UNIT-V MEMORY MANAGEMENT

Swapping:

Swapping is a **memory management technique** in which processes are temporarily moved (swapped) from main memory (RAM) to a backing store (usually disk) and brought back into RAM when needed. This helps the system run more processes than the size of physical memory would normally allow.

How Swapping Works:

1. **When memory is full** and a new process needs to be executed, the OS may **swap out** (move) an inactive or low-priority process from RAM to disk (swap space).
2. The new process is then **swapped in** (loaded) from disk into RAM.
3. When the swapped-out process is needed again, it will be **swapped back** into RAM, possibly replacing another process.

Example of Swapping:

Suppose we have **100 MB RAM** and three processes:

- Process P1 → 40 MB
- Process P2 → 50 MB
- Process P3 → 60 MB

Initially, P1 and P2 fit into RAM (40 + 50 = 90 MB).
P3 (60 MB) cannot fit because only 10 MB is free.

Solution:

- The OS swaps out **P1 (40 MB)** to disk.
- This frees 40 MB, so RAM now has 50 MB (P2) + 40 MB free = 90 MB free.
- P3 (60 MB) can now be swapped in.

So now RAM has:

- P2 (50 MB) + P3 (60 MB) = 110 MB → but only 100 MB allowed.
  To fix, OS manages by replacing P2 with P3, keeping only needed processes active.

Allocation of swap space:

- **Swap space** is a portion of the hard disk used as an extension of main memory (RAM).
- It stores inactive pages or whole processes that are swapped out of RAM when physical memory is full.
- In UNIX, swap space is usually created either as a separate partition or a swap file.

Swap Space (100 blocks on disk)

```
+-----------------------------------------------------+
| 0   1   2   3   4  ...  98   99                     |
|              ALL FREE                               |
+-----------------------------------------------------+
```

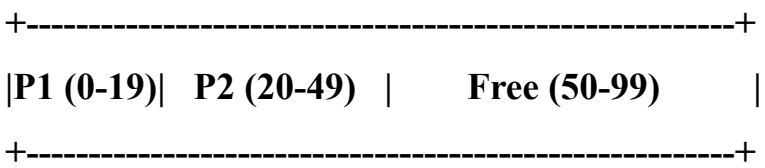
Initial Swap Map:

```
+------------+------------+
| Start Block|   Length   |
+------------+------------+
|    0       |    100     |  <-- entire swap area is free
+------------+------------+
```

After Process Allocations

1. P1 needs 20 blocks → allocates 0–19
2. P2 needs 30 blocks → allocates 20–49

Swap Space:



Swap Map:

Process	Start Block	Length
P1	0	20
P2	20	30
Free	50	50

1. **During Process Creation**
    - When a process is created, the UNIX kernel checks if there is enough **swap space available**.
    - UNIX requires that **swap space is reserved for the entire virtual memory needs of the process** (even if not used immediately).
    - This guarantees that if the process grows, it won't fail due to lack of memory.
2. **Swapping Out**
    - If RAM is full, the kernel moves **entire processes** or some pages to swap space.
    - The location (block numbers) of each swapped page is recorded in the **process table**.
3. **Swapping In**
    - When the process is scheduled to run again, the swapped-out pages are brought back into RAM from swap space.
4. **Types of Swap Allocation in UNIX**
    - **Pre-allocation**: Entire swap space for a process is allocated when the process starts. (older UNIX systems)

- **Lazy Allocation**: Swap space is allocated only when pages are actually swapped out (modern UNIX/Linux).

Example:

- Process P1 needs 20 MB.
- Process P2 needs 50 MB.
- System RAM = 64 MB.

- UNIX allocates **virtual swap space** for P1 (20 MB) and P2 (50 MB) during creation.
- If P1 and P2 both run, part of their memory may be swapped out. For example:
  - P1 keeps 10 MB in RAM, 10 MB in swap.
  - P2 keeps 30 MB in RAM, 20 MB in swap.

This ensures both can run, even though RAM is limited.

Swapping Process In

Swapping in = Loading the process from swap space (disk) back into RAM.

Swapping In is the operation where a process that was previously swapped out (moved to disk) is brought back into the main memory (RAM) so it can continue execution.

