Introduction 1

• What is an Operating System?
  – The software that runs in protected/kernel/supervisor mode
  – The software that provides an abstraction to the hardware for application programs
  – The software that manages the hardware resources for application programs
  – The software that controls system resources and the processes using these resources
Systems get built in layers; one view of the layers is:

- Applications
- Operating System
- Hardware

The lowest layers provide services that are used by the higher layers. Another view of the layers:

1. Scripts
2. Applications and Utilities
3. System software
4. System Libraries
5. System call interface
6. Operating System kernel
7. Hardware

Higher Levels of Abstraction
OS Concepts

• We will talk about:
  – Processes
  – Threads
  – Mutual Exclusion
  – Process Scheduling
  – Deadlocks
  – Memory Management
  – Disk Scheduling - this is less of an issue than it once was because the hardware has gotten much smarter, but it points up very important concepts
  – File systems
  – I/O
  – Security
System Calls 1

• A system call is how a non operating system process communicates with and asks for services from the operating system

• Recall that the operating system runs in a privileged or protected mode, so the system call crosses the protection boundary

• How a system call works:
  – put the parameters in a specific place (including the number of the system call)
  – issue a trap to the OS
  – OS examines the parameters to decide which system call is being invoked - OS performs the work
  – OS puts the return code in a specific place and returns from the trap
System Calls 2

- The scheduler may not schedule the process that invoked the system call next
- System calls can *block* - for example, a disk read may complete some time later - other processes would be run while the original process was blocked
- Examples of system calls include: read, write, open, fork, etc.
- System calls are typically quite expensive compared to ordinary function calls
Processes

• A program in execution; a program and its state
• Process Creation
  – When; How
• Process Termination
  – When; How
• Process Hierarchies
  – Control; Inheritance
• Process States
  – Running; Blocked; Ready
  – Transition Diagram
Processes 1

• A process is a program in execution. The process includes all of the state necessary for the OS to execute, stop, and restart the program. This includes: text, data, registers, interrupt state, signals, etc.
• Generally, every process will have the illusion that it is running alone to completion
• Multiprogramming OS give the illusion that the programs execute alone. In fact, the OS is switching back and forth between processes, this can also be called *pseudo parallelism*
Processes 2

What the processes see - the illusion that they run to completion
Processes 3

Processes are sequential, and the operating system is running them all in turn.

A — system call
B — time expired
C — i/o operation
D — sleep
C — i/o operation
B — i/o operation
A — i/o operation
B etc.

Time
Processes 4

Processes can be organized into hierarchies - in Unix, every process has a parent, and every process is a descendant of init.

Processes can be thought of as having states:
- executing in user mode
- executing in kernel mode
- not executing but ready to execute
- sleeping while it waits for some resource (blocked)

Unix Process State Diagram
Processes 5

Tannenbaum’s process state diagram is slightly different:

1. block on i/o
2. get interrupted by timeout
3. get rescheduled
4. i/o becomes available

Note, there is this new blocked state. We get there by trying to perform i/o that isn’t ready. This is captured in the previous diagram by the OS putting the process to sleep

The scheduler causes transitions 2 & 3. Fair scheduling becomes important here.

A process can be interrupted by the scheduler, or it can interrupt itself.
Processes 6

Implementation

Process Table - has an entry per process - data structure that allows the OS to control a process

Some things found in a process table entry (PTE) registers, stack pointer, CPU time used, process id, parent process id, pointer to data segment, pointer to text segment, pointer to stack segment, current directory, open files, signal masks, etc

We find all of these things in the task structure in Linux.
Look at:
<linux/sched.h> -- task lives here
<asm/processor.h> -- thread struct lives here
Interrupts

Interrupt Vector contains the location of the interrupt handler for each device

When an interrupt occurs, the hardware saves the current processes state and then jumps to the location contained in the interrupt vector
- save the registers and create a new stack (some OS run on the process stack - this makes the default case easier, but error handling gets much harder)
- interrupt routine handles the interrupt
- potentially, we may now have multiple runnable processes, so we return to the scheduler
- scheduler decides which process to run
Threads 1

- Processes have a thread of execution
- Traditionally, one thread of execution per process
- Threads = multiple threads of execution in a process

<table>
<thead>
<tr>
<th>Per Process</th>
<th>Per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>address space</td>
<td>program counter</td>
</tr>
<tr>
<td>global variables</td>
<td>registers</td>
</tr>
<tr>
<td>open files</td>
<td>stack</td>
</tr>
<tr>
<td>child processes</td>
<td>state</td>
</tr>
<tr>
<td>signals</td>
<td></td>
</tr>
<tr>
<td>permissions</td>
<td></td>
</tr>
</tbody>
</table>
Threads 2

- Threads have the same basic states as processes: running, ready, blocked
Threads 3

• Why use threads
  – threads are cheaper to create than processes
  – compute servers; lower variance
  – threads share address space

• Implementation
  – User threads vs. Kernel threads

• User Threads
  – processes each have a thread table
  – processes create stacks for each thread
  – processes save registers

• Kernel Threads
  – kernel has a thread table
Thread Issues

• Blocking
  – user threads one blocks all block

• Scheduling
  – kernel can be cleverer about scheduling; user more flexible

• Scaling Issues
  – with many many threads, user threads scale better

• Control and Implementation
  – user threads; must yield control or have a timer
  – many system calls aren’t thread safe or reentrant
  – signals
  – what does fork do about creating multiple threads
Interprocess Communications

- **Race Conditions** occur when two processes can interact, and the outcome can depend on the order.
  - Imagine Process 1 and Process 2 have a shared variable that is called $x$ in process 1 and $y$ in process 2.
  - Process 1 executes $x = x + 1$
  - Process 2 executes $y = y - 1$ at the same time
  - If the variable contained 5 to begin with, it can contain any of 4, 5, and 6 depending on the order in which things executed
- The operations $x = x + 1$ and $y = y - 1$ are not atomic with respect to each other i.e. either could be interrupted at any time.
- The results are unpredictable - this is bad
Critical Regions

• The part of a program where a shared resource is accessed is called a *critical region*

• Conditions for a solution
  – No two processes in their critical regions at once
  – No assumptions about speed or number of processors
  – No process outside its critical region can block another
  – No process waits forever to enter its critical region

