

## 2006 Comp Review Session: Software Systems

by David Stavens, Marc A.  
Schaub, and Neil Patel

## Contents

1. Concurrency
2. Memory Management
3. Disk and File Systems

## Review for 2006 OS Comp Part 1: Concurrency

Reading List Summarized  
by David Stavens

## Part 1: Concurrency

- Processes
- Threads
- Scheduling
- Synchronization

(or, compressing 7 hours of  
CS 140 into about 15 mins.)

## Processes

- “A process is a program in execution.”
  - Includes:
    - Program code
    - Current activity such as:
      - Program counter
      - Register contents
      - Stack (ie: local variables) and Data (ie: globals)
- Processes “invented” to allow most basic ideas of concurrency on machines.
  - Allows “firmer control and...compartmentalization”

## Context Switching

- Stop Process 1, Run Process 2.
- Requires:
  - Saving entire Process 1 state.
  - Loading entire Process 2 state.
- “Process state” on next slide.
- “Speed varies from machine to machine”
  - “Typical speeds range from 1 to 1000 microseconds”

## Process Control Blocks (PCB)

- PCB = Representation of process in OS
  - Process State: Ready/Running/Waiting/Halted
  - Program Counter (ie: for context switching)
  - Register Data (ie: for context switching)
  - Scheduling Data (ie: priority, etc.)
  - Memory Management (ie: page tables, etc.)
  - Accounting (ie: CPU time used, process ID)
  - I/O Status (ie: lists of open files, etc.)

## Process Creation

- “parent” process spawns a “child” process.
- One common method:
  - `fork()` followed by `exec()`.
    - `fork()` gives child “a copy of the address space.”
    - `exec()` overwrites “process memory space with a new program”

## Example: Unix Shell

- Can write shell-like functionality:

```
/* parent process spawns child */
if (fork() == 0)
{
    /* child process */
    exec( "/some/other/program" );
}
/* parent continues doing something else */
```

## Interprocess Communication

- Many different methods.
- Can communicate over network.
  - Standard client/server model.
- Can even use shared memory.
  - What if `fork()` did not `exec()` right away?
  - Shared memory usually done with threads.
    - Because then local variables are private.

## Part 1: Concurrency

- ✓ Processes
  - Threads
  - Scheduling
  - Synchronization

(or, compressing 7 hours of CS 140 into about 15 mins.)

## Threads

- Also “called a lightweight process (LWP).”
- Multiple control flows in a single process.
  - For example:
    - Web server serving 10 requests at once.
    - Word processor checking spelling while printing.
- Less overhead than multiple processes.
  - Though equal in terms of what can be done.

## Components of a Thread

- All threads in a process share:
  - Code Section
  - Data Section (ie: global variables / heap)
  - Open Files, Signals, etc.
- But each thread has its own:
  - Thread ID
  - Program Counter
  - Register Set
  - Stack (ie: local variables)

## User Threads vs. Kernel Threads

- User Threads are “above the kernel”.
  - “Library provides support for”
    - Creation, scheduling, management, etc.
  - Benefits: “Fast to create and manage”
  - Drawbacks: Entire process blocks if any thread blocks if kernel is single-threaded.
- Kernel Threads
  - Native kernel support for creation, etc.
  - Drawbacks: Slower (ie: more overhead).
  - Benefit: No problems with blocking.
  - Benefit: “Threads on different processors.”

## Multithreading Models

- Many-to-One
  - MANY user threads, ONE kernel thread
  - Problem: Whole process blocks.
  - Problem: No multiprocessing.
- One-to-One
  - ONE kernel thread for EACH user thread
  - Problem: Lots of overhead.
- Many-to-Many
  - N user threads mapped to M kernel threads ( $N > M$ )
  - Trades off the above two.

## Thread Pools

- Bad idea to spawn a thread per request?
  - Latency until “servicing the request”
  - Might get too many threads.
- Can have a pool of threads.
  - Incoming request assigned to existing thread.

## Lots more...

- How to cancel threads?
  - Asynchronously
    - One thread kills another.
  - Deferred
    - Thread checks to see if it should terminate gracefully.
- When a signal comes in, which thread gets it?
  - Essentially every combination is possible:
    - Deliver “to the thread to which the signal applies”.
    - Deliver to every thread.
    - Deliver to certain threads.
    - Assign one thread as signal handler.

## Part 1: Concurrency

- ✓ Processes
- ✓ Threads
- Scheduling
- Synchronization

(or, compressing 7 hours of CS 140 into about 15 mins.)

