System Architecture - Summary

Andreas Leppert

19-03-2007

Contents

1 Tanenbaum chapter 1: Introduction **5**
- 1.1 What is an OS? . . . 5
- 1.2 history of OSs . . . 5
- 1.3 OS Zoo . . . 6
- 1.4 computer hardware review . . . 6
- 1.5 OS concepts . . . 8
- 1.6 system calls . . . 9
- 1.7 OS structure . . . 9

2 Tanenbaum chapter 2: Processes and Threads **10**
- 2.1 Processes . . . 10
- 2.2 threads . . . 11
  - 2.2.1 thread model . . . 11
  - 2.2.2 thread usage . . . 12
  - 2.2.3 Implementing threads in user space . . . 12
  - 2.2.4 Implementing threads in kernel space . . . 13
  - 2.2.5 Hybrid Implementations . . . 14
- 2.3 Interprocess Communication (IPC) . . . 14
  - 2.3.1 Mutual exclusion with Busy Waiting . . . 14
  - 2.3.2 Sleep and Wakeup . . . 16
  - 2.3.3 Semaphores . . . 16
  - 2.3.4 Mutexes . . . 17
  - 2.3.5 Monitors . . . 17
  - 2.3.6 Message Passing . . . 18
  - 2.3.7 Barriers . . . 18
- 2.4 Scheduling . . . 18
  - 2.4.1 Introduction . . . 18
  - 2.4.2 Scheduling in Batch Systems . . . 20
  - 2.4.3 Scheduling in Interactive Systems . . . 21
  - 2.4.4 Policy versus Mechanism . . . 22

3 Tanenbaum chapter 3: Deadlocks **22**
- 3.1 Resources . . . 22
- 3.2 Introduction to Deadlocks . . . 23
  - 3.2.1 Conditions for a deadlock . . . 23
- 3.3 Ostrich Algorithm . . . 23
- 3.4 Deadlock Detection and Recovery . . . 23
  - 3.4.1 Deadlock Detection with One Resource of Each Type . . . 23
  - 3.4.2 Deadlock Detection with Multiple Resource of Each Type . . . 23
  - 3.4.3 Recovery from Deadlock . . . 24
- 3.5 Deadlock Avoidance . . . 24
- 3.6 Deadlock Prevention . . . 24

4 Tanenbaum chapter 4: Memory Management **25**
- 4.1 Basic Memory Management . . . 25
  - 4.1.1 Monoprogramming without Swapping or Paging . . . 25
  - 4.1.2 Multiprogramming with Fixed Partitions . . . 26
  - 4.1.3 Relocation and Protection . . . 26
- 4.2 More complex Memory Management - Swapping . . . 26
  - 4.2.1 Memory Management with Bitmaps . . . 27
  - 4.2.2 Memory Management with Linked Lists . . . 27
- 4.3 Virtual Memory . . . 28
  - 4.3.1 Paging . . . 28
  - 4.3.2 Page Tables . . . 28
  - 4.3.3 Translation Lookaside Buffers . . . 30

4.3.4 Inverted Page Tables . . . 30
4.4 Page Replacement Algorithms . . . 30
4.4.1 Optimal Page Replacement Algorithm . . . 31
4.4.2 The Not Recently Used Page Replacement Algorithm . . . 31
4.4.3 The First-In, First-Out Page Replacement Algorithm . . . 31
4.4.4 The Second Chance Page Replacement Algorithm . . . 31
4.4.5 The Clock Page Replacement Algorithm . . . 32
4.4.6 The Least Recently Used Page Replacement Algorithm . . . 32
4.4.7 Simulating LRU in Software with Not Frequently Used . . . 32
4.4.8 The Working Set Page Replacement Algorithm . . . 33
4.4.9 The WSClock Page Replacement Algorithm . . . 33
4.5 Design Issues for Paging Systems . . . 33
4.5.1 Local versus Global Allocation Policy . . . 33
4.5.2 Load Control . . . 34
4.5.3 Page size . . . 34
4.6 Segmentation . . . 34
4.6.1 Implementation of Pure Segmentation . . . 34
4.6.2 Segmentation with Paging: Multics . . . 35
4.7 Segmentation with Paging: Intel Pentium . . . 35