• One solution is to disable interrupts before entering a critical region

• Lock variable solution
Test and Set

while (TAS(lock)) ;  // busy wait on the lock
critical_section();
lock = FALSE;

Does this satisfy our requirements for mutual exclusion?
Semaphores 1

assume we have two operations $P$ and $V$ such that

$P$(semaphore)
critical section
$V$(semaphore)
non-critical section

satisfies the constraints we set out before

A **binary semaphore** abstracts the TAS or test and set constraint we had before
Semaphores 2

A binary semaphore has two states assume we will call them *open* and *closed*

then if we can make the following atomic, we win

P(semaphore) {
    while(semaphore==CLOSED);
    semaphore = CLOSED;
}

V(semaphore) {
    semaphore = OPEN;
}
We can extend binary semaphores to what are called: \textbf{counting semaphores}. Counting semaphores take on non-negative integer values:

\begin{verbatim}
P(semaphore) {
   while (semaphore == 0);
   semaphore--;
}

\end{verbatim}

\begin{verbatim}
V(semaphore) {
   semaphore++;
}
\end{verbatim}

The body of P is atomic.

Counting semaphores turn out to be surprisingly useful...
Scheduling 1

• Scheduling the processor or scheduling which job should run next

• Schedulers should try to meet the following objectives
  – Fairness - each process gets a fair amount of the processor
  – Efficiency - maximize the amount of usage the CPU gets
  – Response Time - minimize the response times for interactive users
  – Turnaround - minimize time it takes to finish processes
  – Throughput - maximize the number of processes run per unit time
  – Degrade gracefully and predictably under high load conditions
    • in other words, we need to be able to predict how much hardware we need - interestingly enough, Linux is much worse than NT for predictability under high load conditions
  – Repeatable
  – Doesn’t allow cheating

• These are not all achievable at the same time
Scheduling 2

• Preemptive scheduling is when the processor can interrupt one process to schedule another

• Non-preemptive scheduling is what it sounds like

• Scheduling algorithms we will talk about
  – FCFS
  – Round Robin
  – Round Robin with penalties
  – Processor Sharing
  – Shortest Job First
  – Preemptive Shortest Job First
  – Priority Scheduling
  – Priority with multiple queues
  – Other multi queue schemes
  – Various Real-Time scheduling algorithms
FCFS - First Come First Served

• Also called FIFO (first in first out)
• Keep a queue of processes, run the one on the queue until it is done, and then run the next one
• Non preemptive
• The simplest scheduling policy - early systems used this
Round Robin

• Preemptive scheduling policy
• Important question is how long each process gets to run for this is called the quantum (q)
• How it works
  – a running process gets q time units to run.
  – when the timer goes off, the process gets preempted
    • the process gets moved to the ready queue
    • the next process in the ready queue gets to run
  – as q gets large, the Round Robin looks like FCFS
  – as q gets very small, we get processor sharing
• There are many issues involved in choosing how big or small to make q
  – small q makes the system responsive but wastes resources
  – large q makes the system more efficient but less responsive
Shortest Job First

- Given a list of processes (actually jobs) run the one first that will take the least time
- Non preemptive
- With no i/o switching, SJF has the shortest average waiting time
- Of course, predicting the time the job will take to run can be hard...
Preemptive SJF

• When a new job enters the run queue, if it has a shorter time than the remaining time of the job on the run queue, then we run the new job
• Obviously, we can have starvation with this
• One modification is to allow the jobs, as they wait, to decrease their predicted run times. This modification removes the possibility of starvation. This modification of the priority of the job as it waits is called aging
Priority Scheduling

• Assign priorities to processes based on some scheme
• We have a list of processes, and we find the process with the highest priority and run it
• We can penalize or help processes based on things like
  – has been waiting a long time (aging)
  – has been using lots of CPU time
  – is interactive (or not interactive)
• This is a very popular scheduling algorithm
• The risk is that the computation of the priority may get expensive, and the system may spend a great deal of time computing which the *best* process to run next is
Two Level Scheduling

• scheduling when the process may or may not be in memory
• requires a way for the scheduler to know whether the process has been swapped out
• to avoid starvation, the scheduler must have a way to have the process brought back into memory
• Tannenbaum assumes that a higher level scheduler decides when to bring a process back into memory
• We will deal with these issues when we talk about memory management
Scheduling Issues

- Threads
- Very large numbers of processes
- Degrading gracefully
- Scheduling very long running processes
- Scheduling and the swapper interfering
- Scheduling jobs and interactive processes at the same time
- Queuing theory to characterize the behavior of different scheduling algorithms
Thread Scheduling

• User level threads have no effect on OS scheduling because the kernel doesn’t know about them

• Kernel threads allow an interesting possibility which is that the kernel can take into account that scheduling the threads from the same process consecutively can save context switch costs
Deadlocks

• A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

• Deadlocks are caused by processes trying to acquire resources defined as either devices, a piece of information, or a lock

• Resources are:
  – preemptible if the resource can be taken away
  – non-preemptible otherwise
Conditions for Deadlock

• 4 conditions must be present to have a deadlock
  – Mutual Exclusion - each resource is free or owned by 1 process
  – Hold & Wait - processes holding resources can request more
  – No preemption - resources can’t be taken away once granted
  – Circular Wait - there must be a circular chain of at least 2 processes waiting on the next member of the chain
Modeling Deadlocks

• Think of a directed graph where

A wants resource R  A holds resource R

B holds R and wants S; A holds S and wants R; Deadlock
Approaches to Deadlock

• Ignore it
• Detect it and then recover
• Dynamically avoid it by clever resource allocation
• Prevent it by structurally avoiding one of the 4 conditions
Ignoring Deadlock

• This is what most real world OS do
  – The cost of avoiding, preventing, detecting or recovering from it is really high
  – The price paid of simply having things get killed is not that high
  – In real world systems, the frequency is rare and applications can often avoid it themselves
Detecting Deadlock

• Database systems typically try to detect and recover from deadlock
• With a single instance of each resource, we simply do cycle detection in a graph
• If we have multiple instances of some resources, we can be slightly trickier
Recovery

• Preemption
  – take some resource away from a process
  – usually impractical

• Rollback
  – put the process back in some known good state
    where it doesn’t have the resource
  – usually impractical

• Kill! Kill! Kill!
  – kill some process holding resources
  – ugly but effective; most common strategy
Deadlock Avoidance

