Lab 09

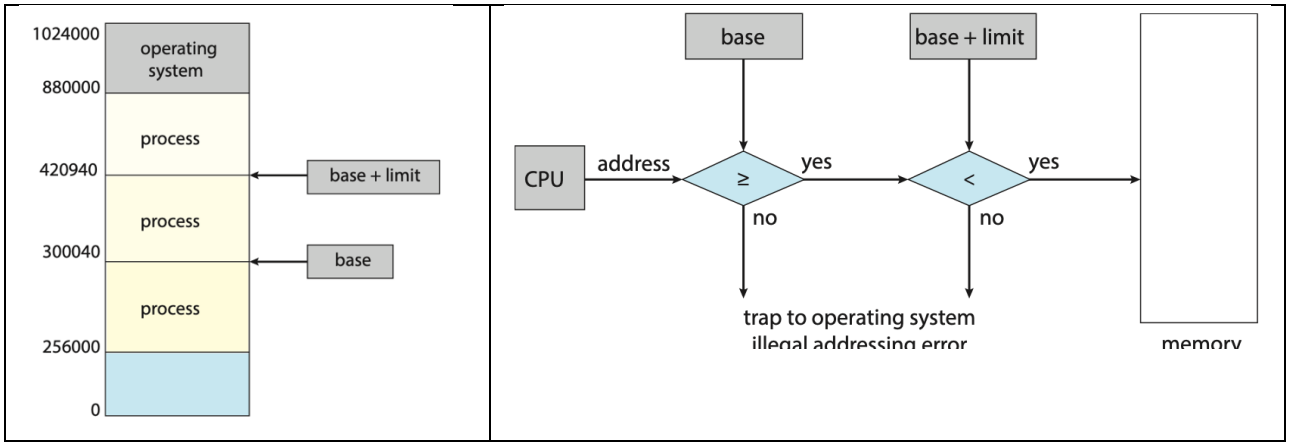
Main Memory

(Chapter 9)

Section 1: Summary – Self-study before entering the lab

Background

1. **Basic Hardware and Protection**
   - **CPU Access**: The CPU can only directly access registers and main memory. Any instructions or data must be in one of these locations before they can be executed.
   - **Memory Stall**: Accessing main memory takes many CPU cycles, which can cause the processor to "stall." To mitigate this, fast **cache** memory is placed between the CPU and main memory.
   - **Protection (Base and Limit Registers)**: To ensure processes do not interfere with each other or the OS, hardware uses a pair of registers:
     - **Base Register**: Holds the smallest legal physical memory address.
     - **Limit Register**: Specifies the size of the memory range.
     - Every address generated by a user process is checked against these registers; an out-of-bounds access results in a **trap** to the OS.



2. **Address Binding**

   Programs are typically stored on disk as binary executables. Before execution, they must be brought into memory and placed in a process context. Addresses go through three stages of binding:

   1) **Compile Time**: If the memory location is known at compile time, absolute code is generated. If the location changes later, the program must be recompiled.
   2) **Load Time**: If the location is unknown at compile time, the compiler generates relocatable code, and final binding happens when the program is loaded into memory.
   3) **Execution (Run) Time**: If a process can move during execution, binding is delayed until run time. This requires special hardware support and is used by most modern operating systems.

3. **Logical vs. Physical Address Space**
   - **Logical (Virtual) Address**: An address generated by the CPU.
   - **Physical Address**: The actual address seen by the memory unit.
   - **MMU (Memory Management Unit)**: A hardware device that performs the run-time mapping from virtual to physical addresses. A simple version uses a relocation register (base) that is added to every logical address generated by a user process

4. **Dynamic Loading and Linking**
   - **Dynamic Loading**: A routine is not loaded into memory until it is actually called. This improves memory utilization by only loading code that is needed (e.g., error-handling routines).
   - **Dynamic Linking (DLLs)**: Linking of system libraries is postponed until execution time. This allows multiple processes to share a single copy of a library (like the standard C library) in memory, saving significant space.