## Scheduling

- N programs want to run.
- But you have M CPUs. And  $N > M$ .
- What to do?
- These slides focus on  $M = 1$ .
  - Because the book does, too.
  - But lots of papers on multiprocessor scheduling...

## Non-Preemptive Scheduling

- Change which process runs only when that process gives up the CPU on its own:
  - By switching from running to waiting
    - I/O request
    - Calling wait()
  - By terminating
- Problem: What if process is infinite loop?

## Preemptive Scheduling

- Still change processes as above:
  - When process blocks/waits or terminates.
- But now also change when a process:
  - Switches from running to ready (eg: interrupt)
  - Switches from waiting to ready (eg: I/O done)
- Add: Hardware timer causes interrupts.
  - So now a process can't run forever.
- Result: Can switch processes at almost any important change of state.

## Issues for Preemptive Scheduling

- Preemptive scheduling dominates today.
- Key Question #1: (Focus of this Section)
  - What process do we run next?
  - How long does it run until pre-empted?
- Key Question #2: (Focus of next Section)
  - How do we keep data consistent?
    - What if a process is modifying a global structure when pre-empted?
    - Especially an issue for threads many-to-one and many-to-many.

## Metrics for Evaluation

- CPU Utilization
  - We want the CPU to be busy.
- Throughput
  - Processes completed per time unit.
- Turnaround Time
  - Time between submission and completion.
- Waiting Time
  - How long a process waits
    - This is the factor actually effected by scheduling.
- Response Time
  - How long until the first response is ready.

## First-Come, First-Served

- No Preemption
- First job to arrive at queue executes
  - until it blocks or terminates.
- Then the next job runs.
- Etc.

## Round-Robin Scheduling

- Similar to First-Come, First-Served
  - But with preemption.
- Time quantum dictates max. running time.
- Time quantum should be larger than the context switch time.
  - To minimize time wasted context switching.

## Shortest-Job-First

- “Know” the length of a process' next CPU burst.
- Then we run the process with the shortest burst.
- Provably optimal!
  - “Gives the minimum average waiting time.”
- How do we know the length of the next burst?
  - We don't. But we can try to predict it.
  - Prediction accuracy = how close to optimal in practice

## More on Shortest-Job-First

- How to predict the next burst length of a process
  - Assume: next burst is similar to its previous bursts.
  - Can use exponential averaging...
- Shortest-Job-First can be preemptive:
  - If current process has  $N$  more ticks and a new job arrives that has  $M$  more ticks and  $M < N$ , preempt.
  - This is called “shortest-remaining-time-first”.

## Priority Scheduling

- Each process has a priority
- Highest priority process gets CPU
  - In case of tie: First-Come, First-Served
- Notice: Shortest-Job-First is special case.
- No consensus: is 0 highest or lowest?

## More on Priority Scheduling

- How do we define priorities?
  - Lots of factors:
    - Time Limits, Memory Requirements, Files Open
    - Average I/O Burst / Average CPU Burst
    - “Importance”, Owing User/Department, etc.
- Can be preemptive or nonpreemptive.
  - As in Shortest-Job-First

## Starvation

- Starvation means “indefinite blocking”.
  - A process wants to run but never gets CPU.
- This concept comes up often.
- Priority scheduling can cause starvation.
- Can be solved by “aging”.
  - “Gradually increasing the priority of processes that wait in the system for a long time”.

## Multilevel Queue Scheduling

- OS has N queues.
  - Each uses "its own scheduling algorithm."
- Also need scheduling at the queue level.
  - Often, "fixed-priority preemptive scheduling."
- Assignment of processes to queues? Fixed:
  - System Processes (highest priority)
  - Interactive Processes
  - Interactive Editing Processes
  - Batch Processes
  - Student Processes (lowest priority)

## Multilevel Queue Feedback Scheduling

- Just like Multilevel Queue Scheduling
  - But now processes can change queues.
- Most flexible scheduling algorithm.
- Also most complicated.
- Numerous simulation and data-driven methods exist for selecting/evaluating parameters.

## Linux Scheduling

- Processes have credits.
  - "Process with the most credits is selected" to run.
- Timer interrupt reduces credits by one.
- New process selected when credits = 0.
- Should "no runnable processes have...credits":
  - Reassign:  $\text{credits} = \text{credits} / 2 + \text{priority}$
- Rewards interactive and I/O-bound processes

## Part 1: Concurrency

- ✓ Processes
- ✓ Threads
- ✓ Scheduling
- Synchronization

(or, compressing 7 hours of  
CS 140 into about 15 mins.)

## Why does synchronization matter?

- Consider the following simple code

```
int global;
int foo(void)
{
    global++;
    return global;
}
```

- What if two threads run foo() in parallel?
  - I think we can agree global should be += 2.

