Memory

Swapping

What is Swapping?

Temporarily moves process (or part) to backing store and reloads to main mem as needed.

- Allows total address space > physical mem → increases multiprogramming.
- Works with process scheduling ("roll out/roll in" lower priority processes).

Why use Swapping?

- Run programs larger than physical mem.
- Increase degree of multiprogramming.
- Improve CPU utilization & throughput (more ready processes).
- Efficient process mem allocation.
- Reduce I/O vs loading full program at once.

Standard Swapping

Moves entire process image between main mem & backing store.

Disadvantages:

- High transfer time for large processes (included in context switch time).

Swap Space Management

Secondary storage for swapping pages. Raw partition or large file. Primarily backs stack/heap.

Executable pages → demand-paged from file system & discarded if unmodified.

Swapping with Paging (Demand Paging)

Swaps individual pages instead of full process.

- If page not in mem → page fault → load page from backing store.
- No free frames → page-replacement selects victim frame.
- Modified victims written back; unmodified discarded.

Pros: Run large virtual address space on small physical mem.

Cons: Frequent page faults → performance hit.

Allocation Methods

Contiguous Mem Allocation

Each process occupies single block of physical mem.

- Uses base & limit registers → maps logical to physical addresses → enforces protection.

Allocation Strategies (Dynamic Storage Allocation)

- **First Fit:** First hole large enough (faster than Best Fit).
- **Best Fit:** Smallest hole large enough; smallest leftover hole.
- **Worst Fit:** Largest hole; largest leftover hole.
- **First/Best > Worst** for speed & utilization.

Non-Contiguous Mem Allocation

- **Segmentation:**

  Program divided into logical segments → segment table: base address + limit.

  Logical address = <segment #, offset>. Suffers external fragmentation.

- **Paging:**

  Physical mem → fixed-size **frames**.

  Logical mem → fixed-size **pages**.

  No external fragmentation.

Effective Access Time (EAT)

Formula:

EAT = (hit ratio * (TLB access + mem access)) + (miss ratio * (TLB access + page table access + mem access))

Impact:

High TLB hit ratio → minimizes extra overhead.

Without TLB, paging doubles access time.

Memory Address Translation

Purpose: Maps logical → physical addresses. Enables multiple processes in mem concurrently & ensures mem protection.

Logical vs Physical Addresses

- **Logical Address (Virtual Address)** — generated by CPU; part of process's *logical address space*.
- **Physical Address** — actual location in main mem; part of *physical address space*.

Address Binding

- **Compile Time** — If mem location known, generate absolute code. If moved → recompile.
- **Load Time** — If location unknown, generate relocatable code. Binding happens at load time.
- **Execution Time** — Binding done during execution; allows moving process in mem. Requires hardware support.

Protection & Relocation (Base + Limit Registers)

- CPU checks each address against **base** and **limit** registers → prevents illegal access.
- Registers only modifiable by OS; MMU adds base to logical address → forms physical address.

`Although segments are a non-contiguous memory management approach, external fragmentation is still possible.`

True. External fragmentation is still possible in segmentation because free memory is divided into variable-sized segments, which can leave small unusable gaps between allocated segments.

`The translation lookaside buffer (TLB) is stored in main memory.`

False The TLB is a hardware cache stored inside the CPU, not in main memory.

`The # entries in an inverted page table is equal to the number of frames in logical memory.`

False The number of entries in an inverted page table equals the number of **physical frames**, not logical memory size.

`The amount of virtual memory used by a process can exceed physical memory.`

True Virtual memory allows a process to use more memory than is physically available by swapping pages in and out of disk.

`The working set model is an approach used to help manage the number of frames a process needs in order to execute smoothly and avoid thrashing.`

True The working set model tracks the set of pages a process actively uses to ensure enough frames are allocated to prevent thrashing.

Paging

Purpose: Allows non-contiguous physical address space

→ solves external fragmentation.

Key Idea: Logical page can reside in any physical frame.

