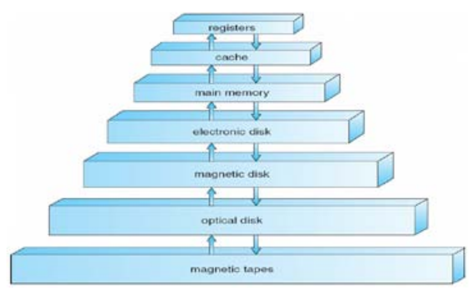
Study Guide to Accompany *Operating Systems Concepts 9th Ed* by Silberschatz, Galvin and Gagne

By Andrew DeNicola, BU ECE Class of 2012

Ch.1 - Introduction

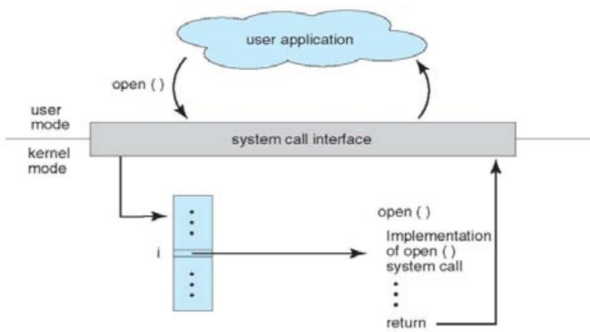
- An OS is a program that acts as an intermediary between a user of a computer and the computer hardware
- Goals: Execute user programs, make the comp. system easy to use, utilize hardware efficiently
- Computer system: Hardware ↔ OS ↔ Applications ↔ Users (↔ = 'uses')
- OS is:
  - Resource allocator: decides between conflicting requests for efficient and fair resource use
  - Control program: controls execution of programs to prevent errors and improper use of computer
- Kernel: the one program running at all times on the computer
- Bootstrap program: loaded at power-up or reboot
  - Stored in ROM or EPROM (known as firmware), Initializes all aspects of system, loads OS kernel and starts execution
- I/O and CPU can execute concurrently
- Device controllers inform CPU that it is finished w/ operation by causing an interrupt
  - Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
  - Incoming interrupts are disabled while another interrupt is being processed
  - Trap is a software generated interrupt caused by error or user request
  - OS determines which type of interrupt has occurred by polling or the vectored interrupt system
- System call: request to the operating system to allow user to wait for I/O completion
- Device-status table: contains entry for each I/O device indicating its type, address, and state
  - OS indexes into the I/O device table to determine device status and to modify the table entry to include interrupt
- Storage structure:
  - Main memory – random access, volatile
  - Secondary storage – extension of main memory That provides large non-volatile storage
  - Disk – divided into tracks which are subdivided into sectors. Disk controller determines logical interaction between the device and the computer.
- Caching – copying information into faster storage system



- Multiprocessor Systems: Increased throughput, economy of scale, increased reliability
  - Can be asymmetric or symmetric
  - Clustered systems – Linked multiprocessor systems
- Multiprogramming – Provides efficiency via job scheduling
  - When OS has to wait (ex: for I/O), switches to another job
- Timesharing – CPU switches jobs so frequently that each user can interact with each job while it is running (interactive computing)
- Dual-mode operation allows OS to protect itself and other system components – User mode and kernel mode
  - Some instructions are only executable in kernel mode, these are privileged
- Single-threaded processes have one program counter, multi-threaded processes have one PC per thread
- Protection – mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of a system against attacks
- User IDs (UID), one per user, and Group IDs, determine which users and groups of users have which privileges

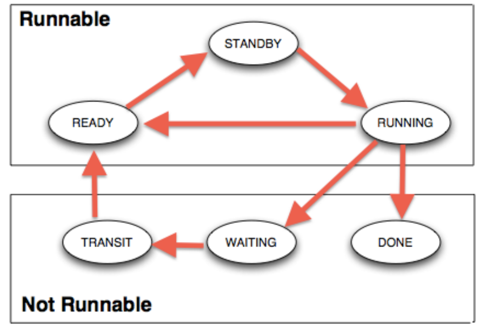
Ch.2 – OS Structures

- User Interface (UI) – Can be Command-Line (CLI) or Graphics User Interface (GUI) or Batch
  - These allow for the user to interact with the system services via system calls (typically written in C/C++)
- Other system services that a helpful to the user include: program execution, I/O operations, file-system manipulation, communications, and error detection
- Services that exist to ensure efficient OS operation are: resource allocation, accounting, protection and security
- Most system calls are accessed by Application Program Interface (API) such as Win32, POSIX, Java
- Usually there is a number associated with each system call
  - System call interface maintains a table indexed according to these numbers
- Parameters may need to be passed to the OS during a system call, may be done by:
  - Passing in registers, address of parameter stored in a block, pushed onto the stack by the program and popped off by the OS
  - Block and stack methods do not limit the number or length of parameters being passed

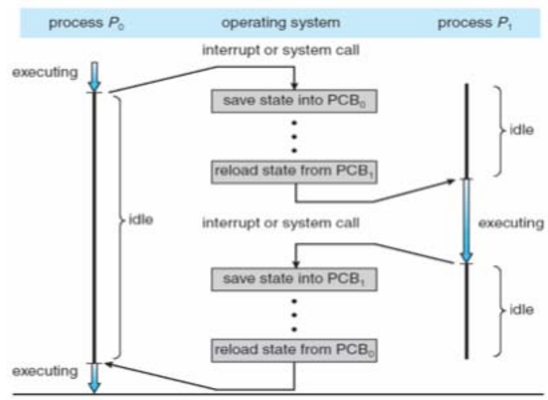


- Process control system calls include: end, abort, load, execute, create/terminate process, wait, allocate/free memory
- File management system calls include: create/delete file, open/close file, read, write, get/set attributes
- Device management system calls: request/release device, read, write, logically attach/detach devices
- Information maintenance system calls: get/set time, get/set system data, get/set process/file/device attributes
- Communications system calls: create/delete communication connection, send/receive, transfer status information
- OS Layered approach:
  - The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
  - With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
- Virtual machine: uses layered approach, treats hardware and the OS kernel as though they were all hardware.
  - Host creates the illusion that a process has its own processor and own virtual memory
  - Each guest provided with a 'virtual' copy of the underlying computer
- Application failures can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory

Ch.3 – Processes



- Process contains a program counter, stack, and data section.
  - Text section: program code itself
  - Stack: temporary data (function parameters, return addresses, local variables)
  - Data section: global variables
  - Heap: contains memory dynamically allocated during run-time
- Process Control Block (PCB): contains information associated with each process: process state, PC, CPU registers, scheduling information, accounting information, I/O status information
- Types of processes:
  - I/O Bound: spends more time doing I/O than computations, many short CPU bursts
  - CPU Bound: spends more time doing computations, few very long CPU bursts
- When CPU switches to another process, the system must save the state of the old process (to PCB) and load the saved state (from PCB) for the new process via a context switch
  - Time of a context switch is dependent on hardware
- Parent processes create children processes (form a tree)
  - PID allows for process management
  - Parents and children can share all/some/none resources
  - Parents can execute concurrently with children or wait until children terminate
  - fork() system call creates new process
    - exec() system call used after a fork to replace the processes' memory space with a new program
- Cooperating processes need interprocess communication (IPC): shared memory or message passing
- Message passing may be blocking or non-blocking
  - Blocking is considered synchronous
    - Blocking send has the sender block until the message is received
    - Blocking receive has the receiver block until a message is available
  - Non-blocking is considered asynchronous
    - Non-blocking send has the sender send the message and continue
    - Non-blocking receive has the receiver receive a valid message or null



