Final Review
Operating System Design

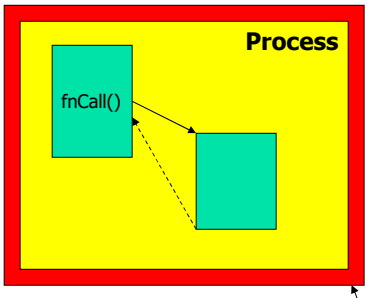
University of Illinois at Urbana-Champaign

CS 423

1

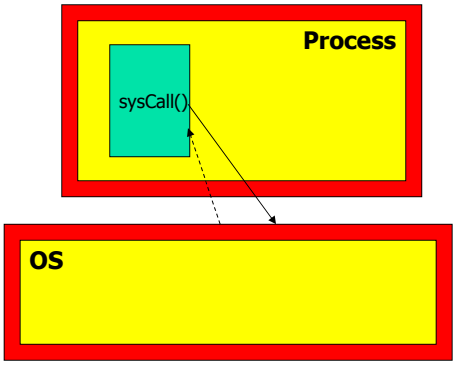
System Calls

Function Call



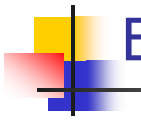
Caller and callee are in the same Process
- Same user
- Same "domain of trust"

System Call



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

2



Examples of System Calls

- Example:
  - getuid() //get the user ID
  - fork() //create a child process
  - exec() //executing a program
- Don't mix system calls with standard library calls
  - Differences?
  - Is printf() a system call?
  - Is rand() a system call?

3



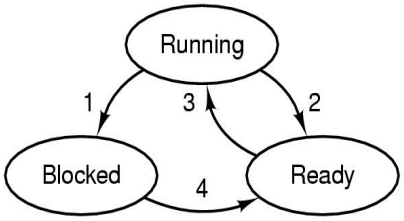
I/O Library Calls versus System Calls

- **Each system call has analogous procedure calls from the standard I/O library:**

System Call	Standard I/O call
open	fopen
close	fclose
read/write	getchar/putchar
	getc/putc
	fgetc/fputc
	fread/fwrite
	gets/puts
	fgets/fputs
	scanf/printf
	fscanf/fprintf
lseek	fseek

4

Processes



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated
- Question: in a single processor machine, how many process can be in running state?

5

Creating a Process – fork()

- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- fork() returns
  - 0 if child
  - -1 if fork fails
  - Child's PID if parent process
- Child gets new program counter, stack, file descriptors, heap, globals, pid!

6



exec() Function

- Exec function allows child process to execute code that is different from that of parent
- Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
- Exec functions return -1 and sets errno if unsuccessful

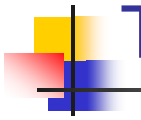
7



Threads and Processes

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

8



Threads

POSIX function	description
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

9



Kernel versus User Threads: Trade-offs?

- Kernel thread packages
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors
- Threads in User-level
  - Fast context switch
  - Customized scheduling

10



Signals

- Signal is *generated* when the event that causes it occurs.
- Signal is *delivered* when a process receives it.
- The *lifetime* of a signal is the interval between its generation and delivery.
- Signal that is generated but not delivered is *pending*.
- Process *catches* signal if it executes a *signal handler* when the signal is delivered.
- Alternatively, a process can *ignore* a signal when it is delivered, that is to take no action.
- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains the set of signals currently blocked.

11



Examples of POSIX Required Signals

Signal	Description	default action
SIGABRT	process abort	implementation dependent
SIGALRM	**alarm clock**	**abnormal termination**
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	**interactive attention signal (usually ctrl-C)**	**abnormal termination**
SIGKILL	**terminated (cannot be caught or ignored)**	**abnormal termination**

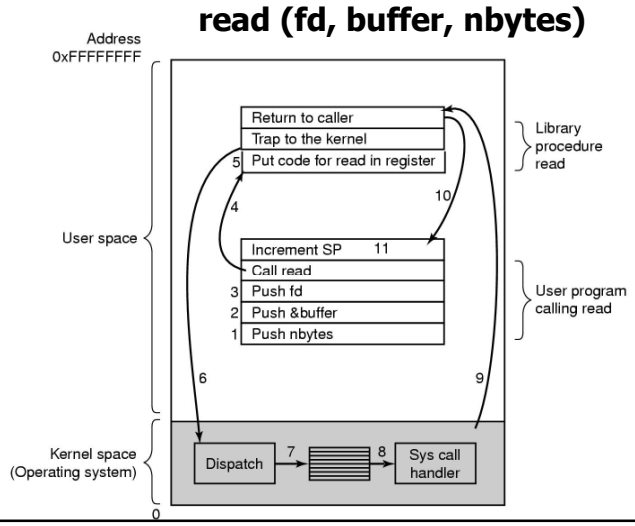
12

Command Line Generates Signals

- You can send a signal to a process from the command line using **kill**
- kill -l will list the signals the system understands
- kill [-signal] pid will send a signal to a process.
  - The optional argument may be a name or a number (default is SIGTERM).
- To unconditionally kill a process, use:
  - kill -9 pid which is
    kill -SIGKILL pid.

13

Steps in Making a System Call



14



Some System Calls For Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

15

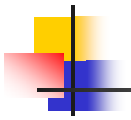


Some System Calls For File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

16

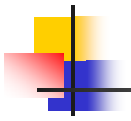


Some System Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

17

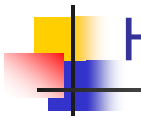


Some System Calls For Miscellaneous Tasks

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

18



Hardware Interrupts

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an "Interrupt controller"
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passes an interrupt number
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler

19



Why Hardware Interrupts?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)

20



Other Interrupts

- Software Interrupts:
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Example: The system call interrupt
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel

21



Other Interrupts

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

22



Bottom Halves

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
- The deferred portion of interrupt processing is called the "Bottom Half"

23



soft_irq

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses raise_softirq() to mark that a given soft_irq must execute the bottom half
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled

24



soft_irq Types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- …

25



Tasklets

- Bottom halves multiplexed on top of soft_irq's
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)

26

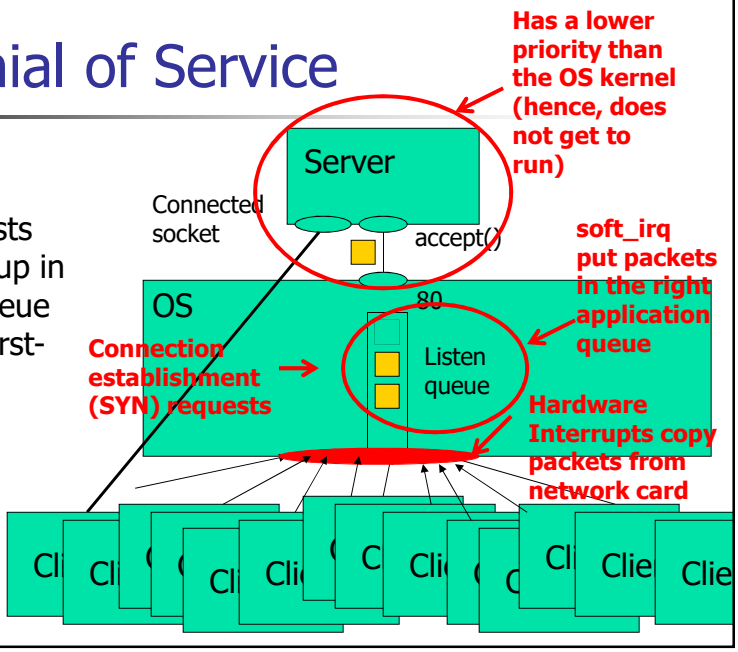
Work Queues

- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)

27

Denial of Service

Client requests get queued-up in the listen queue
First-come first-served





Semaphores in POSIX

- int sem_init(sem_t *sem, int pshared, unsigned value);
- int sem_destroy(sem_t *sem);
- int sem_trywait(sem_t *sem);
- int sem_wait(sem_t *sem);
- int sem_post(sem_t *sem);

29



POSIX Mutex-related Functions

- int **pthread_mutex_init**(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
  - Also see PTHREAD_MUTEX_INITIALIZER
- int **pthread_mutex_destroy**(pthread_mutex_t *mutex);
- int **pthread_mutex_lock**(pthread_mutex_t *mutex);
- int **pthread_mutex_trylock**(pthread_mutex_t *mutex);
- int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