Logical Address Space → Pages

Physical Mem → Frames

Pages loaded into any available frame.

Steps of MMU:

1. Extract page number (p) from logical address.
2. Index into page table → frame number (f).
3. MMU forms physical address → frame f + page offset (d).

Page Table:

- Maps logical page numbers → physical frame numbers.
- Stored in main mem → **PTBR** points to it.
- Each entry: frame number + protection bits + valid-invalid bit.

Valid-Invalid Bit:

- **Valid:** page in mem.
- **Invalid:** page not in mem → triggers page fault.

Paging Hardware & TLB

- Without TLB → 2 mem accesses per reference (page table + data).
- **TLB:** Cache of recent page # → frame # translations.

TLB hit: quick access.

TLB miss: must access page table → load translation into TLB.

ASID: lets TLB entries coexist for multiple processes.

Virtual Memory

Definition: Execution of processes not fully in mem.

Separates logical from physical mem.

Benefits:

- Run larger programs than physical mem.
- More programs concurrently.
- Less I/O (unused pages not loaded).
- Shared libs & files → page sharing.
- Efficient process creation (copy-on-write).

Mem Initialization:

Zero-out free frame → avoid security issues.

Page Fault Handling

Detection: MMU sees invalid bit → triggers trap.

Handling Steps:

1. Save registers/process state.
2. Validate page ref → locate page on disk.
3. Find free frame → schedule I/O.
4. Load page → update page table → set valid bit.
5. Resume/restart instruction.

No Free Frame: page replacement required.

If victim is dirty → write back first → adds delay.

Performance Impact:

Page faults → millisecond delays (vs nanosecond mem access).

High fault rate → poor performance.

Fragmentation

Fragmentation → Wasted space due to imperfect fit btwn allocated space and requested space.

Two types:

- **External Fragmentation**
  - Free space exists but not contiguous → can't satisfy request.
  - Occurs in *contiguous allocation*, *segmentation*, *contiguous file allocation*.
  - Example: lots of small holes scattered in memory.
- **Internal Fragmentation**
  - Allocated block > requested size → unused space *inside* allocated block.
  - Happens with *paging* (last frame), *fixed-size allocations*, *clustered file allocation*, *buddy system*.

Where it appears:

- **Main Memory**
  - *Contiguous allocation* → external fragmentation (small scattered holes).
  - *Paging* → internal fragmentation (last frame of process).
  - *Buddy system* → internal fragmentation from rounding to power of 2.
- **File Systems**
  - *Contiguous allocation* → external fragmentation.
  - *Linked allocation, Indexed allocation* → no external fragmentation, but internal if using large clusters.
  - *Extents* reduce external frag but large extents can still waste space.

Solutions:

- External frag → *compaction* (expensive, requires dynamic relocation).
- Internal frag → tune *page/block/cluster size* to balance overhead vs. fragmentation.

File System Allocation Methods

Contiguous Allocation:

Each file → set of contiguous blocks.

Pros: fast sequential access, easy direct access.

Cons: external fragmentation; hard to grow file.

Linked Allocation:

File → linked list of blocks.

Pros: no external fragmentation.

Cons: poor random access; pointer overhead.

Indexed Allocation:

File → index block with pointers to data blocks.

Pros: efficient direct access, no external fragmentation.

Cons: index block overhead.

Large Files:

- **Linked Index Blocks**
- **Multilevel Index**
- **Combined Scheme (UNIX Inode):** direct, single, double, triple indirect pointers.

Thrashing

Cause: Too many processes → excessive paging.

Cycle:

- New process faults → steals frames → more faults → CPU idle → OS adds more → worse.

Detection: High page fault rate + low CPU utilization.

Solutions:

- Decrease degree of multiprogramming.
- Local replacement → isolate offending process.
- Use **Working Set Model** or **Page Fault Frequency (PFF)** to manage frame allocation.

Working Set

Working Set Window (Δ): defines current working set.

