

Practice Final Questions (MCQ + Other)

What is thrashing (from both a system-wide and per process perspective)?

What should you do when the system is thrashing? What should be done when a process is thrashing?

Thrashing occurs when excessive time is spent swapping pages between memory and disk, leaving little time for useful computation. System-wide thrashing happens when the total working sets of all processes exceed physical memory; per-process thrashing occurs when a single process lacks enough frames for its working set and generates constant page faults. To address system thrashing, reduce multiprogramming, apply the working set model, and tune page replacement. For a thrashing process, allocate more frames or suspend it.

One big advantage of paging is that it eliminates external fragmentation. However, internal fragmentation can occur, especially as the page size increases. This would argue for making page sizes smaller. Do you see anything wrong with this argument? Typical page sizes are between 512 bytes and 4K bytes. Why not make page sizes 128 bytes or even less? Name one advantage, if any, of having page sizes 4K?

Making page sizes very small (e.g. 128 bytes) greatly increases page table size and management overhead, since more pages are needed to cover the same memory; this leads to more page table accesses and TLB misses. Additionally, disk I/O becomes less efficient when transferring many small pages. A 4K page size offers a good balance: it keeps page tables manageable and allows efficient disk I/O, while keeping internal fragmentation at an acceptable level.

What does "virtual memory" mean (try to give a concise definition)? Describe how it works? If there is only 1 process executing, do we really need virtual memory?

Virtual memory is an abstraction that allows processes to use more memory than physically available by giving them a large, uniform address space. The CPU generates virtual addresses, which the MMU translates into physical addresses via a page table; pages not in memory are paged in from disk as needed. If only one process is running and fits in physical memory, virtual memory isn't strictly necessary for protection or isolation, but it still simplifies programming, enables demand paging, and supports large address spaces.

Explain the difference between local and global page replacement. In a virtual memory system? Why might using only 1 strategy be sub-optimal?

Local page replacement allows a process to replace only its own pages; its frame allocation is fixed and cannot be borrowed by other processes. Global page replacement lets a process replace any page in memory, even from other processes, with all frames shared system-wide. Using only local replacement can cause thrashing if a process needs more frames but can't obtain them, while using only global replacement risks instability and unfairness, as one process may evict pages from others.

If an application could alert the operating system that it will access a particular file's data in a sequential manner, how might the OS exploit this information to improve performance?

If the OS knows a file will be accessed sequentially, it can improve performance by prefetching file blocks ahead of time and adjusting cache policies (e.g. evicting older blocks sooner), which reduces I/O latency and improves throughput.

Could you implement the equivalent of a multi-level directory structure with a single-level directory structure in which arbitrarily long names can be used? Explain how you can do so. What problems, if any, do you see with such an approach?

You could simulate multi-level directories in a single-level system by encoding directory paths into long filenames (e.g. dir1_dir2_filename), but this is non-standard and leads to long, unwieldy names; it also lacks true hierarchy, making operations like listing or navigating directories inefficient and error-prone.

What is the purpose of the virtual file system layer in the OS? How does it interact with the physical file system layer?

How it works - Applications make system calls . These calls go through the VFS layer, which defines a standard set of operations. The VFS then calls the appropriate functions in the driver for the specific physical file system, which handles the actual disk operations

Purpose - Provides abstraction and portability. Allows the OS to support multiple file systems simultaneously. Simplifies application development - apps don't need to care what file system is being used.

Discuss the differences and merits of linked file allocation versus using a file allocation table (FAT).

Linked file allocation stores each file as a linked list of disk blocks, with each block pointing to the next; it avoids external fragmentation and makes growing files easy, but offers poor random access and adds pointer overhead. FAT stores these block links in an in-memory table, enabling faster access and easier random access, but the table can consume significant memory on large disks and requires maintaining consistency between memory and disk.

Def - Page - A fixed size block of memory into which a process is divided for management by the OS. Both physical and logical memory are divided into pages of the same size

Def - Paging - Mem management scheme that allows a process's physical address space to be non-contiguous, avoiding external fragmentation

Def - Degree of multiprogramming - Number of processes loaded into memory at the same time

Def - Image - Process image, the complete state of a process stored in memory. Code, data, stack, heap, registers, PC, etc. Everything needed to re - make that process.

Def - Context Switch - Saving state of running process and loading state of another, allowing CPU to switch from one to another.

Def - Page Fault - Occurs when a program tries to access a page of memory that is not current in main memory. It must fetch a missing memory page from secondary storage.

Def - Page Replacement Algorithm - Method used by OS to decide which memory page to remove when a page fault occurs and no free frames are available. Minimizes future page faults.

Def - Thrashing - When a process spends more time paging than executing, leads to severe performance degradation

Def - Holes - Gaps of free memory between allocated blocks caused by processes being loaded and removed; leads to external fragmentation in contiguous memory allocation systems.

