Memory Management - Short Notes

---

1. Basic Bare Machine

Definition

- **Bare Machine**: Computer system with no operating system
- User programs have direct access to all hardware resources
- Single program executes at a time

Characteristics

- **No protection**: Programs can access any memory location
- **No multiprogramming**: Only one job at a time
- **Manual loading**: Operator loads programs manually
- **Inefficient**: CPU idle during I/O operations
- **Simple addressing**: Physical addresses only

Problems

- Poor CPU utilization
- No memory protection
- Manual intervention required
- Resource wastage

Exam Questions:

1. What are the main limitations of a bare machine architecture?
2. Why is CPU utilization poor in bare machine systems?
3. Explain why memory protection is impossible in bare machines.

---

2. Resident Monitor

Definition

- **Resident Monitor**: Primitive OS that remains in memory permanently
- First step toward modern operating systems

- Provides basic job scheduling and I/O control

Structure

- **Monitor resides in low memory** (typically 0 to k)
- **User program area** (k to max)
- Monitor always resident, never swapped out

Functions

- **Job sequencing**: Automatically loads next job
- **I/O control**: Handles device management
- **Memory protection**: Protects monitor area from user programs
- **Control transfer**: Returns control after job completion

Advantages

- Reduced setup time between jobs
- Better CPU utilization than bare machine
- Automatic job sequencing
- Basic protection for monitor

Limitations

- Still single job execution
- Memory underutilization (one job at a time)
- CPU idle during I/O of current job

Exam Questions:

1. How does a resident monitor improve upon bare machine architecture?
2. Explain the memory layout in a resident monitor system.
3. What are the key limitations of resident monitors?

---

3. Multiprogramming with Fixed Partitions

Concept

- Memory divided into **fixed-size partitions** at system startup
- Multiple programs loaded in different partitions simultaneously

- Each partition holds one process

Partition Schemes

Equal-Size Partitions:

- All partitions same size
- Simple to implement
- Internal fragmentation problem
- Small jobs waste space

Unequal-Size Partitions:

- Partitions of different sizes
- Better memory utilization
- Reduces internal fragmentation
- More flexible

Allocation Strategies

Fixed Assignment:

- Job waits for specific partition size
- Simple queue per partition
- May cause underutilization

Dynamic Assignment:

- Job allocated to smallest sufficient partition
- Better utilization
- Single queue for all jobs

Problems

- **Internal Fragmentation**: Wasted space within partition
- **Degree of multiprogramming limited** by number of partitions
- **Fixed partition sizes** can't adapt to job mix
- Large jobs may not fit in any partition

Formula

- **Internal Fragmentation** = Partition Size - Process Size

Exam Questions:

1. Differentiate between equal and unequal size partitions with examples.
2. What is internal fragmentation? How does it occur in fixed partitioning?
3. Compare fixed and dynamic assignment strategies in fixed partitioning.
4. If a 100KB job is placed in a 150KB partition, calculate internal fragmentation.

---

4. Multiprogramming with Variable Partitions

Concept

- Partitions created **dynamically** based on process requirements
- No predefined partition sizes
- Partitions vary in size during execution

Allocation Strategies

First Fit:

- Allocate **first hole large enough**
- Fast (stop searching when found)
- May create small holes at beginning

Best Fit:

- Allocate **smallest hole that fits**
- Minimizes wasted space per allocation
- Slow (must search entire list)
- Creates tiny unusable holes

Worst Fit:

- Allocate **largest available hole**
- Leaves large leftover holes
- Poor performance in practice

Next Fit:

- Like First Fit but continues from **last allocation point**
- Distributes holes more evenly

- Slightly worse performance than First Fit

Problems

- **External Fragmentation**: Free memory scattered in small holes
- Total free memory sufficient but not contiguous
- Requires periodic compaction

Solutions

- **Compaction**: Shuffle processes to create one large hole
  - Time consuming
  - Requires relocatable programs
  - System must pause

Exam Questions:

1. Explain the difference between First Fit, Best Fit, and Worst Fit with examples.
2. What is external fragmentation? How is it different from internal fragmentation?
3. Describe the compaction process and its drawbacks.
4. Given memory holes of 100KB, 500KB, 200KB, 300KB and a request for 200KB, show allocation for each strategy.

