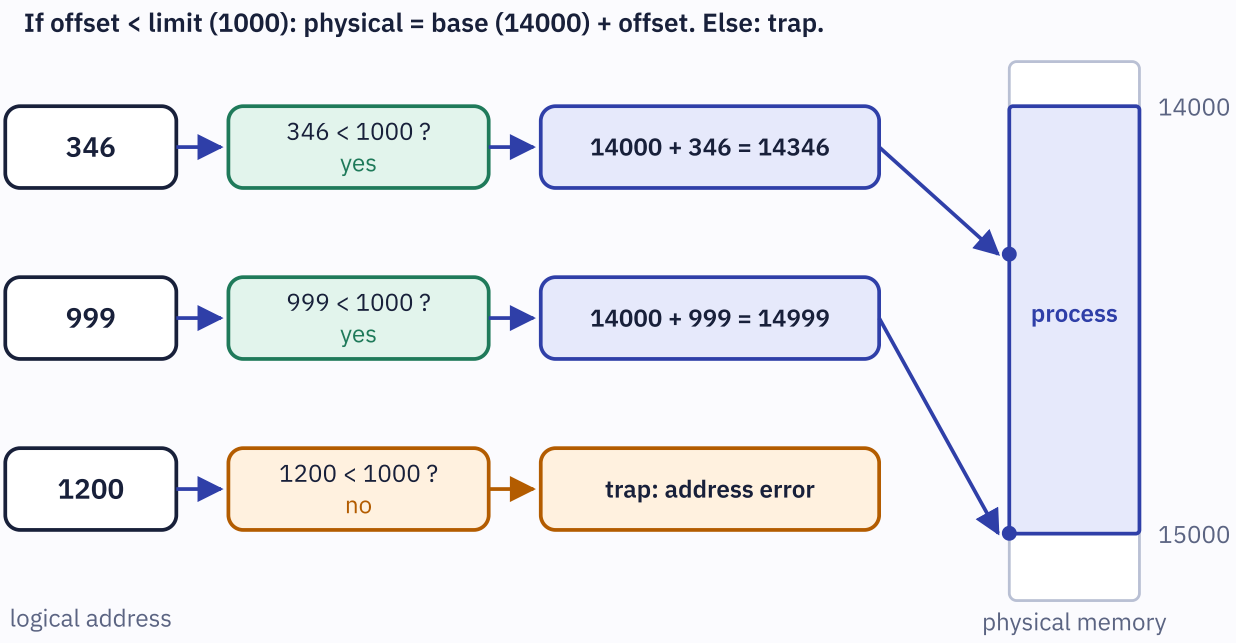


Figure 4.1.1 Every address is checked against the limit before the base is added: 346 and 999 are below 1000 and land at 14346 and 14999, while 1200 is past the end and traps.

The memory-management unit: base plus limit

Translation is done by the memory-management unit (MMU), a hardware block between CPU and memory. The simplest MMU has two registers per process.

The base register holds the smallest physical address of the process. Some syllabi call it the relocation register. The limit register holds the size of the process in bytes or words. It is a size, not the last valid address.

For every memory reference the CPU sends an offset. The MMU applies one rule.

```
if offset ≥ limit → trap; else physical = base + offset
```

base is the process start in physical memory; offset is the logical address; limit is the process length; physical is the translated physical address; trap means an address error sent to the OS.

The order matters. Check the limit first, then add the base. If you add first, a large offset can land inside another process and read or overwrite it.

The check is protection. In a multiprocessor or time-sharing machine, each process has its own base and limit. The OS loads the pair on a context switch (see an earlier lesson, Process scheduling queues and context switching). A trap goes to the OS, which usually terminates the offending process.

Allocation strategies and library loading

Translation answers where an address goes. Allocation answers where the OS puts each process.

Contiguous allocation gives each process one continuous block. It is simple, but free memory breaks into holes; choosing a hole is covered in 4.3, and moving a process out is 4.2.

Non-contiguous allocation splits a process. Paging cuts memory into fixed frames and the process into same-size pages, so pages can sit anywhere (4.4, 4.5). Segmentation uses variable pieces such as code, stack and heap, each with its own base and limit (4.6). Virtual memory adds demand paging later (4.7).

