Study Notes

Operating Systems: Internals and Design Principles

Eighth Edition

William Stallings

TABLE OF CONTENTS

CHAPTER 1 COMPUTER SYSTEM OVERVIEW

Computer system components
- CPU, memory, I/devices, system bus – fig 1.1

Instruction execution
- fetch, decode, execute cycle – fig 1.2

Interrupts
- types of interrupts – table 1.1
- program flow with/without interrupts – fig 1.5
- instruction cycle with interrupts – figs 1.6, 1.7
- efficiency gain – figs 1.8, 1.9
- interrupt processing – figs 1.10, 1.11
- multiple interrupts – disable or prioritize – figs 1.12, 1.13

Memory Hierarchy
- quantity, speed, cost - relationships
- memory hierarchy – fig 1.14
- two-level memory – fig 1.15
- conditions (a-d) necessary for speedup – locality of reference
- secondary (auxiliary) memory – disk cache

Cache memory
- varying speeds of processor, registers, cache, main memory
- principles – figs 1.16, 1.1.7
- blocks (K words), slots (lines) of K words each – fig 1.17
- example – fig 1.18
- cache size, block size, mapping function, replacement algorithm, write policy

Two-level memories
- main memory cache (implemented in hardware)
- virtual memory, disk cache (both at least partially implemented in OS)
- locality, locality, locality – sequential execution, iteration,
- procedure call depth behavior (fig 1.21)
- data – local variables and parameters, arrays, large records
- spatial locality, temporal locality
- figs 1.22, 1.23, 1.24

CHAPTER 2 OPERATING SYSTEM OVERVIEW

OS objectives and functions

- convenience, efficiency, adaptability
- layers and views – Fig 2.1
- OS services – program dev., execution, IO, files, system resource access
- error detection and response, usage stats
- resource management – fig 2.2
- forces for change – hardware changes, new services, bug fixes
-

Evolution of OS

- serial processing - waste in scheduling and setup time
- simple batch systems – monitor, resident monitor – Fig 2.3
- memory protection, timer, privileged instructions, interrupts, user vs kernel mode
- multiprogrammed batch systems – tables 2.1, 2.2, memory management, scheduling
- time-sharing systems, user transaction processing, time slicing, minimize response time

Major Achievements

- **processes** – synchronization, mutual exclusion, determinacy, avoiding deadlock, execution context (process state), fig 2.8, context switching
- **memory management** – process isolation, automatic allocation and management, support of modular programming, protection and access control, long-term storage, virtual memory and files, virtual address, paging – figs 2.9, 2.10
- **information protection and security** – availability, confidentiality, integrity, authenticity
- **scheduling and resource management** – fairness, differential responsiveness, efficiency, round robin, priority queue, IO queueing, interrupt handling,

Developments leading to modern OS

- monolithic kernel,
- microkernel architecture
- multithreading (thread, process)
- symmetric multiprocessing
- distributed OS

- OOD

MS Windows

- Windows history
- single-user multitasking, Architecture – fig 2.15s
- OS Organization – "modified" microkernel architecture
- kernel-mode components: executive, kernel, HAL (hardware abstraction layer), device
- drivers, windowing and graphics system
- executive manager modules: IO, cache., object., plug and play, power, security, virtual
- memory, process/thread, configuration, local procedure call (LPC)
- user mode processes – special system support, service, environment, user apps
- client-server model
- threads and SMP
- windows objects – encapsulation, object class and instance, inheritance, polymorphism
- control objects, dispatcher objects

Traditional UNIX (System V release 3, and before)

- UNIX history
- architecture – fig 2.16
- kernel – fig 2.17
- single processor

Modern UNIX

- kernel – fig 2.18
- Versions

Linux

- Linux history
- modular structure -- loadable modules – dynamic linking, stackable modules
- fig 2.19 – example list of Linux kernel Modules
- Kernel components – signals, system calls, processes and scheduler, virtual memory, file systems, network protocols, character device drivers, bock device drivers, network

CHAPTER 3 PROCESS DESCRIPTION AND CONTROL

What is a process?