## Running foo()

- |   |  |
|---|--|
| • Thread 1  | • Thread 2   |
| • Load value of global <ul style="list-style-type: none"><li>– Global = 0</li></ul> |  |
| • Increment global <ul style="list-style-type: none"><li>– Global = 1</li></ul>     |  |
| • Save value of global <ul style="list-style-type: none"><li>– Global = 1</li></ul> |  |
|   | • Load value of global <ul style="list-style-type: none"><li>– Global = 1</li></ul>      |
|   | • Increment value of global <ul style="list-style-type: none"><li>– Global = 2</li></ul> |
|   | • Save value of global <ul style="list-style-type: none"><li>– Global = 2</li></ul>      |

## Running foo()

- Thread 1
  - Load value of global
    - Global = 0
  - Increment global
    - Global = 1
  - Save value of global
    - Global = 1
- Thread 2
  - Load value of global
    - Global = 0
  - Increment value of global
    - Global = 1
  - Save value of global
    - Global = 1

## Race Condition

- So “the outcome of the execution depends on the particular order in which the access takes place”...
- ...this is called a race condition.

## Critical Section

- Critical Section is an abstraction.
  - Sections of code that update global data.
  - Only one process in a critical section at once.
    - (For processes that work on the same data.)

```
int foo(void)
{
    global++; // Critical Section
    return global;
}
```

## Mutexes

```
int foo(void)
{
    lock(&global_mutex);
    global++; // Critical Section
    unlock(&global_mutex);
    return global;
}
```

- Idea: Only one thread at a time may have a lock on “global\_mutex”. How do we implement this?

## Mutexes: 3 Requirements

- Mutual Exclusion
  - Only 1 process in the critical section at once.
- Progress
  - “If no process is executing in [a critical section]”
  - “and [N] processes wish to enter [the section]”
    - Only these N may decide who goes next.
    - And the decision must occur in finite time.
- Bounded Waiting
  - “Starvation” applied to mutexes.
  - “There exists a bound on the number of times that other processes are allowed to enter [the critical section]” while another has been waiting to enter.

## N Process Software Mutex

- Initially:

```
boolean choosing[n] = {false};
int number[n] = {0};
```
- Lock:

```
choosing[i] = true;
number[i] = max(number[0], number[1], ... number[n-1]) + 1
choosing[i] = false;
for(j = 0; j < n; j++) {
    while(choosing[j]) ;
    while( (number[j]!=0) && ((number[j],j)<(number[i],i)) );
} /* ((a,b) < (c,d) if a<c or if a=c and b<d) */
```
- Unlock:

```
number[i] = 0;
```

## Hardware Improves Efficiency

- Problem: A software solution is inefficient.
- Idea: Disable interrupts in critical sections.
  - But makes multiprocessing very ugly.
  - Can effect system clock.
- Idea: “Atomic” complex instructions.
  - These instructions are indivisible.
  - And do more than one thing.

## Atomic TestAndSet()

- Atomic:

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```

Initially, lock = false;
- Lock:

```
while( TestAndSet(lock) );
```

(Notice that the bounded waiting constraint is not satisfied.)
- Unlock:

```
lock = false;
```

(Also, note “busy waiting”).

## Atomic Swap()

- Atomic:

```
boolean Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Initially, lock = false;
- Lock:

```
key = true;
while( key == true )
    Swap(lock, key);
```

(Notice that the bounded waiting constraint is not satisfied.)
- Unlock:

```
lock = false;
```

(Also, note “busy waiting”).

## Semaphores

- ```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```
- ```
void wait(semaphore S) {
    S.value--;
    if (S.value < 0) {
        add process to S.L;
        block();
    }
}
```
- ```
void signal(semaphore S) {
    S.value++;
    if (S.value <= 0) {
        remove P from S.L;
        wakeup(P);
    }
}
```
- Lock:

```
wait(mutex);
```
  - Unlock:

```
signal(mutex);
```
  - This does everything we want provided that wait() and signal() are atomic...
    - A recursive problem.
  - We use “busy waiting” to guarantee atomicity.
    - This is inefficient, but it only lasts a few instructions.

## Problems

- Notice if even one process’ programmer reverses wait() and signal() then the entire methodology can break down.
  - And the error may not consistently occur.
- Solution: high-level language constructs
  - One is called the monitor.

## Monitors

- “The monitor construct ensures that only one process at a time can be active within the monitor.”
- Monitors cannot model all synchronization schemes. So they are often combined with other constructs.
  - The other constructs use wait() and signal() in a manner similar to semaphores.