5 Input / Output **36**
5.1 Principles of I/O Hardware . . . 36
5.1.1 I/O Devices . . . 36
5.1.2 Device Controllers . . . 36
5.1.3 Memory Mapped I/O . . . 36
5.1.4 Direct Memory Access . . . 36
5.1.5 Interrupts revisited . . . 38
5.2 Principles of I/O Software . . . 38
5.2.1 Goals of the I/O Software . . . 38
5.2.2 Programmed I/O . . . 38
5.2.3 Interrupt-Driven I/O . . . 39
5.2.4 I/O using DMA . . . 39
5.3 I/O Software Layers . . . 39
5.3.1 Interrupt Handlers . . . 39
5.3.2 Device Drivers . . . 40
5.3.3 Device-Independent Software . . . 40
5.3.4 User-Space I/O Software . . . 41
5.4 Disks . . . 41
5.4.1 Disk Hardware . . . 41
5.4.2 Disk Formatting . . . 42
5.4.3 Disk Arm Scheduling Algorithms . . . 42
5.4.4 Error Handling . . . 42

6 File Systems **42**
6.1 Files (from user's view) . . . 43
6.1.1 File Naming . . . 43
6.1.2 File Structure . . . 43
6.1.3 File Types . . . 43
6.1.4 File Access . . . 43
6.1.5 File Attributes . . . 44
6.1.6 File Operations . . . 44
6.1.7 Memory-Mapped Files . . . 45
6.2 Directories . . . 45
6.2.1 Single-Level Directory Systems . . . 45
6.2.2 Two-Level Directory Systems . . . 45
6.2.3 Hierarchical Directory Systems . . . 45
6.2.4 Directory Operations . . . 45

6.3 File System Implementation 46
6.3.1 File System Layout 46
6.3.2 Implementing Files 46
6.3.3 Implementing Directories 48
6.3.4 Shared Files 48
6.3.5 Disk Space Management 48

1 Tanenbaum chapter 1: Introduction

- purpose of OS: computers are equipped with a layer of software whose job is to manage all the devices and provide user programs with a simpler interface to the hardware
- computer system consists of several levels: physical devices; microarchitecture level; machine language; OS; compiler, editors, command interpreter; browser etc.
- hardware, system programs, application programs
- OS in detail hides complexity of hardware programming and gives a more convenient set of instructions to work with the hardware
- OS is (usually) that portion of the software which runs in kernel mode (it's not correct for embedded or interpreted systems which have no kernel mode or use interpretation instead of hardware); there are also some programs like change password program which run in usermode but which definitely belongs to the OS

1.1 What is an OS?

- extended machine (top-down view)
  - OS hides truth about the hardware from the programmer and presents a nice view of named files that can be read and written
  - function of the OS is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than the underlying hardware
- resource manager (bottom-up view)
  - manage all the pieces of a complex system
  - users often need to share not only hardware, but information like files, databases; thus primary task of OS is to keep track of who is using which resource, to grant resource requests, to aaccount for usage and to mediate conflicting requests from different programs and users
  - time multiplexing/sharing: CPU, printer, etc.; who goes next and for how long is task of the OS
  - space multiplexing/sharing: main memory, etc.