---

5. Protection Schemes

Need for Protection

- Prevent processes from accessing unauthorized memory
- Protect OS from user processes
- Isolate processes from each other

Hardware Support Mechanisms

Base and Limit Registers:

- **Base Register**: Starting address of partition
- **Limit Register**: Size of partition
- Check: Base $\leqslant$ Address $<$ Base + Limit
- Trap if violation occurs

Relocation Register:

- Added to every address generated by process
- Enables **dynamic relocation**
- Address translation: Physical = Logical + Relocation

Bounds Register:

- Specifies maximum logical address
- Used with relocation register
- Provides protection during address translation

Address Translation

```
Physical Address = Logical Address + Base/Relocation Register
Valid if: Logical Address < Limit/Bounds Register
```

Protection Mechanisms

- **Privileged instructions**: Only OS can execute
- **Memory protection keys**: Protect memory blocks
- **Dual mode operation**: User mode vs Kernel mode
- **Access control bits**: Read/Write/Execute permissions

Exam Questions:

1. Explain base and limit register scheme for memory protection.
2. How does relocation register enable dynamic relocation?
3. What happens when a process tries to access memory outside its bounds?
4. Calculate physical address if logical address = 1000, base = 50000, limit = 10000.

---

6. Paging

Concept

- **Logical memory**: Divided into fixed-size **pages**
- **Physical memory**: Divided into fixed-size **frames**
- Page size = Frame size (typically 4KB, 8KB)
- Eliminates external fragmentation

Components

Page Table:

- Maps page number to frame number
- One entry per page
- Stored in main memory
- Each process has its own page table

Address Structure:

```
Logical Address = <Page Number, Page Offset>
Physical Address = <Frame Number, Page Offset>
```

Address Translation

1. Extract page number (p) and offset (d) from logical address
2. Look up page table: Frame number (f) = PageTable[p]
3. Physical address = f × frame_size + d

Formulas

- **Number of pages** = ⌈Process Size / Page Size⌉
- **Page Number** = Logical Address / Page Size
- **Page Offset** = Logical Address % Page Size
- **Physical Address** = Frame Number × Frame Size + Offset

Advantages

- No external fragmentation
- Simple allocation (any free frame)
- Easy to implement swapping
- Shared pages possible

Disadvantages

- **Internal fragmentation** (last page may not be full)
- **Page table overhead** (space for page table)
- **Memory access time** (double access: table + data)

Page Table Entry (PTE)

- **Frame number**: Physical frame location
- **Valid/Invalid bit**: Page in memory or not
- **Protection bits**: R/W/X permissions
- **Reference bit**: Used for page replacement
- **Modify/Dirty bit**: Page modified or not

Exam Questions:

1. Explain paging with a neat diagram showing address translation.
2. If logical address = 2050, page size = 1024 bytes, calculate page number and offset.
3. What is the purpose of valid/invalid bit in page table entry?
4. A process of 72KB with page size 4KB requires how many pages? Calculate internal fragmentation.
5. Compare paging with variable partition schemes.

---

7. Segmentation

Concept

- Memory divided into **logical segments** of varying sizes
- Each segment represents logical unit: code, data, stack, heap
- User view of memory (logical division)

Segment Table

- **Base**: Starting physical address of segment
- **Limit**: Length of segment
- Each process has segment table

Address Structure

```
Logical Address = <Segment Number, Offset>
```

Address Translation

1. Extract segment number (s) and offset (d)
2. Check: d < Limit[s] (protection check)
3. Physical address = Base[s] + d

4. Trap if d ⩾ Limit[s]

Advantages

- **Logical organization**: Matches user view
- **Protection**: Different segments different permissions
- **Sharing**: Easy to share segments (code sharing)
- **Dynamic growth**: Segments can grow
- **No internal fragmentation**

Disadvantages

- **External fragmentation**: Segments of varying sizes
- **Allocation complexity**: Need variable partition allocation
- **Segment table overhead**

Comparison: Paging vs Segmentation

Feature	Paging	Segmentation
Division	Fixed size	Variable size
User view	Transparent	Visible (logical)
Fragmentation	Internal	External
Protection	Page level	Segment level
Sharing	Difficult	Easy

Exam Questions:

1. Explain segmentation with address translation example.
2. How does segmentation provide better protection than paging?
3. Compare and contrast paging and segmentation.
4. If segment table has base=4000, limit=600 and offset=550, calculate physical address.
5. Why is external fragmentation a problem in segmentation?