Exercise: Hardware Address Protection

Scenario: A computer system uses **base and limit registers** for memory protection. A process is currently loaded into memory with the following register values:
- Base Register: 300,040
- Limit Register: 120,900

Task: Determine whether the following **Logical Addresses** generated by the CPU are legal. If they are legal, calculate the resulting **Physical Address**.
1. Logical Address: 14,346
2. Logical Address: 125,000
3. Logical Address: 0

Solution:

1. Logical Address: 14,346
   - Check Legality: Is 14,346 < 120,900? Yes.
   - Physical Address: 300,040 + 14,346 = 314,386

2. Logical Address: 125,000
   - Check Legality: Is 125,000 < 120,900? No.
   - Result: This access results in a trap to the operating system (illegal addressing error).

3. Logical Address: 0
   - Check Legality: Is 0 < 120,900? Yes.
   - Physical Address: 300,040 + 0 = 300,040 (This is the start of the process's physical memory space).

Continuous Memory Allocation

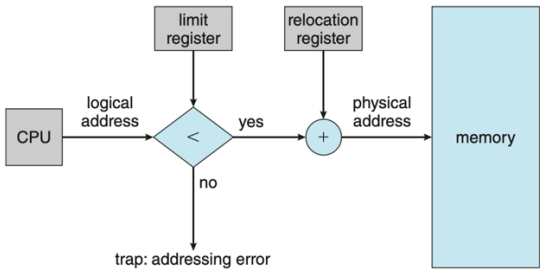
5. **Basic Memory Partitioning**

Main memory must support both the operating system and user processes. In this scheme:
- **Resident Operating System:** Usually held in low memory addresses along with the interrupt vector (though some systems use high memory).
- **User Processes:** Held in the remaining memory. Each process is allocated one contiguous block of space.

6. Hardware Support (Protection)

To prevent processes from accessing memory they do not own, the system uses relocation (base) and limit registers:

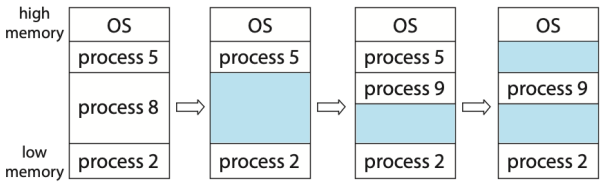
- **Relocation Register:** Stores the value of the smallest physical address of the process.
- **Limit Register:** Stores the range of logical addresses. Every address generated by the CPU must be less than the limit register.
- **Translation:** The MMU maps logical addresses dynamically by adding the relocation register value to them before they are sent to memory.



7. Variable Partition Scheme

The OS maintains a table of holes (blocks of available memory).

- When a process arrives, it is allocated space from a hole large enough to fit it.
- When a process terminates, it frees its block, and adjacent holes are merged into one larger hole.



8. Dynamic Storage-Allocation Strategies

To decide which hole to use for a request of size n, three strategies are commonly used:

- **First-fit**: Allocate the first hole found that is big enough.
- **Best-fit**: Allocate the smallest hole that is big enough (requires searching the entire list). It produces the smallest leftover hole.
- **Worst-fit**: Allocate the largest hole (requires searching the entire list). It produces the largest leftover hole.

9. Fragmentation

- **External Fragmentation**: Total memory exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation**: Memory allocated to a process is slightly larger than requested; the unused difference is "internal" to that partition.
- **Compaction**: A solution to external fragmentation where the OS shuffles memory contents to place all free memory together in one large block.

Example 1: Address Relocation and Protection

A process has a Relocation Register of 100,040 and a Limit Register of 74,600.

1. Logical Address 50,000:
   - Check: 50,000 < 74,600 (Limit)? Yes (Legal).
   - Physical Address: 100,040 + 50,000 = 150,040.
2. Logical Address 80,000:
   - Check: 80,000 < 74,600 (Limit)? No (Illegal).
   - Result: A trap (addressing error) occurs.

Example 2: Storage Allocation Strategies

Given memory holes of sizes: 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order). A process requests 212 KB.

- First-fit: Skips 100 KB, selects the 500 KB hole (the first one that fits).
- Best-fit: Selects the 300 KB hole (the smallest that fits; leftover = 88 KB).
- Worst-fit: Selects the 600 KB hole (the largest available; leftover = 388 KB).

