CHAPTER 6: DEADLOCKS

A deadlock is said to occur whenever: 2 or more processes are blocked. Each of these processes is waiting for a resource that is held by another blocked process. Examples: 2 friends have exchanged insults, Each is expecting the other to apologize 1st. A rebel grp doesn’t want to cease the hostilities before being recognized by the gov’t, The gov’t is ready to negotiate but only after hostilities have ceased. Relations: Process P holds on/owns resource R. Process P needs/wants resource R. Serially reusable resources: Memory space, buffer space, disk space, USB slot to insert a flash drive. Exist only in a limited quantity. 1 process may have to wait for another process to release the resources it is currently holding. Handling deadlocks: Do nothing: ignore the problem. Deadlock prevention: build deadlock-free systems. Deadlock avoidance: avoid system states that could lead to a deadlock. Deadlock detection: Detect & break deadlocks. Haberman’s conditions: 4 necessary conditions must all be in effect for deadlocks to happen: Mutual Exclusion. Hold & wait. No Preemption. Circular wait. Mutual exclusion: at least 1 of the processes involved in the deadlock must claim exclusive control of some of the resources it requires. Hold & Wait: Processes can hold the resources that have already been allocated to them while waiting for additional resources. No preemption: once a resource has been allocated to a process, it cannot be take away or borrowed from that process until the process is finished with it. Circular wait: There must be a circular chain of processes such that each process in the chain holds some resources that are needed by the next process in the chain. Formal equivalent to what we call a vicious circle. Deadlock prevention: any system that prevents any of the 4 necessary conditions for deadlocks will be deadlock-free. Must find the easiest condition to deny. Denying mutual exclusion: prevent any process from claiming exclusive control of any of the resources. Drawbacks: Many resources can only be used by 1 process at a time. Cannot hold on a message & send it at the same time. Denying hold & wait: Require processes to get all the resources they will need or none of them. Drawbacks: Forces processes to acquire ahead of time all the resources they might need. Does not apply the consumable resources such as messages. Allowing preemption: let processes take away or borrow the resources they need from the processes that hold on them. Drawbacks: Will result in lost work when a process steals storage space from another process. Cannot force processes to send messages. Denying Circular wait: Impose a total order on all resource types & force all processes to follow that order when they acquire new resources. If a process needs more than 1 unit of a given resource type it should acquire all of them or none. Works very well for resources like CPU & mem. Drawbacks: would force messages to move in only 1 direction. Processes could not exchange messages.

**CHAPTER 7: MEMORY MANAGEMENT**

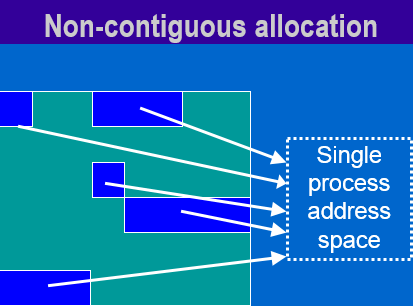
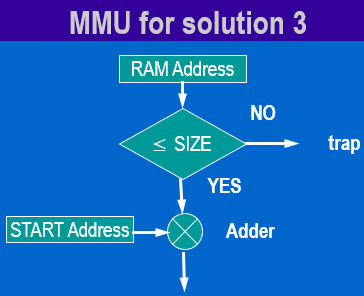
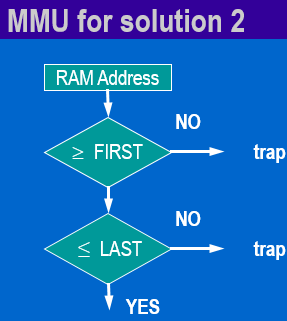
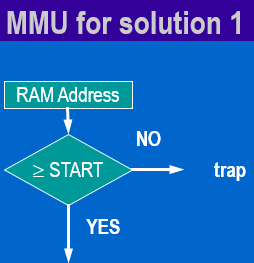
Solution 0: No memory management. Very 1st computers had no OS at all. Each programmer had access to whole main mem of the computer, had to enter the bootstrapping routine loading his or her program into main memory. Advantage: programmer is in total control of the whole machine. Disadvantage: much time is loster entering manually the bootstrapping routine.

Solution 1: Uniprogramming. Every system includes a memory-resident monitor: Invoked every time a user program would terminate. Would immediately fetch the next program in the queue (batch processing). Should prevent user program from corrupting the kernel. Must add a MMU. Assuming that the monitor occupies memory locations 0 to START-1. MMU will prevent the program from accessing memory locations 0 to START-1. Advantage: no time is lost re-entering manually the bootstrapping routine. Disadvantage: CPU remains idle every time the user program does an I/O.

Solution 2: Multiprogramming with fixed partitions, requires I/O controllers and interrupts. OS dedicates multiple partitions for user processes, Partition boundaries are fixed. Each process must be confined btwn its first and last address. Computer often had a foreground partition(FG) and Several background partitions(BG0, …). Advantage: no CPU time lost while system does I/O. Disadvantages: partitions are fixed while processes have different memory requirements. Many systems were requiring process to occupy specific partition.

Solution 3: Multiprogramming w/ variable partitions. OS allocates contiguous extents of mem to processes, initially each process gets all the memory space it needs and nothing more. Processes that are swapped out can return to any main mem location. Initially everything works fine: 3 process occupy most of memory, unused part of mem is very small. [Monitor|P0|P1|P2|/|]. When P0 terminates: Replaced by P3, P3 must be smaller than P0. Start wasting mem space. [Monitor|P3|/|P1|P2|/|]. When P2 terminates: replaced by P4, P4 must be smaller than P0 plus the free space. Start wasting more mem space. [Monitor|P3|/|P1|P4|//|. External fragmentation: Happens in all systems using multiprogramming w/ variable partitions. Occurs b/c new process must fit in the hole left by terminating process, very low probability that both processes will have exactly the same size, typically the new process will be a bit smaller than the terminating process. An Analogy: replacing an old book by a new book on a bookshelf. New book must fit in the hole left by old book, very low probability that both books have exactly the same width, we will end w/ empty shelf space btwn books. Solution is to push books left and right. Memory compaction: When external fragmentation becomes a problem we push processes around in order to consolidate free spaces. Works very well when memory sizes were small. Dynamic address translation: Process don’t occupy fixed locations in main memory, will let them run as if they were starting at location 0, MMU hardware will add the right offset, will test first that process does not try to access anything outside its boundaries. Is it virtual or real? MMU translates: Virtual addresses used by the process into Real addresses in main mem. An Analogy: Living or visiting places that makes us believe we are in a different country: Little Italy in San Francisco, Bazaar del Mundo in San Diego, Chinatown everywhere. Subdivisions with “romantic” Spanish names in California. Streets with names of Ivy League schools or towns hosting them.