---

8. Paged Segmentation

Concept

- **Hybrid approach**: Combines benefits of both paging and segmentation
- Segments divided into pages
- User view: Segmentation; Physical implementation: Paging

Structure

- Each segment has a **page table**
- **Segment table** contains base address of page table for each segment
- Eliminates external fragmentation of pure segmentation

Address Structure

```
Logical Address = <Segment Number, Page Number, Offset>
```

Address Translation

1. Use segment number to index segment table → get page table base
2. Use page number to index page table → get frame number
3. Combine frame number with offset → physical address
4. Check segment limit before page table access

Advantages

- **No external fragmentation** (pages are fixed size)
- **Logical organization** (user sees segments)
- **Protection and sharing** at segment level
- **Efficient memory use**

Disadvantages

- **Complex implementation**
- **Multiple memory accesses** for address translation
- **Large table overhead** (segment table + multiple page tables)

Example Systems

- **Intel x86 architecture** (before pure paging in 64-bit)
- Combines benefits while managing complexity

Exam Questions:

1. Explain how paged segmentation combines advantages of both schemes.
2. Draw the address translation mechanism in paged segmentation.
3. Why does paged segmentation eliminate external fragmentation?
4. Compare memory overhead in paging, segmentation, and paged segmentation.

---

9. Virtual Memory Concepts

Definition

- Technique allowing execution of processes **not entirely in memory**
- Logical address space can be much larger than physical memory
- Only needed parts loaded in memory

Key Ideas

- **Separation** of logical and physical memory
- **Partial loading**: Load only required pages/segments
- **On-demand loading**: Bring pages when needed
- **Swapping**: Move pages between memory and disk

Advantages

- **More processes in memory** (higher multiprogramming)
- **Large programs** can execute (size > physical memory)
- **Better CPU utilization**
- **Faster process creation** (less to load initially)
- **User programs** not limited by physical memory

Requirements

- **Hardware support**: MMU, page tables
- **Demand paging/segmentation**: Load on reference
- **Page replacement**: Remove pages when memory full
- **Backing store**: Fast disk space (swap space)

Address Translation

- **MMU (Memory Management Unit)**: Hardware device
- Translates logical to physical addresses

- Generates page fault if page not in memory

Components

- **Valid/Invalid bit**: Indicates if page in memory
- **Page fault handler**: OS routine to handle missing pages
- **Swap space**: Disk area for swapped pages
- **Page replacement algorithm**: Decides which page to remove

Exam Questions:

1. Define virtual memory and explain its advantages.
2. How does virtual memory allow programs larger than physical memory to execute?
3. What hardware support is required for virtual memory implementation?
4. Explain the role of MMU in virtual memory systems.

---

10. Demand Paging

Concept

- **Lazy loading**: Pages loaded only when demanded (referenced)
- Initially, no pages in memory (or minimal set)
- Page brought in when **page fault** occurs

Working

1. Process starts with no pages in memory
2. CPU tries to access page → page fault
3. OS loads page from disk to memory
4. Update page table
5. Restart instruction

Page Fault Handling

1. **Trap to OS** (page fault interrupt)
2. **Save user registers** and process state
3. **Check validity**: Legal reference or illegal?
4. **Find free frame** (or select victim page)
5. **Read page from disk** to free frame

6. **Update page table** (valid bit = 1)
7. **Restart instruction** that caused fault

Pure Demand Paging

- Start process with **no pages** in memory
- Every page loaded only when accessed
- High initial page fault rate
- Never bring page before it's needed

Components Required

- **Page table with valid/invalid bit**
- **Secondary storage** (swap space/backing store)
- **Instruction restart capability**
- **Free frame list**

Advantages

- Less I/O needed initially
- Less memory needed
- Faster program start
- More processes in memory

Performance Factors

- **Page fault rate** (p): $0 \leqslant p \leqslant 1$
- **Effective Access Time (EAT)**
- **Memory access time** (ma)
- **Page fault service time** (pf_time)

Exam Questions:

1. Explain demand paging with the page fault handling sequence.
2. What is pure demand paging? What are its drawbacks?
3. Describe the steps involved when a page fault occurs.
4. Why is instruction restart capability important in demand paging?