Ch.4 – Threads

- Threads are fundamental unit of CPU utilization that forms the basis of multi-threaded computer systems
- Process creation is heavy-weight while thread creation is light-weight
  - Can simplify code and increase efficiency
- Kernels are generally multi-threaded
- Multi-threading models include: Many-to-One, One-to-One, Many-to-Many
  - Many-to-One: Many user-level threads mapped to single kernel thread
  - One-to-One: Each user-level thread maps to kernel thread
  - Many-to-Many: Many user-level threads mapped to many kernel threads
- Thread library provides programmer with API for creating and managing threads
- Issues include: thread cancellation, signal handling (synchronous/asynchronous), handling thread-specific data, and scheduler activations.
  - Cancellation:
    - Asynchronous cancellation terminates the target thread immediately
    - Deferred cancellation allows the target thread to periodically check if it should be canceled
  - Signal handler processes signals generated by a particular event, delivered to a process, handled
  - Scheduler activations provide upcalls – a communication mechanism from the kernel to the thread library.
    - Allows application to maintain the correct number of kernel threads

Ch.5 – Process Synchronization

- Race Condition: several processes access and manipulate the same data concurrently, outcome depends on which order each access takes place.
- Each process has critical section of code, where it is manipulating data
  - To solve critical section problem each process must ask permission to enter critical section in entry section, follow critical section with exit section and then execute the remainder section
  - Especially difficult to solve this problem in preemptive kernels
- Peterson's Solution: solution for two processes
  - Two processes share two variables: `int` **`turn`** and `Boolean` **`flag[2]`**
  - **`turn:`** whose turn it is to enter the critical section
  - **`flag:`** indication of whether or not a process is ready to enter critical section
    - `flag[i] = true` indicates that process P_i is ready
  - Algorithm for process P_i:

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                critical section
        flag[i] = FALSE;
        remainder section
} while (TRUE);
```

- Modern machines provide atomic hardware instructions: Atomic = non-interruptable
- Solution using Locks:

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

- Solution using Test-And-Set: Shared boolean variable lock, initialized to FALSE

```
boolean TestAndSet (boolean *target){
        boolean rv = *target;
        *target = TRUE;"
        return rv:
}
```

```
do {
        while ( TestAndSet (&lock ))
                        ; // do
nothing
        // critical section
        lock = FALSE;
        // remainder section
} while (TRUE);
```

- Solution using Swap: Shared bool variable lock initialized to FALSE; Each process has local bool variable key

```
void Swap (boolean *a, boolean *b){
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

```
do {
        key = TRUE;
        while ( key == TRUE)
                        Swap (&lock,
&key );
        // critical section
        lock = FALSE;
        // remainder section
} while (TRUE);
```

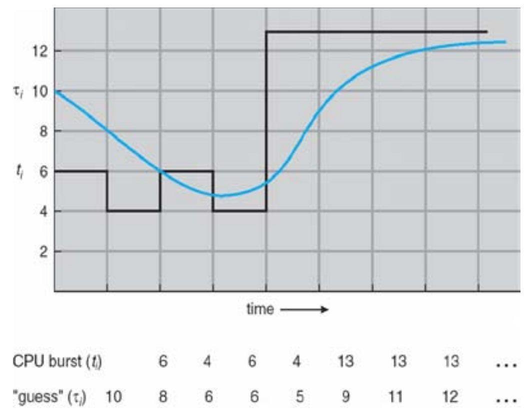
- Semaphore: Synchronization tool that does not require busy waiting
  - Standard operations: wait() and signal() ← these are the only operations that can access semaphore S
  - Can have counting (unrestricted range) and binary (0 or 1) semaphores
- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (most OSes do not prevent or deal with deadlocks)
  - Can cause starvation and priority inversion (lower priority process holds lock needed by higher-priority process)

Ch.5 – Process Synchronization Continued

- Other synchronization problems include Bounded-Buffer Problem and Readers-Writers Problem
- Monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
    - Only one process may be active within the monitor at a time
    - Can utilize condition variables to suspend a resume processes (ex: condition x, y;)
        - x.wait() – a process that invokes the operation is suspended until x.signal()
        - x.signal() – resumes one of processes (if any) that invoked x.wait()
    - Can be implemented with semaphores

Ch.6 – CPU Scheduling

- Process execution consists of a cycle of CPU execution and I/O wait
- CPU scheduling decisions take place when a process:
  - Switches from running to waiting (nonpreemptive)
  - Switches from running to ready (preemptive)
  - Switches from waiting to ready (preemptive)
  - Terminates (nonpreemptive)
- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler
  - Dispatch latency- the time it takes for the dispatcher to stop one process and start another
- Scheduling algorithms are chosen based on optimization criteria (ex: throughput, turnaround time, etc.)
  - FCFS, SJF, Shortest-Remaining-Time-First (preemptive SJF), Round Robin, Priority
- Determining length of next CPU burst: Exponential Averaging:
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \le \alpha \le 1$ (commonly α set to 1/2)
  4. Define: $\tau_{n+1} = \alpha * t_n + (1-\alpha)\tau_n$



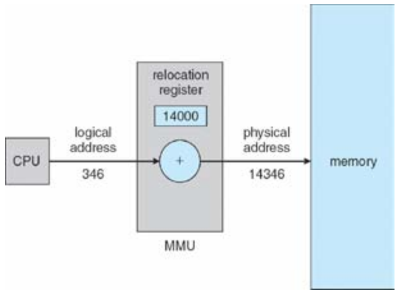
- Priority Scheduling can result in starvation, which can be solved by aging a process (as time progresses, increase the priority)
- In Round Robin, small time quantums can result in large amounts of context switches
  - Time quantum should be chosen so that 80% of processes have shorter burst times that the time quantum
- Multilevel Queues and Multilevel Feedback Queues have multiple process queues that have different priority levels
  - In the Feedback queue, priority is not fixed → Processes can be promoted and demoted to different queues
  - Feedback queues can have different scheduling algorithms at different levels
- Multiprocessor Scheduling is done in several different ways:
  - Asymmetric multiprocessing: only one processor accesses system data structures → no need to data share
  - Symmetric multiprocessing: each processor is self-scheduling (currently the most common method)
  - Processor affinity: a process running on one processor is more likely to continue to run on the same processor (so that the processor's memory still contains data specific to that specific process)
- Little's Formula can help determine average wait time per process in any scheduling algorithm:
  - $n = \lambda \times W$
  - n = avg queue length; W = avg waiting time in queue; λ = average arrival rate into queue
- Simulations are programmed models of a computer system with variable clocks
  - Used to gather statistics indicating algorithm performance
  - Running simulations is more accurate than queuing models (like Little's Law)
  - Although more accurate, high cost and high risk

Ch.7 – Deadlocks

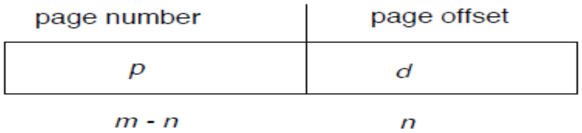
- Deadlock Characteristics: deadlock can occur if these conditions hold simultaneously
  - Mutual Exclusion: only one process at a time can use a resource
  - Hold and Wait: process holding one resource is waiting to acquire resource held by another process
  - No Preemption: a resource can be released only be the process holding it after the process completed its task
  - Circular Wait: set of waiting processes such that P_{n-1} is waiting for resource from P_n, and P_n is waiting for P_0
    - "Dining Philosophers" in deadlock

Ch.8 – Main Memory

- Cache sits between main memory and CPU registers
- Base and limit registers define logical address space usable by a process
- Compiled code addresses bind to relocatable addresses
  - Can happen at three different stages
    - Compile time: If memory location known a priori, absolute code can be generated
    - Load time: Must generate relocatable code if memory location not known at compile time
    - Execution time: Binding delayed until run time if the process can be moved during its execution
- Memory-Management Unit (MMU) device that maps virtual to physical address
- Simple scheme uses a relocation register which just adds a base value to address
- Swapping allows total physical memory space of processes to exceed physical memory
  - Def: process swapped out temporarily to backing store then brought back in for continued execution
- Backing store: fast disk large enough to accommodate copes of all memory images
- Roll out, roll in: swapping variant for priority-based scheduling.
  - Lower priority process swapped out so that higher priority process can be loaded