## Deadlocks

|                                             |                                             |
|---------------------------------------------|---------------------------------------------|
| • Thread 1                                  | • Thread 2                                  |
| Lock (A) // Gets lock.                      |                                             |
| Lock (B) // Gets lock.                      | Lock (A)                                    |
| Unlock (A)                                  | // Gets lock on A.                          |
| Lock (A)                                    | Lock (B)                                    |
| /* Deadlock. Has B and not releasing it. */ | /* Deadlock. Has A and not releasing it. */ |

CS301 comps review presentation

## Memory Management

Marc A. Schaub  
Stanford University  
[firstname.lastname@cs.stanford.edu](mailto:firstname.lastname@cs.stanford.edu)

Source:  
Silberschatz, Galvin, Gagne.  
Operating System Concepts, 6<sup>th</sup> edition.  
Wiley and Sons. 2003.

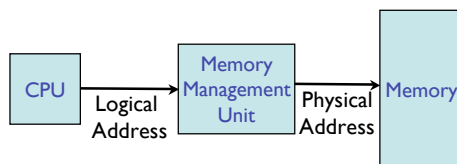
## Outline

- General concepts
- Paging
- Segmentation
- Virtual Memory

## Address Binding

- At compile time (absolute code)
- At load time (relocatable code)
- At execution time
  - Most commonly used
  - Needs hardware support

## Logical vs. physical Address



## Swapping

- The amount of physical memory might be limited.
- A process can be moved temporarily from the memory to a *backing store* (e.g. a hard drive).
- It is possible to keep only the needed instruction and data in memory using *overlays*.

## Memory Allocation

- OS keeps track of which parts of memory are available and which parts are used.
- A block of available memory is called a *hole*.
- When a process arrives, the OS searches a large enough hole and allocates it to the process. If the hole is too large, it is split.
- When the process terminates, the allocated memory becomes a hole again. Adjacent holes are merged.

Skipped !

## Dynamic Storage-Allocation Problem

- Satisfy the requests for memory allocation given a set of holes.
- Possible strategies:
  - First fit (first hole of sufficient size)
  - Best fit (smallest hole that is big enough)
  - Worst fit (largest hole)
    - Produces the largest leftover hole

Skipped !

## Fragmentation

- *External* fragmentation: there is enough memory available, but the holes are not contiguous.
- Solution: compaction (not always possible)
- *Internal* fragmentation: because the memory is subdivided into fixed-size partitions, the memory allocated to a process is slightly larger than the required memory

Skipped !

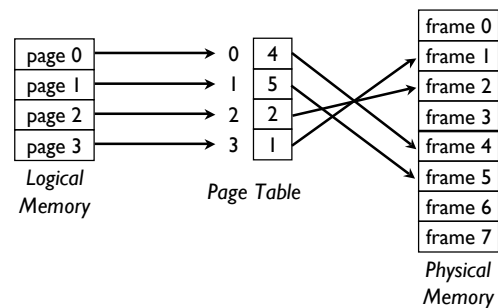
## Outline

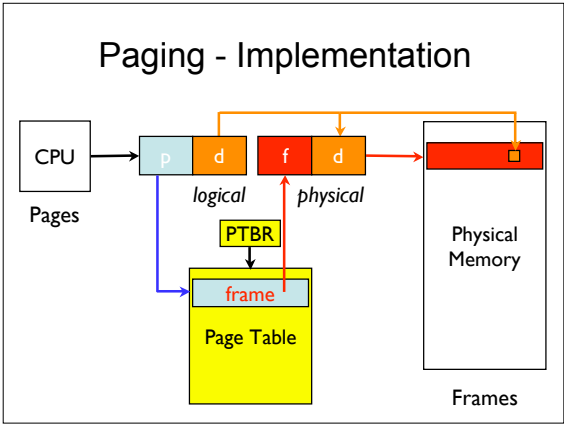
- General concepts
- **Paging**
- Segmentation
- Virtual Memory