Effective Access Time (EAT)

Formula:

```
EAT = (1 - p) × ma + p × pf_time
```

Where:

- **p** = page fault rate ($0 \leqslant p \leqslant 1$)
- **ma** = memory access time (e.g., 200 ns)
- **pf_time** = page fault service time (e.g., 8 ms)

Page Fault Service Time Components

pf_time = Service interrupt + Read page + Restart overhead

Detailed breakdown:

1. **Service page fault interrupt**: ~1-100 μs
2. **Read page from disk**:
   - Seek time: ~5 ms
   - Latency: ~3 ms
   - Transfer time: ~0.05 ms
   - Total: ~8 ms (dominant factor)
3. **Update page table**: ~1-100 μs
4. **Restart process**: ~1-100 μs

Typical values:

- Page fault service time: 8 ms = 8,000,000 ns
- Memory access time: 200 ns

Example Calculation

Given:

- Memory access time = 200 ns
- Page fault service time = 8 ms = 8,000,000 ns
- Page fault rate = p

EAT = (1 - p) × 200 + p × 8,000,000 EAT = 200 + p × 7,999,800 EAT ≈ 200 + 8,000,000p ns

For acceptable performance (say, max 10% slowdown):

- 220 = 200 + 8,000,000p
- p = 20/8,000,000 = 0.0000025
- **p < 0.00025%** (less than 1 fault per 400,000 accesses)

Key Insights

- Page faults are **extremely expensive** (40,000× slower)
- Even small page fault rates significantly degrade performance
- Goal: Keep page fault rate very low (< 0.01%)

Performance Improvement Techniques

- **Prepaging**: Load multiple pages anticipating future use
- **Page size optimization**: Larger pages reduce faults but increase internal fragmentation
- **Working set management**: Keep frequently used pages
- **Efficient page replacement**: Better algorithms reduce faults

Exam Questions:

1. Derive the formula for Effective Access Time in demand paging.
2. If ma = 100ns, page fault time = 10ms, and p = 0.001, calculate EAT.
3. Why are page faults so expensive compared to normal memory access?
4. What page fault rate is acceptable for less than 10% performance degradation? Calculate with typical values.
5. List the components of page fault service time and identify the dominant component.

---

12. Page Replacement Algorithms

Need for Page Replacement

- All frames occupied, page fault occurs
- Must **replace/evict** a page to make room
- Goal: Minimize page fault rate
- Reference string: Sequence of page references

1. FIFO (First In First Out)

Strategy: Replace oldest page (first loaded)

Implementation: Queue (FIFO order)
Advantages:

- Simple to implement
- Fair (equal treatment)

Disadvantages:

- **Belady's Anomaly**: More frames may increase page faults
- Ignores usage patterns
- Poor performance

Example:

```
Reference: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Frames: 3
Page Faults: 9
```

2. Optimal (OPT)

Strategy: Replace page **not needed for longest time** in future

Implementation: Look ahead in reference string (theoretical)

Advantages:

- **Lowest possible page fault rate**
- Benchmark for other algorithms
- No Belady's Anomaly

Disadvantages:

- **Impossible to implement** (needs future knowledge)
- Used only for comparison

3. Least Recently Used (LRU)

Strategy: Replace page **not used for longest time** in past

Implementation:

- Counter: Timestamp each page on reference
- Stack: Keep stack of page numbers
- Hardware support needed for efficiency

Advantages:

- Good approximation of OPT
- Considers usage history
- No Belady's Anomaly

Disadvantages:

- **Expensive** to implement
- Requires hardware support
- Overhead of maintaining timestamps/stack

Example:

```
Reference: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Frames: 3
Page Faults: 10 (better than FIFO)
```

4. LRU Approximation - Second Chance

Strategy: FIFO with reference bit

Algorithm:

1. Check reference bit of oldest page
2. If 0: Replace it
3. If 1: Set to 0, give second chance, move to end
4. Continue until page with ref bit = 0 found

Advantages:

- Cheaper than LRU
- Better than pure FIFO
- Reasonable performance

5. LRU Approximation - Clock (Circular)

Strategy: Second chance with circular queue

Implementation: Pointer rotates (clock hand)

Advantages:

- No shifting of pages needed

- Efficient implementation
- Widely used in practice

Comparison Table

Algorithm	Complexity	Performance	Belady's Anomaly
FIFO	Low	Poor	Yes
Optimal	N/A	Best	No
LRU	High	Good	No
Clock	Medium	Good	No

Belady's Anomaly

- More frames → More page faults (counterintuitive)
- Occurs in FIFO
- Does NOT occur in stack algorithms (LRU, OPT)

Exam Questions:

1. Compare FIFO, LRU, and Optimal page replacement algorithms with examples.
2. What is Belady's Anomaly? Which algorithms suffer from it?
3. Given reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2 and 4 frames, calculate page faults for FIFO and LRU.
4. Why is Optimal algorithm not implementable in practice?
5. Explain the Clock (Second Chance) algorithm with a diagram.

---

13. Thrashing

Definition

- **Thrashing**: Process spends more time **paging** than executing
- Constant page faults, minimal useful work
- CPU utilization drops drastically

Causes

- **Over-multiprogramming**: Too many processes in memory

- **Insufficient frames** per process
- Each process keeps faulting
- Process's **working set** not in memory

Working Set Model

Working Set (WS):

- Set of pages referenced in most recent Δ page references
- Δ = working set window (e.g., 10,000 references)
- **WS(t, Δ)** = pages referenced in (t - Δ, t)

Working Set Size:

- Number of distinct pages in working set
- Varies over time as locality changes

Rule:

- Σ WSS_i ⩽ Total frames available
- If WSS_i > allocated frames → thrashing

Locality of Reference

- Program tends to reference **small set of pages** at any time
- **Temporal locality**: Recent pages likely referenced again
- **Spatial locality**: Nearby addresses likely referenced
- Working set captures locality

Detection

- **Low CPU utilization** despite high multiprogramming
- **High page fault rate**
- **High disk I/O** for paging
- **Low throughput**

Prevention

Reduce multiprogramming degree:

- Suspend some processes
- Swap out processes

Increase frames per process:

- Allocate more frames
- Better matches working set

Working Set Strategy:

- Keep working set in memory
- Don't run if insufficient frames

Page Fault Frequency (PFF):

- Monitor page fault rate per process
- If too high: allocate more frames
- If too low: deallocate frames

Local vs Global Replacement:

- **Local**: Process replaces only its own pages (prevents thrashing spread)
- **Global**: Any process can replace any page (better utilization but thrashing spreads)

Recovery

- **Decrease multiprogramming** level
- **Swap out processes** temporarily
- **Increase memory** (if possible)
- **Improve page replacement** algorithm

Exam Questions:

1. Define thrashing and explain its causes.
2. What is the working set model? How does it prevent thrashing?
3. Explain how over-multiprogramming leads to thrashing with a diagram.
4. Compare local and global page replacement in context of thrashing.
5. What are the indicators that a system is thrashing? How to recover?

---

14. Cache Memory Organization

Concept

- **Cache**: Small, fast memory between CPU and main memory

- Stores copies of frequently accessed data
- Exploits **locality of reference**
- Transparent to programmer

Memory Hierarchy

```
Registers (fastest, smallest, most expensive)
    ↓
L1 Cache (on-chip)
    ↓
L2 Cache (on-chip or off-chip)
    ↓
L3 Cache (shared)
    ↓
Main Memory (RAM)
    ↓
Secondary Storage (slowest, largest, cheapest)
```

Cache Performance

Hit Ratio (h):

- Percentage of memory references found in cache
- h = Cache Hits / Total References

Miss Ratio:

- m = 1 - h
- Percentage of references requiring main memory access

Effective Access Time:

```
EAT = h × Cache_time + (1-h) × (Cache_time + Memory_time)
EAT = h × Tc + (1-h) × (Tc + Tm)
EAT = Tc + (1-h) × Tm
```

Cache Mapping Techniques

1. Direct Mapping:

- Each memory block → specific cache line
- Cache line = Memory block % Number of cache lines
- **Simple**, fast, but high conflict misses

- Formula: Cache Line = (Block Address) mod (Cache Lines)

2. Fully Associative:

- Any memory block → any cache line
- **Most flexible**, lowest conflict misses
- **Expensive** (complex hardware, slow search)

3. Set Associative:

- Compromise: n-way set associative
- Cache divided into sets
- Block → specific set, any line within set
- **Common**: 2-way, 4-way, 8-way
- Formula: Set = (Block Address) mod (Number of Sets)