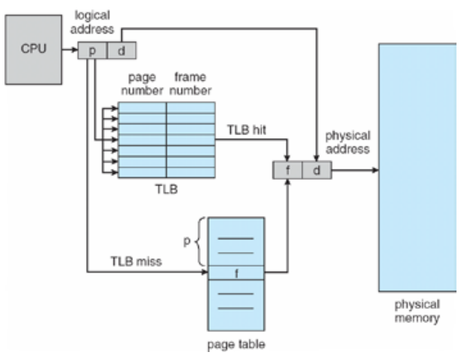
- Solutions to Dynamic Storage-Allocation Problem:
  - First-fit: allocate the first hole that is big enough
  - Best-fit: allocate the smallest hole that is big enough (must search entire list)→ smallest leftover hole
  - Worst-fit: allocate the largest hole (search entire list) → largest leftover hole
- External Fragmentation: total memory space exists to satisfy request, but is not contiguous
  - Reduced by compaction: relocate free memory to be together in one block
    - Only possible if relocation is dynamic
- Internal Fragmentation: allocated memory may be slightly larger than requested memory
- Physical memory divided into fixed-sized frames: size is power of 2, between 512 bytes and 16 MB
- Logical memory divided into same sized blocks: pages
- Page table used to translate logical to physical addresses
  - Page number (p): used as an index into a page table
  - Page offset (d): combined with base address to define the physical memory address
- Free-frame list is maintained to keep track of which frames can be allocated



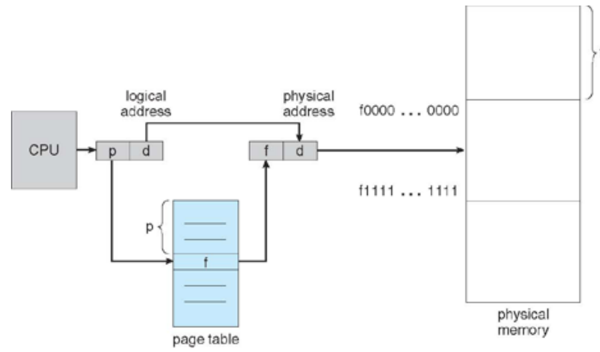
For given logical address space 2^m and page size 2^n

Ch.8 – Main Memory Continued

- Transition Look-aside Buffer (TLB) is a CPU cache that memory management hardware uses to improve virtual address translation speed
  - Typically small – 64 to 1024 entries
  - On TLB miss, value loaded to TLB for faster access next time
  - TLB is associative – searched in parallel

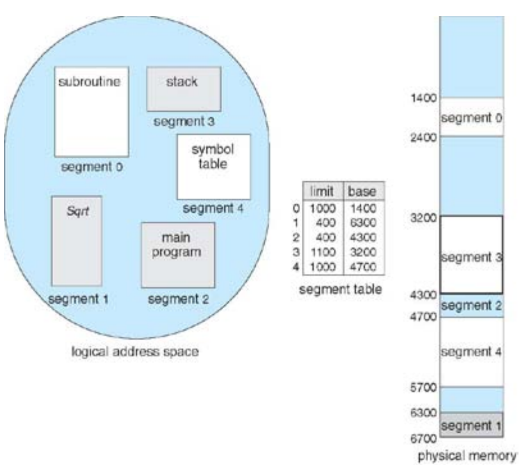


Paging with TLB

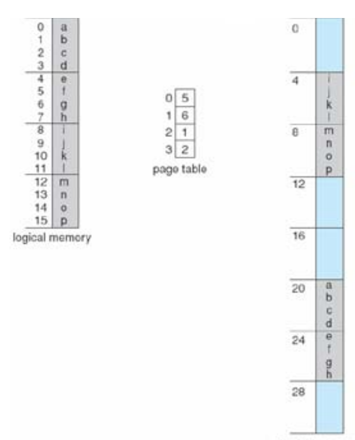


Paging without TLB

- Effective Access Time: EAT = $(1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$
  - ε = time unit, α = hit ratio
- Valid and invalid bits can be used to protect memory
  - "Valid" if the associated page is in the process' logical address space, so it is a legal page
- Can have multilevel page tables (paged page tables)
- Hashed Page Tables: virtual page number hashed into page table
  - Page table has chain of elements hashing to the same location
  - Each element has (1) virtual page number, (2) value of mapped page frame, (3) a pointer to the next element
  - Search through the chain for virtual page number
- Segment table – maps two-dimensional physical addresses
  - Entries protected with valid bits and r/w/x privileges



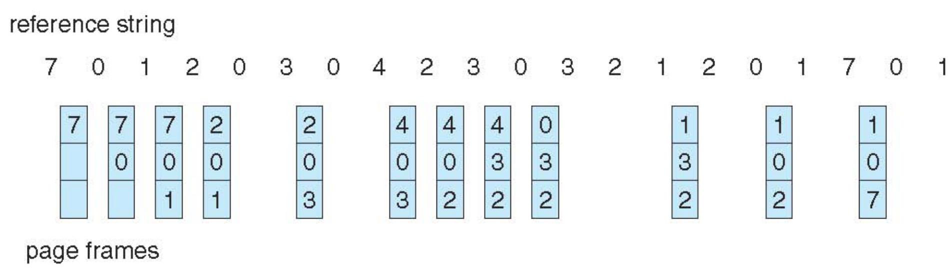
Segmentation example



Page table example

Ch.9 – Virtual Memory

- Virtual memory: separation of user logical memory and physical memory
  - Only part of program needs to be in memory for execution → logical address space > physical address space
  - Allows address spaces to be shared by multiple processes → less swapping
  - Allows pages to be shared during fork(), speeding process creation
- Page fault results from the first time there is a reference to a specific page → traps the OS
  - Must decide to abort if the reference is invalid, or if the desired page is just not in memory yet
    - If the latter: get empty frame, swap page into frame, reset tables to indicate page now in memory, set validation bit, restart instruction that caused the page fault
  - If an instruction accesses multiple pages near each other → less "pain" because of locality of reference
- Demand Paging only brings a page into memory when it is needed → less I/O and memory needed
  - Lazy swapper – never swaps a page into memory unless page will be needed
  - Could result in a lot of page-faults
  - Performance: EAT = [(1-p)*memory access + p*(page fault overhead + swap page out + swap page in + restart overhead)]; where Page Fault Rate 0 ″ p ″ 1
    - if p = 0, no page faults; if p = 1, every reference is a fault
  - Can optimize demand paging by loading entire process image to swap space at process load time
- Pure Demand Paging: process starts with no pages in memory
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- Modify (dirty) bit can be used to reduce overhead of page transfers → only modified pages written to disk
- When a page is replaced, write to disk if it has been marked dirty and swap in desired page
- Pages can be replaced using different algorithms: FIFO, LRU (below)
  - Stack can be used to record the most recent page references (LRU is a "stack" algorithm)



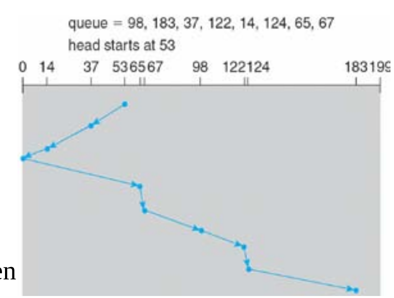
  - Second chance algorithm uses a reference bit
    - If 1, decrement and leave in memory
    - If 0, replace next page
- Fixed page allocation: Proportional allocation – Allocate according to size of process
  - s_i = size of process P_i, $S = \Sigma s_i$, m = total number of frames, a_i – allocation for P_i
  - $a_i = (s_i/S)*m$
- Global replacement: process selects a replacement frame from set of all frames
  - One process can take frame from another
  - Process execution time can vary greatly
  - Greater throughput
- Local replacement: each process selects from only its own set of allocated frames
  - More consistent performance
  - Possible under-utilization of memory
- Page-fault rate is very high if a process does not have "enough" pages
  - Thrashing: a process is busy swapping pages in and out → minimal work is actually being performed
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page