When a swapped-out process needs to run or be resumed, the OS locates its saved state in the swap space (on disk).

The entire process image (code, data, stack, heap, etc.) is read from disk back into RAM.

The process is then restored to the ready or running state, and execution continues from where it left off.

Swapping Process Out

Swapping out = Moving the process from RAM to disk swap space to free memory.

Swapping Out is the operation where a process currently in main memory is moved out to swap space on disk to free RAM.

When the system experiences memory pressure and needs to free RAM for other processes, the OS chooses a process (usually one that's idle or low priority).

The entire process's memory image is copied from RAM into the swap space.

The process is then marked as swapped out or blocked until it is swapped in again.

The RAM used by this process is freed and made available for other processes.

Example:

A text editor (Process A)

A web browser (Process B)

A heavy video editing app (Process C)

Step 1: System Memory Pressure

Process A and B are active and being used.

Process C is idle but loaded in RAM, taking 2 GB.

Now, you open a new heavy program (Process D), which requires 1 GB of RAM.

Your system runs low on free RAM.

Step 2: Swapping Out a Process (Swapping Out)

The OS decides to swap out the idle Process C to free 2 GB RAM.

Process C's memory contents are copied from RAM to the swap space on disk.

Process C is marked as swapped out, and its RAM is freed.

Now, Process D can load into RAM and run.

Step 3: Swapping In a Process (Swapping In)

Later, you want to switch back to Process C.

The OS will swap Process C back into RAM by reading its saved image from the disk swap space.

Process C resumes running from where it was paused.

If RAM is full again, the OS might swap out another less active process to make room.

Command Examples (on Linux):

Check swap space usage:

free -h //Output shows swap space size and usage.

	total	used	free	shared	buff/cache	available
Mem:	7.7G	5.1G	500M	300M	2.1G	2.0G
Swap:	2.0G	0B	2.0G			

total → Total installed memory (RAM).

used → Memory currently used (excluding buffers/cache).

free → Completely unused memory.

shared → Memory used by tmpfs (shared between processes, often for `/dev/shm`).

buff/cache → Memory used by kernel buffers and page cache (can be freed if needed).

available → An estimate of memory available for new applications without swapping.

Here, swap space is 2GB total, currently unused

NOTE:

You cannot manually "swap in" specific data or processes using a bash command.

Swapping is triggered automatically by the OS based on demand.

What you can do is ensure you have enough free RAM and swap space configured.

You can monitor swapping activity but not manually force swap-ins.

Demand Paging

- **Demand Paging** is a memory management scheme used in Unix (and most modern OSes) where **pages of a process are loaded into memory only when they are needed** (on demand).
- Instead of loading the entire process into RAM, the OS loads pages lazily.
- This reduces memory usage and startup time, allowing multiple processes to run efficiently.

How it Works in Unix

1. **Process Creation**
    - When a process is created (e.g., using `fork()` and `exec()`), only part of its address space is loaded.
    - The process image (code, data, stack) exists on **disk** but not fully in **RAM**.
2. **Page Table Setup**
    - The OS maintains a **page table** for each process.
    - Initially, page table entries are marked as **invalid** (not in memory).

3. **Page Fault**
    - When the process tries to access an instruction or data on a page that is **not in memory**, the **MMU (Memory Management Unit)** triggers a **page fault**.
    - Control is transferred to the **OS kernel**.
4. **Handling the Page Fault**
   The OS checks:
    - Is the access valid? (If invalid → segmentation fault).
    - If valid but missing, the OS brings the required page from disk (swap space or executable file) into RAM.
5. **Updating Page Table**
    - After loading, the page table entry is updated as **valid**.
    - The process is restarted from the point of interruption.

Example Scenario:

- Suppose a program has **5 pages**: P1, P2, P3, P4, P5.
- The physical memory (RAM) can hold only **3 pages** at a time.
- The process execution requires pages in the sequence: **P1, P2, P3, P4, P1, P5, P2**.