Example 3: Fragmentation (50-Percent Rule)

Statistical analysis shows that for **N** allocated blocks, approximately 0.5 N blocks are lost to fragmentation.

- If a system has 1,000 allocated blocks, roughly 500 additional blocks' worth of space might be unusable due to fragmentation, meaning one-third of total memory may be wasted.

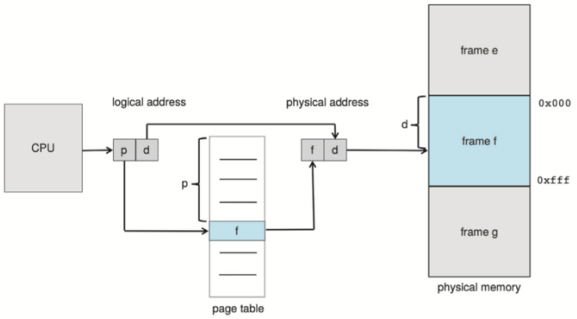
Paging

10. Basic Method

- **Frames and Pages**: Physical memory is broken into fixed-sized blocks called frames. Logical memory is broken into blocks of the same size called pages.
- **Address Translation**: Every address generated by the CPU is divided into two parts:
  - **Page Number (p):** Used as an index into a page table.
  - **Page Offset (d):** Combined with the base address of the frame to define the physical memory address.

11. Hardware Support

- **Page Table Base Register (PTBR):** Points to the page table in main memory.
- **TLB (Translation Look-aside Buffer):** A small, fast-lookup hardware cache. Because accessing the page table in main memory is slow (requires two memory accesses for one data byte), the TLB stores recently used page-table entries.



12. Protection and Sharing

- **Valid-Invalid Bit:** Attached to each entry in the page table to indicate if the page is in the process's logical address space.
- **Shared Pages:** Paging allows multiple processes to share common code (like standard libraries). Only one copy of the "reentrant" code is kept in physical memory.

Exercise 1: Logical to Physical Translation
Scenario: A system has a page size of 4 bytes and a physical memory of 32 bytes (8 frames).
Consider the following Page Table:

- Page 0 → Frame 5
- Page 1 → Frame 6
- Page 2 → Frame 1
- Page 3 → Frame 2

Task: Translate Logical Address 13 to a Physical Address.

1. Find Page Number (p): 13 / 4 = 3 (Integer division).
2. Find Offset (d): 13 % 4 = 1.
3. Lookup Page Table: Page 3 points to Frame 2.
4. Calculate Physical Address: (Frame x Page Size) + Offset = (2 x 4) + 1 = 9.

Exercise 2: Effective Access Time (EAT)

Scenario:

- TLB Lookup time = **10 ns**
- Main Memory Access time = **100 ns**
- TLB Hit Ratio = **95%**

Task: Calculate the EAT.

- **Hit Case:** Time to check TLB + Time to access memory = 10 + 100 = 110 ns.
- **Miss Case:** Time to check TLB + Time to access Page Table (Memory) + Time to access data (Memory) = 10 + 100 + 100 = 210 ns.
- **EAT Formula:** (0.95 x 110) + (0.05 x 210) = 104.5 + 10.5 = 115 ns.

Exercise 3: Page Table Size Calculation
Scenario: A 32-bit logical address space with a 4 KB (2^12) page size. Each Page Table Entry (PTE) is 4 bytes.

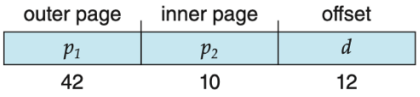
- **Number of Pages:** 2^32 / 2^12 = 2^20 = 1,048,576 pages.
- **Size of Page Table:** 2^20 entries x 4 bytes/entry = 4,194,304 bytes ~= 4 MB.

Structure of Page Tables

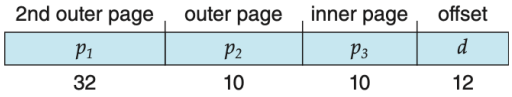
13. Hierarchical Paging

This approach breaks the page table into multiple levels so that the page table itself is paged.