• Assume we know the maximum number of resources a process might need

• Key observation concerns safe vs unsafe states

• A state is safe if there is some scheduling order of the processes that allows them to complete
Avoidance Example

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Free</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Free</td>
<td>5</td>
<td></td>
<td></td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Run</td>
<td>B</td>
<td></td>
<td>Run</td>
<td>C</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td>A</td>
<td>can run</td>
</tr>
</tbody>
</table>

This is safe

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Free</td>
<td>2</td>
<td></td>
<td>Free</td>
<td>0</td>
<td></td>
<td>Free</td>
</tr>
<tr>
<td>Run</td>
<td>B</td>
<td></td>
<td>Run</td>
<td>C</td>
<td></td>
<td>Can’t</td>
</tr>
</tbody>
</table>

If A has one more, this is unsafe
Bankers Algorithm

• Very famous solution by Dijkstra
• When each resource request comes in, check to see whether it leads to a safe state - if not, don’t grant it
• No one actually uses this algorithm because in reality, the maximum number of resources needed isn’t usually known at run time
Deadlock Prevention

• If we can avoid one of the four conditions we can avoid deadlocks
  – Mutual Exclusion
  – Hold and Wait
  – No Preemption
  – Circular Wait
Mutual Exclusion

- One approach is to have any resource have one owner (spooling with a printer)
- Not applicable to all resources
- Not generally practical
Hold and Wait

- Make the process request all resources at start time (MVS does this in some cases)
- Make the process give up all resources when it wants to request a new one -- request everything at once
No Preemption

• Take resources away from a process
• Since not all devices are preemptable, this doesn’t work well
Circular Wait

• Order resources numerically
• Force a process to request resources in numerical order
• Alternatively, don’t allow a process to request a resource lower than it currently holds
• Hard to order everything
Memory Management 1

• Storage Hierarchy in Modern Machines
  – Registers
  – On Chip Cache
  – Off Chip Cache
  – Main Memory
  – Device Memory*
  – Disk
  – Tape
  – Network Devices*

Performance Goes Down
Size Goes Up
Memory Management 2

• We need to move data around in this hierarchy, and we need to be clever about when we move the data
  – Fetch on demand (for example, demand paging)
  – Pre-fetch
    • Pre-fetch of instructions into cache
    • Pre-fetch of data from disk (read ahead)
    • Pre-fetch pages of running processes based on some algorithm
  – Different levels of the hierarchy may require different algorithms to decide whether and what to pre-fetch
    • text vs data
    • kind of device
Address Translation 1

- Convert virtual addresses to physical addresses
  - also called logical to real
- The virtual address is the address in the program
  - `int * x; // x will contain a virtual address`
- The physical address is the address in memory
  - the storage is located somewhere in real memory
- In modern machines, the mapping of virtual to physical is usually done by an memory management unit (MMU)
- In simple machines, the virtual and physical may be the same...
Address Translation 2

• When the translation gets done is of interest
  – At compile time (DOS .com files)
    • compiler generates physical addresses in the executable
    • addresses are fixed for these programs
  – At link time
    • compiler generates relocatable objects
    • objects have external references
    • linker converts the relocatable objects to absolute addresses and also resolves external references
    • old Unix ld acted this way, DOS .exe files look like this, as well
  – At load time looks like link time
Address Translation 3

– At run time
  • addresses are dynamically resolved while the program is running
  • requires hardware support
  • this is the way virtually all modern machines work
Monoprogramming 1

- No swapping or paging
Monoprogramming 2

• No Address Translation done by the OS
• Reload the OS for each job or protect the OS from the job
• If the program needs more memory than we have, use overlays
  – Program is broken into pieces by the programmer
  – A piece of the program is always in memory. This piece calls the OS to load and unload the overlays
  – The programmer has to ensure that an overlay is loaded before it is called (some systems did this automatically)
  – The programmer is responsible for making things fit
  – Very popular in the 60s and in early DOS days
Multiprogramming 1

• The goal is to increase CPU utilization by allowing overlap between CPU and I/O
• If a job waits for I/O, in a monoprogramming environment, the CPU is idle
• If P is the percentage of time spent waiting for I/O, then CPU utilization is 1-P
• Since P is often > 0.5, CPU utilization in a monoprogramming system is bad
Multiprogramming 2

• With a multiprogramming level of N, the CPU utilization is approximately \(1-(P^N)\)

• The idea is that the probability that all of the jobs are waiting for I/O at the same time is low. The probability that all are waiting that the same time is approximated as \(P^N\)

• This is a simple model, but it captures the notion that increasing the multiprogramming level increases CPU utilization

• The limiting factor on the multiprogramming level is the amount of memory
Multiprogramming with Fixed Partitions 1

- Memory divided into $N$ partitions at boot time
- $N$ can be chosen by the operator
- When a job arrives, it is put into the queue for the smallest partition it will fit into
- Any leftover space in a partition is wasted (fragmentation)
- Partition sizes are fixed until reboot
Multiprogramming with Fixed Partitions 2

• We can run into problems if the queue for the small partition is full, and the queue for the large partition is empty

• One solution is to have a single queue
  – Possibly very wasteful if we allow small jobs into big partitions
  – If we don’t, then we run the risk of starving the small jobs

• This was used in IBM OS/MFT
  (Multiprogramming with a Fixed number of Tasks)
Relocation

• We need to decide what base address the program was loaded into (usually with a base register)
  – If the program has a jmp 100, we need to do a jmp 100 + base (think about how the partitions work)
  – The base register lets us do this. Every address in the program will have the base register added to it
  – With base registers, running programs can be moved

• Another solution is to simply modify every address in the program when it is loaded
  – Requires linker support (the linker must make all of the addresses visible in some way
  – OS/MFT did this
Multiprogramming with Variable Partitions2
Multiprogramming with Variable Partitions

- If the jobs are relocatable while running, then we have much more flexibility about moving jobs etc.
- Base registers are one way we get this flexibility
- Dynamic address translations allow us this flexibility
- Number of processes vs. number of holes
  - holes coalesce, processes don’t
  - on average, we expect about twice as many processes as holes
- Which partition to use...
Choosing a Partition 1