## Paging

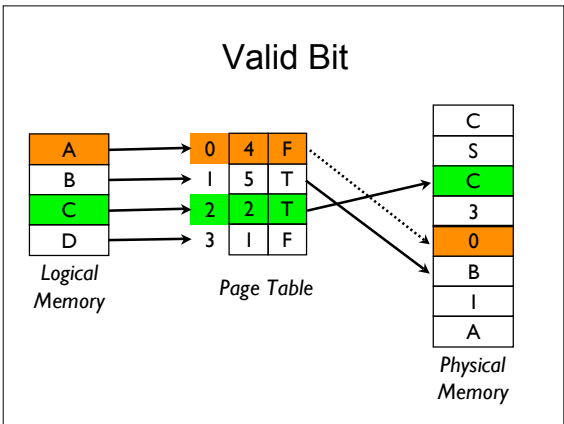
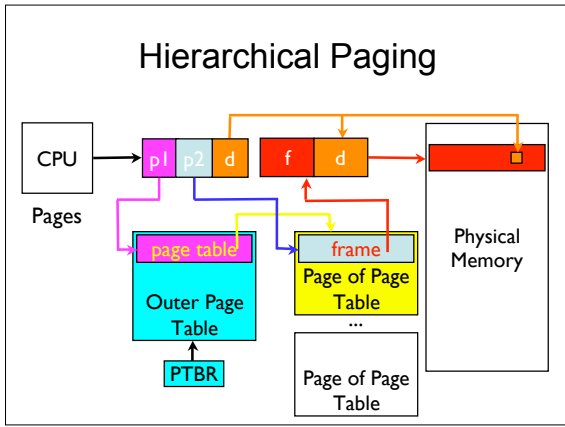
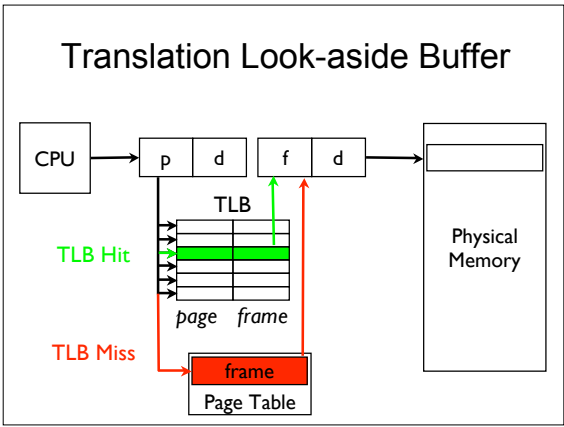
- Motivation: allow the physical-address space of a process to be non-contiguous.
- Traditionally handled in hardware.
- The logical memory is separated into *pages*
- The physical memory is separated into *frames* of the same size

## Paging





- ### Paging
- Benefits
    - Possibility of having protection bits associated with each frame
    - Shared pages
  - Issues:
    - Two memory accesses (page table + frame) needed instead of one.
    - Large sparse page tables for most processes



- ### Paging: Other approaches
- Hashed page tables
    - Commonly used for address spaces larger than 32 bits
  - Inverted Page Table
    - One entry per frame, with PID and logical address
- Skipped !*

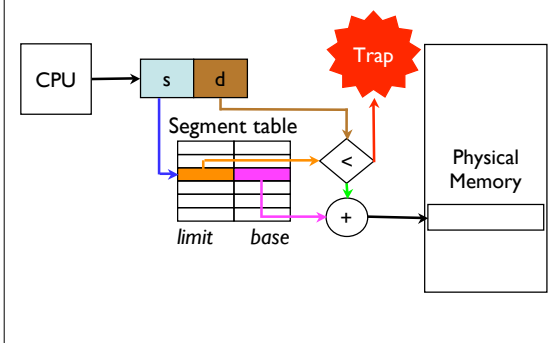
## Outline

- General concepts
- Paging
- **Segmentation**
- Virtual Memory

## Segmentation

- Motivation:
  - From a program point of view, the memory can be seen as segments of various length. For example: global variables, procedure call stack, code, local variables...
- Implemented by having a 2D logical memory space: address = (segment, offset)
- Advantage: protection at the segment level.