- program code, associated data, and a process control block (fig 3.1)

Process states

- trace, dispatcher (figs 3.2, 3.3, 3.4)
- 2-state process model, running, not running (fig 3.5)
- process creation (spawning), child process, process termination
- five state model – round-robin (fig 3.6), transitions, figures 3.7, 3.8a, 3.8b, priority
- suspended processes – fig 3.9 (virtual memory) – reasons for suspension (table 3.3)

Process description

- processes and resources – fig 3.10
- OS control structures – fig 3.11, memory tables, IO tables, file tables, process tables
- process control structures – process location, process control block,
- process image (table 3.4)
- process attributes – typical elements of a PCB – table 3.5
- 3 categories – process identification, processor state info (fig 3.12, table 3.6), process control info (figs 3.13, 3.14)
- role of the PCB – important design issue

Process control

- modes of execution – user mode, kernel (aka system, aka control) mode – table 3.7
- program status word (PSW) – example Itanium has program status register with a 2-bit
- control privilege level (cpl)
- process creation – assign a unique id, allocate space, initialize the PCB, set linkages,
- create or expand other data structures
- process switching – returning control the OS (table 3.8), clock interrupt, I/interrupt, memory fault, trap, supervisor call
- mode switching – interrupt
- change of process state – steps 1-7

Execution of the OS

- nonprocess kernel approach
- execution within user processes
- process-based OS

UNIX SVR4 process management

- process states and transitions – table 3.9, fig 3.17
- process description – table 3.10, user-level context, register context,
- system-level context (tables 3.11, 3.12)
- process control – fork() – steps 1-6 in kernel mode
- 3 options after forking – stay in parent, transfer to child, transfer to another process

CHAPTER 4 THREADS

Processes and threads

- resource ownership (process, task), scheduling/execution (thread, lightweight process)
- multithreading – vs single-threaded (one thread per process) ,
  - —process associations – virtual address, protected access to resources,
  - —thread associations – execution state, saved thread context, execution stack
- static storage, access to memory and resources of its process shared with other threads in that process (fig 4.2)
  - —thread as lightweight process – less time for creation, termination, switch, communication without intervention of the kernel
  - —examples of thread use in single-user system
- thread functionality
  - —thread states – spawn, block, unblock, finish
  - —does the blocking of one thread block the process? (fig 4.3)
  - —multithreading on a uniprocessor (fig 4.4)
  - —thread synchronization – potential for conflicts on shared resources

Types of threads

- User-level threads (ULTs) and kernel-level threads (KLTs – aka kernel-supported threads)
- ULTs – threads library (fig 4.5a), create, destroy, synchronize, scheduling execution, saving and restoring thread contexts
  - —cases of relationships between ULT states and Process states (fig 4.6)
  - —advantages of ULTs over KLTs – saves overhead of mode switching to kernel and back, scheduling can be tailored to application, OS independent
  - —disadvantages of ULTs – a blocking system call from a ULT blocks all threads in the process (jacketing technique can solve this), cannot take advantage of multiprocessing (only one processor is allocated teach process)
- KLTs (fig 4.5b) – user gets an API to the kernel thread facility, kernel does all thread management listed above, this fixes the two disadvantages of the ULTs.
  - —potential order of mag speedup by using KLTs over processes, and another order of mag speedup by using ULTs over KLTs
  - —combined approaches (Solaris) – combined ULT/KLT facility, allow programmer to specify both a number of ULTs and a (larger) number of KLTs, best of both with more design effort on part of programmer

CHAPTER 5 CONCURRENCY: MUTUAL EXCLUSION AND SYNCHRONIZATION

Concurrency

- multiprogramming, multiprocessing, distributed processing, three contexts for concurrency (multiple apps, structured apps, operating system structure)
- difficulties – sharing of global resources, allocation of resources (owner might get suspended), error tracking (which process?, not reproducible?)
- simple example – interleaving error (one or more processors), fixed by mutual exclusion
- race condition – last place writer wins
- OS concerns – OS must manage processes (PCBs), allocation and deallocation of resources (processor time, memory, files, IO), data and resource protection , functional behavior of a process must not depend on its execution speed relative to the speed of other processes
- process interaction – unaware, indirectly aware, directly aware (table 5.2)
  - —competition for resources – mutual exclusion, critical resource, critical section, deadlock, starvation (fig 5.1)
  - —cooperation by sharing – interleaving example again, need mutual exclusion
  - —cooperation by communication – message passing, mutual is not required, but some care needs to be taken to avoid deadlock and starvation
- requirements for mutual exclusion – process that halts in its noncritical section must do so without interfering with others, no deadlock or starvation, if critical section is not in use, any process that requests entry must be permitted without delay, no reliance on relative speeds or number of processes, process must exit critical section after a finite time