- **Two-Level Paging**: In a 32-bit system with 4 KB pages, a logical address is divided into a 10-bit outer page number (p1), a 10-bit inner page offset (p2), and a 12-bit page offset (d).

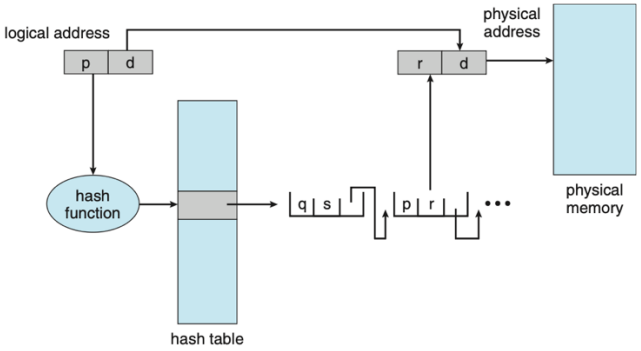


- **64-Bit Systems**: Hierarchical paging is less effective here because it would require many levels (e.g., 7 levels for some architectures), leading to a prohibitive number of memory accesses.



14. Hashed Page Tables

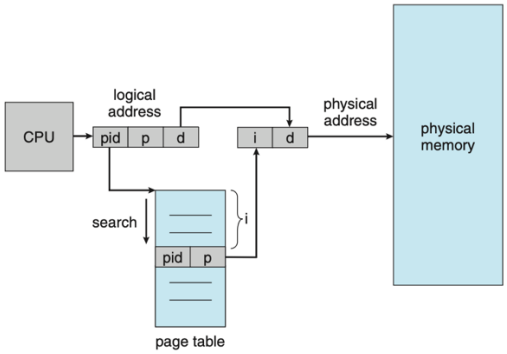
Common for address spaces larger than 32 bits. The virtual page number is hashed into a hash table.



- Each hash table entry contains a linked list of elements that hash to the same location to handle collisions.
- Each element stores the virtual page number and the value of the mapped page frame.

15. Inverted Page Tables

Instead of each process having its own page table, the system has one entry for each real page (frame) of physical memory.



- Each entry consists of the virtual address of the page and information about the process that owns it (Address-Space Identifier or PID).
- **Pros:** Decreases memory needed for page tables.
- Cons: Increases search time; usually requires a hash table to improve performance.

Exercise 1: Hierarchical Paging (Two-Level)

Scenario: A 32-bit system uses 4 KB (2^12) pages. Each page table entry (PTE) is 4 bytes.

- Calculate the number of entries in a single-level page table:
  2^{32} / 2^{12} = 2^{20} ~= 1,048,576 entries.
- Calculate total size of the page table:
  2^20 entries x 4 bytes/entry = 4 MB.
- If we use Two-Level paging (p1, p2, d):
  - How many bits are for p1 and p2 if we want the outer table to fit in one 4 KB page?
  - One 4 KB page holds 4096 / 4 = 1024 (2^10) entries.
  - So, p1 = 10 bits. p2 would also be 10 bits (32 - 10 - 12 = 10).

Exercise 2: Inverted Page Table Memory Savings

Scenario: A system has 512 MB of physical RAM and a 4 KB page size.

1. Calculate the number of entries in the Inverted Page Table:
   - Physical Frames = 512 MB / 4 KB = 524,288 / 4 = 131,072 entries.
2. Compare to Conventional Paging:

If there are 10 active processes, each with a 4 GB virtual address space:

- Conventional: 10 processes x 1 million entries/process = 10 million entries.
- Inverted: Always 131,072 entries total for the whole system.

Exercise 3: Logical Address Breakdown

Scenario: A 64-bit system uses 16 KB pages and 4-level hierarchical paging.

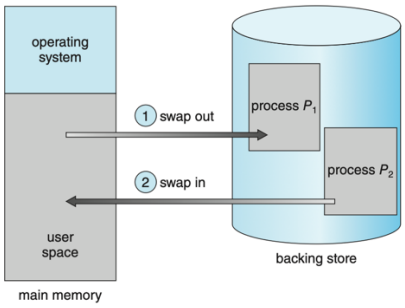
- Offset (d): 16 KB = 2^14, so d = 14 bits.
- Remaining Bits: 64 - 14 = 50 bits.