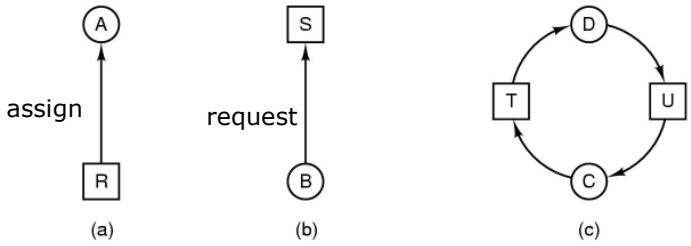
30

Deadlock

- Mutual exclusion
- Hold and wait condition
- No preemption condition
- Circular wait condition

31

Resource Allocation Graph



- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

32

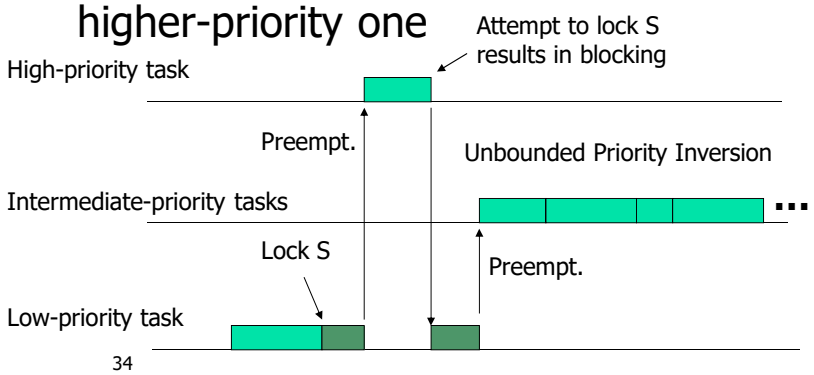
Real-time Scheduling of Periodic Tasks

- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)
- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)

33

Unbounded Priority Inversion

- Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one



34



Priority Inheritance Protocol

- Question: What is the longest time a task can wait for lower-priority tasks?
  - Let there be N tasks and M locks
  - Let the largest critical section of task i be of length B_i
- Answer: ?

35



Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it
- A task that requests a lock R_k is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores (say it belongs to semaphore R_h)
  - The task is said to be blocked by the task holding lock R_h
- A task inherits the priority of the top higher-priority task it is blocking

36



Advanced Configuration and Power Interface (ACPI)

- Defines different power saving states in a platform-independent manner
- The standard was originally developed by Intel, Microsoft, and Toshiba (in 1996), then later joined by HP, and Phoenix.
- The latest version is "Revision 5.0," published in 2011.



Global States

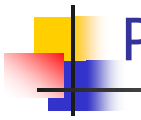
- **G0**: *working*
- **G1**: *Sleeping* and *hibernation* (several degrees available)
- **G2**:, *Soft Off*: almost the same as G3 *Mechanical Off*, except that the power supply still supplies power, at a minimum, to the power button to allow wakeup. A full reboot is required.
- **G3**, *Mechanical Off*: The computer's power has been totally removed via a mechanical switch (as on the rear of a PSU).

Processor "Sleep" States (C-states)

- **C0**: is the operating state.
- **C1** (often known as *Halt*): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.
- **C2** (often known as *Stop-Clock*): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as *Sleep*) is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.

Processor Performance States (P-States)

- **P0** max power and frequency
- **P1** less than P0, voltage/frequency scaled
- **P2** less than P1, voltage/frequency scaled
- …
- **Pn** less than *P(n-1)*, voltage/frequency scaled



Power of Computation

- Terminology
  - R : Power spent on computation
  - V : Processor voltage
  - f : Processor clock frequency
  - R_0 : Leakage power
- Power spent on computation is:
  - $R = k_v V^2 f + R_0$
    where k_v is a constant



Energy of Computation

- Power spent on computation is:
  - $R = k_v V^2 f + R_0$
- Consider a piece of computation of length C clock cycles and a processor operating at frequency f
- The execution time is $t = C/f$
- Energy spent is:
  - $E = R\,t = (k_v V^2 f + R_0)(C/f)$

Linux CPUFreq Governor

- Linux defines multiple DVS modes (called CPUfreq "governors"):
  - Performance (highest frequency)
  - Powersave (lowest frequency)
  - Userspace ("root" user controls frequency)
  - OnDemand (adaptively change frequency depending on load)

Turning Processors Off
The Cost of Wakeup

- Energy expended on wakeup, E_{wake}
- To sleep or not to sleep?
  - Not to sleep (for time t):

    $E_{no\text{-}sleep} = (k_v V^2 f + R_0)\, t$
  - To sleep (for time t) then wake up:

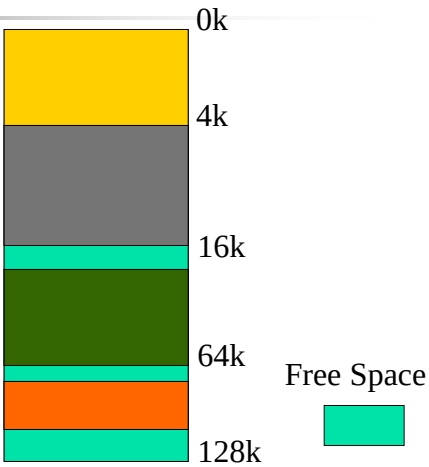
    $E_{sleep} = P_{sleep}\, t + E_{wake}$
  - To save energy by sleeping: $E_{sleep} < E_{no\text{-}sleep}$

    $$t > \frac{E_{wake}}{k_v V^2 f + R_0 - P_{sleep}}$$

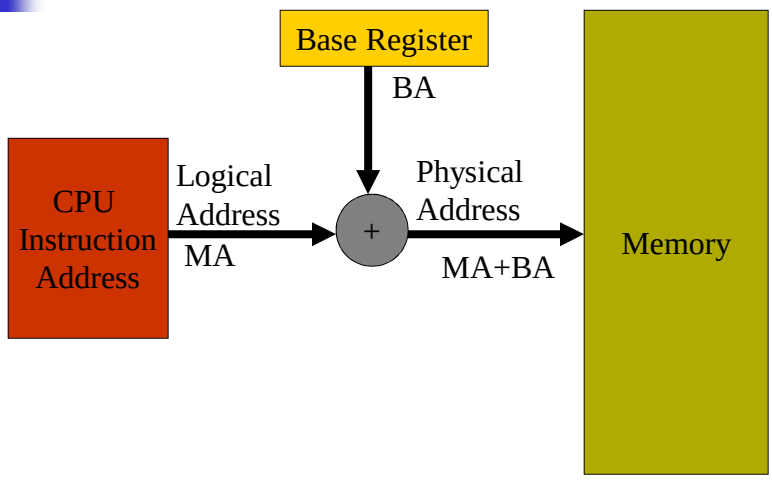
    Minimum sleep interval

History: Multiprogramming with Fixed Partitions

- Divide memory into *n* fixed (possibly unequal) partitions
- Problem:
  - Fragmentation



History: Relocation Register



History: Variable Partition Allocation

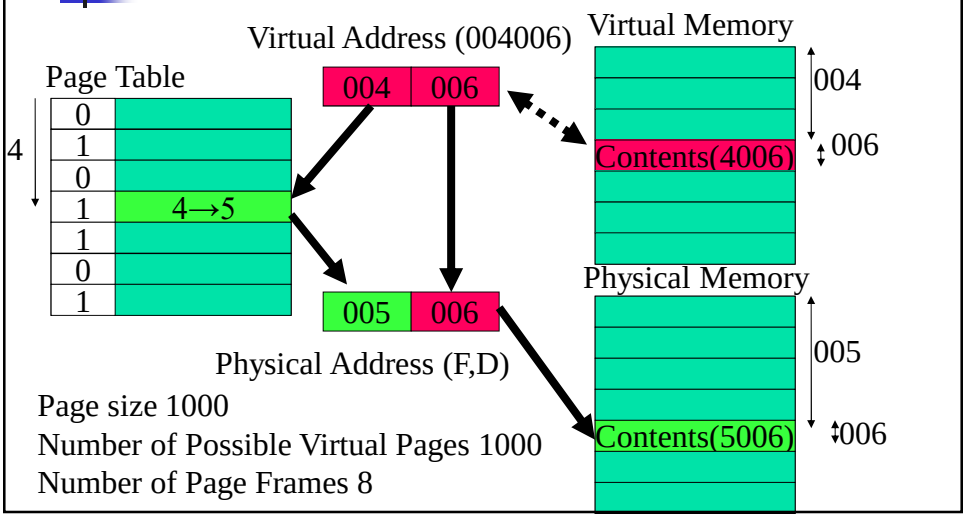


Memory wasted by External Fragmentation

History: Storage Placement Strategies

- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.
- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.
- **Next Fit**.
  - Minor variation of first fit: search from the last hole used.
  - Problem: slightly worse performance than first fit.
- **Worst Fit**.
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.