Solution 4: Non-contiguous allocation. Partition physical memory into fixed-size entities – Page frames. Allocate non-contiguous page frames to processes. Let the MMU take care of the address translation.



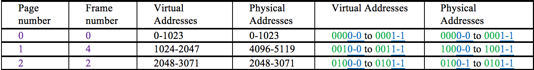
**CHAPTER 8: VIRTUAL MEMORY**

Virtual Memory: Combines 2 big ideas: Non-contiguous memory allocation: processes are allocated page frames scattered all over the main memory. On-demand fetch: Process pages are brought in main mem when they are accessed for the 1st time. MMU takes care of almost everything.

Main Memory: Divided into fixed-size page frames: Allocation units, Sizes are powers of 2 (512B, 1KB, 2KB, 4KB), Properly aligned, Numbered 0, 1, 2, …

Process address space: Divided into fixed-size pages: same sizes as page frames, properly aligned, also numbered 0, 1, 2, …

The Mapping: Will allocate non contiguous page frames to the pages of a process. Assuming 1KB pages & page frames/ Observing that 2^10 = 1000000000 in binary, we write 0-0 for ten zeroes and 1-1 for ten ones. The ten least sig. bits of address do not change. Must only map page numbers into page frame numbers. Same in decimal



The Algorithm: Assume page size = 2^p. Remove p least significant bits from virtual address to obtain the page number. Use page number to find corresponding page frame number in page table. Append p least significant bits from virtual address to page frame number to get physical address.

The Offset: Offset contains all bits that remain unchanged through the address translation process. Function of page size.

The Page Number: Contains other bits of virtual address. Assuming 32-bit addresses:



Internal Fragmentation: Each process now occupies an integer number of pages. Actual process space is not a round number, last page of a process is rarely full. On the average, half a page is wasted, Not a big issue, Internal Fragmentation.

On-Demand Fetch(I): Most processes terminate w/o having accessed their whole address space, ex. Code handling rare error conditions. Other processes go to multiple phases during which they access different parts of their address space, ex. Compilers.

On-Demand Fetch(II): VM systems don’t fetch whole address space of a process when it is brought into memory. They fetch individual pages on demand when they get accessed the 1st time, Page miss or page fault. When memory is full, they expel from memory pages that are not currently in use. On-Demand Fetch(III): The pages of a process that are not in main mem reside on disk. In the executable file for the program being run for the pages in the code segment. In a special swap area for the data pages that were expelled from main memory. On-Demand Fetch(IV): When a process tries to access data that are not present in main memory: MMU hardware detects that the page is missing and causes an interrupt. Interrupt wakes up page fault handler. Page fault handler puts process in waiting state and brings missing page in main memory. Advantages: VM systems us main memory more efficiently than other memory management schemes, Give to each process more or less what it needs. Process sizes are not limited by the size of main memory, Greatly simplifies program organization. Sole Disadvantage: Bringing pages from disk is a relatively slow operation. Takes milliseconds while memory access takes nanoseconds. Ten thousand time to hundred thousand times slower. The Cost of a Page Fault: Let: Tm be the main memory access time. Td the disk access time. f the page fault rate. Ta the avg access time of the VM. Ta = (1-f)Tm + f(Tm + Td) = Tm + fTd

Conclusion: Virtual memory works best when page fault rate is less than a page fault per 100,000 instructions.

Locality Principle: A process that would access its pages in a totally unpredictable fashion would perform very poorly in a VM system unless all its pages are in main memory. Process P accesses randomly a very large array consisting of n pages. If m of these n pages are in main memory, the page fault frequency of the process will be (n-m)/n. Must switch to another algorithm. Fortunately for us most programs obey the locality principle. They access at any time a small fraction of their address space (spatial locality) and they tend to reference again the pages they have recently referenced (temporal locality).

Tuning Considerations: In order to achieve an acceptable performance, a VM system must ensure that each process has in main memory all the pages it is currently referencing. When this is not the case, the system performance will quickly collapse.

Page Table Representation – Page table entries: A page table entry (PTE) contains a page frame number, several special bits. Assuming 32-bit addresses, all fit into 4 bytes. [Page frame number | Bits].

The Special Bits: Valid bit: 1 if page is in main memory. 0 otherwise. Missing bit: 1 if page is not in main memory. 0 otherwise. Dirty bit: 1 if page has been modified since it was brought into main memory, 0 otherwise. A dirty page must be save in the process swap area on disk before being expelled from main memory. A clean page can immediately be expelled. Page-referenced bit: 1 if page has been recently accessed, 0 otherwise. Often simulated in software.

Where to store page tables: use a 3 level approach. Store parts of page table: in high speed registers located in the MMU: the translation lookaside buffer (TLB) (good solution). In main memory (bad solution). On disk (ugly solution).

The translation look aside buffer(TLB): Small high-speed memory: contains fixed number of Page table entries (PTEs). Content-addressable memory, entries include page frame number and page number. [Page number | Page frame number | Bits]

TLB misses: When a PTE can’t be found in the TLB, a TLB miss is said to occur. TLB misses can be handled: by the computer firmware, cost of miss is one extra memory access. By the OS kernel: cost of miss is 2 context switches.

Performance implications: When TLB misses are handled by the firmware, they are very cheap: A TLB hit rate of 99% is very good: Average access cost will be Ta = 0.99Tm + 0.01x2Tm = 1.01Tm Less true if TLB misses are handled by the kernel.

Linear page tables: PTs are too large to be stored in main memory: Store PT in virtual memory (VMS solution). Very large page tables need 2 levels (3 levels on MIPS R3000). Assuming page size of 4KB: Each page of virtual memory requires 4 bytes of physical memory. Each PT maps 4GB of virtual addresses. A PT will occupy 4MB. Storing these 4MB in virtual memory will require 4KB of physical memory.

Multi-level page tables: PT is divided into: A master index that always remains in main memory. Subindexes that can be expelled. Especially suited for a page size of 4KB and 32 bits virtual addresses. Will allocate: 10 bits of the address for the 1st level. 10 bits for the 2nd level. And 12 bits for the offset. Master index and subindexes will all have 2^10 entries and occupy 4KB.

