

Announcements

- Today is the last day for midterm re-grades!
- Design Docs for phase 3 are due on Tuesday
- Design reviews will be Thursday and Friday of next week.
- Project 2 Final Design Docs should be graded by this weekend.

Today's Menu:

- Phase 3 Introduction
- Disk Management and File Layout
- Naming and Directories
- LRU Approximation Algorithms
- Quiz

Nachos Phase III Introduction

- Phase III is much more complicated than Phase II. **START EARLY**
 - o Hopefully this lecture will give you a good start
- In Phase III, you will be working with a SOFTWARE-MANAGED TLB
 - o Processor no longer sees page tables.
 - o It manages a (small) TLB, which is an array of type TranslationEntry
 - Each entry has
 - int vpn
 - int ppn
 - boolean valid
 - boolean readOnly
 - boolean used
 - boolean dirty
 - o Note that the size of TLB is only 4 entries!
 - o The TLB is a private TranslationEntry[4] in Machine.processor
- On each memory reference, MIPS processor looks in TLB for matching entry and uses it if present.
 - o entry.vpn must be same as requested vpn, and entry.valid must be set
 - o If no entry or not valid, an 'exceptionTLBMiss' is generated by the processor
 - o If attempting to write and page is readOnly, an 'exceptionReadOnly' is generated
 - Nothing you need to do about this but cleanly kill the process
 - o Sets entry.used and entry.dirty (if writing)

- Also for this project you will use a global INVERTED PAGE TABLE:
 - o Maps <process ID, vpn> to ppn
 - o OK to use standard java.util.Hashtable (but don't depend on its synchronization properties!)
- How to DEAL WITH A TLB MISS?
 - o Get faulting vaddr from processor register
 - o Look in inverted page table for ppn
 - o Find invalid TLB entry (scan TLB)
 - If none, overwrite a TLB entry
 - Write ppn to TLB entry
- How do we FLUSH THE TLB?

QUESTION: When do we want to do this?

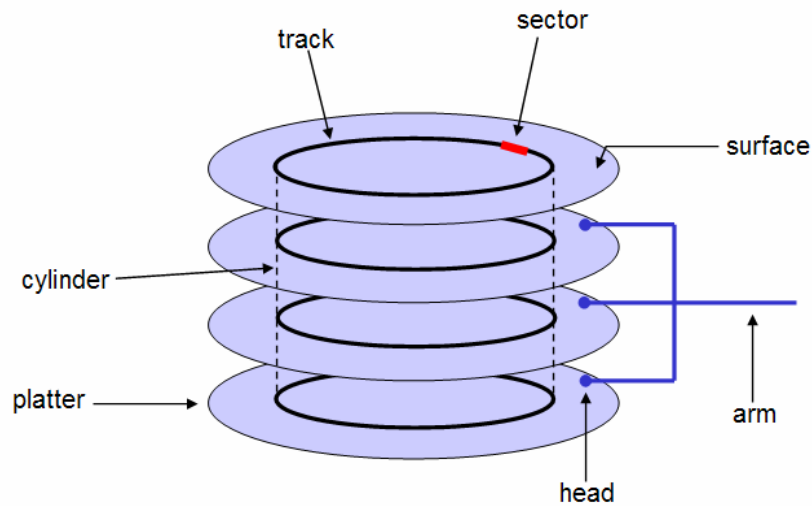
ANSWER: On a context switch.

- o Just set all TLB valid bits to false!
- Also in Phase III: Implement DEMAND PAGING
 - o Simulate much larger physical memory by using disk as backing store
 - o Move contents of memory to/from disk as necessary to provide this illusion -> totally transparent to applications
- Basic idea:
 - o TLB miss causes lookup in page tables to find translation
 - o If not in the inverted page table or entry is invalid
 - Try to allocate a free page
 - If no free pages, choose a page for eviction
 - If evicted page is not dirty, just reuse it
 - If evicted page is dirty, write it to the swap file
 - Read faulting page in from swap file
 - Set valid bit of entry to true
 - Write entry to TLB
 - Return from TLB miss
- ISSUES in Phase III (i.e. Things That Would Be Good To Consider™)
 - o You will need a Core map
 - Provides ppn -> vpn mapping
 - o Since the global page table only contains pages that are actually in physical memory. . .
 - Must maintain a separate data structure for locating pages in swap
 - o What if a page is in use by another process/thread?

