Administrivia

- Midterm?
- NachOS
  - Last minute extension of walkthrough – can hand in by Saturday using dropbox
  - Another help session tomorrow G-7, 5:30pm
- Cross compiler – make sure you have it working; if not, please come and see me
NachOS—An Important Gotcha

- After each system call you have to increment the program counter – otherwise machine will do the same instruction again

```c
DEBUG('q', "Incrementing Registers\n", name);
int pc = machine->ReadRegister(PCReg);
machine->WriteRegister(PrevPCReg, pc);
machine->WriteRegister(PCReg, pc+4);
machine->WriteRegister(NextPCReg, pc+8);
```

- Add this code in the exception handler after you handle the system call
Fineprint for skipping the final

- Problem is there is a lot of important material that will only be tested on the final

- I talked to my lawyers and this is what we came up with:
  - Attend all the remaining classes
  - Score over 75% on the final homework
  - Do NachOS project 3, or if you have collected other bonuses, a proportional part of it (talk to me about details)

- ...or...do the final
Last Time

- Finished Replacement policies + midterm review
- FIFO, LRU, NFU (+aging), NRU, Clock algorithm, Page buffering, working set algorithms
- Hardware/software implications
  - cannot make support needed during memory references complex—it needs to be in hardware
  - can make stuff that happens infrequently (page fault time or context switch time) complex—it should be in software
Virtual Memory for Segmentation

- What are the implications of a segmented virtual memory on
  - Fetching Policy?
  - Replacement Policy?
  - Cleaning Policy?
- Given the cost of compaction, does segmentation make sense?
- End of our memory system coverage....whew!
A couple of loose ends

- Copy-on-write:
  - Remember unix fork()? A new process gets a full copy of parent memory
  - Expensive!
  - copy-on-write: share pages of parent; do a copy only when there is a write to a page

- Memory mapped files:
  - Take advantage of VM to speed up file access
  - File demand paged into memory; swapped out when not being accessed
Today – File System Interface

• Users need “non-volatile” storage for their data that allows flexible access

• The file system interface is perhaps the aspect of the OS most visible to the users

• OS as a user-computer interface and resource manager

• A file is a contiguous logical address space that contains a program or data

• Data: Numeric, character, binary, fields, records, etc..
File System Structure (or Format)

- None – Sequence of words, bytes, etc..

- Simple record structure
  - Lines
  - Fixed length records
  - Variable length records

- Complex structures
  - Formatted document
  - Image
  - Relocatable load file

- Can simulate the last two with the first one using control characters

- Who decides structure? OS or Program?

- File Attributes – what information does the OS need to keep with each file?
File Attributes

- Name: symbolic name given by user or application
- Type: if the system supports more than one type
- Location: where on the device is the file stored
- Size
- Protection information: controls what operations different users are allowed to do on the file
- Time, data, and user identification
- Information about files is kept in the directory structure
  - The directory is similar to a page table for a file system
  - The directory is maintained on disk (why?)
File Operations

- Create: find a space in the file system, and allocate an entry in the directory
- Write
- Read
- Seek: reposition the file pointer in the file
- Delete
- Truncate: keep the file attributes, but delete the contents
- Open: search the directory (on disk) for the file entry and move it to memory (e.g., to an open file table)
- Close: move the entry back to the directory
- Examine/change attributes
- Others?
Type of the File

• Should the OS be made aware of the type/structure of the file?
  – OS can prevent the user from doing unreasonable operations by mistake (e.g., printing a binary executable file, instead of source)
  – Can invoke appropriate operation (view, execute ...)
  – Can support “type-specific” operations

• How to specify/maintain type information?
  – Let OS figure it out (e.g., magic number, parse some of the file...)
  – User specifies, e.g., using extension (.exe .bat .c .o etc...)
Access Methods

- Sequential Access: tape model of a file
  - Information is processed in order
  - Good for batch/streaming data files
  - Difficult to do Random Access

- Direct (or Random) access
  - Can specify the block/offset to read arbitrarily

- Other “logical” organizations can be supported on top of direct access
Logical Organization of the File

• The Pile:
  – Records are stored in the order they are received
  – Records can be of variable size – have to store size
  – Inserting new records inexpensive
  – Retrieval requires exhaustive search

• Sequential File
  – Fixed sized records; keep a key filed with each record
  – Keep records ordered based on key field value
  – Cost of insertion? retrieval?
  – Good for tapes/batch processing; poor for random access
Indexed Files

- Indexed Sequential File:
  - Organization similar to sequential file, but:
    * Add an index to allow quick seeks to desired record
    * Delayed updates: use a log file for new records and, infrequently, re-sort file
  - insertion/retrieval?