## Segmentation



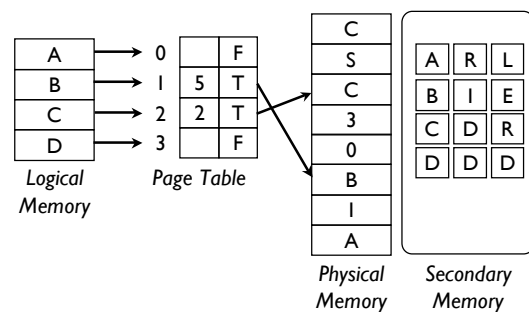
## Outline

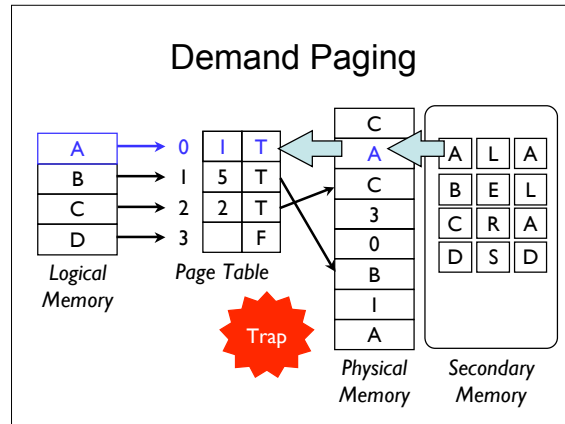
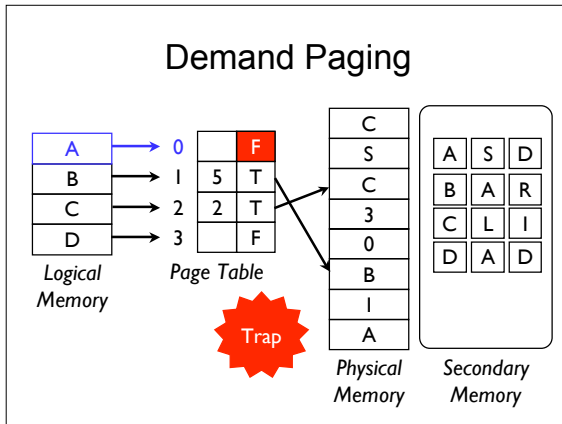
- General concepts
- Paging
- Segmentation
- **Virtual Memory**

## Virtual Memory

- Separation of the logical memory from the physical memory.
- Allows large logical memory when only small physical memory is available.
- Implemented using *Demand Paging*
  - Some pages are swapped out to secondary memory (typically a hard drive)

## Demand Paging





- ### Copy on Write
- After *fork()*, parent and child share the same physical memory.
  - Pages are marked read-only.
  - Pages are duplicated only when one process attempts to write to them.
  - Avoids to copy the whole address space when the child won't use it (e.g. it calls *exec()* immediately after it's creation)
- Skipped !*

- ### Memory-mapped Files
- Treat I/O like memory access.
  - Part of the virtual address space logically associated with a file.
  - File content loaded into memory on first access (page fault).
- Skipped !*

- ### Page replacement
- No more free page in physical memory
  - Need to find a *victim page* that gets written to the disk before the desired page can be loaded from the disk.
  - Two disk accesses
  - Use *modify bit* to avoid writing an unmodified page to disk.

- ### Page replacement algorithms
- FIFO
    - Ignores how often a page is used
    - Belady's anomaly
  - Optimal: Replace the page that will not be used for the longest period of time.
    - Requires future knowledge => hard
  - Least Recently Used (LRU)

## Implementation of LRU

- Counters
  - one time-of-use field per frame
- Stack
  - referenced page added to the top, LRU page at the bottom
- Approximations
  - Reference bit set to one when page accessed
  - Recorded at regular intervals

Skipped !

## Frame Allocation

- Split free memory among processes
- *Equal allocation*: each process gets the same
- *Proportional allocation*: size proportional to the size of the process
- *Global versus local* page replacement:
  - Global: can take frames from other processes
  - Local: select from its own set of frames

Skipped !

## Trashing

- Number of pages insufficient
- Process page faults, page is replaced.
- Replaced page still in active use, page faults again soon.
- Severe paging activity results in performance issue
- Local replacement algorithm avoids the propagation of trashing to other processes

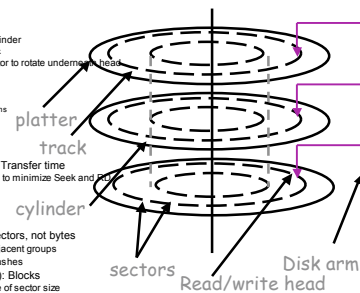
## Review for 2006 Software Systems Comp Part 3: Disks and File Systems By Neil Patel

## Part 3: Disk and File Systems

- Disk Structure and Management
- File-System Interface and Implementation
- Distributed File Systems
- Real-life Systems: NFS, FFS

## Disk Organization

- To read or write disk:
  - Seek: Position heads over cylinder
    - ~ 10 ms to move across disk
  - Rotational Delay: wait for sector to rotate underneath head
    - ~ 8 ms per rotation
  - Select head
  - Transfer data
    - 4 MB/s, 1KB/sector => 25 ms



- Cost of access is Seek + RD + Transfer time
  - Key to using disk effectively is to minimize Seek and Rotational Delay
- Disk reads/writes in terms of sectors, not bytes
  - read/write single sector or adjacent groups
  - Unit of atomicity, even with cross heads
- Logical transfer unit (within OS): Blocks
  - Typically blocks are a multiple of sector size

## Disk Management

- How we organize files on disk
- Must support several file usage patterns
  - Sequential
  - Random
  - Keyed
- Most files are small
- Most I/O due to few very large files that take up most of disk
- Need to handle both efficiently!