1.2 history of OSs

- mainframes to run a job (program or set of programs)
- batch systems to run several programs directly
  - with OS/360 came multiprogramming - memory was partitioned and each job was in one distinct partition
  - while one job was waiting for I/O to complete another job could run on the CPU
  - also sppoling was introduced - ability to load a new job into a partition when one job has finished
- timesharing systems
  - variant of multiprogramming
  - have quick response times for users with online terminals

1.3 OS Zoo

- mainframe OS
  - higher I/O capacity than PCs
  - server rooms full of computers
  - OS has to deal with many jobs at once - batch, transaction processing or timesharing
- server OS
  - run on servers
  - serve multiple users at once over a network
  - allow users to share hardware and software
  - provide services for users
- multiprocessor OS
  - multiple CPUs in a single system
- personal computer OS
  - good interface for a single user
- real time OS
  - time is the key parameter
  - hard realtime and soft realtime systems
- embedded OS
  - small and smaller computers with less functionality
- smart card OS
  - very small
  - needs primitive OS
  - can only handle few functions

1.4 computer hardware review

- simple scheme: bus, CPU, memory and devices
- processor
  - fetch, decode and execute cycle
  - each CPU has a specific set of instructions it can execute
  - registers inside CPU to hold variables and results since accessing memory takes longer than executing an instruction
  - special registers (program counter, stack pointer, program status word)
  - with time multiplexing, OS has to switch programs; each time OS must save all registers and load other ones (of program which runs next)
  - pipelining for efficiency which leads to more complex compilers and OSes
  - even more performance with superscalar CPUs: multiple execution units for integer arithmetic, floating-point arithmetic, boolean operations etc look into a holding buffer which holds fetched instructions

  - user mode vs. kernel mode: when running in kernel mode, CPU can execute every instruction of its instruction set, while running in user mode means only a subset is available
  - user programs needing a kernel service invoke a system call which traps into the kernel and invokes the OS (TRAP instruction)
  - also other traps mostly through hardware and software (interrupts and exceptions) which have to be handled by the OS

- memory
  - goals: fast, large, cheap
  - hierarchy: registers, caches, main memory, magnetic disks, magnetic tapes
  - other forms of memory: ROM, EEPROM, flash ROM, CMOS
  - main memory problems:
    * how to protect programs from one another and the kernel from them all?
    * how to handle relocation?
    * solution: base register and limit register (need of special hardware)
      · when a program is run the base register is set to point to the start of its program text and the limit register tells how long the combined program text and data are
      · check whether program counter is less than limit register value and add it to base register, send the sum to memory
      · device that performs the check and the mapping is called MMU (memory mapping unit) and sits between memory and CPU
  - performance issues:
    * cache hides slow speed of memory (if one program is long enough in memory cache lines are full of that program lines)
    * however cache lines have to be replaced after switching to another program
    * MMU registers have to be changed each time a program switch occurs
    * -> keep in mind: context switches are expensive businesses

- I/O devices
  - consist mostly of controller and the device itself: controller receive commands from OS and carry them out to the device and vice versa
  - mostly, the controller provides a simpler interface to the OS
  - software that talks to the controller is called device driver
  - how to put device driver into kernel mode
    * relink kernel with device driver and reboot system
    * make an entry in a system file telling the kernel that it needs the device driver and reboot the system
    * install drivers on the fly without rebooting
  - controller have normally set of device register that are used to communicate with it
  - Input / Output
    * busy waiting: user issues system call, kernel translates it into procedure call to the appropriate driver, driver starts I/O and sits in tight loop continously polling the device to see if it's done, until data is put where it's needed, returns
    * interrupts: driver starts device and asks it to give an interrupt when it is finished, driver returns; OS blocks caller and does other jobs; when controller detects end of transfer it generates an interrupt, to signal completion
    * DMA: communication between memory and controller without using CPU; CPU sets up DMA with necessary information and does other work; when DMA chip is done it causes an interrupt, which is handled as above

  - what happens exactly when interrupts are used?
    * driver tells controller what to do by writing necessary information into the device registers
    * when controller has finished it signals the interrupt controller using certain bus lines
    * if interrupt controller accepts interrupt, it asserts a pin on the CPU chip informing it
    * interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished
    * CPU takes the interrupt: program counter and PSW are stored on current stack, CPU switches into kernel mode
    * find the address of the interrupt handler by using the device number as an index into part of special memory (interrupt vector)
    * start the interrupt handler which removes the stacked program counter and PSW and saves them, queries device to learn its status
    * when handler is finished, reload program counter and PSW and return to the first instruction of the user program that was not yet executed