Three loading ideas save memory. Dynamic loading loads a routine only when it is first called. Dynamic linking postpones linking until run time: a small stub finds the library routine. A shared library is one copy on disk and in memory, mapped by many processes; the code is shared, while each process keeps its own data.

WORKED EXAMPLE

Question. A system uses execution-time binding with a base register of 14000 and a limit register of 1000, in word addresses. For each logical address, give the physical address or state the trap: (a) 346 (b) 999 (c) 1200.

1. Read the rule: if offset ≥ limit, trap; else physical = base + offset. Here base = 14000 and limit = 1000.
2. (a) offset = 346. 346 < 1000, so the access is valid. physical = 14000 + 346 = 14346.
3. (b) offset = 999. 999 < 1000, so the access is valid. physical = 14000 + 999 = 14999.
4. (c) offset = 1200. 1200 ≥ 1000, so the access is invalid. The MMU raises a trap and no physical address is produced.

Answer. The valid physical addresses are 14346 and 14999; logical address 1200 traps with an address error.

EXAM TRAPS

- Adding the base before checking the limit. A bad offset can then read another process; the check must come first.
- Quoting a logical address as if it were physical. Only base + offset after a valid check gives a physical address.
- Treating the limit register as the last valid address. It is the size, so offset equal to limit traps.
- Assuming compile-time binding is normal. Absolute code cannot be moved; execution-time binding needs MMU hardware.
- Swapping the base and limit. Base is the start address; limit is the length.

CHECK YOURSELF

– REMEMBER **What are the three moments at which address binding can occur?** 2 marks

Compile time, load time and execution time.

– REMEMBER **Define logical address and physical address.** 2 marks

A logical address is generated by the CPU and used by the process; many syllabi call it a virtual address. A physical address is the address seen by the memory unit.

– UNDERSTAND **Why must the MMU check the limit before adding the base?** 3 marks

The limit check rejects offsets outside the process. If the base were added first, an oversized offset could point into another process and break protection.

– UNDERSTAND **State the difference between the base register and the limit register.** 3 marks

The base register holds the start of the process in physical memory. The limit register holds the process size, not the last valid address.

– APPLY **A process has base = 8000 and limit = 600. Find the physical address for logical address 250, and state what happens for logical address 600.** 4 marks

250 < 600, so physical = 8000 + 250 = 8250. For 600, 600 ≥ 600, so the MMU traps.

– ANALYZE **A program is bound at compile time. The OS loads it at a different start address. Predict the result and explain.** 4 marks

The absolute addresses inside the code no longer match the load address. The program will access wrong locations and usually crash; compile-time absolute code must load at its fixed address.

– ANALYZE **Two processes share a CPU. P1 has base 5000, limit 1000; P2 has base 7000, limit 1000. Both use logical address 900. Explain why their physical addresses differ and what protection the limit gives.** 5 marks

For P1: 900 < 1000, physical = 5000 + 900 = 5900. For P2: 900 < 1000, physical = 7000 + 900 = 7900. The same logical address maps to different physical addresses because each process has a private base. The limit stops either process from using an offset of 1000 or more, so it cannot reach outside its own block.

BEYOND THE SYLLABUS

Not for the exam. Real CPUs add a translation look-aside buffer (TLB) to cache recent translations, and modern OSes randomize load addresses (ASLR) for security. Kernel allocators such as the buddy system and slab allocator manage free memory inside the OS using related free-list ideas. Virtual machines add a second layer of address translation. Search next: TLB, ASLR, buddy allocator, slab allocator.

4.2 Swapping of processes

When memory is full and another process needs space, the operating system can copy one whole process out to disk and later copy it back. The disk area that holds these copies is the backing store, also called swap space. This is swapping: roll out, then roll in. It lets more processes take turns in a small memory than could fit at once. The catch is time. Moving a whole process costs far more than the CPU work it saves, so classic whole-process swapping has largely given way to moving individual pages.

You need: 4.1 for what lives in memory, and context switching and scheduling queues (an earlier lesson).