## Allocation Schemes

- Contiguous Allocation
- Linked Files
- Indexed
- Multilevel indexed files (inode)

## Contiguous Allocation

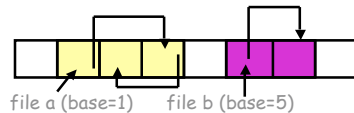
- User says in advance how big file will be
- Search bitmap of free disk space (using best/first fit)
- File header contains:
  - First sector in file
  - Size



- Pros
  - Fast sequential access
  - Easy Random Access
- Cons
  - External Fragmentation
  - Hard to grow files

## Linked Files

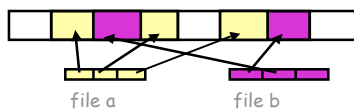
- Each block has a pointer to next one on disk
- File Header points to first block



- Pros
  - Can grow files dynamically
  - Free List managed the same as files
- Cons
  - Sequential access requires seek between each block
  - Random access is horrible... requires full traversal

## Indexed files

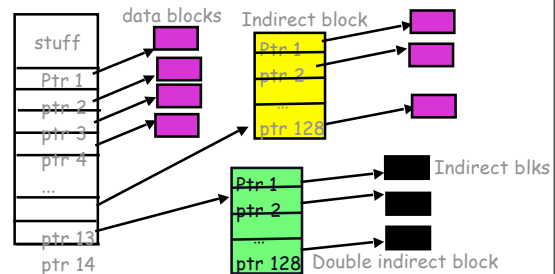
- User specifies max file size, system allocates file header with an array of pointers big enough to point to requisite number of blocks



- Pros
  - Can easily grow to the allocated space
  - Fast random access
- Cons
  - Awkward to grow file bigger than initial allocation
  - Sequential access still slow, since blocks can be anywhere on disk

## Multilevel indexed files

- Key idea: be efficient for small files, but still support large ones



## Multilevel Indexed files explained

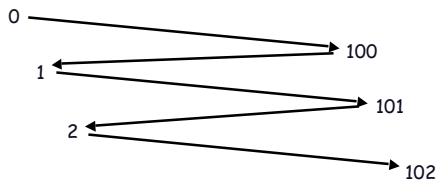
- Allocate 14 pointers
  - First 12 are to data blocks
  - 13th to an indirect block; a block of pointers to data blocks
  - 14th is doubly indirect
- Pros
  - Relatively simple
  - Scales to large files well
  - Small files are cheap and easy
- Cons
  - Lots of indirection when reading large files

## Disk Scheduling

- Disk can only handle one request at a time, so what order do we process requests?
- Several schemes to try and minimize seek time
  - FIFO
  - SSTF
  - SCAN

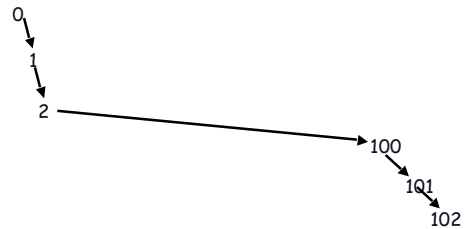
## FIFO

- Schedule disk requests in order received.
  - Fair but may have huge seeks for no good reason.
- Example: read from cylinders 0, 100, 1, 101, 2, 102



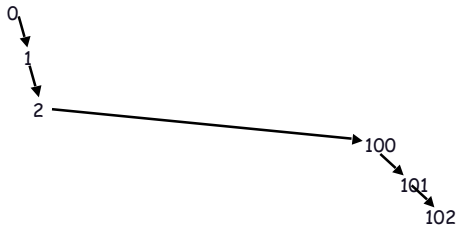
## Shortest Scan Time First

- Pick the request that's the closest on the disk
  - Optimize seek time, but may result in starvation
- Example: read from cylinders 0, 100, 1, 101, 2, 102



## SCAN

- Elevator algorithm: take the closest request in the direction of travel
  - No starvation, and retains flavor of SSTF
  - Unfairly biased towards blocks in the middle of the disk
- Example: read from cylinders 0, 100, 1, 101, 2, 102

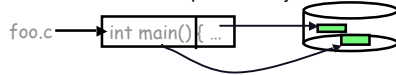


## Part 3: Disk and File Systems

- ✓ Disk Structure and Management
- File-System Interface and Implementation
- Distributed File Systems
- Real-life Systems: NFS, FFS

## File-System Interface

- The File abstraction:
  - user's view: named sequence of bytes



- File operations:
  - create a file, delete a file
  - read from file, write to file
- Directories
  - One-level, two-level, tree-structure

## File-System Implementation

- Files are identified and managed by File Headers
  - a.k.a file descriptors, i-nodes
  - Hold file size, access times, owner and group id, protection bits
- Inodes are stored in a fixed sized array
  - Size of array determined when disk is initialized and can't be changed. Array lives in known location on disk.