Replacement Policies (Cache)

- **LRU**: Least Recently Used
- **FIFO**: First In First Out
- **Random**: Random selection
- **LFU**: Least Frequently Used

Write Policies

Write Through:

- Write to **both cache and memory**
- Ensures consistency
- Slower (every write to memory)

Write Back:

- Write only to **cache**
- Memory updated when block replaced
- **Dirty bit** indicates modified block
- Faster but complex

Types of Cache Misses

- **Compulsory**: First access (cold start)
- **Capacity**: Cache too small for working set

- **Conflict**: Multiple blocks map to same location

Exam Questions:

1. Explain cache memory organization and its role in memory hierarchy.
2. Calculate effective memory access time if cache hit ratio = 0.85, cache time = 10ns, memory time = 100ns.
3. Compare direct mapping, fully associative, and set associative cache organizations.
4. Differentiate between write-through and write-back policies.
5. What are the three types of cache misses? Give examples of each.

---

15. Locality of Reference

Definition

- **Locality of Reference**: Tendency of processor to access same set of memory locations repetitively
- Foundation for cache and virtual memory effectiveness
- Programs don't access memory randomly

Types of Locality

1. Temporal Locality:

- If memory location accessed, likely to be accessed again **soon**
- Recent references likely repeated
- Examples:
  - Loop variables
  - Frequently called functions
  - Stack variables in active function

2. Spatial Locality:

- If memory location accessed, **nearby locations** likely accessed soon
- Sequential access patterns
- Examples:
  - Array traversal
  - Sequential instruction execution
  - Structure/object field access

Why Locality Exists

Program Structure:

- **Loops**: Same instructions/data repeatedly
- **Sequential execution**: Instructions in sequence
- **Functions**: Localized code and data
- **Arrays**: Sequential access patterns
- **Data structures**: Related data stored together

Programmer Behavior:

- Loops naturally create temporal locality
- Arrays create spatial locality
- Modular programming creates locality

Working Set

- Set of pages in **current locality**
- Changes as program moves through phases
- **Phase transition**: Shift from one locality to another

Exploitation by Systems

Cache Memory:

- Temporal: Keep recent data in fast cache
- Spatial: Fetch entire cache line (multiple words)

Virtual Memory:

- Working set model based on locality
- Demand paging works because of locality
- Page replacement assumes locality continues

Prefetching:

- Predict future references based on spatial locality
- Load next blocks in advance

Measuring Locality

Reference String Analysis:

- Working set size over time
- Page fault patterns
- Cache hit rates

Locality Phases:

- Programs execute in distinct phases
- Each phase has its own locality
- Transitions cause temporary increase in misses

Impact on Performance

- **Good locality** → High cache hit rate → Fast execution
- **Poor locality** → Thrashing → Slow execution
- Compiler optimizations improve locality

Programming for Locality

Good Practices:

- Access arrays sequentially (row-major in C)
- Keep working set small
- Reuse data while in cache
- Group related data in structures

Bad Practices:

- Random access patterns
- Large working sets
- Column-major access in row-major languages
- Pointer chasing in linked structures

Exam Questions:

1. Define locality of reference and explain its two types with examples.
2. How does locality of reference make cache memory and virtual memory effective?
3. Give programming examples that exhibit good and poor locality.
4. Explain the concept of working set in relation to locality.

5. Why do loops exhibit both temporal and spatial locality?

📌 QUICK SUMMARY - Exam Ready

Key Formulas

Paging:

- Number of Pages = [Process Size / Page Size]
- Page Number = Logical Address / Page Size
- Offset = Logical Address % Page Size
- Physical Address = Frame Number × Frame Size + Offset

Effective Access Time (Virtual Memory):

- EAT = (1 - p) × ma + p × pf_time
- Where p = page fault rate, ma = memory access time

Cache Performance:

- EAT = h × Tc + (1-h) × (Tc + Tm)
- Simplified: EAT = Tc + (1-h) × Tm
- h = hit ratio, Tc = cache time, Tm = memory time

Segmentation:

- Physical Address = Base[segment] + Offset
- Check: Offset < Limit[segment]

Working Set:

- Total WSS ⩽ Total Frames (to avoid thrashing)