Hashed page tables: Only contain page4s that are in main memory, PTs are much smaller. Also known as inverted page tables.

Discussion: We have much fewer PTEs than w/ regular PT, Whole PT can reside in main memory. Hashed/inverted PTEs occupy 3 times more space than regular PTEs, Must store page number, page frame number and a pointer to next entry.

Selecting the right page size: Increasing the page size: increases the length of the offset. Decreases the length of the page number. Reduces the size of page tables, less entries. Increases internal fragmentation. 4KB seems to be a good choice.

Page Replacement Policies – Function: Selecting which page to expel from main memory when: Memory is full. Must bring in a new page.

Objectives: A good page replacement policy should: Select the right page to expel (victim). Have a reasonable run-time overhead. 1st objective was more important when memory was extremely expensive. 2nd objective has been more important since the mid-eighties.

Classification: 4 classes of page replacement policies: Fixed-size local policies. Global Policies. Variable-size local policies. Hybrid policies (part global and part local).

Fixed-size local policies: Assign to each process a fixed number of page frames. Whenever a process has used all its page frames, it will have to expel one of its own pages from main memory before bringing in a new page. 2 policies: local FIFO and local LRU. Local FIFO: Expels the page that has been in main memory for the longest period of time. Very easy to implement: Can organize the pages frames into a circular queue. Very bad policy: Does not take into account how the page was used. Local LRU: Expels the page that has not referenced for the longest period of time. Best fixed-size replacement policy: Has an extremely high overhead: Must keep track of all page accesses. Never used for VM.

Global policies: Treat whole memory as a single pool of page frames. Whenever a page fault happens and memory is full, expel a page from any process, Processes “steal” page frames from each other. Many policies. Global FIFO & Global LRU: Global variants of local FIFO and local LRU. Same advantages and disadvantages.

MULTICS Clock policy: Organizes page frames in a circular list. When a page fault occurs, policy looks at next frame in list: if PR bit = 0, the page is expelled and the page frame receives the incoming page. If PR bit = 1, the PR bit is reset and policy looks at the next page in list. STEP 1: reset PR bit to 0. STEP 2: reset PR bit to 0. STEP 3: expel this page.

BSD Implementation: Designed for architectures lacking a PR (Page Replacement) bit. Uses the valid bit to simulate the PR bit: Resets valid bit to zero instead of resetting the PR bit to zero. When page is referenced again an interrupt occurs and the kernel sets the valid bit back to one. Requires 2 context switches. STEP 1: mark page invalid. STEP 2: mark page invalid. STEP 3: expel this page. A first problem: When memory is overused, had of clock moves too fast to find pages to be expelled. Too many resets, too many context switches. Berkeley UNIX limited CPU overhead of policy to 10% of CPU time. No more than 300 page scans/second.

Mach Policy: Stores pages from all process in single FIFO pool. Expelled pages go to end of a global LRU queue where they wait before being actually expelled from main memory: Can be rescued if they were expelled by error. FIFO policy makes many errors. When page fault occurs, VM system looks first into the global LRU queue to find whether the missing page is not in the queue. If the missing page is found in queue, it is reclaimed and brought back into the FIFO pool: Cost of reclaim is 2 context switches. Much cheaper than a disk access.

Variable-space local policies: Working set policy let each process keep into main memory all pages it had accessed duping its last T references. Provided excellent performance. Was never implemented due to its very high cost. Influenced research efforts to design better page replacement policies. Not covered anymore.

Hybrid policies: Window page replacement policy combines aspects of local and global policies.

Windows policy: Allocates to each process a private partition that it manages using a FIFO policy. Pages expelled by the FIFO policy are put at the end of a large global LRU queue form which they can be reclaimed, predates by several years use of same solution by Mach. Major Advantage: Supports real-time applications. Most VM systems are poorly suited to real time applications, unpredictable paging delays. Policy allows VM to allocate to a process enough page frames to hold all its pages, Process will never experience a page fault. Major Disadvantage: Hard to decide how many frames to allocate to each process: Allocating too many frames leaves not enough space for the global LRU queue, Page fault rate will become closer to that of a global FIFO policy. Not allocating enough frames would cause too many reclaims and too many context switches.

Windows solution: Each process is allocated a minimum and maximum working set size. Processes start with their minimum allocation of frames. If the main memory is not full, the VM manager allows processes to grow up to their maximum allocation.

As the main memory become full, the VM manager starts trimming the working sets of processes. Processes that exhibit a lot of paging can regain some of their lost frames if enough frames remain available.

Virtual Memory Tuning – The Problem: With virtual memory: most processes run w/o having all their pages in main memory, Can have more processes in main memory: reduces CPU idle times, increases the system throughput. Zone 1: Optimal Behavior: Throughput increases w/ multiprogramming level. Little or no impact of page faults on system performance. Zone 2: Unstable Behavior: Page fault impact on throughput increases. Any surge of demand may move the system performance to zone 3. Situation is analogous to that of a freeway just below its saturation point: Cars still move fast but any incident can cause a slowdown. Zone 3: Thrashing: Active pages are constantly expelled from main memory to be brought back again and again. Paging device becomes the bottleneck. Situation is analogous to that of a freeway above its saturation point: Cars barely move. Prevent Thrashing: Have enough main memory. Start suspending processes when paging rate starts increasing. Old empirical rule: Keep utilization of paging disk below 60%.

**CHAPTER 9: FILE SYSTEMS**

General Organization – The File System: Provides long term storage of info. Will store data in stable storage (disk). Cannot be RAM b/c: Dynamic RAM loses its contents when powered off. Static RAM is too expensive. System crashes can corrupt contents of the main memory. Overall Organization: Data managed by the file system are grouped in user-defined data sets called files. The file system must provide mechanism for naming these data: Each file system has its own set of conventions, All modern operating systems use a hierarchical directory structure.

Windows Solution: Each device and each disk partition is identified by a letter. A: and B: were used by the floppy drives. C: is the first disk partition of the hard drive. If drive has no other disk partition, D: denotes the DVD drive. Each device and each disk partition has its own hierarchy of folders. In a hierarchical file system files are grouped in directories and subdirectories. The folders and subfolders of Windows. These directories and subdirectories form one tree in each disk partition.