- Given that we want to find some space for a process either to be created or to be swapped in, how do we choose?
  - Best Fit - search through all of the holes, and choose the one that is closest (but bigger) **Slow**
  - First Fit - find the first hole large enough and use that **Leaves slivers, Uses big holes**
  - Worst Fit - find the biggest hole **Difficult to place big processes after a while**
  - Quick Fit - keep a list of common sizes (2K, 4K, etc) and use **Finding a neighbor to merge is expensive**
Partition Choice Implementation

• Linked List
  – Keep a doubly list of allocated and free memory (sorted by address)
  – Each item is either a Hole or a Process and has the starting location and the length
  – When a process exits, the entry is found and can be coalesced with its neighbors

• Bit Map
  – Keep a bit map of memory
  – If each bit represents a big amount of memory, we have fragmentation
  – If each is small, we have lots of bits to store and handle
Virtual Memory 1

- Divide the memory of the program into fixed size pieces called **pages**
- Divide the memory of the machine (the real or physical memory) into fixed size pieces called **page frames**
  - The size of the pages and the size of the page frames is **always** the same
- Keep a mapping of pages to page frames. This mapping, called the **page table**. The entries in this table show us, for a page p1, which frame contains p1
Virtual Memory 2

Assume we have 4K pages, and 16K of actual, physical, memory.
Page Faults 1

• As we saw in the previous picture, some pages have frames associated with them at any given time, and some don’t

• When a process tries to access a page that does not have a frame associated with it, a page fault is generated
  – The process blocks
  – The OS picks a frame that contains something that won’t be used again for a while and it (possibly) writes the contents of that frame to backing store (that virtual page has been paged out)
  – The OS gets the contents of the virtual page we want from the backing store and puts it in the page frame (the location will be in the page table entry
Page Faults 2

- How the OS chooses the page frame will be a topic of much discussion...
- Much cleverness can be expended on picking the pages to page out
- Much cleverness can be expended on picking which pages should be brought into memory
- Page faults are expensive, and all of this works because programs usually have a working set and locality of reference
- Working set - the pages that are accessed in a given time window tend to be drawn from a fairly small set because the addresses used by the program are not random
Page Tables

• Assume a machine with 32 bit addresses, and 4k pages, the machine has 1 million pages, so the page table has 1 million entries
  – Since every process has its own page table, this is going to get expensive
  – Looking things up in a table with 1 million entries is expensive, too
• Every memory reference turns into an address translation
  – Many instructions access multiple memory locations, so huge numbers of accesses to the page table happen
• The page table has the potential to be a giant bottleneck
Page Table Implementations

• Page Table implementations are governed by: big, cheap, fast pick two

• We will discuss:
  – Where the page table is kept
    • all in main memory
    • in special hardware
  – Multi level page tables
    • avoid many of the flaws of single page tables
    • added complexity
  – Inverted page tables
  – Translation Lookaside Buffer - a cache of page to frame mappings
1 Page Table in Memory

• Recall our picture of a processes memory
  – most of the pages in the processes address space are empty (the stack and heap are growing towards them)
• If the page table has entries for all of these pages (which it has to), then it will be big
• Having it in memory will be slow
• A single page table in memory is really only practical if the address spaces are small
  – PDP 11

<table>
<thead>
<tr>
<th>Page Table Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

5/6/2002 Moran OS Lecture 13
Multi Level Page Tables 1

- Add a level of indirection, so we have a page table that rather than pointing to page frames points to other pages

- Assume we have one page table
  - break it into page sized pieces (a single page table entry is fairly small, so we get a reasonable number in a page)
  - each of these page size pieces is a 2nd level page table
  - the first level page table will point to these 2nd level page tables
  - the first level page table will be in memory, the 2nd level tables will be paged in as needed
Multi Level Page Tables 2

Stack
↓
↓
Heap
Data
Text
Multi Level Page Tables 3

• One way to structure the virtual addresses is as 3 parts
  – pt1 pt2 offset
  – assume there are 1024 entries in every page table
  – then we need 10 bits for pt1 and pt2, and the offset will be 12 bits (on a 32 bit machine)
  – PT1 is the index in page table 1 PT2 is the index in page table 2 Offset is the offset in the page

<table>
<thead>
<tr>
<th>PT1</th>
<th>PT2</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
Multi Level Paging 4

- We can obviously extend the two level scheme to three or more levels
  - on a 32 bit machine anything beyond three levels is probably overkill
- The VAX used two level paging
- The SPARC uses three level paging
Structure of Page Table Entries

• What does the page table contain? (on some Unix)
  – The physical address of the page (the frame)
  – Protection bits telling whether the process can read/write/execute the page
  – Bits for the following
    • Valid - says the contents of the page are valid
    • Reference - tells whether a process recently accessed the page
    • Modify - tells whether a process recently modified the page
    • Copy on write - do we create a new copy when the page is written
    • Age - how long the page has been in the processes working set
Page Replacement Algorithms 1

• Given a page fault, how do we decide which page we will sacrifice to bring a new one in?

• Programs have a working set and locality of reference
  – If we access an address at time t0, we are more likely to access that same address again in the near future
    • This tells us we should keep things around
  – If we access an address at time t0, we are more likely to access addresses near it in the near future
    • This tells us we do well to bring in more than a single word
  – Normal programs don’t simply access random addresses
Page Replacement Algorithms 3

• Algorithms we will look at
  – Random PRA
  – Optimal PRA
  – Not Recently Used (NRU) PRA
  – FIFO PRA
  – Second Chance PRA
  – Clock PRA
  – Least Recently Used (LRU) PRA
  – Not Frequently Used (NFU) PRA
Random PRA

• Pick a page to page out at random
  – Performance is lousy
  – No locality of reference
  – No working set
  – Easy to implement
  – Any other page replacement algorithm should work better than this for a normal program
Optimal PRA

• Replace the page whose next reference will be farthest in the future
  – Imagine that we know the exact sequence of pages the process will access, then we choose the page that it will be the longest time before we need it again
  – Has the best possible performance (provably)
  – Impossible to implement because we don’t know, in advance, what pages a process will access
FIFO PRA

• Pick the page that has been around the longest, and page it out
  – Not a very good algorithm
  – Bad locality of reference
  – Bad working set behavior
  – Easy to implement
2nd Chance PRA