in memory

- I/O Interlock: Pages must sometimes be locked into memory

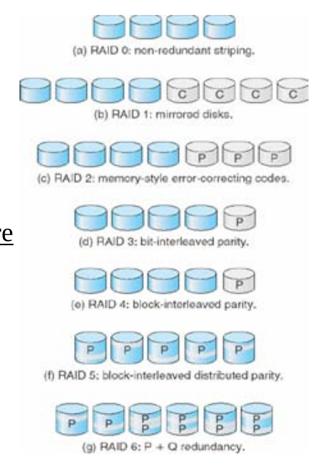
Ch.10 – Mass-Storage Systems

- Magnetic disks provide bulk of secondary storage – rotate at 60 to 250 times per second
    - Transfer rate: rate at which data flows between drive and computer
    - Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency)
    - Head crash: disk head making contact with disk surface
- Drive attached to computer's I/O bus – EIDE, ATA, SATA, USB, etc.
    - Host controller uses bus to talk to disk controller
- Access latency = Average access time = average seek time + average latency (fast ~5ms, slow ~14.5ms)
- Average I/O time = avg. access time + (amount to transfer / transfer rate) + controller overhead
    - Ex: to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead = 5ms + 4.17ms + 4KB / 1Gb/sec + 0.1ms = 9.27ms + .12ms = 9.39ms
- Disk drives addressed as 1-dimensional arrays of logical blocks
    - 1-dimensional array is mapped into the sectors of the disk sequentially
- Host-attached storage accessed through I/O ports talking to I/O buses
    - Storage area network (SAN): many hosts attach to many storage units, common in large storage environments
        - Storage made available via LUN masking from specific arrays to specific servers
- Network attached storage (NAS): storage made available over a network rather than local connection
- In disk scheduling, want to minimize seek time; Seek time is proportional to seek distance
- Bandwidth is (total number of bytes transferred) / (total time between first request and completion of last transfer)
- Sources of disk I/O requests: OS, system processes, user processes
    - OS maintains queue of requests, per disk or device
- Several algorithms exist to schedule the servicing of disk I/O requests
    - FCFS, SSTF (shortest seek time first), SCAN, CSCAN, LOOK, CLOOK
        - SCAN/elevator: arm starts at one end and moves towards other end servicing requests as it goes, then reverses direction
        - CSCAN: instead of reversing direction, immediately goes back to beginning
        - LOOK/CLOOK: Arm only goes as far as the last request in each directions, then reverses immediately



SCAN

- Low level/physical formatting: dividing a disk into sectors that the disk controller can read and write – usually 512 bytes of data
- Partition: divide disk into one or more groups of cylinders, each treated as logical disk
- Logical formatting: "making a file system"
- Increase efficiency by grouping blocks into clusters - Disk I/O is performed on blocks
    - Boot block initializes system - bootstrap loader stored in boot block
- Swap-space: virtual memory uses disk space as an extension of main memory
    - Kernel uses swap maps to track swap space use
- RAID: Multiple disk drives provide reliability via redundancy – increases mean time to failure
    - Disk striping uses group of disks as one storage unit
    - Mirroring/shadowing (RAID 1) – keeps duplicate of each disk
    - Striped mirrors (RAID 1+0) or mirrored striped (RAID 0+1) provides high performance/reliability
    - Block interleaved parity (RAID 4, 5, 6) uses much less redundancy

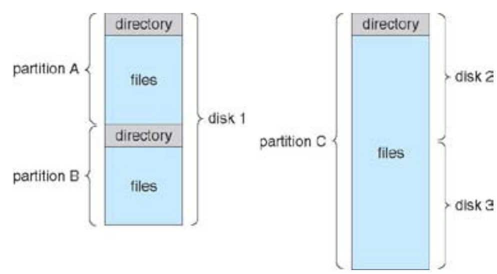


- Solaris ZFS adds checksums of all data and metadata – detect if object is the right one and whether it changed
- Tertiary storage is usually built using removable media – can be WORM or Read-only, handled like fixed disks
- Fixed disk usually more reliable than removable disk or tape drive
- Main memory is much more expensive than disk storage

Ch.11 – File-System Interface

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

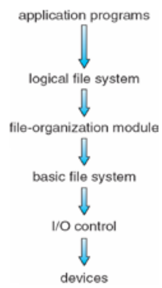
- File – Uniform logical view of information storage (no matter the medium)
  - Mapped onto physical devices (usually nonvolatile)
  - Smallest allotment of nameable storage
  - Types: Data (numeric, character, binary), Program, Free form, Structured
  - Structure decided by OS and/or program/programmer
- Attributes:
  - Name: Only info in human-readable form
  - Identifier: Unique tag, identifies file within the file system
  - Type, Size
  - Location: pointer to file location
  - Time, date, user identification
- File is an abstract data type
- Operations: create, write, read, reposition within file, delete, truncate
- Global table maintained containing process-independent open file information: open-file table
  - Per-process open file table contains pertinent info, plus pointer to entry in global open file table
- Open file locking: mediates access to a file (shared or exclusive)
  - Mandatory – access denied depending on locks held and requested
  - Advisory – process can find status of locks and decide what to do
- File type can indicate internal file structure
- Access Methods: Sequential access, direct access
  - Sequential Access: tape model of a file
  - Direct Access: random access, relative access
- Disk can be subdivided into partitions; disks or partitions can be RAID protected against failure.



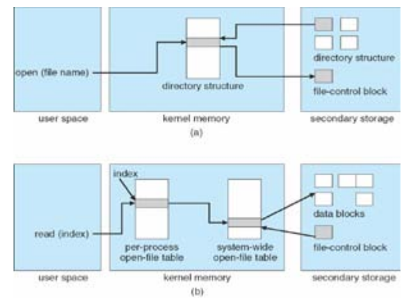
File-System Organization

  - Can be used raw without a file-system or formatted with a file system
  - Partitions also knows as minidisks, slices
- Volume contains file system: also tracks file system's info in device directory or volume table of contents
- File system can be general or special-purpose. Some special purpose FS:
  - tmpfs – temporary file system in volatile memory
  - objfs – virtual file system that gives debuggers access to kernel symbols
  - ctfs – virtual file system that maintains info to manage which processes start when system boots
  - lofs – loop back file system allows one file system to be accessed in place of another
  - procfs – virtual file system that presents information on all processes as a file system
- Directory is similar to symbol table – translating file names into their directory entries
  - Should be efficient, convenient to users, logical grouping
  - Tree structured is most popular – allows for grouping
  - Commands for manipulating: remove – rm<file-name> ; make new sub directory - mkdir<dir-name>
- Current directory: default location for activities – can also specify a path to perform activities in
- Acyclic-graph directories adds ability to directly share directories between users
  - Acyclic can be guaranteed by: only allowing shared files, not shared sub directories; garbage collection; mechanism to check whether new links are OK
- File system must be mounted before it can be accessed – kernel data structure keeps track of mount points
- In a file sharing system User IDs and Group IDs help identify a user's permissions
- Client-server allows multiple clients to mount remote file systems from servers – NFS (UNIX), CIFS (Windows)
- Consistency semantics specify how multiple users are to access a shared file simultaneously – similar to synchronization algorithms from Ch.7
  - One way of protection is Controlled Access: when file created, determine r/w/x access for users/groups

Ch.12 – File System Implementation



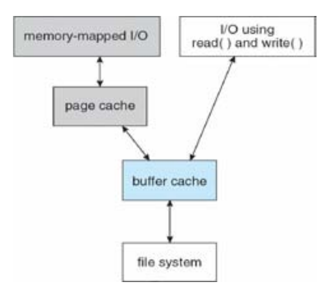
- File system resides on secondary storage – disks; file system is organized into layers →
- File control block: storage structure consisting of information about a file (exist per-file)
- Device driver: controls the physical device; manage I/O devices
- File organization module: understands files, logical addresses, and physical blocks
  - Translates logical block number to physical block number
  - Manages free space, disk allocation
- Logical file system: manages metadata information – maintains file control blocks
- Boot control block: contains info needed by system to boot OS from volume
- Volume control block: contains volume details; ex: total # blocks, # free blocks, block size, free block pointers
- Root partition: contains OS; mounted at boot time
- For all partitions, system is consistency checked at mount time
  - Check metadata for correctness – only allow mount to occur if so
- Virtual file systems provide object-oriented way of implementing file systems
- Directories can be implemented as Linear Lists or Hash Tables
  - Linear list of file names with pointer to data blocks – simple but slow
  - Hash table – linear list with hash data structure – decreased search time
    - Good if entries are fixed size
    - Collisions can occur in hash tables when two file names hash to same location