Roll out and roll in

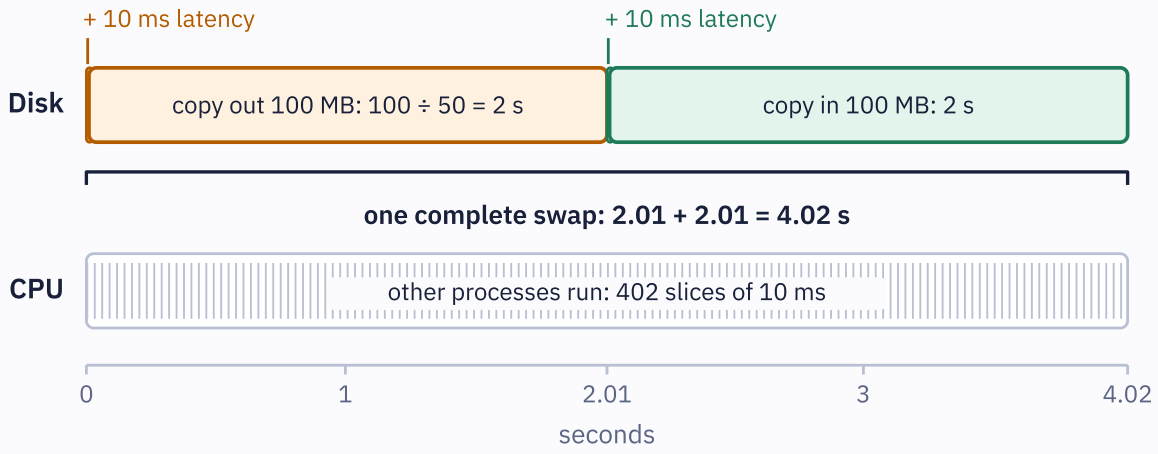
The memory manager wants the CPU busy. The CPU is busy only if some ready process sits in memory. When memory is full and a new or higher-priority process needs frames, one option is to move a process out of the way entirely.

Swapping does exactly that. The OS picks a victim, usually a process that is blocked or has low priority, and copies every byte of it from memory to a disk area. That is swap out, also called roll out. Later, when the process should run again, the OS copies it back: swap in, or roll in. The disk area is the backing store, also called swap space or the swap device.

The backing store must hold copies of every process that might be out at the same time, and it must allow direct access, so any process can be read back without scanning past the others.

The classic rule: a process is either entirely in memory or entirely on disk. Half-swapped processes do not exist in this scheme. So the moment a victim is chosen, all its frames become free at once, and the waiting process can be loaded into that space.

Two decisions belong to the OS: which process to swap out, and when. Swapping out a process that will be needed again soon wastes disk bandwidth and delays it.



4.02 s ÷ 0.010 s = 402 time slices pass before the swapped process is back.

Figure 4.2.1 Swapping a 100 MB process out and back in again takes 2.01 s each way at 50 MB/s plus 10 ms latency: 4.02 s in all, which is 402 time slices of 10 ms.

The cost is transfer time

A swap is pure copying, so its cost is set by the disk, not the CPU. Each direction takes the process size divided by the transfer rate, plus one disk latency for the seek and rotation.

T_swap = 2 × (S ÷ R + L)

T_swap is the total swap time in seconds; S is the process size in megabytes; R is the disk transfer rate in MB/s; L is the disk latency in seconds; the factor 2 counts the copy out and the copy in.

Latency is a few milliseconds on a hard disk, so the transfer dominates for any decent-sized process. Notice what the formula says: swap time grows in a straight line with process size.

PROCESS SIZE	COPY OUT (S)	COMPLETE SWAP (S)
10 MB	0.21	0.42
50 MB	1.01	2.02
100 MB	2.01	4.02
200 MB	4.01	8.02

Values in the table use R = 50 MB/s and L = 10 ms. If the next process to run is not in memory, the context switch cannot finish until the copy in completes. So context-switch time is not a fixed few microseconds; it becomes proportional to process size (see an earlier lesson).

Why whole processes rarely swap now

Compare the two numbers. A 100 MB process costs 4.02 s to swap. On the same machine a time slice might be 10 ms. That swap is worth 402 time slices. No scheduler can win that trade, even if the swapped process would use every slice fully. Meanwhile the CPU sits idle, or at best runs the other processes, while the disk copies.