- In FIFO, we would keep a queue of the pages, and we’d remove (page out) the first one in the queue.
- In 2nd Chance, if the first one in the queue has the read bit clear (it hasn’t been read recently) we page it out.
- If it does have the read bit set, we move it to the back of the queue and clear the read bit
  - we walk down the queue until we find a page with the R bit set.
- Fairly good performance
- Somewhat expensive to implement
Clock PRA 1

- 2nd Chance is somewhat inefficient because we are moving things in the list
- Keep the pages in a circular list, we imagine a clock hand sweeping through the pages
  - If the clock hand hits a page with the read access set, we turn the read access off
  - If the clock hand hits a page with the read access off, we page it out
- One modification is to add a 2nd hand, the second hand does the paging out, and the first hand turns the read access off
Least Recently Used PRA

• Approximation of the Optimal PRA
• Assumptions
  – pages used in the last few instructions will be used again in the next few instructions
  – pages not used for a long time probably won’t be used again soon
• Pick the page that has gone the longest without being used
• Difficult to implement (without very special hardware)
• Excellent behavior
Not Frequently Used PRA 1

• Since LRU is difficult to implement, NFU is an approximation

• How it works
  – We associate a counter with each page
  – At each clock interrupt, we increment the counter of every page that has the reference bit set (clearing the reference bit in the process)
  – When we have a page fault, we remove the page with the lowest count

• Doesn’t work too well since it remembers everything
Not Frequently Used PRA 2

• A straight-forward modification is to shift the counter (assume 8 bits) to the right and add on the left
• This has the advantage that the counts get “aged”
• Fairly easy to implement
• Good performance
• Excellent working set behavior
Modeling Paging Algorithms

• Intuition says that more page frames should decrease the number of page faults

• Counter example, for FIFO, is called Belady’s Anomaly
  – 0 1 2 3 0 1 4 0 1 2 3 4
  – three page frames gives 9 page faults
  – four page frames gives 10 page faults
Design Issues for Paging

• A PRA is local if the victim frame is chosen from the frames the process already has

• A PRA is global if the victim frame is chosen from the frames of all processes

• No PRA is truly local, since at startup, a process has no frames, and needs to get some

• Global algorithms run the risk of thrashing (getting nothing done because the system is only doing paging)
  – Global LRU and RR scheduling
  – By the time a process runs all its pages have gone
Local vs Global PRA

• For global algorithms, we need to strike a balance between giving each process an equal number of pages and allowing one process to use all of the pages.
• If we give each process an equal number of pages, we have the possibility that some processes have many more than their working set and some have many fewer.
• One way to decide is to give processes with many page faults more pages; we can swap out processes that cannot be satisfied.
Page Size

• Must be disk block size multiple

• Large Pages
  – Good for User I/O. I/O less likely to cross page boundaries
  – Good for Demand Paging
    • One big page paged in rather than several small ones
    • However we run the risk that the page contains info that isn’t really local to each other
  – Large internal fragmentation
  – Smaller page table

• Small Pages have the opposite properties
Implementation Issues 2

• Page Faults (what really happens)
  – Hardware traps to the kernel (save some state)
  – Save state and call the OS (assembly language)
  – If we know which page is needed fine, otherwise, we need to figure it out from the instruction being executed
  – Check whether address is valid, if not kill process
  – Find a frame, if it is dirty schedule a write. Block the process. Also, keep anyone else from using the frame.
  – When the frame is clean, schedule a read to populate the frame. (The process is still blocked)
  – When the read has completed, restart the instruction
  – Put the process in the ready state
  – When the process is ready to run again, asm routine runs to restore state
Pages for Processes 1

• How to decide how many page frames a process gets
  – we don’t want to thrash

• Basic answer is that we want to give every process the number of pages in its working set

• Page Fault Frequency
  – Keep track of the number of faults compared to the number of references (using a time decay)
  – If the process has too high a rate, give it more frames
  – If not all processes can be satisfied, reduce the level of multi-programming by swapping some out
Working Set PRA

• Basic idea is to keep track of the working set and fault out pages not in the working set
• Keep track, for each PTE virtual time of last use and a referenced bit
• Scan all pages, any page with R=1 gets the last use updated to now
• Any page with R=0, compare last use time to now if the difference is “big enough” grab that frame
• If we get no pages doing this, and there were any pages with R=0, fault out the one used longest ago
• Otherwise if none are R=0 fault one out at random
WSClock

• Arrange the PTEs as in Clock with a reference bit, a modify bit, and a last use time (like working set)

• Scan like clock, if $R=0$ and it is “old” then
  – if it is not modified grab it
  – if it is modified, schedule a write and continue

• If we get back to the start
  – if no writes have been scheduled, grab a page that is clean
  – otherwise, keep going until we get to a clean page
Segmentation

• We have been assuming that the virtual address space is contiguous
  – This doesn’t work very well when a process has two or more dynamically growing regions
  – With two regions, we can start them on opposite sides of the virtual address space (stack and heap)

• Alternatively, we could have a number of virtual address spaces that start at address 0
  – Called segments
  – Will be user visible
  – Makes sharing and protection somewhat easier
## Segmentation vs. Paging

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Demand Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer Aware</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td># of Addr Spaces</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>VA size &gt; PA size</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Protection</td>
<td>Pages</td>
<td>Procedures</td>
</tr>
<tr>
<td>Accommodate elems</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Changing size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease Sharing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal Fragmentation</td>
<td>Yes</td>
<td>No, in theory</td>
</tr>
<tr>
<td>External Fragmentation</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Placement Question</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Replacement Question</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
General Segmentation

- Allows fine grained sharing and protection
- Segments can be variable sized
- Addresses become Segment,Offset
- Memory layout
  - All segments in memory to run a program
  - Demand segmentation
  - Combine with demand paging
File Systems

• Requirements for Data Storage
  – Size - we need to store very large amounts of data
  – Persistence - the data needs to stay around after the creating process terminates
  – Access - multiple processes can access the data at the same time

• Store data in files

• The part of the OS that manages the files is called the file system
File Names 1

• How does a user or program refer to and find the collection of data (the file)?
  – The file system provides a way to name the files

• Not every file will have a unique name

• Often, the file name will consist of two parts: of the form XXXXXXXX.YYY where the .YYY part is called the file extension

• The file extension is usually used to tell what kind of file it is
  – .com or .exe on a DOS or Windows box
File Types