1. If we use 4 levels where the top level is 14 bits and the next three are 12 bits each:

Address: [p1 (14) | p2 (12) | p3 (12) | p4 (12) | d (14)].

Note: In practice, x86-64 uses 48-bit virtual addresses with four 9-bit levels and a 12-bit offset.

Swapping



16. Basic Concept

A process can be **swapped** temporarily out of memory to a **backing store** (typically a fast disk) and then brought back into memory for continued execution. This allows the system to support more processes than can fit in RAM simultaneously.

17. Standard Swapping

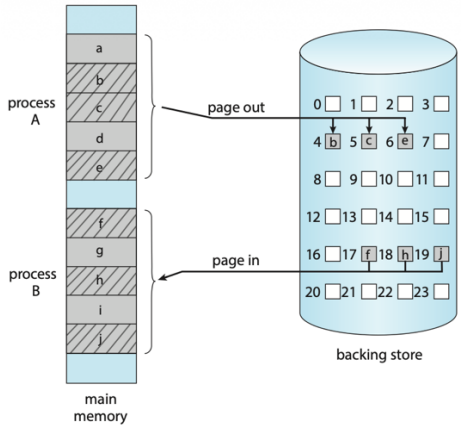
In standard swapping, entire processes are moved between main memory and the backing store.

- **Roll out, roll in:** A variant used for priority-based scheduling; lower-priority processes are swapped out so higher-priority processes can be loaded and executed.
- **Ready Queue:** The system maintains a ready queue of processes whose images are on the backing store or in memory.

18. Swapping with Paging (Modern Approach)

Standard swapping is rarely used in modern operating systems because moving entire processes is time-consuming. Instead, modern systems use **Page Swapping** (or Paging).

- Only specific **pages** of a process are swapped rather than the entire process.
- The term "swapping" in modern contexts usually refers to **paging**, where a "page out" moves a page from memory to the backing store, and a "page in" moves it back.



19. Constraints on Swapping

- **Pending I/O:** A process cannot be swapped if it has a pending I/O operation. If the I/O is using the process's memory buffers, swapping the process out would cause the I/O to over-write the memory of a different process that was moved into that space.
- **Solution:** Either never swap processes with pending I/O or execute I/O operations only into operating-system buffers (which are then copied to user memory).

Exercise 1: Calculating Transfer Time

Scenario: A process of size **100 MB** needs to be swapped out to a backing store. The transfer rate of the hard disk is **50 MB/s**.

1. **Calculate the Swap-out time:**
   - Time = Size / Transfer Rate
   - 100 MB / 50 MB/s= 2 seconds.
2. **Calculate total Swap time (Out and In):**
   - If the process is swapped out and then a 100 MB process is swapped in:
   - 2 s (out) + 2 s (in) = 4 seconds.

Note: This exercise illustrates why swapping entire processes is slow and why modern systems prefer paging.

Exercise 2: Impact of Latency

Scenario: Using the same **100 MB** process and **50 MB/s** disk, assume the disk has a **latency (seek time)** of **8 milliseconds (0.008 s)**.

1. **Calculate the effective swap-out time:**
   - Time = Latency + (Size / Transfer Rate
   - 0.008 s + 2 s = 2.008 seconds.
2. **Conclusion:** For large processes, the transfer time dominates the latency. For very small pages (e.g., 4 KB), the latency would become the more significant factor.

Exercise 3: Paging vs. Full Swapping

Scenario: A process is **1 GB**, but only **10 MB** of its code is currently active.

1. **Standard Swapping Time:** (Assume 100 MB/s disk)
   1024 MB / 100 MB/s ~= 10.24 seconds.

2. **Page-in Time (Demand Paging):**
   If only the active 10 MB is brought in:
   10 MB / 100 MB/s = 0.1 seconds.

   Observation: Paging is over 100 times faster in this scenario, highlighting why modern OSs do not swap entire processes.