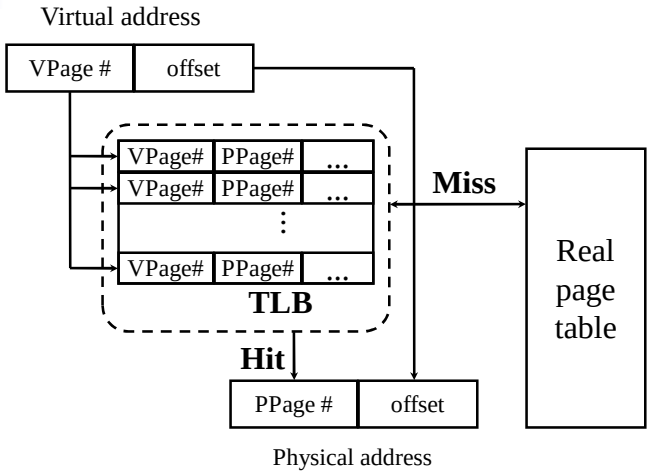
Page Mapping Hardware



Page Fault

- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS's VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

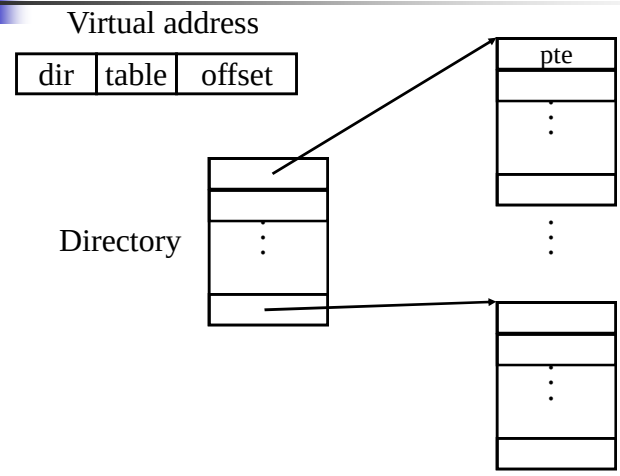
Optimization: Translation Lookaside Buffer (TLB)



Effective Access Time

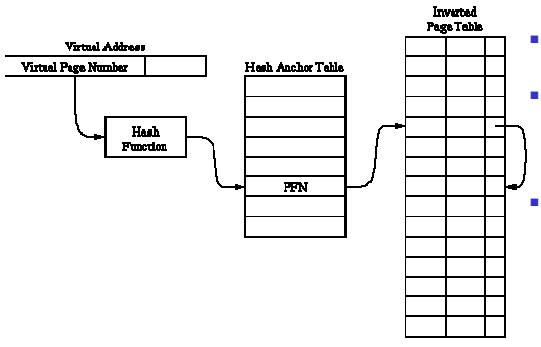
- TLB lookup time = σ time unit
- Memory cycle = m μs
- TLB Hit ratio = η
- Effective access time
  - $Eat = (m + \sigma)\,\eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m\,\eta$

Multilevel Page Tables



What does this buy us? Sparse address spaces and easier paging

Inverted Page Tables



- Hash the process ID and virtual page number to get an index into the HAT.
- Look up a Physical Frame Number in the HAT.
- Look at the inverted page table entry, to see if it is the right process ID and virtual page number. If it is, you're done.
- If the PID or VPN does not match, follow the pointer to the next link in the hash chain. Again, if you get a match then you're done; if you don't, then you continue. Eventually, you will either get a match or you will find a pointer that is marked invalid. If you get a match, then you've got the translation; if you get the invalid pointer, then you have a miss.

Belady's Anomaly (for FIFO)

FIFO with 4 physical pages

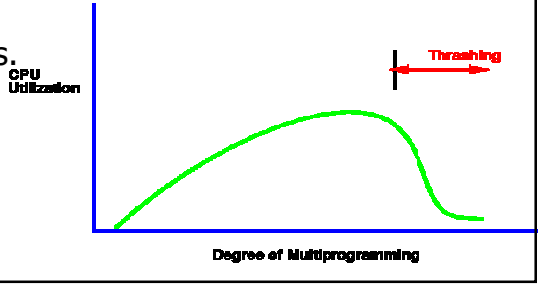
12 references, 10 faults

As the number of page frames increase, so does the fault rate.

Page Refs	4 Page Frames				
	Fault?	Page Contents			
A	yes	A			
B	yes	B	A		
C	yes	C	B	A	
D	yes	D	C	B	A
A	no	D	C	B	A
B	no	D	C	B	A
E	yes	E	D	C	B
A	yes	A	E	D	C
B	yes	B	A	E	D
C	yes	C	B	A	E
D	yes	D	C	B	A
E	yes	E	D	C	B

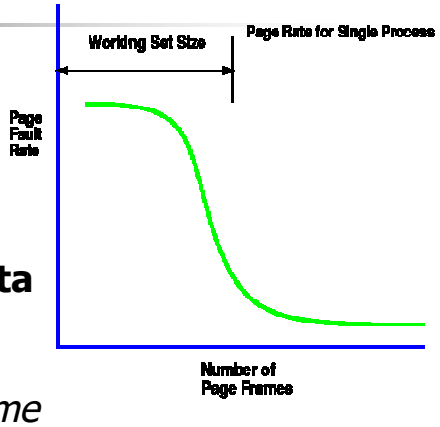
Thrashing and CPU Utilization

- As the page rate goes up, processes get suspended on page out queues for the disk.
- the system may try to optimize performance by starting new jobs.
- starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.
- system throughput plunges.



Working Set

- the working set model assumes locality.
- **the principle of locality states that a program clusters its access to data and text temporally**.
- *As the number of page frames increases above some threshold, the page fault rate will drop dramatically*.



Page Size Considerations

- Small pages
  - Reason:
    - Locality of reference tends to be small (256)
    - Less fragmentation
  - Problem: require large page tables
- Large pages
  - Reason
    - Small page table
    - I/O transfers have high seek time, so better to transfer more data per seek
  - Problem: Internal fragmentation, needless caching



What's a Virtual Machine?

- Virtual machine is an entity that emulates a guest interface on top of a host machine
  - Language view:
    - Virtual machine = Entity that emulates an API (e.g., JAVA) on top of another
    - Virtualizing software = compiler/interpreter
  - Process view:
    - Machine = Entity that emulates an ABI on top of another
    - Virtualizing software = runtime
  - Operating system view:
    - Machine = Entity that emulates an ISA
    - Virtualizing software = virtual machine monitor (VMM)



Purpose of a Virtual Machine

- Emulation
  - Create the illusion of having one type of machine on top of another
- Replication
  - Create the illusion of multiple independent smaller guest machines on top of one host machine (e.g., for security/isolation, or scalability/sharing)
- Optimization
  - Optimize a generic guest interface for one type of host



Taxonomy

- Process VMs
  - Same ISA
    - Multiprogrammed systems
  - Different ISA
    - Dynamic translators (Java), emulators
- System VMs
  - Same ISA
    - Classic VMs
    - Hosted VMs (VMWare)
  - Different ISA
    - Whole system VMs



Basic Interpretation

- Problem: Emulate guest ISA on host ISA
- Solution: Basic Interpretation

```
inst = code (PC)
opcode = extract_opcode (inst)
switch (opcode) {
    case opcode1 : call emulate_opcode1 ()
    case opcode2 : call emulate_opcode2 ()
    …
}
```