Memory Allocation Strategies

Strategy	First Fit	Best Fit	Worst Fit
Search	First adequate	Smallest adequate	Largest
Speed	Fast	Slow	Slow

Strategy	First Fit	Best Fit	Worst Fit
Fragmentation	Moderate	Small holes	Large holes
Usage	Common	Memory constrained	Rarely used

Page Replacement Comparison

Algorithm	Implementation	Performance	Anomaly
FIFO	Queue	Poor	Yes
Optimal	Impossible	Best	No
LRU	Stack/Counter	Good	No
Clock	Circular + Ref bit	Good	No

Fragmentation Quick Guide

Internal Fragmentation:

- Occurs in: Fixed partitions, Paging (last page)
- Wasted space: Within allocated partition/page
- Formula: Partition/Page Size - Actual Need

External Fragmentation:

- Occurs in: Variable partitions, Segmentation
- Wasted space: Between allocated regions
- Solution: Compaction (expensive)

Critical Concepts

Memory Protection:

- Base + Limit registers for partitions
- Valid/Invalid bits for paging
- Segment limit checks for segmentation

Virtual Memory Success Factors:

- Low page fault rate (< 0.01%)
- Good locality of reference
- Appropriate working set size
- Efficient page replacement

Thrashing Indicators:

- CPU utilization ↓
- Page fault rate ↑
- Disk I/O ↑ (paging activity)
- Throughput ↓

Thrashing Prevention:

- Don't over-multiprogramming
- Ensure WSS ⩽ Allocated frames
- Use local replacement
- Monitor PFF (Page Fault Frequency)

Cache Organization

Mapping Types:

- **Direct**: Simple, fast, high conflicts
- **Fully Associative**: Flexible, expensive, slow
- **Set Associative**: Balanced (most common)

Write Policies:

- **Write Through**: Safe, slower
- **Write Back**: Fast, needs dirty bit

Locality Principles

Temporal Locality:

- Recently used → likely used again

- Examples: loops, function calls, variables

Spatial Locality:

- Nearby addresses → likely accessed next
- Examples: arrays, sequential code

Why It Matters:

- Makes caching effective
- Enables virtual memory
- Basis for working set model

Important Terms Checklist

□ **Resident Monitor** - Early OS in memory □ **Multiprogramming** - Multiple jobs in memory □ **Fragmentation** - Internal vs External □ **Paging** - Fixed-size pages/frames □ **Segmentation** - Variable logical units □ **Virtual Memory** - Logical > Physical memory □ **Demand Paging** - Load on reference □ **Page Fault** - Referenced page not in memory □ **Thrashing** - Excessive paging □ **Working Set** - Active pages set □ **Locality** - Temporal + Spatial □ **Cache** - Fast memory buffer □ **Hit Ratio** - Cache success rate

---

Common Exam Calculations

1. Page/Frame Calculations:

```
Process = 72KB, Page Size = 4KB
Pages = [72/4] = 18 pages
Internal Fragmentation = (18×4) - 72 = 0KB
```

2. Address Translation (Paging):

```
Logical = 2050, Page Size = 1024
Page# = 2050/1024 = 2
Offset = 2050 % 1024 = 2
If Frame[2] = 5:
Physical = 5 × 1024 + 2 = 5122
```

3. EAT Calculation:

```
ma = 200ns, pf = 8ms, p = 0.001
EAT = 0.999×200 + 0.001×8,000,000
EAT = 199.8 + 8,000 = 8,199.8 ns
Slowdown = 8200/200 = 41× slower!
```

4. Cache EAT:

```
h = 0.9, Tc = 10ns, Tm = 100ns
EAT = 10 + (1-0.9)×100
EAT = 10 + 10 = 20ns
```

Quick Tips for Exam

✓ **Fixed vs Variable Partitions**: Internal vs External fragmentation ✓ **Paging always better** than fixed/variable for fragmentation ✓ **Page faults are expensive**: 1 fault ≈ 40,000 normal accesses ✓ **Thrashing**: Too many processes, too few frames ✓ **Working Set**: Must fit in allocated frames ✓ **LRU better than FIFO**, but Optimal is best (theoretical) ✓ **Locality**: Foundation of caching and virtual memory ✓ **Cache hit ratio**: Higher is better (aim for > 0.8)

Good Luck! 🎯