Step-by-Step Execution:

1. **P1 requested** → not in RAM → Page fault → Load P1.
   RAM = [P1]
2. **P2 requested** → not in RAM → Page fault → Load P2.
   RAM = [P1, P2]

3. **P3 requested** → not in RAM → Page fault → Load P3.
   RAM = [P1, P2, P3]

4. **P4 requested** → RAM full → Use replacement policy (say FIFO).
   → Remove P1 → Load P4.
   RAM = [P2, P3, P4]

5. **P1 requested** → not in RAM → Page fault → Replace P2.
   RAM = [P1, P3, P4]

6. **P5 requested** → Page fault → Replace P3.
   RAM = [P1, P5, P4]

7. **P2 requested** → Page fault → Replace P4.
   RAM = [P1, P5, P2]

Final Outcome:

- **Total Requests** = 7
- **Page Faults** = 6 (almost every time, since memory is small)

Data Structures in Demand Paging

1. Page Table

- Each process has a page table.
- It maps virtual page numbers to physical frame numbers.
- Each entry contains:
    - Valid/Invalid bit → tells if the page is in RAM.
    - Frame number → location in physical memory.
    - Protection bits → read/write/execute permissions.

    - Referenced/Modified bit → for page replacement decisions.

2. Frame Table

- Keeps track of all physical memory frames.
- Shows which frame is free or which process/page occupies it.
- Helps the OS find a free frame when a new page must be loaded.

3. Swap Area (Secondary Storage Table)

- Keeps information about where each page is stored on disk.
- When a page fault occurs, the OS checks here to bring the page into RAM.

4. Free Frame List

- A list of available (unused) frames in RAM.
- Used to quickly allocate memory for a demanded page.
- If the list is empty → page replacement algorithm (FIFO, LRU, etc.) is triggered.

Example:

- Virtual Address Space = 4 pages {P0, P1, P2, P3}
- Physical Memory = 2 frames (F0, F1)

Execution Sequence: P1 → P2 → P3

1. Access P1 → Page fault
    - Allocate F0 to P1.

    - Update page table: P1 → (Valid=1, Frame=F0).
    - Free list = [F1].
2. Access P2 → Page fault
    - Allocate F1 to P2.
    - Update page table: P2 → (Valid=1, Frame=F1).
    - Free list = [].
3. Access P3 → Page fault
    - No free frames → Replacement needed.
    - Suppose FIFO → Replace P1 in F0.
    - Update page table: P3 → (Valid=1, Frame=F0), P1 → (Valid=0).

Page Stealer Process (Page Replacement Process)

- In demand paging, when a page fault occurs and no free frames are left in RAM, the OS needs to free some space. It ensures that RAM always has room for the next needed page.
- The Page Stealer Process is the part of the OS that decides which page to remove (steal) from RAM to bring in the required page.

How it works

1. A process needs a page → page fault occurs.
2. OS checks free frame list.
    - If empty → Page Stealer is activated.
3. Page Stealer selects a page in RAM to evict (based on replacement policy like FIFO, LRU, etc.).

4. If the selected page was modified → it is written back to disk.
5. The new required page is loaded into that freed frame.

Example:

- RAM has 3 frames: [P1, P2, P3]
- Process requests P4, but memory is full.
- Page Stealer Process is called.
- Suppose replacement policy = FIFO → P1 is removed.
- P4 is loaded in its place.
- RAM = [P4, P2, P3]

Page Fault

A page fault occurs when a program tries to access a page that is not in main memory (RAM) but is needed for execution.

Steps in Handling a Page Fault:

1. CPU checks the page table → finds **invalid bit** (page not in RAM).
2. OS **traps a kernel** (page fault handler).
3. If the page is on disk (swap area), the OS finds a free frame.
    - If no free frame → page replacement (page stealer).
4. The required page is loaded into RAM.
5. The page table is updated.
6. Process resumes from the faulting instruction.

Example:

- Process has pages P1, P2, P3.
- RAM can hold 2 pages.
- If CPU requests P3 but only P1 & P2 are in RAM → **Page Fault occurs**.
- OS replaces one page (say P1) with P3.

Types of Page Faults

1. **Minor (Soft) Page Fault** – page is in memory but not mapped correctly.
2. **Major (Hard) Page Fault** – page is not in memory; must be read from disk (slower).