- Even if the OS thinks a file is just a byte stream, there may be types of files
- Directories - maintain the structure of the file system
- Special Files - used to name devices
- Links - used to give multiple names to a file
- Strong Typing
  - can, for example, prevent files from being used or renamed to a different type
  - not used anymore
File Use Without Types

- On a Unix box, what does ./foo do
  - first two (or several) bytes in the file that tell the OS what kind of executable file it is
  - examine the first two characters, and if they are ‘#!’ then execute the rest as a command interpreter
  - if they aren’t, lookup in a table to find what kind of executable format the file has:
    - COFF
    - a.out
File Attributes

- Protection
- Password
- Creator
- Owner
- Read Only Flag
- Hidden Flag
- System Flag
- Archive Flag
- Text Flag
- Random Access Flag
- Temporary Flag

- Lock Flags
- Record Length
- Key Position
- Key Length
- Creation Time
- Access Time
- Change Time
- Current Size
- Max Size
- Audit Flags
File Operations 1

- Create (creat in Unix) How we create a new file (may be subsumed in open)
- Delete (unlink in Unix) How we make the name go away - may or may not reclaim the storage
- Open How we map the name to internal OS structures (saves having to do this on every access)
- Close How we remove the map of the name to the internal structures (may flush buffers to disk)
- Read How we get data from the file into memory
File Operations 2

• Write How we take data from memory and put it in the file (usually writes to the current location, it may truncate)

• Seek Make the current location different (for subsequent file operations)

• Get Attributes (fstat in Unix) How we check the values of various attributes

• Set Attributes

• Rename How we give the file a new name (Tanenbaum is wrong about copy/delete)
Directories 1

• A directory contains some number of entries
  – one per file name (possibly multiple names for the same file)
  – the primary way of structuring the storage of files

• Depending on the system, there may be one or more directories
  – some have one directory per user
  – some have one directory for the whole system
  – modern systems tend to have many directories

• With many directories, directories are typically entries in another directory
Directories 2

- Hierarchical Directories
  - We have a way of finding a “path” to a directory
  - The path consists of a sequence of directories starting at some “root”
  - There is a path separator
    - On DOS it is ‘\’
    - On Unix it is ‘/’
    - On Multics it was ‘>’
    - It can really be anything that *cannot* appear in a legal file or directory name
  - A path is of the form /a/b/c/d
  - Possible different roots for network file systems
Naming Directories and Contents

• We want to be able to name every directory
• We want to be able to name the contents of every directory
• Example: foo is a directory; bar is a directory in foo; baz.txt is a file in bar
  – Unix has foo/bar/baz.txt
  – DOS has foo\bar\baz.txt
  – Other systems have used >,;,[, etc.
• The important point is that the separator allows us to distinguish the components
Absolute vs Relative Paths

• Processes have a notion of a current directory
  – where the process currently is in the file system

• File can be named with respect to the current directories

• Absolute path names allow is to find the file from anywhere

• Relative paths require that we start at the appropriate current directory

• Unix has a notion of ‘.’ and ‘..’ to support the idea of relative paths
Directory Operations 1

• Create
  – create an empty directory (‘.’ & ‘..’ are there)

• Delete
  – delete an empty directory

• Opendir
  – as with files, this creates the internal structures necessary to access the directory

• Closedir
  – as with files
Directory Operations 2

• Readdir
  – read the contents of a directory. People used to use
    read for this, but that exposes the directory structure,
    and that is thought to be very bad thing

• Rename
  – as with files

• Link
  – Given an entry in a directory, add a new name for it

• Unlink
  – Remove an entry in the directory
Implementing Files

• Disk Hardware
  – Disks operate on sectors; they don’t operate on single words
  – Disk blocks (the unit seen by the file system) are usually multiple sectors

• Allocating the blocks in a file
  – Contiguous Allocation
  – Linked Allocation
  – File Allocation Table (FAT)
  – Inodes
Contiguous Allocation

- At creation time, all of the space for the file is allocated
- If the file were ever resized, it would be necessary to move the whole thing (like Multiprogramming with fixed partitions)
- Simple and fast for access
- Not useful for general purpose systems, but it works well for read only and other systems where file size is known at creation time
- Used by some older mainframe file systems
Linked Allocation

- Directory entry contains a link to the first block in the file
- Each block contains a pointer to the next block
- Has a number of serious drawbacks
  - Random access is horrendous
  - Blocks have lost some space, and are not power of two anymore
- No real systems do things this way
File Allocation Table

• Directory entry points to the first block in the file
• A table is maintained in memory that, for block N contains the next block in the file (for the same file)
• This is a linked implementation, but the links
  – Are stored separately from the blocks
  – Are kept in memory so random access isn’t as bad
• The table requires one word per disk block
  – 4k blocks 4 byte words means 1M table per 1G disk
  – Requires that disk blocks are fairly large
• DOS and Windows do it this way
Inodes

- Directories point to index nodes (Inodes)
- Inodes have several kinds of pointers
  - direct blocks
  - indirect blocks
  - double indirect blocks
  - triple indirect blocks
- Inodes are kept in memory for open files
  - Many references take one I/O
  - Some take two
  - Very few take three
- Unix did things this way
Inode Contents

- File Owner (uid)
- File Type (regular, directory, special, FIFO)
- File Permissions
- File Access Times (access, modify, inode modify)
- Number of links to the file
- Table of contents of the disk blocks
- File Size
Inode Pointers

- 10 direct blocks
- 1 single indirect
- 1 double indirect
- 1 triple indirect
- With 1K disk blocks
  - 10K in the directs (1k x 10)
  - 256K in the single indirect (256 x 1k)
  - 64M in the double indirect (256 x 256 x 1k)
  - 16G in the triple indirect (256 x 256 x 256 x 1k)
Implementing Directories

- Directory contains a name and a pointer to the inode
- File meta-data (owner, etc) is in the inode
- Directories are just files (with internal structure), so subdirectories are simply name/inode pairs in the directory
- For long filenames, some differences of where the file name is actually stored
- How we resolve names
Links to Files

- Tannenbaum calls these “shared files”
- Idea is to give a file multiple names
- Unix supports two ways of doing this
  - hard links
  - soft or symbolic links (Windows has these too)
- Hard links create a new directory entry for an existing inode
- Symbolic links are files that contain the path of another file
Managing Disk Space