## Directories

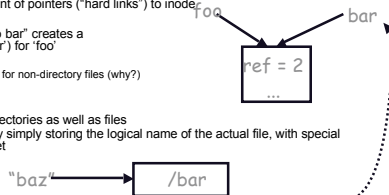
- Mapping of names to file index (i-number)
- Directories stored on disk just like regular files
  - inode contains special flag bit set
- users can read just like any other file
  - "ls" command works this way
- only OS is permitted to modify directories

## Directory example

- Given: /joe/abcde/file1
- What's the disk I/O to read the first byte of file1?
  1. Read in file header for root (always the same)
  2. Read in first data block for root
  3. Read in file header for joe
  4. Read in first data block for joe
  5. Read in file header for abcde
  6. Read in first data block for abcde
  7. Read in file header for file1
  8. Read in first data block for file1
- Caching makes all of this efficient

## Linking

- Hard links: more than one dir entry can refer to a given inode
  - Unix stores count of pointers ("hard links") to inode
  - to make: "ln foo bar" creates a synonym ("bar") for "foo"
  - Can only be used for non-directory files (why?)
- Soft links:
  - Can point to directories as well as files
  - Implemented by simply storing the logical name of the actual file, with special "sym link" bit set



- No Protection from deletion ... may be "dangling" or point to wrong file
- When the file system encounters a symbolic link it automatically translates it (if possible).

## Protection and Access Control

- Goals of Protection
  - Prevent accidental or maliciously destructive behavior
  - Ensure fair resource usage
- Access rights
  - Files
    - Read, write, execute
  - Directories
    - List, modify, delete
- Access Control Lists
  - Keep list of access rights for each domain (user, group) with each object

```
File3:
    User A: rwr
    Group B: rw
    ...
```
- Capability Lists
  - Keep lists of access rights for each object with each domain

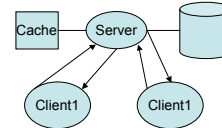
```
User A:
    File 1: rw
    File 2: r
    ...
```
- In practice systems use a combination approach
  - ACL for objects, plus assign "group" or "role" capabilities to users

## Part 3: Disk and File Systems

- ✓ Disk Structure and Management
- ✓ File-System Interface and Implementation
- Distributed File Systems
- Real-life Systems: NFS, FFS

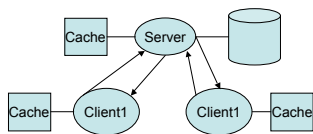
## Distributed File Systems

- Provide access to files stored on a remote disk
- Simple solution:
  - RPC for every file system request to remote server



- Main issue: Performance
  - Network is slower than local memory, network traffic builds up
  - Server becomes bottleneck
- Solution: Caching (of course!)

## NFS



- Now open/read/write/close can be done locally
- Issues: client/server crashes and cache consistency

## NFS – Handling Failures

- What if the server crashes?
  - We want this to be transparent to the client
- NFS supports idempotent operations
  - If client doesn't get an acknowledgement, it's ok to repeat request multiple times
- Delayed-write caching policy
  - Cache's data periodically checked, flushed to server if dirty
- NFS is stateless... no client information is kept on server

## NFS Protocol – Handling Cache Consistency

- With multiple clients reading and writing the same files, how do we know we have the most up-to-date files?
  - In NFS, clients poll server for changes
  - Close-to-open cache consistency: If read starts more than 30 seconds after last write, get new copy. Otherwise, who knows?
- What if multiple clients write to same file
  - In NFS, changes can come from one, the other, or both

## FFS

- Used in Unix (a.k.a. UFS)
- Important properties
  - Big blocks 4-8K
    - But allows chopping into "fragments"
  - Clustering
    - Tries to put sequential blocks in adjacent sectors (access one block, probably access next)
    - Tries to keep inode in same cylinder as file data (if you look at inode, most likely will look at data too)
    - Tries to keep all inodes in a dir in same cylinder group (access one name, frequently access many... "ls -l")

## Acknowledgments

- Much of this section was material adapted from to lecture notes from:
  - CS 140 (Rosenblum)
  - UC Berkeley's CS 162 (Joseph)