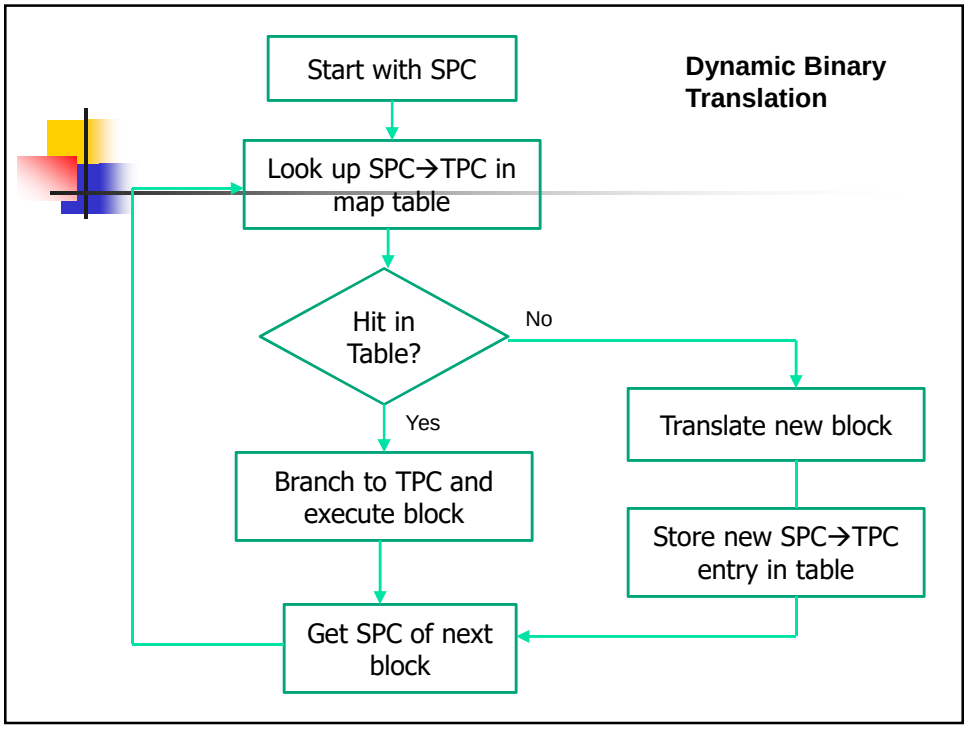
Binary Translation

- Emulation:
  - Guest code is traversed and instruction classes are mapped to routines that emulate them on the target architecture.
- Binary translation:
  - The entire program is translated into a binary of another architecture.
  - Each binary source instruction is emulated by some binary target instructions.



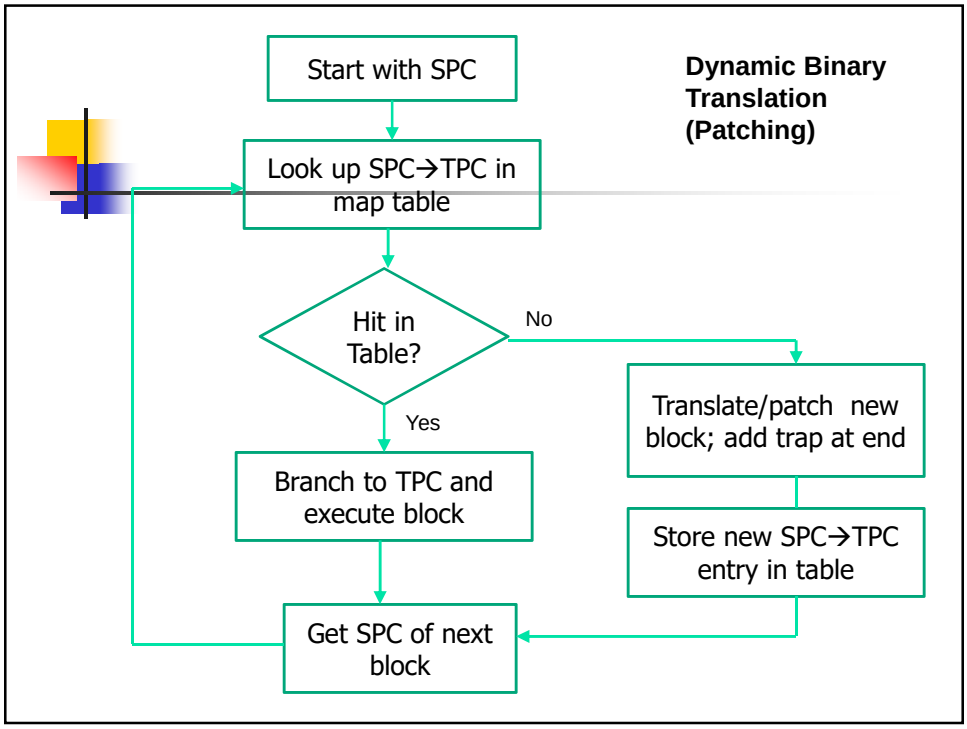
Dynamic Basic Blocks

- Program is translated into chunks called "dynamic basic blocks", each composed of straight machine code of the target architecture
  - Block starts immediately after a jump instruction in the source binary
  - Block ends when a jump occurs
- At the end of each block (i.e., at jumps), emulation manager is called to inspect jump destination and transfer control to the right block with help of map table (or create a new block and map table entry, if map miss)



Sensitive Instructions

- Any instructions that directly affect resource allocation or access privileges are *sensitive*
- Guest OSes should not be allowed to perform sensitive instructions directly because they may gain unfair advantage over other VMs
  - Note: Guest OSes usually run in privileged mode.
  - When emulated, they run in user mode.
  - Privileged instructions result in traps
  - If all sensitive instructions are privileged, they will trap to VMM, which can decide how to execute them



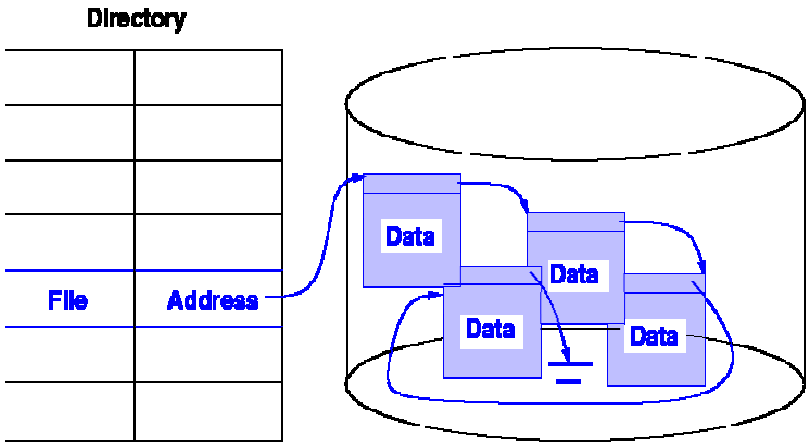
Paging

- Architectural assumptions:
  - Paging hardware knows where page table is by looking up value of page table pointer register
  - Page tables have agreed-upon structure that the hardware knows
  - OS maintains value of page table pointer register to point to the page table of the running application
- Virtualization: Page table pointer register is virtualized, and all accesses to the register trap to VMM (why?)
  - Read access: VMM returns the virtual page table pointer value (the guest page table pointer register)
  - Write access: VMM updates the virtual page table pointer value and sets the real page table pointer register to point to the corresponding shadow table.

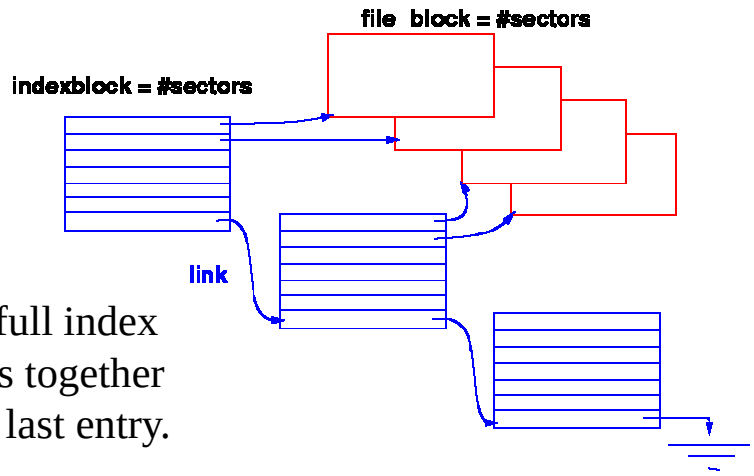
Interrupts in Binary Translation

- When an interrupt occurs, the emulated code may be in non-interruptible state
  - Determine which block is currently running
  - Unchain the block from the next by replacing the jump at the end of the block to a transfer of control to the emulation manager.
  - Let the block finish
  - Control is transferred to emulation manager which invoked interrupt handler.