All the figures are captured from the textbook "Silberschatz, A. et al., "Operating Systems Concepts", 10th Edition, John Wiley, 2018"

Section 2: Discussion – selected questions only (1 Hour)

1. Consider a system with 16MB of memory and fixed partitions, all of size 64KB, answer the following:

    a. Identify the minimum number of bits needed in an entry in the process table to record the partition to which a process has been allocated.
    b. Identify the number of bits in the limit register.

2. Consider a memory with a contiguous allocation scheme consisting of 4 partitions of 600K, 400K, 500K, and 300K (in order) respectively. Given a set of processes with 200K, 300K, 470K, and 500 K (in order):

    a. Using first fit, how would you allocate the memory to them?
    b. Using best fit, how would you allocate the memory to them?
    c. Using worst fit, how would you allocate the memory to them?

3. Consider a logical address space of sixty-four pages of 1024 words each mapped onto a physical memory of 32 frames.

    a. How many bits are there in the logical address?
    b. How many bits are there in the physical address?

4. a. Consider a single level paging system with the page table stored in memory.

    Assume the usage of Translation Look-aside Buffer (TLB) with 70 percentage hit ratio, and also assume that it takes 30 nanoseconds to search the TLB and 200 nanoseconds to access memory. Identify the effective memory access time.

    b. Repeat the above calculations for a two level paging system.

5. On a simple paging system, Translation Lookaside Buffer (TLB) holds the most active page entries and the full page table is stored in the main memory. If reference to TLB takes 10 ns, and memory reference takes 180ns, identify the hit ratio to achieve an effective access time of 200ns.

6. Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512MB of physical memory. How many entries are there in each of the following:

    a. A conventional single-level page table
    b. An inverted page table

7. Identify the page numbers and offsets for the following address references (Given as decimal numbers). Assume the page size is 1-KB.

    a. 19366
    b. 30000
    c. 256

8. Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0, 430
b. 1, 10
c. 2, 500
d. 3, 400
e. 4, 112

9. Consider a system that uses a two-level page table with page size of 4KB and 32-bit logical addresses. The first 8 bits of the address serve as the index into the first-level page table.

a. How many bits are needed to specify the second-level index?
b. How many entries are in a level-one page table?
c. How many entries are in a level-two page table?
d. How many pages are in the virtual address space?

Section 3: Practical exercises

Exercise 1: [Logical vs Physical Address]
In Linux, every C program operates in a **Virtual (Logical) Address Space**. The addresses you see when printing a pointer are not physical RAM locations; they are translated by the MMU.

```
//p1.c
#include <stdio.h>
#include <unistd.h>

int main() {
    int x = 10;
    // This prints the LOGICAL address.
    // The OS/MMU maps this to a PHYSICAL address in RAM.
    printf("Logical Address of x: %p\n", (void*)&x);
    sleep(100);
    return 0;
}
```

Compile and run the program and observe the output.

```
$gcc -o p1 p1.c
$./p1&
```

& → run the process in the background.

Obtain the process id for process p1.

```
$cat /proc/[pid]/maps
```

Exercise 2: [Contiguous Memory Allocation & Fragmentation**]**

Compile and run the program and observe the output.

$gcc -o p2 p2.c
$./p2

```
//p2.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("--- Contiguous Allocation & Fragmentation Demo ---\n\n");

    // 1. Simulate allocating several contiguous blocks
    // Imagine these are three processes being loaded into memory
    void* block1 = malloc(128); // Process A
    void* block2 = malloc(128); // Process B
    void* block3 = malloc(128); // Process C

    printf("Allocated three 128-byte blocks.\n");
    printf("Block 1 Address: %p\n", block1);
    printf("Block 2 Address: %p\n", block2);
    printf("Block 3 Address: %p\n\n", block3);
```

```
    // 2. Simulate "External Fragmentation"
    // We free the middle block (Process B), creating a 'hole' in memory.
    printf("Freeing Block 2 (Process B) to create a hole...\n");
    free(block2);

    // 3. The Dilemma:
    // We now have 128 bytes free in the middle and more free space at the end.
    // However, if we try to allocate a contiguous 200-byte block,
    // the 'hole' left by Block 2 is too small, even if total system memory is
enough.

    printf("Attempting to allocate a 200-byte block...\n");
    void* block4 = malloc(200);

    if (block4 != NULL) {
        printf("Block 4 allocated at: %p\n", block4);
        printf("Note: The allocator had to find a NEW contiguous space,\n");
        printf("leaving the 128-byte hole from Block 2 unused (External
Fragmentation).\n");
    }

    // Clean up
    free(block1);
    free(block3);
    free(block4);

    return 0;
}
```