(a) open() ***(b) read()***

- Contiguous allocation: each file occupies set of contiguous blocks
  - Simple, best performance in most cases; problem – finding space for file, external fragmentation
  - Extent based file systems are modified contiguous allocation schemes – extent is allocated for file allocation
- Linked Allocation: each file is a linked list of blocks – no external fragmentation
  - Locating a block can take many I/Os and disk seeks
- Indexed Allocation: each file has its own index block(s) of pointers to its data blocks
  - Need index table; can be random access; dynamic access without external fragmentation but has overhead
- Best methods: linked good for sequential, not random; contiguous good for sequential and random
- File system maintains free-space list to track available blocks/clusters
- Bit vector or bit map (n blocks): block number calculation → (#bits/word)*(# 0-value words)+(offset for 1st bit)
-
  - Example:

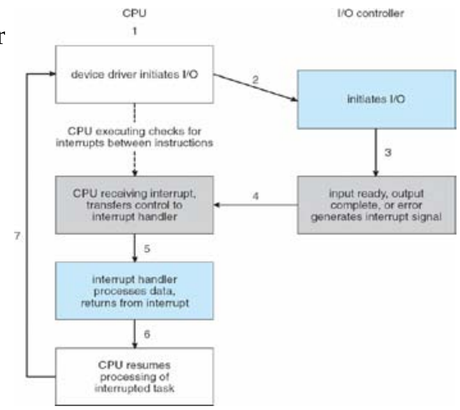
    block size = 4KB = 212 bytes
    disk size = 240 bytes (1 terabyte)
    n = 240/212 = 228 bits (or 256 MB)
    if clusters of 4 blocks -> 64MB of memory



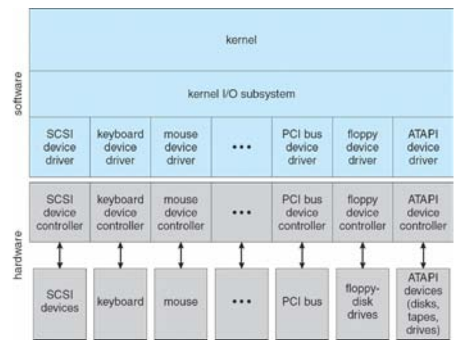
- Space maps (used in ZFS) divide device space into metaslab units and manages metaslabs
  - Each metaslab has associated space map
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS – no buffering
-
  - Asynchronous writes are more common, buffer-able, faster
- Free-behind and read-ahead techniques to optimize sequential access
- Page cache caches pages rather than disk blocks using virtual memory techniques and addresses
  - Memory mapped I/O uses page cache while routine I/O through file system uses buffer (disk) cache
- Unified buffer cache: uses same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching

Ch.13 – I/O Systems

- Device drivers encapsulate device details – present uniform device access interface to I/O subsystem
- Port: connection point for device
- Bus: daisy chain or shared direct access
- Controller (host adapter): electronics that operate port, bus, device – sometimes integrated
  - Contains processor, microcode, private memory, bus controller



- Memory-mapped I/O: device data and command registers mapped to processor address space
  - Especially for large address spaces (graphics)
- Polling for each byte of data – busy-wait for I/O from device
  - Reasonable for fast devices, inefficient for slow ones
  - Can happen in 3 instruction cycles
- CPU interrupt-request line is triggered by I/O devices – interrupt handler receives interrupts
  - Handler is maskable to ignore or delay some interrupts
  - Interrupt vector dispatches interrupt to correct handler – based on priority; some nonmaskable
  - Interrupt chaining occurs if there is more than one device at the same interrupt number
  - Interrupt mechanism is also used for exceptions



- Direct memory access is used to avoid programmed I/O for large data movement
  - Requires DMA controller
  - Bypasses CPU to transfer data directly between I/O device and memory
- Device driver layer hides differences among I/O controllers from kernel
- Devices vary in many dimensions: character stream/block, sequential/random access, synchronous/asynchronous, sharable/dedicated, speed, rw/ro/wo
- Block devices include disk drives: Raw I/O, Direct I/OU
  - Commands include read, write, seek
- Character devices include keyboards, mice, serial ports
  - Commands include get(), put()
- Network devices also have their own interface; UNIX and Windows NT/9x/2000 include socket interface
  - Approaches include pipes, FIFOs, streams, queues, mailboxes
- Programmable interval timer: used for timings, periodic interrupts
- Blocking I/O: process suspended until I/O completed – easy to use and understand, not always best method
- Nonblocking I/O: I/O call returns as much as available – implemented via multi-threading, returns quickly
- Asynchronous: process runs while I/O executes – difficult to use, process signaled upon I/O completion
- Spooling: hold output for a device – if device can only serve one request at a time (ex: printer)
- Device Reservation: provides exclusive access to a device – must be careful of deadlock
- Kernel keeps state info for I/O components, including open file tables, network connections, character device states
  - Complex data structures track buffers, memory allocation, "dirty" blocks
- STREAM: full-duplex communication channel between user-level process and device in UNIX
  - Each module contains read queue and write queue
  - Message passing used to communicate between queues – Flow control option to indicate available or busy
  - Asynchronous internally, synchronous where user process communicates with stream head
- I/O is a major factor in system performance – demand on CPU, context switching, data copying, network traffic

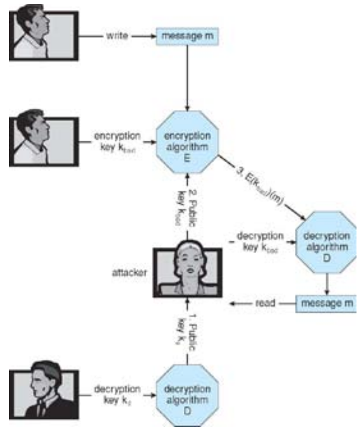
Ch.14 – Protection

- <u>Principle of least privilege</u>: programs, users, systems should be given just enough privileges to perform their tasks
- Access-right = <obj-name, rights-set> w/ rights-set is subset of all valid operations performable on the object
    - <u>Domain</u>: set of access-rights
        - UNIX system consists of 2 domains: user, supervisor
        - MULTICS domain implementation (domain rings) – if j<i → D_i □ D_j
- <u>Access matrix</u>: rows represent domains, columns represent objects
    - Access(i,j) is the set of operations that a process executing in Domain_i can invoke on Object_j
    - Can be expanded to dynamic protection

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- Access matrix design separates <u>mechanism</u> from <u>policy</u>
    - Mechanism: OS provides access-matrix and rules – ensures matrix is only manipulated by authorized users
    - Policy: User dictates policy – who can access what object and in what mode
- Solaris 10 uses <u>role-based access control (RBAC)</u> to implement least privilege
- <u>Revocation</u> of access rights
    - <u>Access list</u>: delete access rights from access list – simple, immediate
    - <u>Capability list</u>: required to locate capability in system before capability can be revoked – reacquisition, back-pointers, indirection, keys
- <u>Language-Based Protection</u>: allows high-level description of policies for the allocation and use of resources
    - Can provide software for protection enforcement when hardware-supported checking is unavailable