UNIX Solution: Each device and disk partition has its own directory tree: Disk partitions are glued together through the mount operation to form a single tree, Typical user doesn’t know where his files are stored. Devices form a separate device hierarchy, Can also be automounted. After mount, root of second partition can be accessed as /usr

File Organizations: Earlier file systems organized file into user-specified records, They were read and written atomically. Starting with UNIX modern file systems organize file as sequence of bytes, Can be read or written to in an arbitrary fashion.

Files are stored on disk using fixed-size records called blocks: All files stored on a given device or disk partition have the same block size. Block sizes are transparent to the user, he rarely knows them.

The case for fixed-size blocks: Programmer defined records were often too small: A grade file would have had one record per student, Around 100 bytes. Can pack around 40 student records in a single 4KB block, one single read replaces 40 reads. Could not read a file w/o knowing its record format, Hindered the development of utility programs.

Protection – Objective: To provide controlled access to info. Both Windows and UNIX let file owners decide who can access their files and what they can do, Not true for more secure file systems. They enforce security restrictions.

Enforcing controlled access: 2 Basic solutions: Access control Lists. Tickets. Each has its advantages & disadvantages.

Access control lists: Table specifying what each user can do w/ the file. Main advantage: Very flexible: can easily add new users to change/revoke permissions of existing users. Main disadvantages: Very slow: must authenticate user at each access. Can take more space than the file itself.

Tickets: Specify what the ticket holder can do. Must prevent users from forging tickets. Use encryption: Similar to using patterns that are hard to forge on bills. Let kernel keep them: Similar to bank doing all the bookkeeping for our accounts.

Main advantage: Very fast: must only check that the ticket is valid. Main disadvantages: Less flexible than access control lists: Cannot revoke individual tickets. Ticket holders can make copies of tickets and distribute them to other users.

Conclusion: Best solution is to combine both approaches. Use access control lists for long-term management of permission. Once a user has been authenticated, give him a ticket. Limit ticket lifetimes to force users to be authenticated from time to time.

UNIX Solution: UNIX: Checks access control list of file whenever a file is opened. Lets file descriptor act as a ticket until the file is closed. File owner can specify 3 access rights: read, write, execute for user (herself), a group, and others (all other users).

rwx - - - - - - : Owner can do everything she wants w/ her file & nobody else can access it. rw – r - - r - - : Owner can read from and write to the file, everybody else can read the file. rw – rw - - - - : Owner and any member of group can read from and write to the file. UNIX access control lists: Main Advantage: Takes very little space: 9 bits plus 32 bits for group-ID. Main disadvantage: Less flexible than full access control lists: Groups are managed by system admin. Works fairly well as long as groups remain stable.

Selecting the block size: Much more important issue than selecting the page size of a VM system b/c: Many very small files, small UNIX test files. Some very large files, Music, Video. The 80 – 20 rule: We can roughly say that: 80% of the files occupy 20% of the disk space. Remaining 20% occupy the remaining 80%. The dilemma: Small block sizes: Minimize internal fragmentation, best for storing small files. Provide poor data transfer rates for large file, Too many small data transfers. There is no single optimum block size, Depends too much on file sizes.

UNIX Implementation – Version 7: Each disk partition contains: Superblock containing parameters of the file system disk partition. An i-list w/ one i-node for each file or directory in the disk partition and a free list. The data blocks (512 bytes).

The i-node: Each i-node contains: The user-id and the group-id of the file owner. The file protection bits. The file size. The times of file creation, last usage, and last modification. The number of directory entries pointing to the file and a flag indicating if the file is a directory, an ordinary file, or a special file. 13 block addresses. The file name(s) can be found in the directory entries pointing to the i-node. Storing block addresses: 10 direct blocks. Block size = 512 B. 3 levels of indirection 128^3 block addresses. How it works: 1st 10 blocks of file can be accessed directly from i-node: 10 x 512 = 5, 120 bytes. Indirect block contains 512/4 = 128 addresses: 128 x 512 = 64 kilobytes. With 2 levels of indirection we can access 128 x 128 = 16K blocks: 16k x 512 = 8 megabytes. With 3 levels of indirection we can access 128 x 128 x 128 = 2M blocks: 2M x 512 = 1 gigabyte. Maximum file size is 1 GB + 8 MB + 64 KB + 5 KB. Explanation: File sizes can vary from a few 100 bytes to a few gigabytes w/ a hard limit of 4 gigabytes. The designers of UNIX selected an i-node organization that: wasted little space for small files, Allowed very large files. Discussion: What is the true cost of accessing large file? UNIX caches i-nodes and data blocks. When we access sequentially a very large file we fetch only once each block of pointers, Very small overhead. Random access will result in more overhead. FFS Modifications: BSD introduced the “Fast File System” (FFS). Superblock is replicated on cylinders of disk. Disk is divided into cylinder groups. Each cylinder group has its own i-node table, It minimizes disk arm motions. Free list replaced by bit maps. Cylinder groups: In the old UNIX file system i-nodes were stored apart from the data blocks. Too many long seeks, Affected disk throughput. FFS partitions the disk into cylinder groups containing both i-nodes and data blocks. Most file have their data blocks in the same cylinder group as their i-node. Problem solved. The FFS i-node: i-node has now 15 block addresses. Minimum block size is 4k, 15th block address is never used. FFS organization: Block size b>= 4kb. 12 direct blocks. b/4 indirect blocks. b/4 x b/4 double indirect blocks. How it works: In a 32 bit architecture, file size is limited to 2^32 bytes, that is 4GB. When block size is 4KB, we can access: 12 x 4KB = 48KB directly from i-node. 1,024 x 4KB = 4MB with one level of indirection. 4GB – 48KB – 4MB with 2 levels of indirection. The bit maps: Each cylinder group contains a bit map of all available blocks in the cylinder group. The file system will attempt to keep consecutive blocks of the same file on the same cylinder group. Block sizes: FFS uses larger blocks allows the division of a single file system block into 2, 4, or 8 fragments that can be used to store: small files. Tails of larger files. Explanations: Increasing the block size to 4K eliminates the 3rd level of indirection. Keeping consecutive blocks of the same file on the same cylinder group reduces disk arm motions. Allocation full blocks and block fragments: allows efficient sequential access to large files, minimizes disk fragmentation. Using 4K blocks w/o allowing 1K fragment would have wasted 45.6% of the disk space, this would not be true today. Metadata issues: Most of the good performance of FFS is due to its extensive use of I/O buffering, Physical writes are totally asynchronous. Metadata updates must follow a strict order: FFS uses blocking writes for all metadata updates, more recent file systems use better solutions. Deleting a file: Correct sequence is: 1.) Write to disk directory block containing deleted directory entry. 2.) Write to disk i-node block containing deleted i-node. Leaves the file system in a consistent state. Creating a file: Correct sequence is: 1.) write to disk i-node block containing new i-node. 2.) Write to disk directory block containing new directory entry. Leaves the file system in a consistent state. Handling metadata updates: Out-of-order metadata updates can leave the file system in temporary inconsistent state: Not a problem as long as the system doesn’t crash btwn the 2 updates. Systems are known to crash. FFS Solution: FFS performs synchronous updates of directories and i-nodes: Requires many more seeks. Causes a serious performance bottleneck. BETTER SOLUTIONS: Log-structured file systems: BSD-LFS. Soft updates. Journaling file systems (most popular approach).