Linked File Allocation

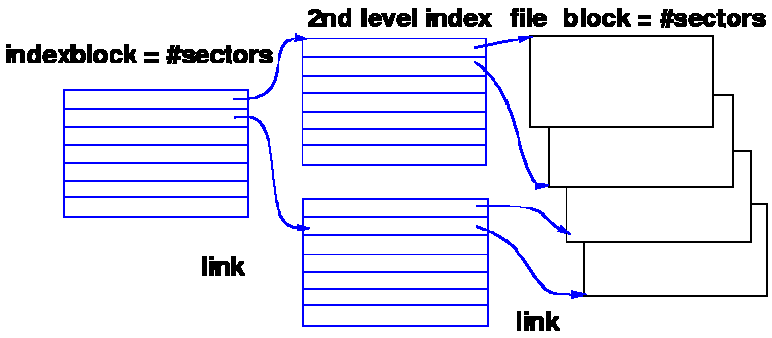


Indexed File Allocation



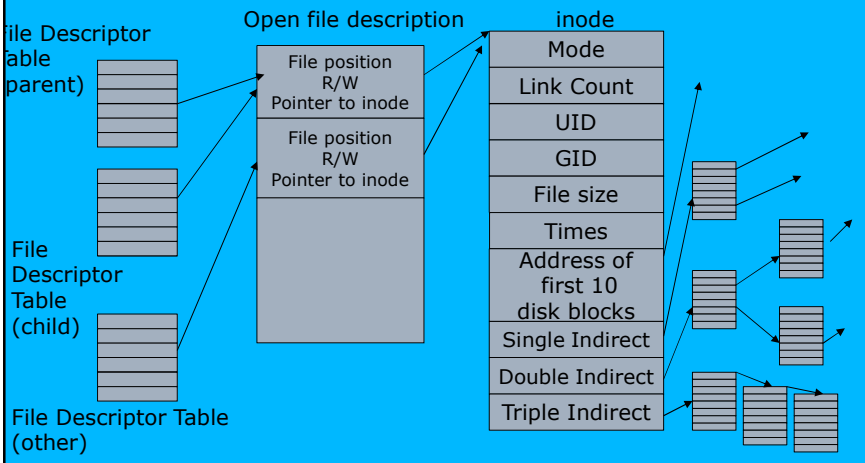
Link full index blocks together using last entry.

Other Forms of Indexed File - Multilevel Index



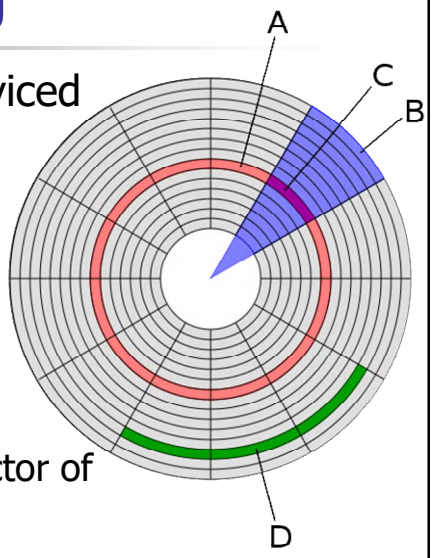
Multiple levels of index blocks

UNIX file structure implementation



Disk Scheduling

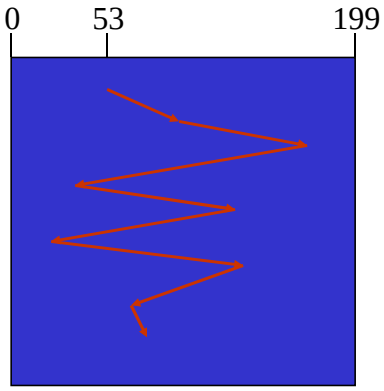
- Which disk request is serviced first?
  - FCFS
  - Shortest seek time first
  - Elevator (SCAN)
  - C-SCAN (Circular SCAN)



A: Track. B: Sector. C: Sector of Track. D: File

FIFO (FCFS) order

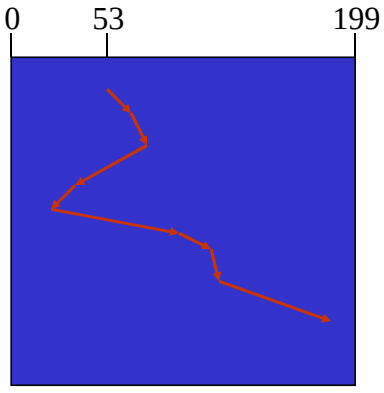
- Method
  - First come first served
- Pros
  - Fairness among requests
  - In the order applications expect
- Cons
  - Arrival may be on random spots on the disk (long seeks)
  - Wild swing can happen
- Analogy:
  - FCFS elevator scheduling?



98, 183, 37, 122, 14, 124, 65, 67

SSTF (Shortest Seek Time First)

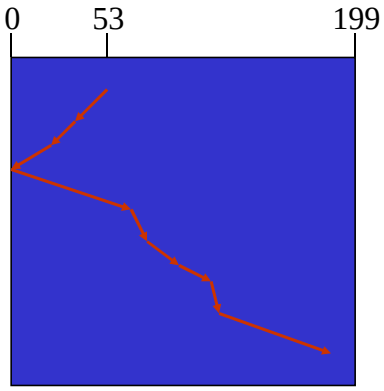
- Method
  - Pick the one closest on disk
- Pros
  - Try to minimize seek time
- Cons
  - Starvation
- Question
  - Is SSTF optimal?
  - Can we avoid starvation?



98, 183, 37, 122, 14, 124, 65, 67
(65, 67, 37, 14, 98, 122, 124, 183)

Elevator (SCAN)

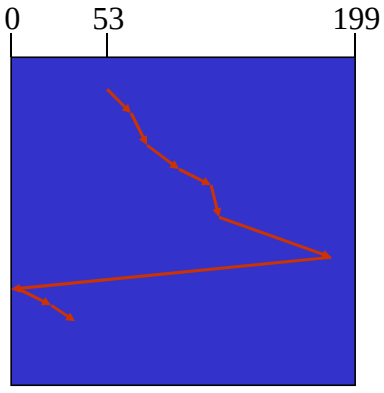
- Method
  - Take the closest request in the direction of travel
- Pros
  - Bounded time for each request
- Cons
  - Request at the other end will take a while



98, 183, 37, 122, 14, 124, 65, 67
(37, 14, 65, 67, 98, 122, 124, 183)

C-SCAN (Circular SCAN)

- Method
  - Like SCAN
  - But, wrap around
- Pros
  - Uniform service time
- Cons
  - Do nothing on the return



98, 183, 37, 122, 14, 124, 65, 67
(65, 67, 98, 122, 124, 183, 14, 37)

Estimate Sustained Average Transfer Rate

- Suppose that a disk drive spins at 7200 RPM (revolutions per minute), has a sector size of 512 bytes, and holds 160 sectors per track.
- What is sustained average transfer rate of this drive in megabytes per second?

Disk spins 120 times per second (7200 RPM/60)

Each spin transfers a track of 80 KB (160 sectors x0.5K)

Sustained average transfer rate is 120x80 = 9.6MB/s.

Average Performance of Random Access

- Suppose that a disk drive spins at 7200 RPM (revolutions per minute), has a sector size of 512 bytes, and holds 160 sectors per track.
- Average seek time for the drive is 8 milliseconds
- Estimate # of random sector I/Os per second that can be done and the effective average transfer rate for random-access of a sector?

Average Performance of Random Access

Disk spins 120 times per second
Average rotational cost is time to travel half track: 1/120 * 50%=4.167ms

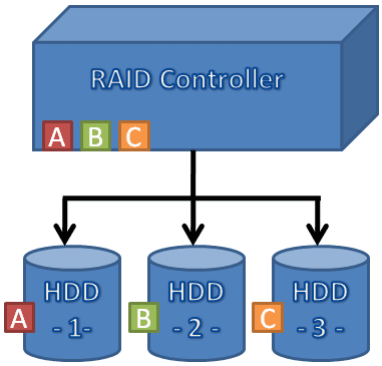
Transfer time is 8ms to seek
+ 4.167 ms rotational latency
+ 0.052 ms (reading one sector takes 0.0005MB/ 9.6MB).
= 12.219 ms

# of random sector access/second = 1/0.012219=81.8

Effective transferring rate: 0.5 KB * 81.8= 0.0409 MB/s.