Ch.15 – Security

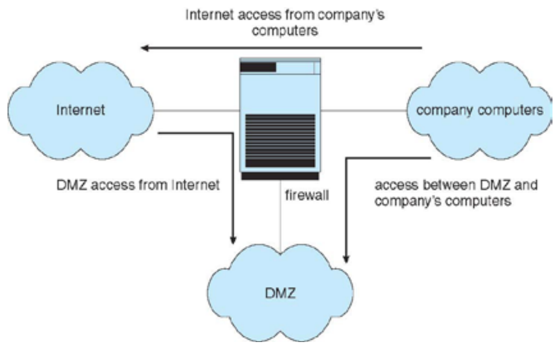


Man-in-the-middle attack - Asymmetric Cryptography

- System secure when resources used and accessed as intended under all circumstances
- Attacks can be accidental or malicious
    - Easier to protect against accidental than malicious misuse
- Security violation categories:
    - Breach of confidentiality – unauthorized reading of data
    - Breach of integrity – unauthorized modification of data
    - Breach of availability – unauthorized destruction of data
    - Theft of service – unauthorized use of resources
    - Denial of service – prevention of legitimate use
- Methods of violation:
    - Masquerading – pretending to be an authorized user
    - Man-in-the-middle – intruder sits in data flow, masquerading as sender to receiver and vice versa
    - Session hijacking – intercept and already established session to bypass authentication
- Effective security must occur at four levels: physical, human, operating system, network
- Program threats: trojan horse (spyware, pop-up, etc.), trap door, logic bomb, stack and buffer overflow
- Viruses: code fragment embedded in legitimate program; self-replicating
    - Specific to CPU architecture, OS, applications
    - Virus dropper: inserts virus onto the system
- Windows is the target for most attacks – most common, everyone is administrator
- Worms: use spawn mechanism – standalone program
- Port scanning: automated attempt to connect to a range of ports on one or a range of IP addresses
    - Frequently launched from zombie systems to decrease traceability
- Denial of service: overload targeted computer preventing it from doing useful work
- Cryptography: means to constrain potential senders and/or receivers – based on keys
    - Allows for confirmation of source, receipt by specified destination, trust relationship
- Encryption: [K of keys], [M of messages], [C of ciphertexts], function E:K to encrypt, function D:K to decrypt
    - Can have symmetric and asymmetric (distributes public encryption key, holds private decipher key) encryption
        - Asymmetric is much more compute intensive – not used for bulk data transaction
        - Keys can be stored on a key ring
- Authentication: constraining a set of potential senders of a message
    - Helps to prove that the message is unmodified
    - Hash functions are basis of authentication
        - Creates small, fixed-size block of data (message digest, hash value)
- Symmetric encryption used in message-authentication code (MAC)
- Authenticators produced from authentication algorithm are digital signatures
- Authentication requires fewer computations than encryption methods
- Digital Certificates: proof of who or what owns a public key
- Defense in depth: most common security theory – multiple layers of security
- Can attempt to detect intrusion:
    - Signature-based: detect "bad patterns"
    - Anomaly detection: spots differences from normal behavior
        - Both can report false positives or false negatives
    - Auditing, accounting, and logging specific system or network activity

Ch.15 – Security Continued

- Firewall: placed between trusted and untrusted hosts
    - Limits network access between the two domains
    - Can be tunneled or spoofed
- Personal firewall is software layer on given host
    - Can monitor/limit traffic to/from host
- Application proxy firewall: Understands application protocol and can control them
- System-call firewall: Monitors all important system calls and apply rules and restrictions to them



CHAPTER 1

Introduction

Practice Exercises

1.1 What are the three main purposes of an operating system?
Answer:
The three main puropses are:

- To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
- To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as fair and efficient as possible.
- As a control program it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and control of I/O devices.

1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?
Answer:
Single-user systems should maximize use of the system for the user. A GUI might "waste" CPU cycles, but it optimizes the user's interaction with the system.

1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?
Answer:
The main difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system it is running. Therefore when writing an operating system for a real-time system, the writer must be sure that his scheduling schemes don't allow response time to exceed the time constraint.

1.4 Keeping in mind the various definitions of **operating system,** consider whether the operating system should include applications such as Web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

Answer:

An argument in favor of including popular applications with the operating system is that if the application is embedded within the operating system, it is likely to be better able to take advantage of features in the kernel and therefore have performance advantages over an application that runs outside of the kernel. Arguments against embedding applications within the operating system typically dominate however: (1) the applications are applications - and not part of an operating system, (2) any performance benefits of running within the kernel are offset by security vulnerabilities, (3) it leads to a bloated operating system.

1.5 How does the distinction between kernel mode and user mode function as a rudimentary form of protection (security) system?

Answer:

The distinction between kernel mode and user mode provides a rudimentary form of protection in the following manner. Certain instructions could be executed only when the CPU is in kernel mode. Similarly, hardware devices could be accessed only when the program is executing in kernel mode. Control over when interrupts could be enabled or disabled is also possible only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode, thereby enforcing protection of critical resources.

1.6 Which of the following instructions should be privileged?

a. Set value of timer.

b. Read the clock.

c. Clear memory.

d. Issue a trap instruction.

e. Turn off interrupts.

f. Modify entries in device-status table.

g. Switch from user to kernel mode.

h. Access I/O device.

Answer:

The following operations need to be privileged: Set value of timer, clear memory, turn off interrupts, modify entries in device-status table, access I/O device. The rest can be performed in user mode.

1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

Answer:

The data required by the operating system (passwords, access controls, accounting information, and so on) would have to be stored in or passed through unprotected memory and thus be accessible to unauthorized users.

1.8 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

Answer:

Although most systems only distinguish between user and kernel modes, some CPUs have supported multiple modes. Multiple modes could be used to provide a finer-grained security policy. For example, rather than distinguishing between just user and kernel mode, you could distinguish between different types of user mode. Perhaps users belonging to the same group could execute each other's code. The machine would go into a specified mode when one of these users was running code. When the machine was in this mode, a member of the group could run code belonging to anyone else in the group.

Another possibility would be to provide different distinctions within kernel code. For example, a specific mode could allow USB device drivers to run. This would mean that USB devices could be serviced without having to switch to kernel mode, thereby essentially allowing USB device drivers to run in a quasi-user/kernel mode.

1.9 Timers could be used to compute the current time. Provide a short description of how this could be accomplished.

Answer:

A program could use the following approach to compute the current time using timer interrupts. The program could set a timer for some time in the future and go to sleep. When it is awakened by the interrupt, it could update its local state, which it is using to keep track of the number of interrupts it has received thus far. It could then repeat this process of continually setting timer interrupts and updating its local state when the interrupts are actually raised.

1.10 Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Answer:

Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

1.11 Distinguish between the client–server and peer-to-peer models of distributed systems.

Answer:

The client-server model firmly distinguishes the roles of the client and server. Under this model, the client requests services that are provided by the server. The peer-to-peer model doesn't have such strict roles. In fact, all nodes in the system are considered peers and thus may act as *either* clients or servers—or both. A node may request a service from another peer, or the node may in fact provide such a service to other peers in the system.

For example, let's consider a system of nodes that share cooking recipes. Under the client-server model, all recipes are stored with the server. If a client wishes to access a recipe, it must request the recipe from the specified server. Using the peer-to-peer model, a peer node could ask other peer nodes for the specified recipe. The node (or perhaps nodes) with the requested recipe could provide it to the requesting node. Notice how each peer may act as both a client (it may request recipes) and as a server (it may provide recipes).

Operating System Structures

CHAPTER 2

Practice Exercises

2.1 What is the purpose of system calls?
Answer:
System calls allow user-level processes to request services of the operating system.

2.2 What are the five major activities of an operating system with regard to process management?
Answer:
The five major activities are:

a. The creation and deletion of both user and system processes

b. The suspension and resumption of processes

c. The provision of mechanisms for process synchronization

d. The provision of mechanisms for process communication

e. The provision of mechanisms for deadlock handling

2.3 What are the three major activities of an operating system with regard to memory management?
Answer:
The three major activities are:

a. Keep track of which parts of memory are currently being used and by whom.

b. Decide which processes are to be loaded into memory when memory space becomes available.

c. Allocate and deallocate memory space as needed.

2.4 What are the three major activities of an operating system with regard to secondary-storage management?
Answer:
The three major activities are:

- Free-space management.
- Storage allocation.
- Disk scheduling.

2.5 What is the purpose of the command interpreter? Why is it usually separate from the kernel?
Answer:
It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel since the command interpreter is subject to changes.

2.6 What system calls have to be executed by a command interpreter or shell in order to start a new process?
Answer:
In Unix systems, a *fork* system call followed by an *exec* system call need to be performed to start a new process. The *fork* call clones the currently executing process, while the *exec* call overlays a new process based on a different executable over the calling process.