- Indexed File:
  - Variable size records
  - Create an index for every field you want to search on
  - Insertion/retrieval?
  - Hashed/Direct file: use hashing instead of index
Directories

- File systems can be extensive – Terabytes, hundreds of thousands of files, etc..
- Need to be able to organize these files
- Two parts to the solution:
  - Divide the file-system into partitions
    * Usually each partition refers to a device (disk) or a portion of a device
  - For each partition, use a directory to maintain information about the files on it
Directory Organization

● How should the directory be organized?

● What functions would we like from a directory
  – Search for a file (by name)
  – Create a file
  – Delete a file
  – List a directory: what are the files in the directory
  – Rename a file
  – Traverse the file system: list the files and directories in the whole file system

● Directory organization should allow these to be done efficiently
Logical Organization of Directory

- From a user interface perspective; directory should be organized logically to allow
  - Efficiency – user should be able to locate files quickly
  - Naming – convenient to users
    * Two users can have the same file name
    * The same file can have several different names
  - Grouping – logical grouping of files
Single-Level Directory

- A directory is just a linear list of files
- Advantage: simple to implement; easy to share files
- Disadvantages?
- How do you create/delete/renam..?
Two-Level Directory

- Idea: allow a separate directory for each user
- Advantages/Disadvantages? Think about user perspective
- How to share files? How to execute “system programs”?
Tree-Structured Directories

• Advantages/Disadvantages?

• Concept of “current directory” or working directory

• Absolute vs. Relative path

• How to share files?
Acyclic-Graph Directories

- Allow the same file or directory to have different names - Why?
  - One actual file, but multiple names/directory entries

- Implementation
  - Soft-links: create a pointer to the file’s original entry
  - Hard-links: duplicate information in both directories
Acyclic-Graph Directories

- Several issues to solve – need to solve differently for soft vs. hard links

- What happens when one of the “names” is deleted? When can the file be actually deallocated?

- What happens if a shared directory is modified?

- How do we prevent the cycles?
  - Allow links only to files?
  - Run a cycle detection algorithm (when?)
  - Allow cycles – complicates things (e.g., garbage collection)
Protection

- File owner/creator should be able to determine
  - What can be done
  - By whom

- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

- Most conservative: no sharing, only owner has permissions

- Most liberal: anyone can do anything
Protection (cont’d)

- Access Lists: For every file, keep a list of “what can be done and by whom”
  - Very flexible, but very expensive as well

- Modern solutions, try to aggregate some of this information

- Unix Solution:
  - Modes of access: read, write, and execute (others?)
  - Three classes of users: 1) owner; 2) group; 3) public
  - 9 protection bits associated with every file (ls -ls):
    - `rwxr-xr-x` 1 nael users ... test*
    - `rw-r--r--` 1 nael users ... test.cc
File System Structure

- Layered Design
  - Upper layers present the logical file interface to the user
  - Middle layers map logical representation to physical representation
  - Lower layers implement physical file system; transferring file from disk to memory and back
Example

- When a file is created (Logical Operation):
  - Application issues a system call to create the file to the logical file system
  - The logical file system reads appropriate directory to memory, updates with new entry and saves it back to disk

- Writing some data to the new file:
  - Searches directory for the file
  - Check permissions
  - Find physical Location
  - Issue physical writes to basic file system
  - Basic file system deals with I/O and device drivers

- Inefficient? On every file access we have to go to the directory (another disk access)

- We are concerned with implementing the logical file system (upper and middle layers)
Mounting and Open File Tables

<table>
<thead>
<tr>
<th>index</th>
<th>File name</th>
<th>permissions</th>
<th>access dates</th>
<th>pointer to disk block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>abc</td>
<td>rw rw rw</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>mail</td>
<td>rw</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Opening a file allows caching of the file entry from the directory to the *open file table*
  - similar to the TLB for memory management
  - Returns a file descriptor (pointer to the open file table) to the program

- A filesystem must be *mounted* before it can be used
  - Mounting provides an attachment of the physical device into the logical file system
  - OS reads the directory for the mounted system and verifies format is correct
  - Provides a logical attachment of the file system (maps it into the “logical directory structure”)
File Allocation

- A disk consists of blocks that are allocated to files as they need them.
- How should blocks on the disk be allocated to files?
  - Problem similar to memory allocation
  - We deal with “fixed partitions” only (the size of the block)
- What are some of the approaches we used for allocating memory frames?
- Is there “virtual file system” like virtual memory?