- buses
  - many buses for handling more traffic efficiently
  - plug and play mechanism on Intel PCs using the BIOS

1.5 OS concepts

- processes
  - a process is a program in execution
  - associated with it is its address space (a list of memory locations from some minimum to some maximum, which the process can read and write
  - address space contains the executable program, program's data, program's stack, set of registers
  - information about processes other than address space is stored in the process table which is an array of structures, one for each process currently in existence
  - since processes can create new processes (childs) interprocess communication is needed

- deadlocks
  - when two or more processes are acting on shared resources, they can sometimes get themselves into a stalemate siuation they cannot get out of
  - for example consider resources cdrom and tape drive with two processes requesting the resources in vice versa order

- memory management
  - simplest one
    * only one program at a time is in memory
    * to run a second program, the first one has to be removed and the second one placed in memory
  - more complex one
    * more programs at once in memory, with protection mechanism which is done in hardware but controlled by the OS
  - what happens when there is more address space than real memory?
    * answer: use virtual memory
    * OS keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed

- input/output

  - I/O subsystems for managing the devices

- files
  - clean, abstract model of device-independent files
  - concept of directories to group files which leads to file hierarchies
  - before a file can be read or written it must be opened at which time the permissions are checked; if succeeded, a file descriptor, a small integer, is returned
  - special files are introduced to make I/O devices look like ordinary files (using same system calls as for normal files):
    * block special files
    * character special files

1.6 system calls

- interface between OS and the user programs is defined by the set of system calls provided by the OS
- example: count = read(fd, buffer, nbytes);
  - program pushes parameters onto the stack, in reverse order
  - actual call to library procedure (normal procedure call instruction used to call all procedures)
  - library procedure, mostly written in assembler, puts the syscall number in a place where the OS expects it, such as a register
  - it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel
  - kernel code examines syscall number and then dispatches to the correct syscall handler, usually via a table of pointers to syscall handlers indexed on syscall numbers
  - syscall handler runs
  - once the syscall handler completes, control may be returned to the user-space library procedure at the instruction following the TRAP instruction
  - this procedure returns to the user program in the usual way procedure calls return
  - user program cleans up the stack, as it does after any procedure call

1.7 OS structure

- monolithic systems
  - "the big mess", since there is normally no structure at all
  - collection of procedures which can call all other procedures if they like to
  - no information hiding since every procedure is visible to every other procedure
  - basic structure possible (three layers)
    * main program that invokes the requested service procedure
    * set of service procedures carrying out the system calls
    * set of utility procedures helping the service procedures

- layered systems
  - organize OS as a hierarchy of layers, each one constructed upon the one below it
  - THE:
    * layer 0: processor allocation, multiprogramming
    * layer 1: memory and drum management
    * layer 2: operator-process communication

        * layer 3: input/output management
        * layer 4: user programs
        * layer 5: the operator

- virtual machines
    - time sharing system providing multiprogramming and an extended machine with a more comvenient interface than the bare hardware
    - the virtual machine monitor runs on the hardware and does the multiprogramming providing several virtual machines to the next layer up
    - each virtual machine is an exact copy of the bare hardware, thus distinct OSes can be used
- exokernels
    - give each user a clone of the actual computer, but with a subset of the resources
    - at the bottom layer runs the exokernel which allocates resources to virtual machines and checks accesses
    - again, each virtual machine can run its own distinct OS
    - mapping is a bit simpler with exokernel instead of virtual machines
- client-server model
    - leave a minimal microkernel by moving up code into higher layers
    - realize most of the OS in user processes
    - more secure than other kernel models since most of work is done in user space
    - few functions have to be done in kernel mode

2 Tanenbaum chapter 2: Processes and Threads

2.1 Processes

- consider the difference between pseudoparallelism through context switching and true hardware parallalism through multiprocessors
- process is just an executing program including the current values of the program counter, registers, and variables
- switching back and forth between processes is called multiprogramming
- process creation
    - happens while
        * system initialization
        * execution of a process creation system call by a running process
        * user request to create a new process
        * initiation of a batch job
    - in all cases, a new process is created by having an existing process execute a process creation system call
    - after process is created, both the parent and the child have their own distinct address spaces
- process termination
    - happens with
        * normal exit (voluntary)

        * error exit (voluntary)
        * fatal error (involuntary)
        * killed by another process (involuntary)
    - in Windows and Linux, if a process is killed, their children are not killed!