So modern systems still swap, but in much smaller units. A page is typically 4 KB. Copying a page at 50 MB/s takes about 0.08 ms, slow but survivable, especially since most needed pages are already in memory. Page-level work is covered in 4.7 and 4.8.

- Unix-style systems keep a swap area and send pages to it only when free memory falls below a threshold; otherwise they free memory by dropping clean file pages and writing back dirty ones.
- Windows uses page files, written page by page, sized by the system or the user.
- Linux also has a swap partition or swap file, but it moves pages, not whole processes.
- Android and iOS normally do not swap to disk. Android may swap into a compressed area of RAM (zram); iOS keeps no swap device. With no disk swap, both end idle background apps when memory runs short.

How swap space is laid out on disk is a file-system topic; see a later lesson.

WORKED EXAMPLE

Question. A process of 100 MB must be swapped out of memory. The backing store transfers at 50 MB/s and has a disk latency of 10 ms per transfer. The CPU time slice is 10 ms. Find (a) the copy-out time, (b) the time for one complete swap, and (c) how many time slices that swap is worth.

1. Copy-out transfer time = S ÷ R = 100 MB ÷ 50 MB/s = 2 s.
2. Add one disk latency: 2 s + 0.010 s = 2.01 s to write the process to swap space.
3. Copy in is the same size at the same rate: 2 s + 0.010 s = 2.01 s to read it back.
4. One complete swap = 2.01 s + 2.01 s = 4.02 s.
5. Time slices lost = 4.02 s ÷ 0.010 s per slice = 402 slices.

Answer. Copy out takes 2.01 s, one complete swap takes 4.02 s, and that is worth 402 time slices of 10 ms.

EXAM TRAPS

- Counting the transfer only once. The process goes out and comes back, so the transfer cost doubles; 100 MB at 50 MB/s is 4 s of transfer, not 2 s.
- Mixing MB (megabytes) with Mb (megabits). 100 MB at 50 Mb/s is 16 s, not 2 s. Divide by 8 when the sizes are in different units.
- Thinking swap time is a small constant. It is linear in process size, which is exactly why whole-process swapping lost to paging.
- Forgetting the disk latency, or adding it once instead of twice. Each of the two transfers pays its own latency.
- Treating swapping and paging as the same thing. Classic swapping moves a whole process and leaves it entirely in memory or entirely on disk; paging moves individual pages and keeps the rest in memory.

CHECK YOURSELF

– REMEMBER **Define backing store and give its two other common names.** 2 marks

The backing store is the disk area that holds copies of processes that have been swapped out of memory. It is also called swap space or the swap device.

– UNDERSTAND **Why does the swap-time formula multiply by 2?** 3 marks

A swap has two transfers: the victim is written to the backing store and later read back. Both move the same number of bytes at the same rate and each pays its own latency, so the single-transfer cost is doubled.

– APPLY **A 200 MB process is swapped at 40 MB/s with a latency of 5 ms. Find the copy-out time and the time for one complete swap.** 4 marks

Transfer = 200 ÷ 40 = 5 s. Copy out = 5 + 0.005 = 5.005 s. Complete swap = 2 × 5.005 = 10.01 s.

– APPLY **With the swap of 10.01 s above and a time slice of 10 ms, how many time slices does the swap cost?** 3 marks

10.01 s ÷ 0.010 s = 1001 time slices.

– ANALYZE **A system swaps a whole process out on every context switch, and another back in. Its time slices are 10 ms and its processes average 50 MB at 50 MB/s with 10 ms latency. Explain what happens to CPU utilisation.** 5 marks

One complete swap = 2 × (1 s + 0.01 s) = 2.02 s, about 202 slices. The machine spends 2.02 s of disk copying to deliver 10 ms of CPU work, so utilisation collapses and the system behaves like thrashing. Whole-process swapping on every switch is unusable; that is why pages are swapped instead.

– ANALYZE **Give two reasons why phones such as Android and iOS generally do not use disk swap.** 4 marks

Flash storage is slower per operation than RAM and wears out with repeated writes, so constant paging would hurt both speed and device life. Phone memory pressure is handled instead by ending idle background apps, and by compressing some pages in RAM.