When you call malloc(128), the allocator searches for a single, unbroken sequence of 128 bytes. In the code above, block1, block2, and block3 are typically placed one after another in the heap, mimicking how an OS places processes in physical memory.

When we free(block2), we create a **hole** of 128 bytes.

- Total free memory = 128 bytes (the hole) + all memory after block3.
- If a new request (e.g., 200 bytes) arrives, it **cannot** fit into the 128-byte hole. The allocator must skip that hole and look further down. The 128-byte gap remains wasted unless a smaller process comes along to fill it.

3. Internal Fragmentation (Implicit)

In many C libraries, if you request malloc(10), the library might actually allocate 16 or 32 bytes due to alignment requirements.

- **Calculation:** If you request 10 bytes and the system allocates 16:
- **Internal Fragment** = 16 - 10 = 6 bytes wasted inside the block.

Exercise 3: [Memory Allocation and Page Touching]

This program allocates exactly **10 pages** of memory. Because of **Dynamic Loading**, the OS won't give the program physical frames until we actually "touch" (write to) them.

```
//p3.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    long page_size = sysconf(_SC_PAGESIZE);
    int num_pages = 10;
    size_t total_size = page_size * num_pages;

    printf("Step 1: System Page Size is %ld bytes.\n", page_size);
    printf("Step 2: Allocating %d pages (%zu bytes total).\n", num_pages,
total_size);

    // Logical Address Space is created here (Virtual memory)
    char *memory = malloc(total_size);

    printf("Process ID: %d\n", getpid());
    printf("Memory allocated at logical address: %p\n", (void*)memory);
    printf("Press Enter to 'touch' 5 pages and see physical memory increase...");
    getchar();

    // Accessing memory triggers the MMU to map logical pages to physical frames
    for (int i = 0; i < 5; i++) {
        memory[i * page_size] = 'A'; // Write to the start of each page
    }

    printf("Touched 5 pages. Check 'RSS' in top/ps now.\n");
    printf("Press Enter to exit...");
    getchar();

    free(memory);
    return 0;
}
```

1. Compile and Run:

```
gcc p3.c -o p3 && ./p3
```

2. Observe Virtual vs. Physical (In a second terminal):

Run `ps -o pid,vsz,rss,comm -p [PID]`

- VSZ (Virtual Set Size): This represents the Logical Address Space. It will be large because it includes the 10 allocated pages.
-
- RSS (Resident Set Size): This is the Physical Address Space currently in RAM.

3. The "Touch" Effect:

- Before pressing Enter: RSS will be low (the OS hasn't mapped the frames yet).
- After pressing Enter: RSS will increase by approximately **20 KB** (5 pages x 4 KB).

Theory Connection: This demonstrates **Execution-time binding**. The logical address is only bound to a physical frame when the reference is made.

4. Deep Inspection with **pmap**:

   To see the actual Page Table mappings for your process, run:

   pmap -x [PID]

   - Address: The starting Logical Address of your memory block.
   - Kbytes: The size of the allocation.
   - RSS: How much is actually in physical memory.
   - Mapping: Shows if it's the [heap], stack, or a shared library.

Exercise 4: [structure of the page table]

While C programs cannot directly "see" the hardware page table (the OS abstracts this), we can write a program that simulates how a 32-bit address is broken down into a two-level structure using bitwise operations.