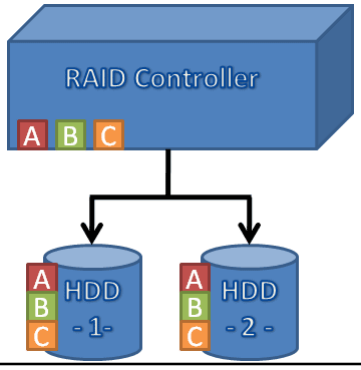
Raid Level 0

- Level 0 is nonredundant disk array
- Files are striped across disks, no redundant info
- High read throughput
- Best write throughput (no redundant ir
- Any disk failure results in data loss



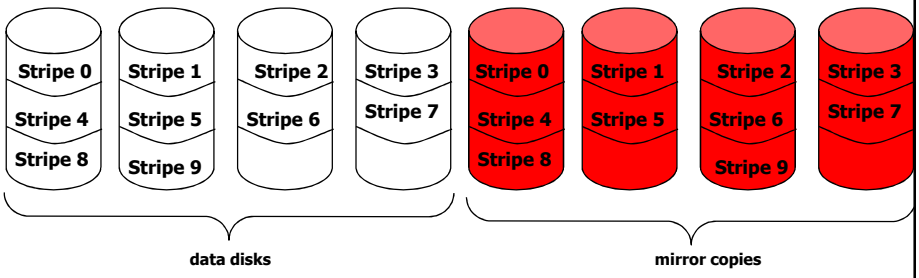
Raid Level 1

- Mirrored Disks
- Data is written to two places
  - On failure, just use surviving disk (easy to rebuild)
- On read, choose fastest to read
  - Write performance is same as single drive, read performance is 2x better
- Expensive (high space overhead)



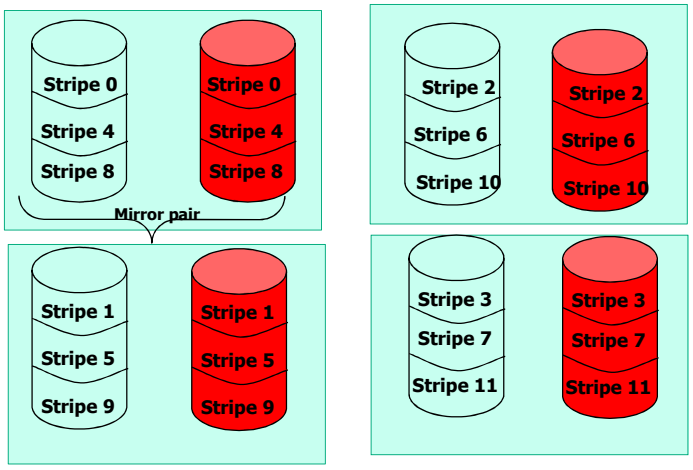
Raid Level 0+1

- Stripe on a set of disks
- Then mirror of data blocks is striped on the second set.



Raid Level 1+0

- Pair mirrors first.
- Then stripe on a set of paired mirrors



Distributed File Systems

- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.
- A *Distributed File System* (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- The resources on a particular machine are *local* to itself. Resources on other machines are *remote*.

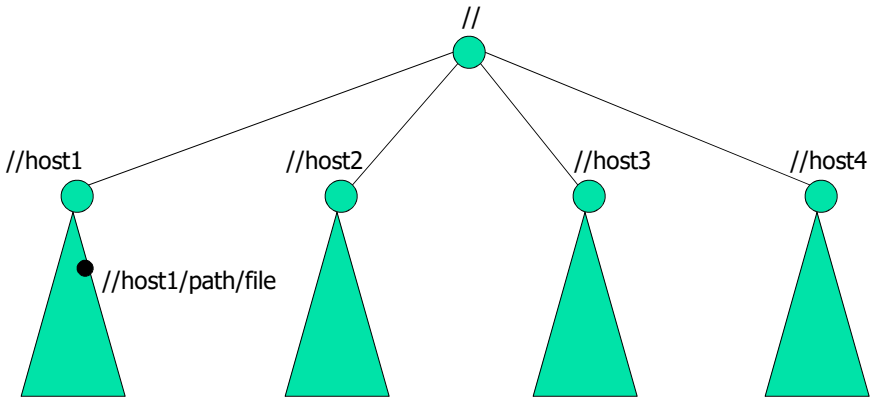
86

Naming and Location Transparency

- *Naming:* mapping between logical and physical objects.
- *Location transparency:*
  - The name of a file does not reveal any hint of the file's physical storage location.
- *Location independence:*
  - The name of a file doesn't need to be changed when the file's physical storage location changes.

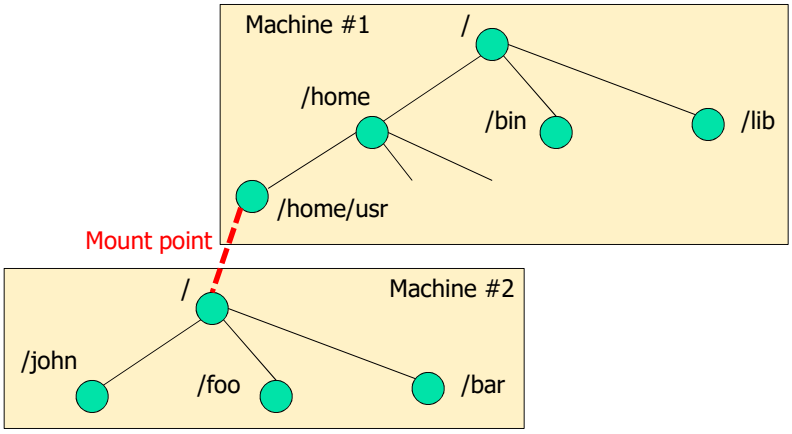
Example #1:

- No location transparency



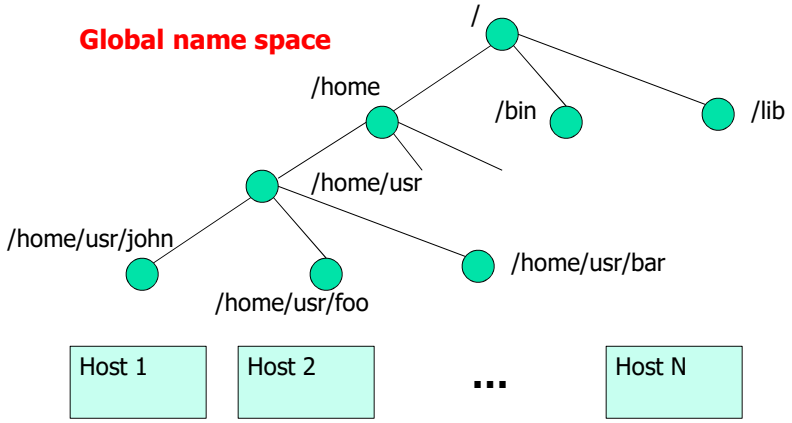
Example #2:

- Location transparency in NFS: Mount operation

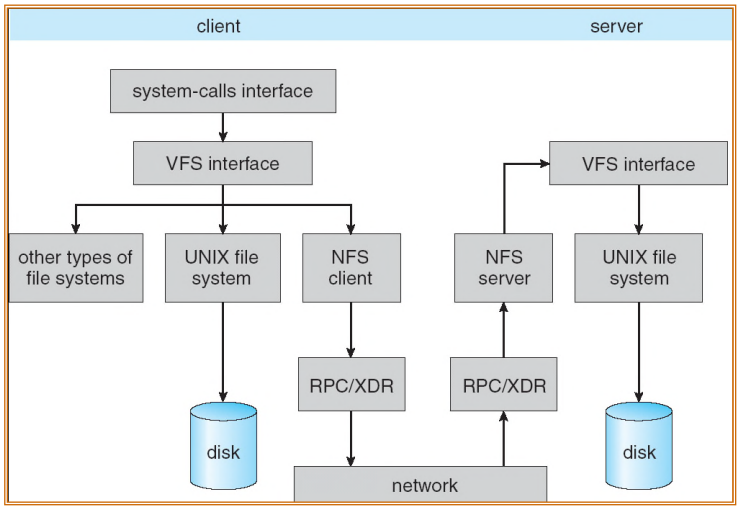


Example #3:

- Location independence in Andrew



Schematic View of NFS Architecture



Network File System (NFS)

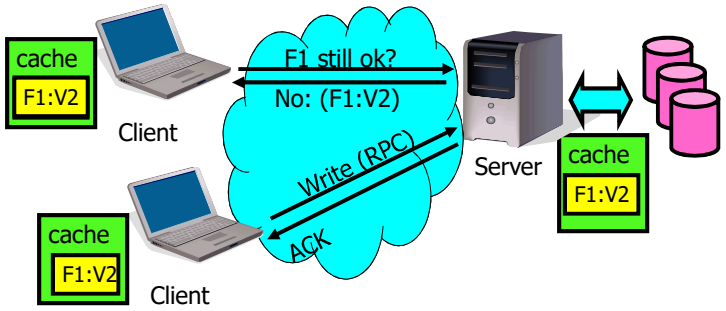
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes!