- I.e. we are doing a memory copy to/from the page, or loading it from disk?
 - Need to ensure these pages are not touched during page replacement
 - Page "pinning": keep it locked in physical memory for a short amount of time
 - What happens if all pages are "pinned" during sweep?
 - Need to sleep until page is no longer "pinned"
- Format of SWAP FILE is up to you. . . however, a suggestion:
 - Start out with a zero-length file
 - Every time we need new space, just grow swap file by one page
 - Reuse free pages (maintain linked-list of free entries)
 - When writing to swap: Find empty swap file page and write memory page to it, reset page table entry
 - When reading from swap: Find swap page number, read swap page into memory, reset page table entry
- TESTING WILL BE EXTREMELY IMPORTANT FOR THIS PROJECT.
 - There are many, many cases in the above operations.
 - How are you going to test them all?
 - Black Box Testing:
 - Use coff files to debug your VM.
 - Some suggestions in the spec.
 - Evaluation of Page-Replacement Algorithm
 - Collect stats about page faults
 - Compare your page replacement policy with random

Disk Management

- Vocabulary:
 - Disks: multiple big platters filled tracks.
 - Tracks: circuits around the disk sectioned into sectors.
 - Blocks: one or more sectors, the logical unit of disk space allocation.
 - Sectors: group of magnetic bits on the platter. Sometimes interchanged with blocks.
 - Cylinder: all of the sectors in a vertical column.



QUESTION: When laying out a file on a disk, what kinds of things do I need to consider?

ANSWER: Many. Easy to grow file? Fast? (i.e. minimize seeks) Fragmentation? Ease of file creation? Ease of deletion? Favor small or large files? (whats the common case?) Where do oft-used files go?

- Disk Performance
 - o Performance depends on a number of steps
 - seek: moving the disk arm to the correct cylinder
 - depends on how fast disk arm can move
 - seek times aren't diminishing very quickly
 - rotation: waiting for the sector to rotate under head
 - depends on rotation rate of disk
 - rates are increasing, but slowly
 - transfer: transferring data from surface into disk controller, and from there sending it back to host
 - depends on density of bytes on disk
 - increasing, and very quickly
 - o When the OS uses the disk, it tries to minimize the cost of all of these steps
 - particularly seeks and rotation
- Methods of allocating disk to files
 - o Contiguous: allocate all the blocks in a row!

QUESTION: Pros? Cons?

ANSWER: Fast sequential access, easy random access, but causes external fragmentation and its hard to grow files.

- Linked: allocate blocks separately, with a pointer in each block pointing to the next block.

QUESTION: Pros? Cons?

ANSWER: Growing files is easy. However, HORRIBLE random access, lose a node and break the list = loss of file.

- Indexed: hold an array of pointers to all the blocks

QUESTION: Pros? Cons?

ANSWER: Random access is good, growing file is easy. However, fixed file size per index, and still lots of seeks for sequential access.

- How its actually done in Unix: Multilevel Indexed
 - Have a master index that points to secondary indexes.
 - Still an upper bound on file size, lots of seeks
 - . . .but, its simple, small files are easy and cheap, and files can easily expand.

QUESTION: When creating or growing a file, how is a free block found?

ANSWER: Maintain a bit map of free blocks (or sectors) on the disk.