- process states
    - three-state-model
        * running
        * ready
        * blocked

- implementation of processes
    - OS maintains process table which is an array of structures, with one entry (process control block) per process
    - contains everything that has to be saved when the process is switched from running to ready or blocked state, i.e. program counter, stack pointer, memory allocation, open files, accounting and scheduling information, etc.

- interrupt scenario:
    - each I/O device is associated with a location called the interrupt vector which contains the address of the interrupt service procedures
    - user process 3 is running when a disk interrupt occurs
    - user process 3's program counter, PSW, registers are pushed onto the current stack by the interrupt hardware
    - computer jumps to the address specified in the disk interrupt vector (hardware has done its job here)
    - all interrupts start by saving the registers, often in the process table entry for the current process
    - information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler
    - call to a C procedure to do the rest of the work for this specific interrupt type
    - after finishing its job, the scheduler is called to see who to run next
    - after that, control is passed back to asm code to load up the registers and memory map for the now-current process and start it running

2.2 threads

- need of multiple threads of control in the same address space running in quasi-parallel
- multiple threads of execution to share a set of resources so they can work together closely to perform some task

2.2.1 thread model

- processes are used to group resources together, threads are entities scheduled for execution on the CPU
    - threads has a program counter, registers, stack
    - analogy:
        * multiple processes running in one computer sharing physical memory, disks, printers etc.
        * multiple threads running in one process sharing the same address space, open files, etc.
    - since threads have some properties of processes they are called lightweight processes
    - multithreading means that multiple threads are used in one process

  - same illusion with thread switching as it was with process switching: threads are switched back and forth
  - difference: since every thread can access every memory address within the process address space, it could read, write or destroy every variable, stack etc. of other threads (in the same process)
  - but: threads started by one user collaborate and do not fight with each other! thus there is no need for protection
  - per process items: address space, global variables, open files, child processes, pending alarms, signals and signal handlers, accounting information
  - per thread items: program counter, registers, stack, state
  - states:
    * running
    * blocked
    * ready
    * terminated

2.2.2 thread usage

- processes start normally with a single thread present, new threads can be added with thread_create
  - when a thread has finished his work, thread_exit is called and the thread is no longer schedulable
  - one thread can wait for a specific thread to exit using thread_wait
  - thread_yield is used to allow a thread to voluntarily give up the CPU to let another thread run
- problems with fork and threads:
  - should the child also have the threads of the parent?
  - What happens when a thread in the parent process was blocked, should the thread in the child also be blocked?
- problems with shared data structured and threads:
  - what happens when a parent thread closes a file while a child thread reads from it?
  - What happens when parent thread allocates memory, a thread switch occurs, and the child thread allocates memory again?
- reasons for threads
  - multiprogramming (with shared address space): in many applications multiple activities are going on at once
  - more efficient than processes regarding creation and termination since they do not have any resources attached to them
  - better performance when there is substantial computing and I/O
  - better for systems with multiprocessors

2.2.3 Implementing threads in user space

- threads package is entirely in user space
- the kernel knows nothing about them, it is managing only single-threaded processes
- threads run on top of a run-time system which is a collection of procedures that manage threads like thread_create, thread_exit, etc.
- each process needs its own private thread table to keep track of the threads in that process