– UNDERSTAND **State the classic rule about where a swapped process lives, and say what it means for free frames.** 3 marks

A process is entirely in memory or entirely on disk. So when a victim is swapped out, all of its frames are released together, giving one large free region for the incoming process.

BEYOND THE SYLLABUS

Not for the exam. Linux now ships compressed RAM swap (zram) and a swap cache layer (zswap) that trade CPU cycles for avoiding disk writes entirely, which changes the arithmetic in this chapter. Container platforms lean on memory overcommit and the out-of-memory killer rather than classical swapping, and cloud providers bill partly on memory pressure. Android's low-memory killer and iOS memory limits are worth reading as real designs of "what to do when memory is full". Search for zram, zswap, memory overcommit, and the Linux OOM killer.

4.3 Contiguous memory allocation and fragmentation

Contiguous memory allocation gives each process one continuous block of physical memory. The idea is simple: a process starts at one address and grows upward, with no gaps inside its block. Fixed partitions, made at boot, waste space inside a partition. Variable partitions create free holes at run time. The OS must choose a hole for every new process

using first fit, best fit, worst fit or next fit. The choice decides how much memory is wasted. This scheme is the baseline that paging and segmentation improve on.

> You need 4.1 for base and limit, and 4.2 for what happens when no hole is big enough.

Variable partitions and holes

In variable partitioning, the OS does not pre-cut memory. It starts with one large free block. When a process arrives, the OS takes a hole—a continuous free block—large enough for it and gives the process the first part. The rest of the hole stays free. When the process exits, its block becomes a hole again. The OS merges that hole with any free neighbours, so two adjacent holes become one bigger hole. Without merging, memory breaks into many small pieces.

- First fit: scan holes from the start and take the first hole that fits.
- Best fit: scan all holes and take the smallest hole that fits.
- Worst fit: scan all holes and take the largest hole that fits.
- Next fit: like first fit, but resume scanning from the hole after the last placement, not from the start.

leftover = hole size − request size

leftover is the free space left in the chosen hole, in KB here; hole size and request size are also in KB. The formula applies only when hole size ≥ request size.

Holes in address order: 100, 500, 200, 300, 600 KB

request	212 KB	417 KB	112 KB	426 KB
First fit first hole that fits	500 → 288 hole → left over	600 → 183 hole → left over	288 → 176 hole → left over	**waits** no hole ≥ 426
Best fit smallest hole that fits	300 → 88 hole → left over	500 → 83 hole → left over	200 → 88 hole → left over	600 → 174 hole → left over
Worst fit largest hole	600 → 388 hole → left over	500 → 83 hole → left over	388 → 276 hole → left over	**waits** no hole ≥ 426

Only best fit places all four; it keeps the 600 KB hole whole for the 426 KB request.

Figure 4.3.1 Each cell gives the hole chosen and what is left over, in KB. First fit and worst fit break up the large holes early, so the 426 KB request waits; best fit keeps the 600 KB hole and places all four.

Fixed partitions and internal fragmentation

In fixed partitioning, the OS cuts memory into a fixed number of partitions at boot. The partition sizes stay fixed for the life of the system. A process must fit completely inside one partition. If a 300 KB process sits in a 512 KB partition, 212 KB inside that partition is wasted. This waste is internal fragmentation: the unused space is inside the allocated block. The process cannot share the leftover with another process. The number of partitions limits how many processes can be in memory at once. Some systems use unequal partitions to fit common process sizes. Fixed partitions are simple and need no hole search, but every process that does not fill its partition pays the waste.

External fragmentation, compaction and the 50-percent rule

Variable partitions avoid internal waste but create external fragmentation. Free memory is split into holes between allocated blocks. Total free memory may be large enough for a new process, yet no single hole is big enough, so the process waits even though memory is free. Compaction fixes this by sliding allocated blocks together, leaving one large hole. Moving a running process changes its physical addresses. Compaction therefore needs execution-time relocation: the CPU must translate every logical address to a physical address at run time, and the OS updates the process's base register after the move. A program that uses absolute physical addresses cannot be moved safely.