Def - Base Register - Holds the starting physical address of a process's allocated memory block. Used during address translation in contiguous memory allocation to map a process's logical address to a physical address.

Def: backing store - Secondary storage device used to temporarily hold processes that have been swapped out of main memory. Typically fast disk or NVM device.

1.4.1 Running on Air!

Summary:

AIR-0 prototype is slow. Two possible improvements:

- Add a TLB with access time = $(1/4) * M$
- Use faster memory $M' = 0.8 * M$

Given:

- No page faults
- Page table fits in memory
- TLB hit ratio $h = 80\%$

Goal: Compare Effective Access Time (EAT) for both options.

(a) Write EAT expressions for both cases

Case 1 (with TLB):

- TLB hit (80%):
 $\rightarrow (\frac{1}{4})M + M = (\frac{5}{4})M$
- TLB miss (20%):
 $\rightarrow (\frac{1}{4})M + M + M = (\frac{9}{4})M$

Overall EAT:

$$EAT_{TLB} = h * (5/4)M + (1 - h) * (9/4)M$$

$$EAT_{TLB} = 0.8 * (5/4)M + 0.2 * (9/4)M$$

$$EAT_{TLB} = (1.0M) + (0.45M) = 1.45M$$

Case 2 (Faster Memory $M' = 0.8M$):

- NO TLB \rightarrow always access page table first, then memory

$$EAT_{FasterMem} = M' + M' = 0.8M + 0.8M = 1.6M$$

(b) Suppose $M = 100ns$, now calculate EAT

Also change TLB hit ratio to $h = 60\%$.

Case 1 ($h = 0.6$):

- TLB hit $\rightarrow 25ns + 100ns = 125ns$
- TLB miss $\rightarrow 25ns + 100ns + 100ns = 225ns$

Overall EAT:

$$EAT_{TLB} = 0.6 * 125ns + 0.4 * 225ns$$

$$EAT_{TLB} = 75ns + 90ns = 165ns$$

Case 2 (Faster mem):

$$M' = 0.8 * 100ns = 80ns$$

$$EAT_{FasterMem} = 80ns + 80ns = 160ns$$

Conclusion: Now faster memory is better ($160ns < 165ns$).

(c) When is each case better?

If Case 1 (TLB, $h = 80\%$) is better, how fast must M' be to beat it?

Want:

$$2 * M' < 1.45M \rightarrow M' < 0.725M \rightarrow \text{Memory must be } 27.5\% \text{ faster than } M (M' < 72.5ns \text{ if } M = 100ns).$$

If Case 2 is better (e.g. $h = 60\%$), how much must M' improve for TLB to be better?

Want:

$$125h + 225(1 - h) < 160 \rightarrow \text{solve:}$$

$$100h + 225 < 160 \rightarrow 100h < -65 \rightarrow h > 65\%$$

\rightarrow Hit ratio must be $> 65\%$ for TLB to be better than fast memory at $M' = 80ns$.

Final Summary

Case EAT (80% hit) EAT (60% hit)

$$TLB \quad 1.45M / 145ns = 165 ns$$

$$\text{Faster Mem} \quad 1.6M / 160ns = 160 ns$$

\rightarrow TLB better if $h > 60\%$, but faster memory wins if h drops to 60%.

Def - Inode - Data structure used to store metadata about a file, such as its size, permissions, timestamps, and pointers to data blocks

Def - Extents - A contiguous block of storage allocated to a file; multiple extents allow flexible file growth while reducing fragmentation

Def - 50% percent rule - In first fit mem allocation, about half as many blocks as allocated are lost to external fragmentation. So if n blocks are allocated, around $n/2$ blocks become unusable

Def - Bitmap - A data structure where each bit represents the status (free or allocated) of a block of memory or storage. Used to efficiently track free space and locate available blocks

Def - Base Register (Relocation Register) - holds the smallest legal physical memory address for a process

Def - Limit Register - Specifies the size of the process's logical address range

Def - Trap - A software-generated interrupt triggered by a program (e.g. system call, divide by zero, invalid memory access); transfers control to the OS to handle the event.

Def - Working set - Set of pages a process has referenced within a recent time window; it represents the process's current locality and determines how many frames it needs to avoid thrashing

FIFO	6	5	3	6	2	4	3	7	6	2	4	2	1	3	5	4	6	0	2	3	3	2	4	1
Frame 0	6	6	6	6	6	4	4	4	4	4	4	4	3	3	3	0	0	0	0	0	1			
Frame 1	5	5	5	5	5	7	7	7	7	7	7	5	5	5	5	2	2	2	2	2	2			
Frame 2	3	3	3	3	3	6	6	6	6	6	6	4	4	4	3	3	3							
Frame 3	2	2	2	2	2	2	2	2	2	2	2	1	1	1	6	6	6	6	6	6	4	4		
Faults	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Faults: 17																								
optimal	6	5	3	6	2	4	3	7	6	2	4	2	1	3	5	4	6	0	2	3	3	2	4	1
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	0	3	3	2	3	3
Frame 1	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Frame 2	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	5	5	5	2	2	2	2	2
Frame 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Faults: 12	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
LRU	6	5	3	6	2	4	3	7	6	2	4	2	1	3	5	4	6	0	2	3	3	2	4	1
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
Frame 1	5	5	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Frame 2	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	5	5	5	2	2	2	2	2
Frame 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Faults: 19	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

FIFO - Evict oldest page when full

Optimal - Evict the page that will not be used for the longest time

LRU - Evict the least recently used page

2.4.1 - "Oh Grandma! What big files you have!"