JOURNALING FILE SYSTEMS: Key Idea: Record metadata updates, 1st on a log (the journal), Later at their proper location. When recovering from a crash, use the journal to finalize all incomplete metadata updates. Step 1: Update buffer and journal. Step 2: update the file system (can now remove update record from the journal). Explanations: Metadata update are written twice on disk: 1st in the journal. Then the file system. All other updates remain non-blocking. Advantage: writing metadata updates twice is still cheaper than using a single blocking write b/c: Journal is organized as a log and all writes are sequential. Second update is non-blocking. Implementation rules: Journaling file system must ensure that: every update is written first in the journal before the file system is updated. Journal entries corresponding to updates that have been propagated to the file system are removed from the journal. Complicates I/O buffer design. Synchronous JFSes: Write all metadata updates one by one in the journal w/o any delay. Guarantee file system will always recover to a consistent state. Guarantee that metadata updates will never be lost. Asynchronous JFSes: Buffer their writes to the journal are buffered until an entire buffer is full. Guarantee file system will always recover to a consistent state. Don’t guarantee that metadata updates will never be lost. Are much faster than synchronous JFS.

RECENT FILE SYSTEMS – Linux file systems: 1st Linux file system was a port of Minx file system: Essentially a “toy” file system, Maximum file size was 64MB. Many more recent file systems: Ext1, Ext2, Ext3, Ext4, … and others. Ext2: Was essentially analogous to the UNIX fast file system we discussed before: 15 block addresses per i-node, Cylinder groups are called block groups. Major differences include: larger maximum file size: 16GB – 2TB, various extensions ex. online compressions, full ACLs. Ext3: Journaling file system w/ 3 levels of Journaling. Journal: journals metadata and data updates. Ordered: guarantees that data updates will be written to disk before associated metadata are marked as committed. Writeback: makes no such guarantees. Windows File System (NTFS): Another journaling file system. Each file is an object composed of one or more data streams.

“Only the main stream of a file is preserved when it is copied to a FAT-formatted USB drive, attached to an e-mail, or uploaded to a website.” NTFS data structures: Master File Table (MTF): contains most metadata, equivalent to UNIX i-node table. Each file can have one or more MFT records depending on file size and attribute complexity. MFT records contain: pointers to data blocks for most files, Contents of very small files. NTFS block allocation policy: Allocates block clusters instead of individual blocks. Each cluster has space for several contiguous blocks. Cluster size is defined when the disk drive is formatted. Improves performances but increases internal fragmentation.

UNIX FILE SEMANTICS – File types: 3 types of files: Ordinary files: uninterrupted sequence of bytes. Directories: accessed through special system calls. Special files: allow access to hardware devices. Ordinary files (I): 5 basic file operations are implemented: open() – returns a file descriptor. read() – reads so many bytes. write() – writes so many bytes. lseek() – changes position of current byte. close() – destroys the file descriptor. All reading and writing are sequential. The effect of direct access is achieved by manipulating the offset through lseek(). Files are stored into fixed-size blocks. Block boundaries are hidden from the users, Same as in MS-DOS/Windows. The file metadata: Include file size, file owner, access rights, last time the file was modified, but not the file name. Stored in the file i-node. Accessed through special system calls: chmod(), chown(), …I/O buffering: UNIX caches in main memory: i-nodes of opened files, Recently access file blocks. Delayed write policy: increases the I/o throughput, Results in lost writes whenever a process or the system crashes. Terminal I/O are buffered one line at a time.

Directories (I): Map file names with i-node addresses. Don not contain any other information. 2 directory entries can point to the same i-node. Directory subtrees can’t cross file system boundaries unless a new file system is mounted somewhere in the subtree. To avoid loops in directory structure, directory file can’t have more than one pathname.

Special Files (I): Not files but devices: /dev/tty is your current terminal. dev/fd0 is your floppy drive. Advantage: allows access to devices such as floppy drive, tape drive, …as if they were regular files. Disadvantage: We want to see floppy drives and flash drives as file systems, not as single files. A better solution is to mount them automatically when they get inserted (automount): MS-DOS/Windows solution, Should be able to recognize FAT formatted devices.

MAPPED FILES – Virtual memory and I/O buffering (I): In a VM system, we have: Implicit transfers of data btwn main memory and swap area (page faults, etc.) Implicit transfers of information btwn the disk drive and the system I/O buffer. Explicit transfers of info btwn the I/O buffer and the process address space. I/O buffering greatly reduces number of disk accesses. Each I/O request must still be serviced by the OS: 2 context switches per I/O request. Mapped files (I): When a process opens a file, the whole file is mapped into the process virtual address space: no data transfer takes place. File blocks are brought in memory on demand. File contents are accessed using regular program instructions (or library functions). Shared files are in shared memory segments.

Mach Implementation: Process virtual address space, usual VM pager, swap area, External pager, file system.

Linux Implementation (I): mmap(…): Maps files or devices into memory. Implements demand paging, File blocks are brought on demand, Lazy approach. Can map a portion of a file (offset + number of bytes). Syntax: #include <sys/mman.h> void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset); Can only map opened files!!!! A few options and flags: Setting addr to NULL lets the system choose the start address of the mapped file. Flag MAP\_SHARED makes updates to the mapping visible to all processes that map the file. Flag MAP\_PRIVATE keeps these updates private. Flag MAP\_ANONYMOUS along with flag MAP\_SHARED creates a shared memory segment. Linux implementation (II): mmsync(…) int msync(void \*addr, size\_t length, int flags); Flushes back to disk all changes made to the mapping in main memory. Flushes back to disk all changes made in main memory from address addr to address addr + length – 1. Many flag options. Discussion: solution requires very large address spaces. Most programs will continue to access files through calls to read() and write(): function calls instead of system calls. NO context switches! Problems: 2 major problems: Harder to know the exact size of a file. Much harder to emulate the UNIX consistency model in a distributed file system.