The 50-percent rule is a rule of thumb for first fit. Over a long run, with many processes arriving and leaving, about half of memory ends up in holes, on average, and half in allocated blocks. The number of holes grows too, so searching takes longer. The rule is an average, not a limit. Fixed-size blocks, covered in 4.4, remove external fragmentation by allowing any free frame to hold any page. Variable-length logical pieces in 4.6 face a related placement problem.

WORKED EXAMPLE

Question. Free holes in address order are 100 KB, 500 KB, 200 KB, 300 KB and 600 KB. Requests arrive in this order: 212 KB, 417 KB, 112 KB and 426 KB. For first fit, best fit and worst fit, show the hole used by each request, the leftover after each successful placement, and whether every request is placed.

1. Start with holes H = [100, 500, 200, 300, 600] KB.
2. First fit, 212 KB: 100 is too small; 500 fits. Take 500. Leftover = 500 − 212 = 288 KB. H = [100, 288, 200, 300, 600] KB.
3. First fit, 417 KB: 100, 288, 200 and 300 are too small; 600 fits. Take 600. Leftover = 600 − 417 = 183 KB. H = [100, 288, 200, 300, 183] KB.
4. First fit, 112 KB: 100 is too small; 288 fits. Take 288. Leftover = 288 − 112 = 176 KB. H = [100, 176, 200, 300, 183] KB.
5. First fit, 426 KB: largest hole is 300 KB, which is less than 426 KB. No hole fits; request waits.
6. Best fit, 212 KB: fitting holes are 300, 500 and 600; smallest is 300. Take 300. Leftover = 300 − 212 = 88 KB. H = [100, 500, 200, 88, 600] KB.
7. Best fit, 417 KB: fitting holes are 500 and 600; smallest is 500. Take 500. Leftover = 500 − 417 = 83 KB. H = [100, 83, 200, 88, 600] KB.
8. Best fit, 112 KB: fitting holes are 200 and 600; smallest is 200. Take 200. Leftover = 200 − 112 = 88 KB. H = [100, 83, 88, 88, 600] KB.
9. Best fit, 426 KB: 600 fits. Take 600. Leftover = 600 − 426 = 174 KB. H = [100, 83, 88, 88, 174] KB. All four requests are placed.
10. Worst fit, 212 KB: largest hole is 600. Take 600. Leftover = 600 − 212 = 388 KB. H = [100, 500, 200, 300, 388] KB.
11. Worst fit, 417 KB: largest fitting hole is 500; 388 is too small. Take 500. Leftover = 500 − 417 = 83 KB. H = [100, 83, 200, 300, 388] KB.
12. Worst fit, 112 KB: largest hole is 388. Take 388. Leftover = 388 − 112 = 276 KB. H = [100, 83, 200, 300, 276] KB.
13. Worst fit, 426 KB: largest hole is 300 KB, which is less than 426 KB. No hole fits; request waits.

Answer. First fit places 212, 417 and 112 KB and leaves leftovers 288, 183 and 176 KB; best fit places all four and leaves 88, 83, 88 and 174 KB; worst fit places 212, 417 and 112 KB and leaves 388, 83 and 276 KB. All values are in KB.

EXAM TRAPS

- Best fit is not always the best overall. It leaves the smallest leftovers, which often become tiny holes too small to use.
- External fragmentation is holes between allocated blocks, in variable partitions. Internal fragmentation is waste inside a fixed partition or the last page of a process.
- Next fit does not rescan from the start; it continues from the hole after the last placement, so its holes chosen can differ from first fit.
- Compaction needs execution-time relocation. Moving a process is safe only if its addresses are translated at run time and its base register is updated.
- The 50-percent rule is an average for first fit, not a rule that half of RAM must stay free or a fixed limit.

CHECK YOURSELF

– REMEMBER **Define a hole in variable partitioning.** 2 marks

A hole is a continuous free block of physical memory.

– UNDERSTAND **Why does fixed partitioning cause internal fragmentation?** 3 marks

A process smaller than its partition leaves unused space inside the allocated partition, and no other process can use that space.

– APPLY **Holes are 200 KB, 450 KB and 300 KB. A request of 250 KB arrives. Which hole does best fit choose, and what is the leftover?** 4 marks

Best fit takes the smallest hole that fits: 300 KB. Leftover = 300 − 250 = 50 KB.