- Once files have been laid out on disk, how do we schedule Disk Accesses?
 - FIFO - server requests in order of arrival
 - SSTF - shortest seek time first. Go to nearest request to current head position (greedy algorithm)
 - SCAN - move arm in one direction, servicing requests until it reaches the other end of the disk, then reverse direction and do the same thing (elevator algorithm)
 - CSCAN - (circular scan) service request only in one direction to the end of the disk. Then return arm to beginning of the disk

Naming and Directories

- Need some kind of file header to maintain information about this file!
- In UNIX, this is called an 'i-node'
- inode contents:
 - mode (permission bits, 10: trwxrwxrwx)
 - owners (user and group)
 - timestamps (access, creation, modification)
 - size of file in bytes
 - block count
 - reference count (number of 'directory links' to file)
 - 12 direct block pointers
 - single indirect block pointer

- double indirect block pointer
- triple indirect block pointer
- Max file size with block size at 4096 bytes (can also be 8 KB, 16 KB, etc...)
 - $12 \times 4096 = 48$ KB in direct blocks
 - Indirect block stores 1024 32-bit block ptrs, so 4 MB in indirect block
 - Double indirect block stores 1024×1024 block pointers, so 4 GB
 - Triple indirect block stores $1024 \times 1024 \times 1024$ block pointers, so 4 TB
 - Total size: 4 TB + 4 GB + 4 MB + 48 KB
- Note that triple-indirect block not really used and file size limited by 32-bit size word, so actually 4 GB in practice
- How do we place blocks of files on disk so that we minimize seek time?
 - Original: place all the i-nodes at the beginning of the disk
 - Data blocks start out being allocated sequentially, but then become scattered around disk as disk fragments

QUESTION: Why is this a really bad idea?

ANSWER: LOTS of seeks!

- Fast FS: FS attempts to allocate blocks for a given file from the same cylinder group, to reduce seeks
 - Bookkeeping information staggered across cylinder group to prevent critical information from being on same platter of disk
 - This introduces a requirement that we need to keep space free in each cylinder!
- Log-based FS: Treat the disk as a 'log', write the i-nodes to the end of the log. (What does this fix?)
 - Now need to maintain index into the i-nodes so we can find them, and hence our files!
- A directory is typically just a file that happens to contain special metadata
 - directory = list of (name of file, file attributes)
 - attributes include such things as size, protection, location on disk, creation time, access time, ...
 - the directory list is usually unordered (effectively random)
 - when you type "ls", the "ls" command sorts the results for you
- Let's say you want to open "/one/two/three." What goes on inside the file system?
 - open directory "/" (well known, can always find)
 - search the directory for "one", get location of "one"
 - open directory "one", search for "two", get location of "two"
 - open directory "two", search for "three", get loc. of "three"
 - open file "three"
 - (of course, permissions are checked at each step)

- FS spends lots of time walking down directory paths
 - o this is why open is separate from read/write (session state)
 - o OS will cache prefix lookups to enhance performance
 - /a/b, /a/bb, /a/bbb all share the “/a” prefix

LRU Approximation Algorithms

- Clock Page replacement algorithm
 - o HW keeps use bit per page frame and sets on every reference
 - o On page fault, advance clock hand, OS checks use bit
 - 1 ==> clear use bit, continue
 - 0 ==> replace page
 - o Keep looping until we get a free page
 - o Note that if clock hand goes all the way around (because all pages marked as 'used'), then on next sweep through these pages will be evicted.

- N Chance Page Replacement algorithm
 - o HW keeps use bit per page frame and sets on every reference
 - o OS keeps counter per page frame - # of sweeps
 - o On page fault, advance clock hand, OS checks use bit
 - 1 ==> clear use bit, clear counter, continue
 - 0 ==> increment counter,
 - if (counter < N) continue
 - else replace page

- Second Chance List
 - o Memory split into two parts
 - mapped - FIFO, marked r/w, directly accessible to program
 - unmapped - Second Chance List, LRU, marked invalid (but in memory)
 - o On page fault, OS checks second chance list
 - if in SCL,
 - move page from SCL to end of FIFO
 - move page from head of FIFO to end of SCL
 - set bits
 - if not in SCL,
 - move page from disk to end of FIFO
 - move page from head of FIFO to end of SCL
 - move page form head of SCL to disk
 - set bits

Quiz