**Summer 2015 REVIEW:**

1.] Current implementations of the AMD64 architecture use 48 bit virtual addresses. Assuming a page size of 4KB:

[A] How many bit of the address would be used by the byte offset? 12 bits

[B] How many bits of the address would be used by the page number? 48-12 = 36 bits

[C] How many pages would there be in a process address space? 2^36 pages

2.] Explain why the FIFO page replacement policy: A.] Has very low overhead. Since it always expels the page that was brought first in main memory, it does not have to keep track of the process memory access patterns. B.] Produces more page faults than other policies. It will from time to time expel a page that is currently used by the process.

3.] A 32 bit FFS file system has a block size of 4KB. How many blocks of a 510KB file can be accessed: A.] Directly from the i-node? 12 blocks. B.] With one level of indirection? 128-12 = 116 blocks. C.] With two levels of indirection? 0 blocks. (*Hint: The total of your three answers should equal to the size of the file.*) Note: When you compute the number of blocks occupied by a file, you need to round it up! Otherwise you would truncate the file. In this case, 510 KB/4 KB = 127.5 rounded up to 128.

4.] A.] What is the purpose of the UNIX mount() system call? The mount system call merges the directory trees of two distinct disk partitions into a single tree. B.] How does a *journaling file system* record *metadata updates*? It writes these updates twice on the disk, first on a log and then at their normal place. C.] What is the *main disadvantage* of letting the *kernel* handle *TLB misse*s? Two additional context switches. D.] What is the purpose of the *dirty bit* in a virtual memory system? The dirty bit indicates whether a page frame has been modified since it was brought in main memory or has remained unchanged. E.] Where do UNIX file systems store their *access control lists*? UNIX stores file access control lists in the i-node of each file. F.] Why can *inverted page tables* fully reside in main memory? Because they only contain the addresses of the virtual pages that reside in main memory.

5.] Alice is the owner of the file netsimulator whose protection bits are ‐rwxr‐xr‐x. She has assigned the group networks to the file. A.] What can she do with the file? Read , write and execute the file. B.] What can members of the networks group do? Read and execute it. C.] What can other users do? Read and execute it.

6.] When does *thrashing* happen? Thrashing occurs when there are too many processes in main memory and active pages are constantly expelled from main memory to be brought back again and again. What can we do to *prevent it*? We should limit the utilization of the paging device to 60 percent. OR we should limit the multiprogramming level of the system.

7.] In the *Windows page replacement policy*, what happens when a page is *expelle*d from the resident set of a process? The page expelled from the resident set of a process goes to the end of a global queue from where they can be reclaimed.

**Review 3 Summer 2012:**

True or False: External fragmentation is often the result of using very large page sizes. False: Larger pages sizes increase internal fragmentation but it rarely is an issue in VM systems.

Global page replacement policies cannot handle real-time processes. True: Real-time processes must keep all their pages in memory all the time, which can only be achieved by giving them a private partition containing all the pages they will ever need.

UNIX uses file-specific passwords to control access to files. False: UNIX uses the file access control list to check which file access rights can be granted to a user process.

UNIX stores the name of a file in its i-node: False: see next answer.

A UNIX file can have several names. True: It can have as many names as there are directory entries pointing to it.

UNIX special files are not files. True: They are physical devices – or POSIX named semaphores.

TLB misses: When comparing the hit ratios of 2 TLBs, which question should we ask first? Are TLB misses handled by the firmware or by the OS? If TLB misses are handled by the firmware, the cost of a TLB miss is one extra memory reference. If TLB misses are handled by the OS, the cost of a TLB miss is 2 context switches.

What is the purpose of the dirty bit? The dirty bit tells whether a page has been modified since the last time it was brought into main memory. It is used whenever a page must be expelled from main memory. If it’s dirty bit is ON, the page must be save to disk before being expelled. If it’s dirty bit is OFF, there already is an exact copy of the page on disk.

How does the BSD UNIX page replacement policy simulate a missing page-referenced bit? Whenever the hand of the clock encounters a page that has been recently used, it marks it INVALID instead of clearing its page-referenced bit.

What is the main drawback of the technique? The main drawback of this technique is that an interrupt will occur the next time that page will be accessed. The cost of handling this interrupt will be roughly equal to 2 context switches.

Among the following page replacement policies: Local FIFO, Local LRU, Global FIFO, Global LRU, Original Clock, Berkeley Clock, and Windows:

Which are the poorest performers? Local FIFO and Global FIFO

Which ones cannot be implemented at a reasonable cost? Local LRU and Global LRU

Which ones provide a decent performance and do not require any special hardware support? Berkeley Clock and Windows

Which ones support real-time processes? Windows (older exams mention VMS)

Page Table Organization: A virtual memory system has a virtual address space of 4GB and a page size of 4KB. Each PTE occupies 4 bytes.

A.] How many bits are used for the byte offset? Since 4K = 2^12, the byte offset will use 12 bits.

B.] How many bits are used for the page number? Since 4G = 2^32, we have 32-bit virtual addresses of these 32 bits the page number occupies 32 – 12 = 20 bits.

C.] What is the maximum number of page table entries in a page table? Address space / Page size = 2^32 / 2^12 = 2^20 PTEs.

D.] What is the maximum size of a page table? Since each PTE occupies 4 bytes, 4 x 2^20 = 2^22 bytes = 4MB.

UNIX File Organization: A 32 bit UNIX file system has 16 KB pages and i-nodes with 15 block addresses. How many file bytes can be accesses:

A.] Directly from the i-node? The 12 first blocks that is 12 x 16KB = 192KB.

B.] With 1 level of indirection? An indirect block can store 16KB / 4B = 4K block addresses. Hence we can access the next 4K x 16K = 64MB of the file.

C.] With 2 levels of indirections? We should be able to access 4K x 4K = 2^24 blocks of 16KB = 256GB. Since the maximum file size is 4GB we will be able to access 4GB – 64MB – 192KB.

D.] With 3 levels of indirections? None

Internal Fragmentation: Internal fragmentation is not a big problem in VM systems. Is it always so in file systems? Give a hypothetical example to support your argument. Internal fragmentation can be a big problem whenever the disk partition contains many files than are much smaller than the block size. Consider for instance a partition where many files are smaller than 1KB and the block size is 4KB. More than 75 percent of the space used by these files will be wasted.