2.7 What is the purpose of system programs?
Answer:
System programs can be thought of as bundles of useful system calls. They provide basic functionality to users so that users do not need to write their own programs to solve common problems.

2.8 What is the main advantage of the layered approach to system design? What are the disadvantages of using the layered approach?
Answer:
As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

2.9 List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.
Answer:
The five services are:

a. **Program execution**. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.

b. **I/O operations**. Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to access only devices they should have access to and to access them only when they are otherwise unused.

c. **File-system manipulation**. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.

d. **Communications**. Message passing between systems requires messages to be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

e. **Error detection**. Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media must be checked for data consistency; for instance, whether the number of allocated and unallocated blocks of storage match the total number on the device. There, errors are frequently process-independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles all types of errors. Also, by having errors processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

2.10 Why do some systems store the operating system in firmware, while others store it on disk?

Answer:

For certain devices, such as handheld PDAs and cellular telephones, a disk with a file system may be not be available for the device. In this situation, the operating system must be stored in firmware.

2.11 How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Answer:

Consider a system that would like to run both Windows XP and three different distributions of Linux (e.g., RedHat, Debian, and Mandrake). Each operating system will be stored on disk. During system boot-up, a special program (which we will call the **boot manager**) will determine which operating system to boot into. This means that rather initially booting to an operating system, the boot manager will first run during system startup. It is this boot manager that is responsible for determining which system to boot into. Typically boot managers must be stored at certain locations of the hard disk to be recognized during system startup. Boot managers often provide the user with a selection of systems to boot into; boot managers are also typically designed to boot into a default operating system if no choice is selected by the user.

CHAPTER 3

Processes

Practice Exercises

3.1 Using the program shown in Figure 3.30, explain what the output will be at `Line A`.

Answer:

The result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5.

3.2 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

Answer:

There are 8 processes created.

3.3 Original versions of Apple's mobile iOS operating system provided no means of concurrent processing. Discuss three major complications that concurrent processing adds to an operating system.

Answer: FILL

3.4 The Sun UltraSPARC processor has multiple register sets. Describe what happens when a context switch occurs if the new context is already loaded into one of the register sets. What happens if the new context is in memory rather than in a register set and all the register sets are in use?

Answer:

The CPU current-register-set pointer is changed to point to the set containing the new context, which takes very little time. If the context is in memory, one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. This process takes a little more time than on systems with one set of registers, depending on how a replacement victim is selected.

3.5 When a process creates a new process using the fork() operation, which of the following state is shared between the parent process and the child process?

a. Stack

b. Heap

c. Shared memory segments

Answer:

Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

3.6 With respect to the RPC mechanism, consider the "exactly once" semantic. Does the algorithm for implementing this semantic execute correctly even if the ACK message back to the client is lost due to a network problem? Describe the sequence of messages and discuss whether "exactly once" is still preserved.

Answer:

The "exactly once" semantics ensure that a remore procedure will be executed exactly once and only once. The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages).

The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock. The client will then wait for one of two occurrences: (1) it will receive an ACK from the server indicating that the remote procedure was performed, or (2) it will time out. If the client times out, it assumes the server was unable to perform the remote procedure so the client invokes the RPC a second time, sending a later timestamp. The client may not receive the ACK for one of two reasons: (1) the original RPC was never received by the server, or (2) the RPC was correctly received—and performed—by the server but the ACK was lost. In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed.

3.7 Assume that a distributed system is susceptible to server failure. What mechanisms would be required to guarantee the "exactly once" semantics for execution of RPCs?

Answer:

The server should keep track in stable storage (such as a disk log) information regarding what RPC operations were received, whether they were successfully performed, and the results associated with the operations. When a server crash takes place and a RPC message is received, the server can check whether the RPC had been previously performed and therefore guarantee "exactly once" semanctics for the execution of RPCs.

CHAPTER 4

Threads

Practice Exercises

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.
Answer:

a. A Web server that services each request in a separate thread.

b. A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.

c. An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

4.2 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
Answer:

a. User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads.

b. On systems using either M:1 or M:N mapping, user threads are scheduled by the thread library and the kernel schedules kernel threads.

c. Kernel threads need not be associated with a process whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads as they must be represented with a kernel data structure.

4.3 Describe the actions taken by a kernel to context-switch between kernel-level threads.
Answer:
Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

4.4 What resources are used when a thread is created? How do they differ from those used when a process is created?

Answer:

Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

4.5 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

Answer:

Yes. Timing is crucial to real-time applications. If a thread is marked as real-time but is not bound to an LWP, the thread may have to wait to be attached to an LWP before running. Consider if a real-time thread is running (is attached to an LWP) and then proceeds to block (i.e. must perform I/O, has been preempted by a higher-priority real-time thread, is waiting for a mutual exclusion lock, etc.) While the real-time thread is blocked, the LWP it was attached to has been assigned to another thread. When the real-time thread has been scheduled to run again, it must first wait to be attached to an LWP. By binding an LWP to a real-time thread you are ensuring the thread will be able to run with minimal delay once it is scheduled.

CHAPTER 5

Process Synchronization

Practice Exercises

5.1 In Section 5.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

Answer:

The system clock is updated at every clock interrupt. If interrupts were disabled—particularly for a long period of time—it is possible the system clock could easily lose the correct time. The system clock is also used for scheduling purposes. For example, the time quantum for a process is expressed as a number of clock ticks. At every clock interrupt, the scheduler determines if the time quantum for the currently running process has expired. If clock interrupts were disabled, the scheduler could not accurately assign time quantums. This effect can be minimized by disabling clock interrupts for only very short periods.

5.2 Explain why Windows, Linux, and Solaris implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, adaptive mutex locks, and condition variables. In each case, explain why the mechanism is needed.

Answer:

These operating systems provide different locking mechanisms depending on the application developers' needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. Mutexes are useful for locking resources. Solaris 2 uses adaptive mutexes, meaning that the mutex is implemented with a spin lock on multiprocessor machines. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

5.3 What is the meaning of the term **busy waiting**? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer:
Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

5.4 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.
Answer:
Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

5.5 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.
Answer:
A wait operation atomically decrements the value associated with a semaphore. If two wait operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

5.6 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.
Answer:
The n processes share a semaphore, `mutex`, initialized to 1. Each process P_i is organized as follows:

```
do {
   wait(mutex);

      /* critical section */

   signal(mutex);

      /* remainder section */
} while (true);
```

CPU Scheduling

Practice Exercises

6.1 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of n.

Answer:

$n!$ (n factorial $= n \times n-1 \times n-2 \times \ldots \times 2 \times 1$).

6.2 Explain the difference between preemptive and nonpreemptive scheduling.

Answer:

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process. Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

6.3 Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Burst Time
P_1	0.0	8
P_2	0.4	4
P_3	1.0	1

a. What is the average turnaround time for these processes with the FCFS scheduling algorithm?

b. What is the average turnaround time for these processes with the SJF scheduling algorithm?

c. The SJF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P_1 and P_2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.

Answer:

a. 10.53

b. 9.53

c. 6.86

Remember that turnaround time is finishing time minus arrival time, so you have to subtract the arrival times to compute the turnaround times. FCFS is 11 if you forget to subtract arrival time.

6.4 What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?
Answer:
Processes that need more frequent servicing, for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing, and thus making more efficient use of the computer.

6.5 Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

These algorithms are thus really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of algorithm sets?

a. Priority and SJF

b. Multilevel feedback queues and FCFS

c. Priority and FCFS

d. RR and SJF

Answer:

a. The shortest job has the highest priority.

b. The lowest level of MLFQ is FCFS.

c. FCFS gives the highest priority to the job having been in existence the longest.

d. None.

6.6 Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer:

It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

6.7 Distinguish between PCS and SCS scheduling.

Answer:

PCS scheduling is done local to the process. It is how the thread library schedules threads onto available LWPs. SCS scheduling is the situation where the operating system schedules kernel threads. On systems using either many-to-one or many-to-many, the two scheduling models are fundamentally different. On systems using one-to-one, PCS and SCS are the same.