- Disks are divided into blocks

- Block sizes are a function of hardware
  - Disks have high latency, so very large block sizes make sense
  - Most files are fairly small (< 5K), so very large block sizes result in large fragmentation
  - Almost any block size you can imagine has been tried in the pursuit of performance (time and space)
  - Current typical disk block sizes are in the 1K-8K range
Disk Block Layout

• We would like to lay blocks out in such a way as to maximize performance
  – Blocks accessed together could be close together
  – Huge amounts of effort were devoted to this at one time
  – Hardware has tended to reduce the need for this kind of cleverness as disk cache sizes have gotten larger
  – Clever layout schemes tend to make allocating blocks somewhat expensive
Free Space Management

• At any moment, we have some free and some allocated blocks on the disk

• Ways we can keep track of free blocks
  – In memory bitmap with one bit per free block
    • small block sizes make this get big
  – Paged bitmap with one bit per free block
  – Linked list with each block pointing to the next
    • requires extra I/O to handle fixing pointers
  – Linked list with all of the links stored in the same place
    • looks like an array of links to free blocks
    • store the array in free blocks and keep the first one in memory
File System Buffer Cache

• For performance reasons, the OS will keep some blocks in memory
  – reads will simply use the in memory copy
  – writes update the in memory copy, and the OS will decide when to write back to disk

• On a read, we scan the cache to see if the block is there. If not, bring it in and read from the cache

• Write through means the OS updates the disk block at the same time (slower and safer)

• Write back means the OS updates the disk block at a later time (faster and riskier)
Buffer Cache Replacement

• To free the space to bring in a new block, we need to pick a block to discard (like PRA)

• We can use LRU to decide except:
  – we don’t want blocks to go too long without getting written back
  – blocks that contain file system structure (inodes) should be written back immediately
  – partial disk blocks (probably still being written) shouldn’t be thrown away

• Modified LRU with separate code to write blocks back to the disk before they are discarded
OS Startup 1

• Hardware is turned on...

• Boot Program
  – general purpose standalone pgm to load & execute other standalone pgms
  – standalone means it can run without kernel assistance
  – standalone pgms linked with special i/o libraries that let them read devices -- they have standalone device drivers

• Once boot is loaded, it has to figure out what to load -- usually a hardwired path
OS Startup 2

• Boot loads pgms at addr 0
  – boot was loaded at 0, so it first relocates itself
  – boot has all addresses as real, so it is built with addresses set to where it will relocate

• Boot loads the kernel at addr 0 and starts it running
  – hardware interrupts are blocked
  – address translation is turned off
OS Startup 3

- Kernel will initialize in 3 phases
  - assembly language pieces
  - machine dependent
  - machine independent
Assembly Language Init

• Setup the run time stack
• Identify the CPU type
• Calculate physical memory
• Enable virtual address translation hardware
  – kernel makes sure all real pages map to the correct virtual pages
• Create the hardware context for process 0
• Invoke the initial C based entry point
  \( \text{main}() \)
Machine Dependent Init 1

- \textit{main}() gets called with the number of the first free page
- \textit{main}() calls \textit{startup}()
- \textit{startup}()
  - init error message buffer to display errors and messages on the console
  - allocate memory for system data structs
  - initialize kernel memory allocator
  - autoconfigure & initialize devices
Machine Dependent Init 2

• System data structs
  – many are allocated contiguously
  – sized based on startup params and physical memory i.e. how big a machine, how many users

• Device Configuration
  – System has a list of all possible devices
  – Device drivers know how to probe a device to see if it is there
  – Interrupt handlers are setup for devices
Machine Independent 1

• Complete the setup of process 0 (swapper)
  – put it in the process table
  – mark it as runnable
  – mark it as the currently running process
• Call \textit{vminit()} to set up the virtual memory system
  – includes paging and swapping devices
• Start the real time clock
Machine Independent 2

• Initialize Others
  – network interfaces
  – network communications
  – process management data structs
  – swap space management data structs
  – file system name cache

• Mount the root file system
  – init cwd and root dir for process 0

• Create init
  – simple assembly language stub
Machine Independent 3

• Call \texttt{sched()} in process 0; this never returns
• Init will run then
  – /etc/rc contains startup scripts that will run
  – /etc/getty gets started to manage terminals and logins
    • /bin/login gets run by getty to allow logins
• Users can now login
Devices

• Block devices
  – operate on blocks
  – can perform a seek operation
  – examples: disks, CD ROM, etc

• Character devices
  – operate on characters
  – cannot perform a seek operation
  – examples: tty, network devices, etc

• Some devices, like tape drives may have both character and block interfaces
Device Controllers

• Controllers tend to abstract away from the low level features of the device
  – display controller knows about scan and refresh rates
  – disk controller knows about error checking, buffering, interleaving, read ahead, etc.
  – Ethernet controller knows about ethernet packet assembly, windowing, etc.

• The controller presents an interface to the OS
  – The interface often consists of memory or some registers
  – The memory may be mapped into normal address space
  – The memory may be in a special device address space
I/O Software Goals 1

• Device Independence
  – As much of the OS as possible should not know or care about the specific features/limitations of a given device
  – To this end, the OS provides common system calls for different devices such as read/write
  – Most of the OS shouldn’t know or care whether a file being read is on a hard disk, floppy, network drive, etc.

• Uniform Naming
  – Devices should have a common naming scheme
  – File naming shouldn’t depend on the device they are stored on
I/O Software Goals 2

- Robust Error Handling - OS shouldn’t crash
  - Detection as close to the source as possible
  - Correction if possible
  - Reporting errors in a useful way

- Performance tricks should be invisible to the user
  - asynchronous I/O should look synchronous
  - OS can’t wait, but it must appear to

- OS needs to handle shared vs non shared devices
  - tape drives, printers, etc.