Controlling access to files: Alice, Bob, and Carol want to share a few files on a UNIX while preventing other user to access these files. What should they do? They should ask their system admin to create a group “abc” for the three of them.

Give examples of tickets and access controls lists in the UNIX file system. The file descriptor of an opened file acts as a ticket. The protection bits of a file constitute its access control list.

Metadata updates: Give examples of what could happen if metadata updates are written to disk in the wrong order. How does the “Fast” File System avoid that? What is the main drawback of this approach? Whenever we create a new file, the block containing the new i-node must be written to the disk before the block containing the directory entry pointing to that i-node. Otherwise, a crash happening after the directory block has been written to disk but before the block containing the new i-node has been written to disk would leave the file system in unsafe state. The Fast File System avoids that by requiring all I/O operations involving metadata update to be blocking. This solution dramatically decreases the throughput of the file system.

What are the respective advantages of Journaling file systems using:

A.] Synchronous log updates? Guarantee that no metadata update will be lost when the system crashes

B.] Asynchronous log updates? much faster b/c they do much less physical I/Os

**Final spring 2011:**

b) List the contents of a *UNIX directory entry*. A file name and an i-node number.

c) What would happen to the performance of the UNIX file system if *i-nodes* were *not cached* in main memory?

Each block read or write will now require at least two disk I/Os. Also acceptable: file system would be much slower.

d) How does a *TLB entry* differ from a regular *page table entry*? In addition to the usual contents of a page table entry, a TLB entry also contains a page number. Also acceptable: TLB entries are stored in high-speed registers; TLB contains the

most recently used page table entries.

F) What is the major advantage of tickets over access control lists? They are much faster

A netbook has two Gigabytes of main memory, 32-bit addresses and a page size of four Kilobytes. A.] How many page frames are there in main memory? 2^31 / 2^12 = 2^19 (=512K) frames. B.] How many bits of the virtual address are taken by the page number? 12 bits. C.] How many bits of the virtual address are taken by the byte offset? 20 bits. D.] On avg, how much memory is lost to internal fragmentation? 2Kilo (=2^11) bytes per program segment.

5.] The windows page replacement policy allocates a specific number of page frames to each process. Under which conditions is this number increased or decreased. a) It is increased when the main memory is not full

b) It is decreased when the main memory becomes full

6. A Berkeley UNIX file system has a block size of eight kilobytes. How many indirect blocks will it allocate: a) For a 80-kilobyte file? none indirect blocks b) For a one-megabyte file? one indirect blocks.

*You may use the available space below to detail your computations for potential partial credit. .*

This Berkeley UNIX file system can access: a) 12 blocks or 92 KB directly from the i-node

b) 8 K/4 =2 K blocks or 16 MB with a single indirect block

**Final summer 2012:**

1. Which of the seven following page replacement policies *Global LRU Global FIFO Original Clock Berkeley Clock Mach Windows*

a) Simulate a *page referenced-bit* ? Berkeley Clock

b) Incorporate a *global queue* ? Mach and Windows (I would also accept Gloval LRU and Gloal FIFO)

c) Were designed to handle *real-time* applications? Windows

2. For each of the statements below, indicate in one or two sentence whether the statement is *true* or *false*, and *why*:

a) UNIX lets each user select the block size of his or her files. FALSE, the block size is set by the system administrator for all files in a given disk partition.

b) Internal fragmentation is much more of an issue in file systems than in virtual memory systems. TRUE, because the same disk partition might contain both very small and very small files.

c) In the Fast File System, each *cylinder group* contains *either* inodes *or* data blocks. FALSE, each cylinder group contains both inodes and data blocks.

3. *Questions with short answers*

a) How do you prevent *thrashing* in a virtual memory system?By ensuring that all programs have their working set of pages in physical memory. This requires a good page replacement policy and enough physicial memory.

b) Why are TLB entries *bigger* than regular page table entries? Because they must contain the virtual page number in addition to the page frame number.

c) How does the Fast File System ensure that all *metadata updates* are executed in the *right order*? By forcing all metadata updates to be BLOCKING: this ensures that these operations are executed one by one in strict sequence.

d) Describe the full contents of a UNIX *directory entry*? Each directory entry contains a name and an inode number (and NOTHING ELSE!)

e) In the Windows page replacement policy, what is the cost of *reclaiming* a page from the *global queue*? Two context switches.

f) What is the main advantage of using *asynchronous log updates* in a *journaling file system*? Faster updates.

4. An old virtual memory system has a virtual address space of four Gigabytes and a page size of two

Kilobytes. a) How many bits are used for *the page number*? 32 – 11 = 21 bits

b) How many bits are used for the *byte offset*? log2 2,048 = 11 bits

5. A virtual memory system has 8 Kilobyte pages and 64-bit addresses and uses *inverted page tables*.

a) Given that each page table entry occupies 24 bytes what is the fraction of main memory occupied

by page tables? 24/8K = 24/8,192 = a bit less than 3 percent (2.923 percent to be exact)

b) Why are these page tables entries occupying 24 bytes? Each page table entry must contain a page number, a page frame number and the address of the next entry. Since the system uses 64-bit addresses each of these entries occupies 8 bytes.

6. A UNIX Fast File System has 32-bit addresses, 8 Kilobyte blocks and 15 block addresses in each i-node. How many file blocks can be accessed: a) Directly from the i-node? 12 blocks b) With one level of indirection? 8K/4 =2K blocks

c) With two levels of indirection? 4G/8K –2K – 12 = 512K -2K – 12 = 510K -12 blocks

(The maximum file size is 4GB!) (I would have accepted 512K as a correct answer.)

d) With three levels of indirection? ZERO blocks

**Summer 2013 final:**

A PC has 4 GB of memory, 32-bit addresses and 4 KB pages.

a) How many bits of the virtual address are taken by the page number? 12 bits

b) How many bits of the virtual address are taken by the byte offset? 32- 12 =20 bits

c) How many page frames are there in main memory? 4GB/4KB = 1M = 220 page frames

d) On average, how much memory is lost to internal fragmentation? 4KB/2 = 2KB = 2,048 bytes per code or data segment

2. The Windows page replacement policy allocates a specific number of page frames to each process. Under which conditions is this number decreased? When the memory is tight.