WSS (Working Set Size): number of pages referenced in Δ.

If Σ WSSi > total frames: thrashing occurs → reduce active processes.

Tracking: difficult → approximated w/ ref bits + timer interrupts.

Behavior: process entering new locality → temporary page fault spike → settles when working set loaded.

`Using a hierarchical page table helps to solve what problems?`

- **X Thrashing:** Not directly solved by page table structure; thrashing is caused by insufficient memory frames.
- **Large page table sizes: Correct** — hierarchical page tables break large tables into manageable parts.
- **X Belady's anomaly:** Caused by certain page replacement algorithms, unrelated to page table structure.
- **X Memory sharing:** Achieved by mapping multiple processes to same frames, not by hierarchy.
- **X External fragmentation:** Paging already eliminates external fragmentation.

`Which steps are NOT involved in servicing a page fault?`

- **Flush the TLB: Incorrect** (This IS a step; old translation must be flushed after loading the page.)
- **Find a free frame: Incorrect** (IS a step; must load page into a frame.)
- **Set dirty bit to TRUE: Correct** — dirty bit is set only when writing happens later, not on page fault.
- **Terminate process: Correct** — page fault is normally handled by loading the page, not by killing process.
- **Update page table entry: Incorrect** (IS a step; must mark page as present.)
- **Write referenced page to backing store: Correct** — backing store is written only if evicting a dirty page, not as part of initial fault servicing.

`Which of the following could be beneficial to improving the efficiency of a virtual memory system?`

- **Smaller page sizes: Incorrect** Not necessarily; too small pages increase page table size and overhead.
- **Larger TLB: Correct** — larger TLB improves hit ratio, reduces memory access overhead.
- **Larger backing store: Incorrect** Larger backing store capacity does not improve *speed*; faster is what matters.
- **Faster backing store: Correct** — reduces page fault service time.
- **Page prefetching: Correct** — prefetching reduces page fault frequency.
- **Fewer active processes: Correct** — fewer processes reduce memory pressure and page faults.

Page Table Structure

For large address spaces, page tables can become very large. Fixes:

- **Hierarchical Paging (Multi Level)**
  - Dividing page table into smaller tables, often by paging page table itself, addresses are translated by traversing multiple levels of page tables
- **Hashed Page Tables**
  - Hash function maps page #s to hash table entries, which point to page table entries; useful for sparse address spaces
- **Inverted Page Tables**
  - Having one entry per physical frame, mapping virtual addys and PID stores in frame. Decreases mem for page tables but increases lookup time (often uses hash table → faster search)

Free Space Management

OS needs to manage list of free blocks (or clusters) on storage device

- **Bit Vector (Bitmap)**
  - Each block represented by bit (1 free, 0 allocated), efficient for finding first free block or contiguous free blocks, requires extra space for bitmap
- **Linked List (Free List)**
  - All free blocks linked together with pointers, first free block points to next, so on, no wasted space for a map, not efficient for finding contiguous space
- **Grouping**
  - Modification of linked list where first free block stores addys of next n free blocks, with last pointer pointing to next block containing addresses, improves speed of finding free blocks
- **Counting**
  - Takes advantage of contiguous free blocks by storing addys of first free block and count of contiguous blocks that follow it. Each entry is an address/count pair, more efficient than a simple linked list if counts > 1, similar extent to allocation
- **Space Maps**
  - Divides space into chunks and uses space maps per chunk. Uses counting format and log structured techniques for updates. Designed for very large file systems

Page Replacement

Goal: Select victim page → minimize future page faults.

Belady's Anomaly: More frames can → more faults.

Algorithms:

- **Optimal (OPT/MIN):** Replace page not needed for longest time (theoretical).
- **FIFO:** Oldest page replaced → can evict frequently used pages.
- **LRU Approximation:**
  - **Second Chance (Clock):** Pages w/ reference bit set → given second chance.
  - **Enhanced Second Chance:** Uses both ref bit + modify bit → 4 classes.