NFS Continued

- NFS servers are stateless; each request provides all arguments required for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error

NFS Cache consistency

- NFS protocol: weak consistency
  - What if multiple clients write to same file?
    - In NFS, can get either version (or parts of both)
    - Completely arbitrary!





Andrew File System

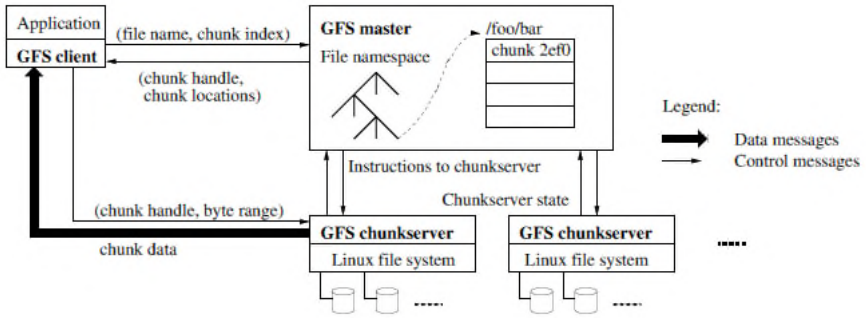
- Andrew File System (AFS, late 80's)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - As a result, do not get partial writes: all or nothing!
    - Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file



Google File System

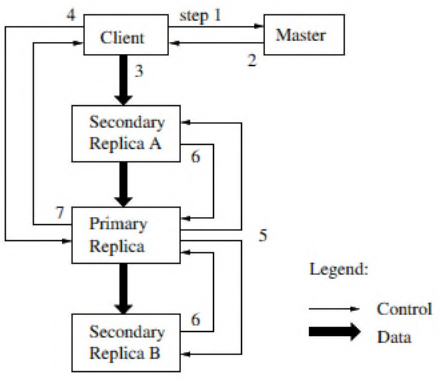
- System is so large that failures are the norm
- Files are very big (multi-GB is the norm)
- Files are modified by "appending" and read usually sequentially

Architecture



- Master, chunk servers and clients
- Chunks (64MB) are replicated on multiple servers
- No file caching
- Clients cache metadata from master

Append



(1) client finds out location of primary and secondary replicas from master

(2) master replies

(3) client sends data to replicas and receives acks

(4) client asks primary replica to do the "append" on the data

(5) primary replica decides on order of appends and asks secondary replicas to follow same order

(6) secondary replicas perform operation and ack primary

(7) primary replies to client



File Copy

- Master received copy (snapshot) request and revokes all leases on its chunks
  - Why revoke?
  - What if it loses communication with a chunk server?
- After all leases are revoked or expire, master duplicates metadata (pointing to the same chunks as original) and increments reference count – no actual data copy is performed
- When a client wants to write to a chunk, the request for primary comes to master who notices that the chunk has a reference count of two and asks chunk servers to replicate it
- A handle is returned to the new chunk copy.



Map/Reduce

- Map/Reduce
  - Programming model from LISP
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes

100



Security

- Data confidentiality: Preventing data exposure
- Data integrity: Preventing tempering with data
- System availability: Preventing denial of service
- Exclusion of outsiders: Preventing un-authorized use

Intruders

- Casual prying
- Insider snooping
- Financial attacks (e.g., programmers for a bank add code to transfer to their account)
- Commercial/military espionage
- Viruses, etc.

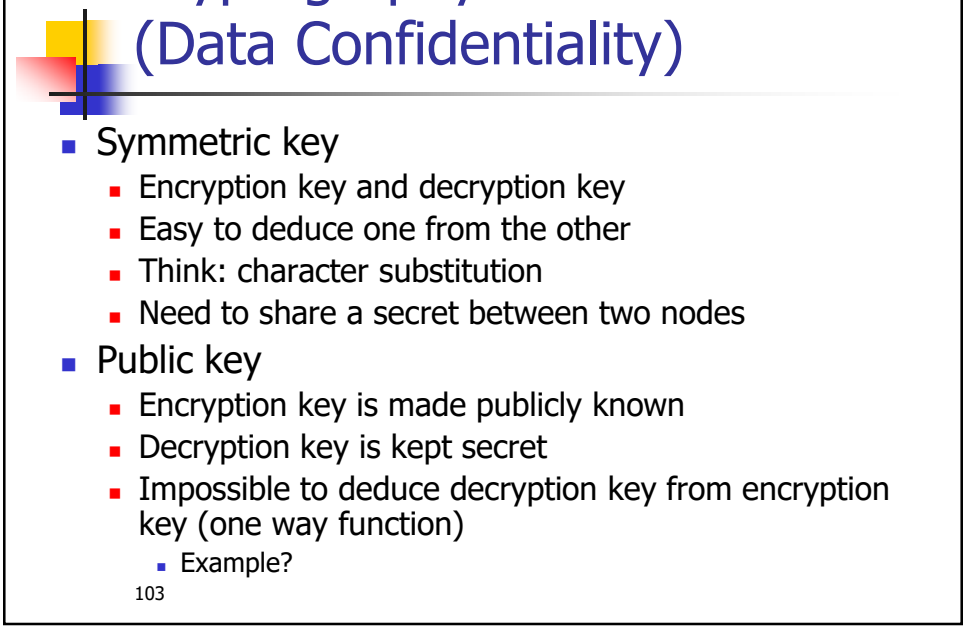
101



Security Attacks

- Authentication
  - E.g., a rogue program, pretending to be the sign-on program, asking for the user's password and then storing it.
- Buffer overflow
  - Give input to a program that will overflow a buffer. If no checks are made, buffer may overwrite stack (including return address) and cause jump to virus code (part of input)
- Operator carelessness
  - E.g., tricking the operator into executing a virus program (such as one named "ls" in a user's directory)
- Residue
  - Interesting information often turns up in wastebaskets; use paper shredders! Information is often left in central memory from a previous user, possibly a system routine; variables that contain sensitive information should be overwritten before they are deallocated!
- Shielding
  - One can inductively ``tap'' a cable, phone line, or in fact any wire over which information passes, without making physical connection to it. Electrical shielding can protect against this.
- Passwords
  - Password guessing, etc.

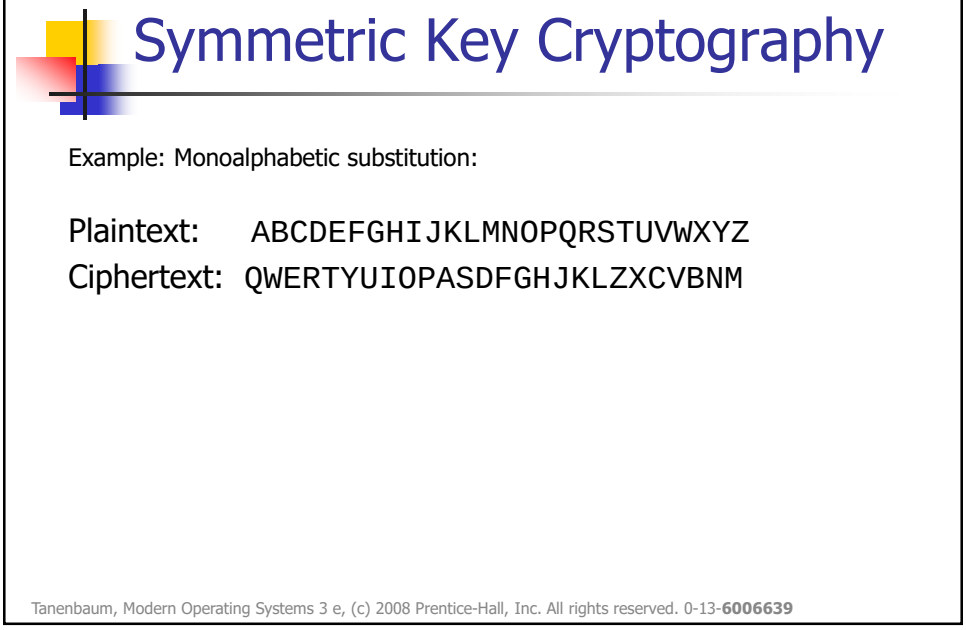
102



Cryptography (Data Confidentiality)

- Symmetric key
  - Encryption key and decryption key
  - Easy to deduce one from the other
  - Think: character substitution
  - Need to share a secret between two nodes
- Public key
  - Encryption key is made publicly known
  - Decryption key is kept secret
  - Impossible to deduce decryption key from encryption key (one way function)
    - Example?