Compile and run the program

```
//p4.c
#include <stdio.h>

// Assuming 4 KB page size (12 bits for offset)
// Level 1: 10 bits, Level 2: 10 bits, Offset: 12 bits
#define OFFSET_BITS 12
#define P2_BITS 10
#define P1_BITS 10

void analyze_address(unsigned int address) {
    unsigned int p1, p2, d;

    // Extract d (the last 12 bits)
    d = address & 0xFFF;

    // Extract p2 (the middle 10 bits)
    p2 = (address >> OFFSET_BITS) & 0x3FF;

    // Extract p1 (the first 10 bits)
    p1 = (address >> (OFFSET_BITS + P2_BITS)) & 0x3FF;

    printf("Logical Address: 0x%08X\n", address);
    printf("  -> Outer Page Table Index (p1): %u\n", p1);
    printf("  -> Inner Page Table Index (p2): %u\n", p2);
    printf("  -> Page Offset (d): %u\n", d);
}

int main() {
    // Example address similar to your tutorial
    unsigned int test_addr = 0x00401234;
    analyze_address(test_addr);
    return 0;
}
```

1. Linux Demo: Observing the Hierarchy

You can observe the complexity of the page table structure on your current machine using Linux tools.

2. Check Paging Levels

Modern x86-64 Linux systems typically use **4-level paging**. You can see the kernel's view of this in the boot logs.

- **Command:** `sudo dmesg | grep -i "paging"`
- **Expected Output:** You may see references to "4-level paging" or "5-level paging" enabled.

3. Inspect Memory Overhead

Because hierarchical page tables save space by not creating tables for unused memory, you can see how much memory the OS is currently using just to *store* these tables.

- **Command:** `grep PageTables /proc/meminfo`
- **Significance:** This value represents the total memory dedicated to the structures.

4. Visualize with pmap

Use the pmap command on a running process to see the "sparse" nature of memory that makes hierarchical tables necessary.

- **Command:** `pmap -v` (using the current shell's PID).
- **Observation:** You will see large gaps between addresses (e.g., between the heap and the stack). A single-level table would require entries for all those empty spaces, but a **hierarchical** or **hashed** table only allocates entries for the segments you actually use.

Concept	Slide Reference	Linux/C Verification
Space Overhead	Page table can be 4MB per process.	`grep PageTables /proc/meminfo` shows total system overhead.
Address Splitting	Address is divided into p1, p2, d.	C code bit-shifting (>>) demonstrates the split.
Sparse Memory	Tables are huge for 64-bit systems.	pmap shows the vast gaps in a process's address space.

Exercise 5: [Swapping]

Standard swapping is usually disabled in modern Linux unless memory is low. This program allocates a massive amount of memory and fills it with data, forcing the OS to consider swapping out idle processes to make room.

Compile and run the program

```
//p5.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main() {
    // Determine 1GB allocation
    size_t size = 1024 * 1024 * 1024;
    printf("Process PID: %d\n", getpid());
    printf("Attempting to allocate and WRITE to 1GB of memory...\n");

    char *buffer = malloc(size);
    if (buffer == NULL) {
        perror("Failed to allocate memory");
        return 1;
    }

    // Swapping with Paging: We must write data to force physical frame allocation
    // This is called "touching" the pages.
    for (size_t i = 0; i < size; i += 4096) {
        buffer[i] = 1;
    }

    printf("1GB written. Check 'vmstat' or 'free' to see Swap usage.\n");
    printf("Press Enter to exit and release memory.\n");
    getchar();

    free(buffer);
    return 0;
}
```

Use these commands while the C program above is running to observe swapping behaviour:

A. Identify the Backing Store

Run: `swapon --show`

- **Concept:** This identifies your active swap partitions or files (the "Backing Store").
- **Observation:** Note the "Type" (partition or file) and the "Used" amount.

B. Observe Real-Time Swapping (Transfer Time)

Run: `vmstat 1`

- **si (swap-in):** Memory being brought back from the backing store.
- **so (swap-out):** Memory being moved to the backing store.
- **Theory Connection:** If these numbers are high, your **context switch time** is suffering due to disk transfer speeds.

C. Process-Specific Swap Usage

Run: `cat /proc/[PID]/status | grep VmSwap`

- **Concept:** This shows exactly how much of your specific C program's logical address space has been moved to the backing store.