Mutual exclusion – hardware support

- interrupt disabling (cli, sti), price is high, won't work with multiple processors
- special machine instructions (atomic) – test and set (fig 5.2a – busy waiting), exchange (fig 5.2b – busy waiting)
- advantages – applicable to any number of processes on one or more processors, simple and easy to verify, can support multiple critical sections (each defined by a variable like bolt)

- disadvantages – busy waiting (yuk!), starvation, deadlock

Semaphores

- counting (general) semaphore, semSignal(s), semWait(s) (fig 5.3), atomic
- binary semaphore, semSignalB(s), semWaitB(s) (fig 5.4)
- strong semaphore (FIF– fig 5.5), weak semaphore
- mutual exclusion using semaphores (figs 5.6, 5.7)
- mutual exclusion using testset, xchg
- producer/consumer problem (fig 5.9 – incorrect solution – table 5.3)
- correct solution to infinite-buffer producer/consumer using binary semaphore (fig 5.10)
- correct solution using counting semaphores
- solution to bounded-buffer producer/consumer using counting semaphores
- Dekker's Algorithm, Peterson's Algorithm, both busy waiting – Appendix A
- implementations of semaphores using testset, interrupt masking (fig 5.14)

Monitors

- use barbershop, producer/consumer examples to motivate
- software module with methods and data
- only one process may be executing in the monitor at a time
- synchronization by use of condition variables, cwait(c), csignal(c)
- structure of a monitor (fig 5.15)
- bounded-buffer producer/consumer using a monitor (fig 5.16)
- alternate model with notify and broadcast

Message passing

- synchronization and communication
- send(destination, msg), receive(source, msg) (table 5.4)
- synchronization
- addressing
- message format
- queuing discipline
- mutual exclusion (fig 5.20)
- bounded-buffer producer/consumer using messages (fig 5.11)

Readers/ Writers

- any number of readers at one time, only one writer at a time, writer excludes readers
- readers have priority solution (fig 5.22)
- writers have priority solution (fig 5.23)

CHAPTER 6 CONCURRENCY: DEADLOCK AND STARVATION

Deadlock

- joint progress diagram (figs 6.2, 6.3)
- reusable resources (fig 6.4)
- consumable resources
- resource allocation graphs (fig 6.5)
- conditions for deadlock – mutual exclusion, hold and wait, no preemption, circular wait

Deadlock prevention

- mutual exclusion – cannot be disallowed
- hold and wait – require a process request all resources at once, block until all are
- available
- no preemption – when blocking on a request, release other resources, or preempt if higher
- priority
- circular wait – impose linear ordering on resource types

Deadlock avoidance

- allows more concurrency than prevention
- process initiation denial – each process must declare needs in advance, matrix of
- requirements, matrix of allocations, new process is not initiated if requirements exceed
- currently available resource remainders
- resource allocation denial – banker's algorithm (figs 6.7, 6.8, 6.9)

Deadlock detection

- marking algorithm, example (fig 6.10)
- recovery – abort all deadlocked processes, rollback all deadlocked processes,
- successively abort deadlocked processes until no deadlock exists, successively preempt
- resources until no deadlock exists, selection criteria

Integrated strategy

- summary (table 6.1)

- group resources into classes, use the linear ordering strategy to prevent deadlocks between classes
- within each class use methods appropriate to that class

Dining philosophers
- semaphore solution – deadlock is possible (fig 6.12)
- semaphore solution with attendant – no deadlock (fig 6.13)
- monitor solution – no deadlock (fig 6.14)