3. A 32-bit Berkeley UNIX file system has a block size of 8 kilobytes. How many *blocks*—*not bytes—*of a given file can be accessed :

a) Using the block addresses stored in the i-node? 12 blocks

b) With one level of indirection? 8K/4 = 2K blocks

c) With two levels of indirection? 4G/8K – 2K – 12 = 510K - 12 blocks

4. Questions with short answers:

a) List the contents of a *UNIX directory entry*. A UNIX directory entry contains a file name and an i-node number.

b) Why is the cost of many system calls equal to *two context switches*? Because these system calls involve trivial requests to the kernel and the cost of the transfers of control to and from the kernel dominates.

c) How does a *TLB entry* differ from a regular *page table entry*? It contains a page number in addition to a page frame number and various bits.

d) What is the major advantage of *tickets* over *access control lists*? They can be validated much faster as the system does not have to check the identity of the entity presenting the ticket.

e) What does the Berkeley Fast File System to reduce *internal fragmentation*? Each block can be partitioned into block fragments that can be allocated individually.

f) How does Berkeley UNIX simulate the *page referenced bit*? Clearing the page referenced bit of a page frame is replaced by marking this page

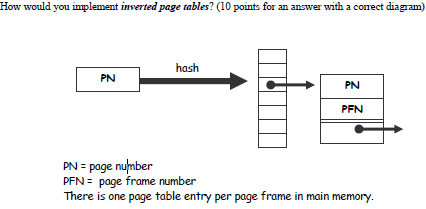
INVALID. Setting the page referenced bit is replaced by marking this page VALID.

5. Given a memory size of two page frames, enumerate the pages that would be *expelled* from main memory while processing the reference string 1 1 0 1 1 0 1 0 2 0

a) under a *LRU* policy: page 1 (when page 2 is fetched into main memory)

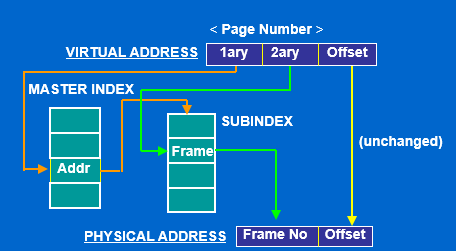
b) under a *FIFO* policy: page 1 (when page 2 is fetched into main memory)

6. How should the type of files stored in a disk partition affect your choice of a block size for that partition? Large block sizes are better when the partition contains very large files. Small block sizes are better when the partition contains many very small files.



6. How does BSD UNIX page replacement policy simulate a missing page-referenced bit? The BSD page replacement policy uses a missing page-referenced bit. Each time it scans a page whose valid bit is one, it resets valid bit to zero. It stops when it encounters a page whose valid bit is zero and it picks this page as the page to expel. What was the main drawback of the technique? The main problem with the policy is its significant context switch overhead. Whenever a page whose valid bit has been reset to zero is referenced again an interrupt occurs and the kernel must set the valid bit back to one. Since is task is trivial, its cost is dominated by the cost of the two required context switches.

Consider the following page table organization applied to a 32-bit architecture and a page size of 4KB:



a) What is the ***size of the page number field***? 32 – 12 = 12 **bits**

b) How would you split it? The first ten bits would be an offset in the MASTER INDEX of the page table

while the remaining ten bits would be an offset in the SECONDARY INDEX.

**Mock Quiz 3:**

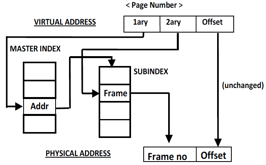
1. A cheap laptop has 2 GB of main memory, 32-bit addresses and a page size of 4 KB. a) How many page frames are there in main memory? 2GB/4KB = 2^31/2^12 = 2^19 = 512K frames [Note: 1KB = 2^10 B, 1MB = 2^20 B, 1GB = 2^30 B, 1TB = 2^40B] b) How many bits of the virtual address are taken by the byte offset? log24K = 12 bits c) How many bits of the virtual address are taken by the page number? (Virtual address bits) 32 - (Offset bits) 12 = (Page Number bits) 20 bits

2. The old UNIX Fast File System required all disk writes involving metadata updates to be performed in a synchronous fashion. Why? To ensure these updates are performed in the right order, thus guaranteeing the consistency of the file system. b) What was the main drawback of this approach? It severely affects the performance of the file system.

3. Consider a 64-bit UNIX file system where all block addresses are 8-byte long and file sizes can exceed 4GB. Assuming a 4KB block size, how many file blocks could it access: a) With one level of indirection? (4K/8) = 2^9 = 512 blocks [Note: Each indirect block can contain 4K/8) = 2^9 = 512 block addresses] b) With two levels of indirection? (512)^2 = 2^18 = 256K blocks c) With three levels of indirection? (512)^3 = 2^27 = 128M blocks

4. Questions with short answers: a) Which page replacement policy supports real-time processes? The Windows policy. b) What is the main advantage of journaling file systems with asynchronous metadata updates? They are much faster than journaling file systems with asynchronous metadata update. In addition, they do not guarantee the durability of recent updates but that’s a disadvantage. c) What is the purpose of the lseek() system call? It moves the read/write file offset of the file thus allowing random access to the contents of the file. d) What is the purpose of the valid bit in a virtual memory system? It tells whether the page is in main memory (“on”) or missing(“off). e) Where do UNIX file systems store file names? In directory entries. f) What is the major disadvantage of very large block sizes in file systems? They cause excessive internal fragmentation whenever the file system contains many very small files.

5. Consider the following multilevel page table organization. It was said in class that that organization was ideally suited to virtual memory systems with 32-bit addresses and 4 KB pages. Why?

Since the page number occupies 20 bits, it can be partitioned in two equal parts so that the master index and all subindexes occupy 4KB. We have 2^10 entries per index and each index entry occupies 4 bytes.

6. Give examples of an access control list and a ticket in the UNIX/LINUX file systems? a) Access control list: The nine protection bits in the file i-node b) Ticket: File descriptors

7. Consider the classical BSD clock replacement policy with a single hand. a) What happens when the hand of the clock reaches a valid page? The page is marked invalid even though it remains in main memory. The idea is to mark the page as not recently referenced. b) What happens when the hand of the clock reaches a page that was marked invalid? The page is expelled from main memory. c) What happens when a process tries to access a page that was marked invalid? The page fault handler is called, finds out that the page is main memory and marks it valid. The overhead of the operation is two context switches.