- the thread table keeps track of the "per-thread-items"
- advantages:
  - threads possible on OSs which do not provide threads out of the box
  - fast thread switch
    * e.g. when one thread is blocked because it has to wait for another thread, then the run-time system stores the thread's registers in the thread table
    * then it looks for a ready thread and reloads the registers with the new values
    * after that, only the program counter and the stack pointer has to be set and the new thread is alive
  - fast thread scheduler
    * operations do not need to enter kernel
    * no trap / context switch / memory cache flushing / etc. is needed
  - specific scheduling algorithms possible
- disadvantages:
  - system call can block entire process
    * when one thread wants to read a character from keyboard, all threads are blocked (with blocking system calls)
      · "solutions": use non-blocking system calls (but then OS has to be changed... and one advantage was to use user level threads without changing OS) or use select as a wrapper for blocking calls like read
  - page faults block entire process
    * when a thread causes a page fault, the kernel which doesn't know anything about threads, naturally blocks the entire process until the page is read in
  - cooperative scheduling necessary
    * if one thread is running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU
    * there are no clock interrupts within the process, so no round-robin fashion is possible (taking turns, "time slices")
    * unless thread enters run-time system of its own free will, the scheduler will never get a chance
  - programmers want thread mostly in applications where threads block often

2.2.4 Implementing threads in kernel space

- managed in kernel space, no run-time system needed
- thread table in kernel that keeps track of all the threads in the system, so there is no need for thread tables in each process
- creation and termination of threads are done via kernel calls
- information stored in kernel thread table is the same as above with threads in user space, but now it is stored in the kernel space
- keep in mind that the process table with its properties still exist (also in kernel space)
- calls that might block threads are system calls, so kernel threads cause greater costs than a call to the run-time system procedure
- only one thread is blocked and other threads of the same process or other processes can be run
- "thread recycling" to reduce the greater costs

  - when a thread is destroyed, it is marked as not runnable
  - the associated data structures are not deleted
  - later, when a new thread is created, the old thread is reactivated, saving some overhead
- kernel level threads
  - do not need non-blocking system calls
  - have no problems with page faults (another thread can be run instead of waiting for the page)

2.2.5 Hybrid Implementations

- combine the advantages of user-level threads with kernel-level threads
- use kernel-level threads and then multiplex user-level threads onto some or all the kernel threads
- each kernel-level thread has some set of user-level threads that take turns using it

2.3 Interprocess Communication (IPC)

- need for communication between processes in a well-structured way not using interrupts
- three issues with IPC
  - how one process can pass information to another (easy for threads since they are in the same address space)
  - how to make sure that two or more processes / threads don't get into each other's way when engaging in critical activities
  - proper sequencing when dependencies are present
- race condition
  - two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - consider "spooler with printer daemon" example
- mutual exclusion: some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
- critical regions / sections: part of the program where the shared memory is accessed
- four conditions for a good solution
  - no two processes may be simultaneously inside their critical regions
  - no assumptions may be made about speeds or the number of CPUs
  - no process running outside its critical region may block other processes
  - no process should have to wait forever to enter its critical region

2.3.1 Mutual exclusion with Busy Waiting

Disabling interrupts

- hardware solution
- each process disables interrupts after entering its critical region and reenable them just before leaving it (no clock interrupts occur)
- unwise to give user processes the power to turn off interrupts (one process could run forever)
- on multiprocessor systems it does not work, since only the interrupts of one processor are disabled, the other ones will continue running and can access the shared data
- it's a useful technique within the OS itself

Lock variables

- software solution
- single, shared lock variable which is initially set to 0
- if a process wants to enter its critical region it sets the lock to 1 and enters
- if the lock is already set to 1, the process waits until it becomes 0
- this does not work since it's the same fatal flaw that we saw in the spooler directory

Strict alternation

- software solution
- variable turn keeps track of whose turn it is to enter the critical region
- if turn is set to x then process x can enter its critical region, all other processes wait in a tight loop until the correct value appears in the turn variable (busy waiting)
- lock that uses busy waiting is called spin lock (which wastes a lot of CPU time)
- this solution violates condition three, since it's possible that a process blocks another process although it is not in its critical region (this happens when one process is much slower than the other)