UNIX concurrency mechanisms
- pipes – shared circular buffers with mutual exclusion, writing to a full pipe blocks, as does reading from an empty pipe
- messages – msgsend, msgrcv, each process has a message queue (like a mailbox), retrieve by FIFO, or by type, attempt to send to a full queue will block , attempt to read from empty queue will block, attempt to read a msg of a certain type will not block on lack of such msg
- shared memory – read-only access, or read-write access
- semaphores – generalization of semWait, semSignal operations, more flexibility
- signals – informs process of occurrences of asynchronous events, single bit in process
- table, no typing, no priorities, otherwise similar to interrupt (table 6.2)

Linux kernel concurrency mechanisms
- atomic operations (table 6.3)
- spinlocks
  —basic spinlock – plain, _irq, _irqsave, _bh
  —spinlock implementation – uniprocessor vs multiprocessor
  —reader-writer spinlock – table
- semaphores
  —kernel semaphores
  —binary (mutex) and counting semaphores
  —reader-writer semaphore (table 6.5)
- barriers (table 6.6)

Solaris thread synchronization primitives
- both within kernel and in threads library
- mutual exclusion lock
- semaphores – sema_p(), sema_v()
- readers/writer lock – rw_enter(), rw_exit(), rw_tryenter(), rw_downgrade(),
- rw_tryupgrade()
- condition variables – cv_wait(), cv_signal(), cv_broadcast(), cv_wait()

Windows concurrency mechanisms
- wait functions

- synchronization objects (table 6.7)
- critical section objects – like mutex, but only for threads of a single process

CHAPTER 7 MEMORY MANAGEMENT

Memory management requirements

- relocation – process swapping
- protection – hardware
- sharing – code and data
- logical organization – modularity – naturally supported by segmentation
- physical organization – two-level scheme, memory management

Memory partitioning

- fixed partitioning – Fig 7.2, if program is too large, programmer must use overlays
  - —small programs waste lots of memory, internal fragmentation
  - —unequal-size partition (7.2b), assign each process to the smallest possible partition, if known, need a scheduling queue for each partition
  - —use a single queue, load a process into smallest possible available partition, if all are available, use a swapping strategy
- dynamic partitioning – space allocated as needed, leads to external fragmentation, overcome with compaction
  - —placement algorithm – best-fit, first-fit, next-fit (fig 7.5)
  - —replacement algorithm – if active processes are all suspended, swap
- buddy system – powers of 2, splitting, coalescing (figs 7.6, 7.7)
- relocation – logical address, relative address, physical address, hardware support (fig 7.8)

Paging

- pages, frames, fig 7.9
- page table, fig 7.10, frame size is power of two

Segmentation

- program and data are divided into variable length segments, similar to dynamic partitioning, but a program may occupy more than one partition, and these need not be contiguous, use a segment table

CHAPTER 8 VIRTUAL MEMORY

Virtual memory: Hardware and control structures

- 2 characteristics of paging and segmentation (logical addresses and page table) pave way for the fundamental breakthrough to virtual memory, resident set
- more processes may be maintained, a process can be larger than main memory
- real memory vs virtual memory (table 8.1)
- locality of code, locality of data (fig 8.1)
- paging – virtual memory (fig8.2a), present bit P, modify bit M, page number, offset, frame number, hardware implementation (figure 8.3)
- on VAX process can have 2G of VM, so if pages are 512 bytes, page table is 16M, unacceptably large for memory, so page tables also kept in VM, two-level hierarchy
- two-level scheme: page directory (page table for the page tables) (figs 8.4, 8.5)
- inverted page table (fig 8.6), uses hash function on the page number, page table has one entry for each real memory page frame, rather than one for each virtual page, chained hashing (fig 8.6)
- translation lookaside buffer – high-speed cache for page table entries (figs 8.7, 8.8), uses associative mapping on page number, only some of which are present (fig 8.9), this is done in hardware, for more info see cache design reference
- virtual mem mechanism must also interact with the main memory cache (not the TLB cache discussed above) (fig 8.10)
- page size considerations (fig 8.11), modern program techniques (objects, threads) decrease locality of references, TLB can become a bottleneck, a larger TLB helps but the TLB interacts with other hardware (main memory cache, # memory accesses per instruction cycle) which limits the increase on size of TLB, larger page sizes can also help, but these can also cause performance degradation (see fig 8.11), so some designers have investigated using multiple page sizes depending on nature of code (large contiguous regions use a small number of large pages, thread stacks use smaller pages)
- segmentation – multiple address space views, segment table for each process (fig 8.2b), resident bit, modify bit, address translation (fig 8.12)
- combined paging and segmentation (figs 8.13, 8.2c)
- protection and sharing (fig 8.14)