6.8 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through the use of LWPs. Furthermore, the system allows program developers to create real-time threads. Is it necessary to bind a real-time thread to an LWP?

Answer:

Yes, otherwise a user thread may have to compete for an available LWP prior to being actually scheduled. By binding the user thread to an LWP, there is no latency while waiting for an available LWP; the real-time user thread can be scheduled immediately.

6.9 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, for process P_2 is 18, and for process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Answer:

The priorities assigned to the processes are 80, 69, and 65 respectively. The scheduler lowers the relative priority of CPU-bound processes.

CHAPTER 7

Deadlocks

Practice Exercises

7.1 List three examples of deadlocks that are not related to a computer-system environment.
Answer:

- Two cars crossing a single-lane bridge from opposite directions.
- A person going down a ladder while another person is climbing up the ladder.
- Two trains traveling toward each other on the same track.

7.2 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.
Answer:
An unsafe state may not necessarily lead to deadlock, it just means that we cannot guarantee that deadlock will not occur. Thus, it is possible that a system in an unsafe state may still allow all processes to complete without deadlock occurring. Consider the situation where a system has 12 resources allocated among processes P_0, P_1, and P_2. The resources are allocated according to the following policy:

	Max	Current	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6

Currently there are two resources available. This system is in an unsafe state as process P_1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes P_0 and P_2 can complete. However, it is possible that a process may release resources before requesting any further. For example, process P_2 could release a resource, thereby increasing the total number of resources to five. This allows process P_0 to complete, which would free a total of nine resources, thereby allowing process P_2 to complete as well.

7.3 Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix **Need**?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

Answer:

a. The values of **Need** for processes P_0 through P_4 respectively are (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2).

b. The system is in a safe state? Yes. With **Available** being equal to (1, 5, 2, 0), either process P_0 or P_3 could run. Once process P_3 runs, it releases its resources, which allow all other existing processes to run.

c. The request can be granted immediately? This results in the value of **Available** being (1, 1, 0, 0). One ordering of processes that can finish is P_0, P_2, P_3, P_1, and P_4.

7.4 A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects $A \cdots E$, deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent the deadlock by adding a sixth object F. Whenever a thread wants to acquire the synchronization lock for any object $A \cdots E$, it must first acquire the lock for object F. This solution is known as **containment**: the locks for objects $A \cdots E$ are contained within the lock for object F. Compare this scheme with the circular-wait scheme of Section 7.4.4.

Answer:

This is probably not a good solution because it yields too large a scope. It is better to define a locking policy with as narrow a scope as possible.

7.5 Prove that the safety algorithm presented in Section 7.5.3 requires an order of $m \times n^2$ operations.

Answer:

Figure 7.1 provides Java code that implement the safety algorithm of the banker's algorithm (the complete implementation of the banker's algorithm is available with the source code download).

```
for (int i = 0; i < n; i++) {
   // first find a thread that can finish
   for (int j = 0; j < n; j++) {
      if (!finish[j]) {
         boolean temp = true;
         for (int k = 0; k < m; k++) {
            if (need[j][k] > work[k])
               temp = false;
         }

         if (temp) { // if this thread can finish
            finish[j] = true;
            for (int x = 0; x < m; x++)
               work[x] += work[j][x];
         }
      }
   }
}
```

Figure 7.1 Banker's algorithm safety algorithm.

As can be seen, the nested outer loops—both of which loop through n times—provide the n^2 performance. Within these outer loops are two sequential inner loops which loop m times. The big-oh of this algorithm is therefore $O(m \times n^2)$.

7.6 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

Answer:

An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run.

An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur.

7.7 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

Answer:

Starvation is a difficult topic to define as it may mean different things for different systems. For the purposes of this question, we will define starvation as the situation whereby a process must wait beyond a reasonable period of time—perhaps indefinitely—before receiving a requested resource. One way of detecting starvation would be to first identify a period of time—T—that is considered unreasonable. When a process requests a resource, a timer is started. If the elapsed time exceeds T, then the process is considered to be starved.

One strategy for dealing with starvation would be to adopt a policy where resources are assigned only to the process that has been waiting the longest. For example, if process P_a has been waiting longer for resource X than process P_b, the request from process P_b would be deferred until process P_a's request has been satisfied.

Another strategy would be less strict than what was just mentioned. In this scenario, a resource might be granted to a process that has waited less than another process, providing that the other process is not starving. However, if another process is considered to be starving, its request would be satisfied first.

7.8 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked waiting for resources. If a blocked process has the desired resources, then these resources are taken away from it and are given to the requesting process. The vector of resources for which the blocked process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process P_0 asks for (2,2,1), it gets them. If P_1 asks for (1,0,1), it gets them. Then, if P_0 asks for (0,0,1), it is blocked (resource not available). If P_2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P_0 (since P_0 is blocked). P_0's *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

a. Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur.

b. Can indefinite blocking occur? Explain your answer.

Answer:

a. Deadlock cannot occur because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process C.

7.9 Suppose that you have coded the deadlock-avoidance safety algorithm and now have been asked to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining **Max**[i] = **Waiting**[i] + **Allocation**[i], where **Waiting**[i] is a vector specifying the resources for which process i is waiting and **Allocation**[i] is as defined in Section 7.5? Explain your answer.

Answer:

Yes. The *Max* vector represents the maximum request a process may make. When calculating the safety algorithm we use the *Need* matrix, which represents *Max* — *Allocation*. Another way to think of this is *Max* = *Need* + *Allocation*. According to the question, the *Waiting* matrix fulfills a role similar to the *Need* matrix, therefore *Max* = *Waiting* + *Allocation*.

7.10 Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.

Answer:

No. This follows directly from the hold-and-wait condition.

CHAPTER 8

Main Memory

Practice Exercises

8.1 Name two differences between logical and physical addresses.
Answer:
A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space. Contrast this with a physical address that refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU). Therefore, physical addresses are generated by the MMU.

8.2 Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base–limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.
Answer:
The major advantage of this scheme is that it is an effective mechanism for code and data sharing. For example, only one copy of an editor or a compiler needs to be kept in memory, and this code can be shared by all processes needing access to the editor or compiler code. Another advantage is protection of code against erroneous modification. The only disadvantage is that the code and data must be separated, which is usually adhered to in a compiler-generated code.

8.3 Why are page sizes always powers of 2?
Answer:
Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

8.4 Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

Answer:

a. Logical address: 16 bits

b. Physical address: 15 bits

8.5 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on the one page have on the other page?

Answer:

By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. "Copying" large amounts of memory could be effected by having different page tables point to the same memory location.

However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user's "copy."

8.6 Describe a mechanism by which one segment could belong to the address space of two different processes.

Answer:

Since segment tables are a collection of base–limit registers, segments can be shared when entries in the segment table of two different jobs point to the same physical location. The two segment tables must have identical base pointers, and the shared segment number must be the same in the two processes.

8.7 Sharing segments among processes without requiring that they have the same segment number is possible in a dynamically linked segmentation system.

a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.

b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

Answer:

Both of these problems reduce to a program being able to reference both its own code and its data without knowing the segment or page number associated with the address. MULTICS solved this problem by associating four registers with each process. One register had the address of the current program segment, another had a base address for the stack, another had a base address for the global data, and so on. The idea is that all references have to be indirect through a register that maps to the current segment or page number. By changing these registers, the same code can execute for different processes without the same page or segment numbers.

8.8 In the IBM/370, memory protection is provided through the use of *keys.* A key is a 4-bit quantity. Each 2K block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal, or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?

a. Bare machine

b. Single-user system

c. Multiprogramming with a fixed number of processes

d. Multiprogramming with a variable number of processes

e. Paging

f. Segmentation

Answer:

a. Protection not necessary, set system key to 0.

b. Set system key to 0 when in supervisor mode.

c. Region sizes must be fixed in increments of 2k bytes, allocate key with memory blocks.

d. Same as above.

e. Frame sizes must be in increments of 2k bytes, allocate key with pages.

f. Segment sizes must be in increments of 2k bytes, allocate key with segments.