- I/O software should be layered to provide abstraction from low level details
Device Drivers

• The software that knows the characteristics of a specific device

• Device drivers tend to have a top half and a bottom half
  – Top half is the piece that provides an interface to the rest of the kernel
  – Bottom half is the piece that provides an interface to the interrupt handler when an I/O completes

• Tannenbaum assumes that the driver is a process; this is somewhat unrealistic
Device Driver Functions

• User issues an I/O request - kernel sends request to the driver
  – if driver was idle; driver formats request for the device and sends it to the device (writing to the hardware)
  – if driver is busy, the request gets queued

• Interrupt arrives (an I/O has completed)
  – driver returns a status code and results (if any)
  – driver finds the next item of work on the queue
    • if the queue has stuff on it, take it and act as if an I/O request was just received
    • otherwise if queue is empty, device will be idle
Device Independent I/O Software

• The goal is to have most of the I/O functionality in this layer

• Provides an abstraction from the actual devices including
  – naming - we want a common naming scheme for different devices (in Unix, /dev provides this)
  – protection - can the user access this device in the way she wants to
  – buffering - convert from user size requests to device driver size requests
  – access issues (keep two people from opening the same tape drive, for example)
Specific Example of I/O Software

- Disk Arm Scheduling
  - Given some large number of disk requests, how do we decide what order we should fulfill them in
  - This is a huge issue with multi user systems
  - Less of an OS concern than it used to be - disk hardware tends to take more control of this than it used to because disks have much larger caches than they used to
  - As usual with scheduling algorithms, we want them to be fair (no starvation), we’d like to avoid cheating, etc.
Disk Arm Scheduling 1

- FCFS - simple but has lousy performance
- Pick - like FCFS, but if we pass something that we want, we service it
- Shortest Seek Time First - greedy algorithm; it tends to stay in the middle of the disk and starve outer requests
- Scan - arm moves one way filling requests, at the end (last request) it reverses direction and does the same going the other way. No starvation. Also called elevator.
Disk Arm Scheduling 2

• Circular Scan - like scan, but only fills request in one direction. Then it goes to the farthest request in the other direction and begins again.
Virtual File System Interface

- File entries reference inodes
- Inodes are unique per file system
- What do we do when we network mount other FS
- Add a new layer - the vnode
- Change all of the interfaces in the system from inode to vnode
- vnode for a local file system refers to an inode
- vnode for a remote file system refers to a protocol control block with location and naming info
vnode

- Flags for locking and attributes like root of FS
- Various ref counts
- A pointer to the mount structure of the FS the vnode is in
- NFS info
- A pointer to the set of vnode operations defined
- Pointer to either the inode or nfsnode
- etc.
Pathname Translation

• We used to do namei, but that doesn’t work

• Vnode pathname translation
  – determine starting point as either root or cwd
  – vnode calls FS specific lookup() and passes path and current lookup dir
  – lookup returns a vnode and we repeat

• Key point - the underlying FS doesn’t know what is being used as a mount point

• Crossing a mount point means we switch FS
Network FS History

• Objective: share files across machines
  – Easiest model is to allow remote machines to mount FS as if they were local

• 3 Important Concerns
  – Semantics
  – Coherence
  – Performance

• Remote Disk
  – a scheme that allowed a machine to talk to another’s disk
NFS Background

- Client/Server Protocol
- Client
  - imports the FS
- Server
  - exports local FS to other machines
- Protocol spec in the public domain
NFS Protocol

• Stateless
• Supports Unix FS Semantics but allows FS with less rich semantics, as well
• Access follows Unix, UID, group model but allows FS with weaker models, as well
• Protocol is transport independent
  – Designed for UDP
  – Ported to TCP and many other protocols
NFS Design Limitations

• Assumes clients & server are on a fast local net
  – Works badly on slow networks
  – Works badly for disconnected machines
• Caching model assumes files won’t be shared
  – If they are, the performance is lousy
• Stateless model loses some Unix FS semantics
  – flock
NFS Operations

• NFS Server is stateless
  – all requests are self contained
  – requests may be sent multiple times if things are slow
  – operations can be idempotent or not
  – for those that aren’t, the server keeps a cache of recent operations so as to avoid having problems with non idempotent ops
## NFS Operation List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETATTR</td>
<td>Get file attributes</td>
<td>yes</td>
</tr>
<tr>
<td>SETATTR</td>
<td>Set file attributes</td>
<td>yes</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Look up file name</td>
<td>yes</td>
</tr>
<tr>
<td>READLINK</td>
<td>Read from a sym link</td>
<td>yes</td>
</tr>
<tr>
<td>READ</td>
<td>Read from a file</td>
<td>yes</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write to file</td>
<td>yes</td>
</tr>
<tr>
<td>CREATE</td>
<td>Create file</td>
<td>yes</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Remove file</td>
<td>no</td>
</tr>
<tr>
<td>RENAME</td>
<td>Rename file</td>
<td>no</td>
</tr>
<tr>
<td>LINK</td>
<td>Create link to a file</td>
<td>no</td>
</tr>
<tr>
<td>SYMLINK</td>
<td>Create a symbolic link</td>
<td>yes</td>
</tr>
<tr>
<td>MKDIR</td>
<td>Create directory</td>
<td>no</td>
</tr>
<tr>
<td>RMDIR</td>
<td>Remove directory</td>
<td>no</td>
</tr>
<tr>
<td>READDRDIR</td>
<td>Read from a directory</td>
<td>yes</td>
</tr>
<tr>
<td>STATFS</td>
<td>Get FS attributes</td>
<td>yes</td>
</tr>
</tbody>
</table>
NFS Server

• Files are globally identified by a file handle
  – this is a token by which a client identifies a file to the server
  – handle is returned by lookup
  – handle is file system id, inode number, generation number of the inode
  – generation number insures that the inode still refers to the same file
  – file handle allows the server to find the file being referred to
NFS Protocol (more)

• Server is stateless
  – doesn’t maintain any info about which clients there are or which files they are accessing
  – RPC message has all info necessary to satisfy a request
  – In fact, the server will maintain a cache of recently accessed files so it isn’t really stateless
Stateless Server

• Can simply startup
  – no state recovery necessary
  – no need to worry about which clients are around
  – network partitions or failures followed by reconnects don’t cause problems
Drawbacks to Stateless Servers

• Local FS have state
  – When a file is unlinked, it is accessible until the last reference goes away
  – NFS can’t do this; it has to delete on the last unlink
  – Similar problems with FS advisory locking (flocks)

• Synchronous writes
  – Writes must be committed to stable store
NFS Client Server Interaction

1. Client does a write() system call
2. Data copied to a kernel buffer on the client and write() returns
3. An nfsiod picks up the dirty buffer and sends the buffer to a server
4. The incoming request is given to an nfsd; the nfsd writes the data to local disk and waits for the i/o to complete
5. The nfsd sends an ack back to the client; the nfsiod gets the ack and marks the buffer as clean