The Berkeley Unix Fast File System (FFS) was created for 2 purposes: to store big files efficiently and to improve performance of file access. (FFS was the forerunner to the Unix file system.) Figure 1 shows how FFS is organized.

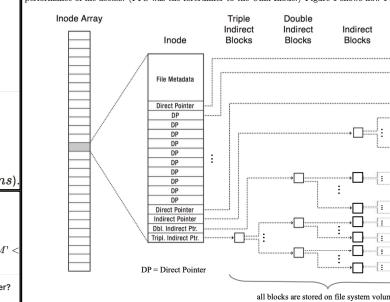


Figure 1: Berkeley FFS

a. From what you can see, explain how FFS works. Describe how a logical reference to some location in a file is processed using FFS to find the data block containing that location. Do you notice any relationship between how FFS works and a multi-level page table? (Hint: All "blocks" in the picture (shown as rectangles) are storage blocks and each storage block is of the same size (in bytes), whether it is used for indication or data. Think about what is in an "indirect" block.) (Use the space below and at the top of the next page.)

(a) How does FFS work?

- FFS uses an Inode to store file metadata + pointers to data blocks.
- The Inode contains:
 - 12 direct pointers → each points to a data block.
 - 1 indirect pointer → points to a block that contains pointers to data blocks.
 - 1 double indirect pointer → points to a block of pointers, each pointing to another block of pointers to data blocks.
 - 1 triple indirect pointer → points to a block of pointers, each pointing to another block of pointers, each of which points to data blocks.

How to find a data block:

- if the offset is small → use a direct pointer.
- if larger → follow indirect, double indirect, or triple indirect pointers level by level to reach the data block.

Relation to multi-level page table:

- Very similar! Both use multiple levels of indirection.
- Both use blocks of pointers of the same size.
- Both translate a logical reference (file offset / virtual address) into a physical location (disk block / physical frame).
- Indirection allows handling large files or large address spaces without a huge flat table.

b. In an inode-based file system implementation like FFS, the inode typically stores 12 direct block (DP) pointers, one 1-level indirect block pointer, one 2-level indirect block pointer, and one 3-level indirect block pointer. Suppose the file system is configured to use a block size of 2^{12} (4096) bytes and all pointers (direct, indirect, double indirect, or triple indirect) takes 4 bytes. What is the maximum file size that can be supported in the FFS file system shown above? Explain your calculation. (Hint: All of the file's data is stored in the data blocks. You need to figure out how many data blocks can be ultimately pointed to by all FFS pointers.)

(b) Max file size calculation

Given:

- Block size = $2^{12} = 4096$ bytes
- Pointer size = 4 bytes → 1 block holds $4096 / 4 = 1024$ pointers

Inode contains:

- 12 direct pointers → 12 data blocks
- 1 indirect pointer → 1024 data blocks
- 1 double indirect pointer → $1024^2 = 1,048,576$ data blocks
- 1 triple indirect pointer → $1,024^3 = 1,073,741,824$ data blocks

Max file size = total data blocks * block size:

$$= (12 + 1024 + 1,048,576 + 1,073,741,824) * 4096 bytes$$

= 1 TB + small extra

Summary: max file size is huge because the triple indirect pointer can reach ~1 billion blocks.

c. Thinking more about the relationship of FFS to page tables, you suddenly have a brilliant idea for speeding up file access. You decide to use the same page table structure for both file and page tables. The idea is that if the data block containing the referenced file byte is found in these most recently referenced blocks (e.g. 8 entries). The idea is that if the data block containing the referenced file byte is found in these most recently referenced blocks (e.g. 8 entries). What needs to be done to make this work? Provide an argument for or against the merits of this idea.

(c) File TLB idea

What should be in the "file TLB":

- Each entry should store:
 - Logical block number within the file
 - Physical block address on disk

How it works:

- On file access, OS first checks the file TLB.
- If the block is cached → directly access physical block.
- If not → walk indirect pointers as usual.

Argument FOR:

- File access often shows locality (sequential reads, nearby block access).
- Speeds up repeated accesses to recently used blocks.
- Small metadata cost, significant performance gain.

Argument AGAINST:

- Adds complexity to inode and file system code.
- Inode is a critical structure → adding TLB increases size and management overhead.
- Not all access patterns benefit (e.g. random or large sequential reads).