Operating system software

- OS policies for VM (table 8.3)

- fetch policy – when is a page brought into main memory? demand paging vs prepaging (bring along a range of contiguous pages, this is different from swapping)
- placement policy – where should the page reside in main memory? important in a pure segmentation system (use best-fit, first-fit, etc.), but irrelevant in a pure paging system or a combined paging/segmentation system, important on NUMA (nonuniform memory access) systems
- replacement policy – among those pages considered for replacement (see next topic – resident set management), which page should be replaced when a new page must be brought in? obvious goal: the one least likely to be referenced in near future
  - —frame locking
  - —basic algorithms – optimal, LRU, FIFO, Clock (fig 8.16)
  - —page buffering – FIF+ two lists (free page and modified page), these act as a cache of pages, modified pages can be written out to disk in clusters
  - —replacement policy and cache size – replacement policies for page buffer,
- resident set management
  - —resident set size – 3 factors at play, fixed-allocation, variable-allocation
  - —replacement scope – local vs global
  - —fixed allocation, local scope
  - —variable allocation, global scope – add page buffering to aid performance
  - —variable allocation, local scope – working set strategy (fig 8.19), virtual time window Δ, use concept to guide strategy – essentially LRU, monitor the page fault rate rather than the working set, page fault frequency strategy, variable-interval sampled working set – driven by three parameters M, L, Q
- cleaning policy – opposite of fetch policy, demand cleaning, precleaning, page buffering with two lists (modified and unmodified)
- load control – number of resident processes (aka multiprogramming level)
  - —thrashing (fig 8.21), page fault frequency algorithm implicitly incorporates load control, another approach L = S criterion, another is the 50% criterion, another is to adapt the clock page replacement algorithm to monitor the rate at which the pointer scans the circular buffer
  - —process suspension – lowest-priority, faulting, last activated, smallest resident set, largest, largest remaining execution window

Unix and Solaris memory management

- paging system (fig 8.22, table 8.5)

—page table, disk block descriptor, page frame data table, swap-use table
—page replacement – refinement of clock policy – two-handed clock (fig 8.23), scanrate, and handspread govern operation

Linux memory management

Windows management

CHAPTER 9 UNIPROCESSOR SCHEDULING

- dispatcher policies: FCFS, round-robin, SPN, SRT, HRRN, feedback, fair-share

CHAPTER 10 MULTIPROCESSOR AND REAL-TIME SCHEDULING

- architecture: master/slave, peer
- static, dynamic process assignment
- single global queue
- process scheduling: nonpreemptive FCFS from single global queue
- thread scheduling:
  —self scheduling, gang scheduling, dedicated processor assignment

CHAPTER 11 I/O MANAGEMENT AND DISK SCHEDULING

- buffering
- disk scheduling policies
- RAID

CHAPTER 12 FILE MANAGEMENT

- file organization: sequential, indexed sequential, indexed, direct
- record blocking
- "portion"
- disk allocation methods and the FAT: contiguous, chained, indexed
- free space management

CHAPTER 16 DISTRIBUTED PROCESSING, CLIENT/SERVER, AND CLUSTERS

- distributed system architectures
- client/server computing
- distributed message passing (RPC)
- clusters

Chapter 18 Distributed Process Management

- process migration and transfer strategies
- global state of a distributed system
- snapshots and "consistent" global state
- distributed snapshot algorithm
- Lamport's timestamping algorithm
- distributed mutual exclusion
- distributed queue and token-passing algorithms
- deadlock in distributed systems: resource allocation and message communication