– ANALYZE **Total free memory is 800 KB, in holes of 300 KB, 300 KB and 200 KB. A 500 KB request waits. Why?** 5 marks

This is external fragmentation. No single hole is at least 500 KB, even though the total free memory is 800 KB.

– UNDERSTAND **What does next fit change compared with first fit?** 2 marks

Next fit resumes the scan from the hole after the last placement, instead of restarting from the first hole.

– APPLY **First fit with holes 100 KB, 500 KB and 200 KB. Requests are 150 KB then 300 KB. Show the placements and leftovers.** 4 marks

150 KB takes 500 KB, leaving 350 KB. Holes become 100, 350 and 200 KB. 300 KB takes 350 KB, leaving 50 KB. Holes become 100, 50 and 200 KB.

– ANALYZE **Compaction can remove external fragmentation. Give one hardware or OS condition and one program condition needed for safe compaction.** 6 marks

The CPU must translate logical addresses to physical addresses at run time, and the OS must update the base register after moving a process. The program must not use absolute physical addresses.

BEYOND THE SYLLABUS

Not for the exam. Real kernels rarely use pure contiguous allocation for user processes; they use paging plus allocators such as buddy and slab. The same fragmentation question appears inside the kernel heap and in long-running servers, where small holes can stop a large allocation. Compaction reappears in garbage-collected runtimes and in memory defragmentation for virtual machines. Search for buddy allocator fragmentation, slab allocator, and execution-time relocation.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Logical address	Address generated by the CPU; the MMU translates it before memory is accessed.	4.1
Physical address	Address seen by the memory unit and placed on the memory bus.	4.1
Memory-management unit (MMU)	Hardware that maps logical addresses to physical addresses using base/limit registers or page tables.	4.1
Base and limit registers	Limit checks the logical address size; physical address = logical address + base after the check passes.	4.1
Swapping	Copying an entire process between main memory and a backing store to free memory.	4.2
Contiguous allocation	Each process occupies one continuous block of physical memory.	4.3
External fragmentation	Free memory splits into small holes that cannot satisfy a request although total free space is enough.	4.3

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– UNDERSTAND **Explain swapping. Why is a backing store needed, and what costs make swapping expensive?** 8 marks

Swapping copies a whole process out and later back in; Backing store is fast disk space for swapped images; Cost includes disk transfer time and context switching; Swapping whole processes is coarse compared with paging; It allows more processes to take turns in limited memory.

Glossary (terms used in this lesson)

Address space	The set of logical addresses a process may use.
Backing store	Disk area that holds swapped-out process images.
Base register	Register holding the start physical address added to a logical address.
Compaction	Moving allocated blocks to combine free memory into one large hole.
Demand paging	Loading pages only when they are referenced.
External fragmentation	Free memory split into holes too small for a request.
Frame	Fixed-size block of physical memory in a paging system.
Hole	A free block of memory in a contiguous allocation scheme.
Internal fragmentation	Wasted space inside an allocated block larger than the request.
Limit register	Register holding the size of a process's logical address space.
MMU	Hardware unit that translates logical addresses into physical addresses.
Page	Fixed-size block of logical memory in a paging system.
Segmentation	Division of a program into variable-length logical segments.
Swapping	Moving a whole process between main memory and a backing store.
Thrashing	High paging activity that leaves little time for useful execution.
TLB	Fast cache of recent page-table translations.
Virtual memory	Technique that lets a process use more logical memory than physical memory.

Sources and further reading

Textbooks

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts*, 10th: Main Memory and Virtual Memory chapters.
- Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems*: Memory Management and Virtual Memory chapters.
- William Stallings, *Operating Systems: Internals and Design Principles*: Memory Management and Virtual Memory chapters.
- Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*: Address Spaces, Paging and Swapping chapters.

Open web resources

Each link below was opened by the book script on 07 Nov 2026 and returned a page; the pages were not read line by line.

- Linux kernel documentation: www.kernel.org/doc/html/latest
- mmap(2) Linux manual page: man7.org/linux/man-pages/man2/mmap.2.html
- GNU C Library manual: Memory Allocation: www.gnu.org/software/libc/manual/html_node/Memory-Allocation.html

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Operating Systems (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.