103



Symmetric Key Cryptography

Example: Monoalphabetic substitution:

Plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext: QWERTYUIOPASDFGHJKLZXCVBNM

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**



Rivest-Shamir-Adelman (RSA) Encryption

- RSA uses two keys: d and e with integer n.
  - pair (e,n) will be the public encryption key;
  - pair (d,n) will be the private key.
- Message m is encrypted as follows: E(m) = (m^e) mod n = C,
- Message m will be decrypted as follows: D(C) = (C^d) mod n

105



Digital Signatures (Data Integrity)

Sender

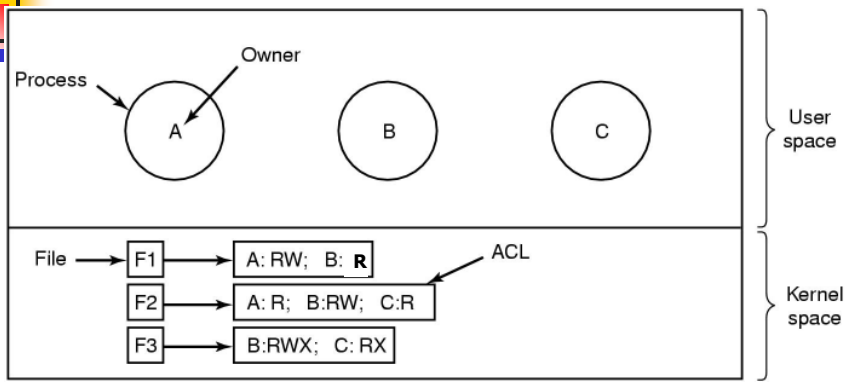
- Use a one-way function whose input is the message being signed and whose output is a digest (e.g., MD5 or SHA-1, SHA-256, …) or hash.
- Encrypt digest using owner's private key
  - Signature block

Receiver

- Compute fixed length digest of the message
- Apply decryption key (sender's public key) to signature block and compare with digest

106

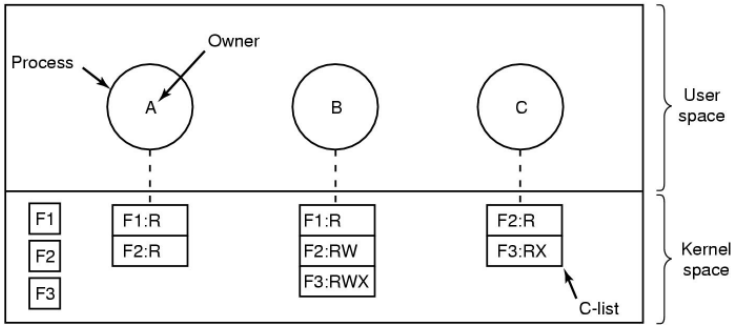
Access Control Lists



Use of access control lists to manage file access in UNIX

107

Capability Lists / C-Lists



When capabilities are used, each process has a capability list.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Multi-level Security

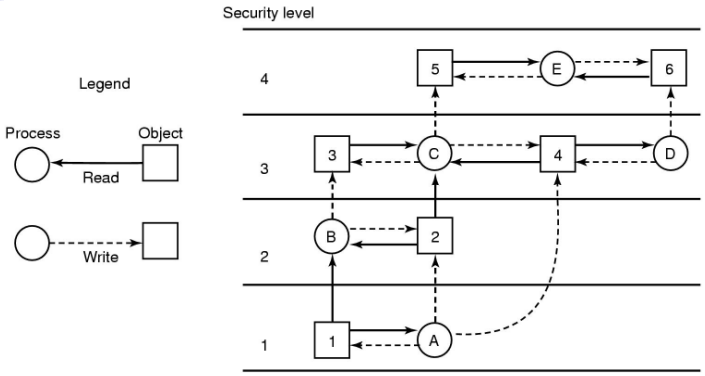
Rules for the Bell-La Padula model:

The simple security principle: A process running at security level k can read only objects at its level or lower.

The * property: A process running at security level k can write only objects at its level or higher.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

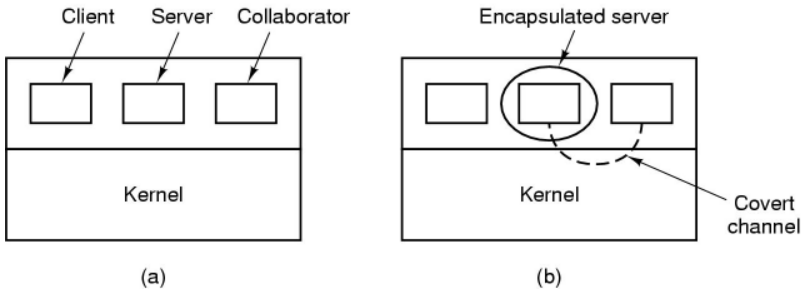
Information Flow



The Bell-La Padula multilevel security model.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Covert Channels



(a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

The BIBA Model

Rules for the Biba model:

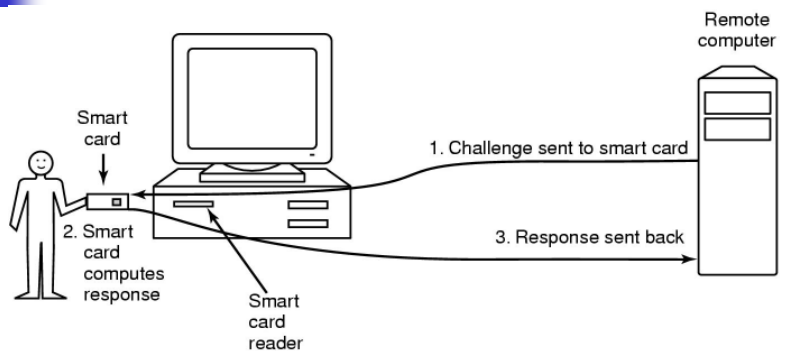
The simple integrity principle: A process running at security level k can write only objects at its level or lower (no write up).

The integrity * property: A process running at security level k can read only objects at its level or higher (no read down).

What does this guarantee?

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Authentication Using A Physical Object



Use of a smart card for authentication.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Authentication Using Biometrics



Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Insider Attacks on Authentication: Trap Doors

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
        printf("login: ");
        get_string(name);
        disable_echoing( );
        printf("password: ");
        get_string(password);
        enable_echoing( );
        v = check_validity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

(b)

(a) Normal code. (b) Code with a trap door inserted.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

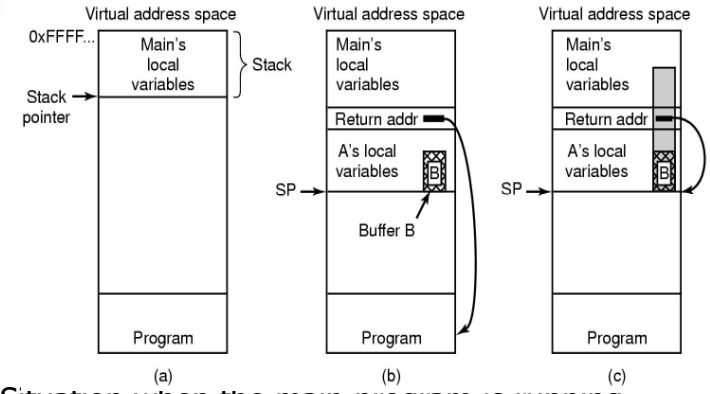
Phishing Attacks and Login Spoofing



(a) Correct login screen. (b) Phony login screen.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Other Attacks to Gain Unauthorized Access:
Buffer Overflow Attacks



(a) Situation when the main program is running.
(b) After the procedure A has been called.
(c) Buffer overflow shown in gray.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Attacks to Prevent Authorized Access:
Denial of Service Attacks

- Attempt to overload the machine (typically with processing of incoming traffic)
- Solution:
  - Network-level: Change the paradigm from push-based to pull-based
  - OS-level: Perform early de-multiplexing and drop unsolicited traffic