Peterson's solution

- software solution
- combine the ideas of taking turns and lock / warning variables
- before entering the critical region, the process has to call enter_region(int process) to make sure that it is safe to enter
- after the critical region, the process uses leave_region(int process) to indicate it is done
- ```
  void enter_region(int process) {
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE) ;
  }
  ```
- ```
  void leave_region(int process) {
    interested[process] = FALSE;
  }
  ```

TSL Instruction

- hardware solution
- TSL RX, LOCK (test and set instruction) which is indivisible
- idea: use shared variable lock, initially set to 0 and updated to 1 using TSL when one process is entering its critical region
- again, use two functions enter_region and leave_region as above
- this only works correctly, when both functions are used at the correct times (just before and just after the critical regions)

- ```
  enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET

  leave_region:
    MOVE LOCK, #0
    RET
  ```

2.3.2 Sleep and Wakeup

- Peterson's solution and TSL are correct but very inefficient (waste a lot of CPU time)
- also, priority inversion problem can appear: two process H and L, with H high priority (always runs when ready) and L low priority; L is in its critical region, and H becomes ready, beginning busy waiting; since L is never scheduled because of the priority, L can never leave its critical region and so H never can enter its critical region and loops forever
- using sleep() / sleep(mem_address) and wakeup(process) / wakeup(mem_address) system calls to avoid busy waiting
- producer-comsumer problem (bounded-buffer problem):
  - two processes share a common, fixed-size buffer
  - producer process puts information into the buffer, consumer process takes it out
  - problems: buffer is full and procuder wants to add information or buffer is empty and consumer wants to get information out of buffer
  - possible solution: when buffer is full, producer goes to sleep; when buffer is empty, consumer goes to sleep
  - however, race condition possible when wakeup-signal is lost (could be resolved by adding a wakeup waiting bit)

2.3.3 Semaphores

- semaphore is an integer which could have the value 0 indicating that no wakeups were saved or some positive value if one or more wakeups are pending
- down (P) and up (V) operations which decrement / increment the semaphore which are atomic operations
- if the value is greater than 0 on a down operation, the value is just decremented; if it is 0, the process is put to sleep without completing the down operation on the semaphore
- if one or more processes are sleeping on that semaphore and an up operation is invoked, one process is chosen and is waked up (to complete its down operation)
- with semaphores the producer-consumer-problem can be solved
  - single processor systems: up and down as system calls which briefly disable all interrupts
  - multiprocessor systems: each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore
- using semaphores in two different ways:
  - as a mutex semaphore for mutual exclusion
  - semaphore for synchronization (guarantees that certain event sequences do or do not occur)
  ```

2.3.4 Mutexes

- when semaphore's ability to count is not necessary, a simplified version called a mutex is used
- only good for mutual exclusion
- two states: locked or unlocked
- can be implemented in user space if TSL is available
- ```
  mutex_lock:
      TSL REGISTER, MUTEX
      CMP REGISTER, #0
      JZE ok
      CALL thread_yield
      JMP mutex_lock
  ok: RET

  mutex_unlock:
      MOVE MUTEX, #0
      RET
  ```
- almost the same as enter_region above, but without busy waiting (instead: thread_yield)
- the order of the down's and up's is important, it must not be mixed up

2.3.5 Monitors

- higher-level synchronization primitive
- monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package
- only one process can be active in a monitor at any instant (this is done by the compiler which uses normally a mutex or a binary semaphore)
- condition variables along with wait and signal for blocking and unblocking processes
- wait and signal are not system calls but language constructs
- similar to sleep and wakeup but one crucial difference
  - sleep and wakeup failed because when one process was trying to go to sleep, the other one was trying to wake it up
  - with monitors that cannot happen since only one process can be active in the monitor
- e.g. the consumer can wake up its sleeping partner by doing a signal on the condition variable the partner is waiting on
- what happens when signal is invoked:
  - Hoare: the newly awakened process runs
  - Brinch Hansen: process doing a signal must exit the monitor immediately (so signal is the last instruction in the monitor procedure)
  - let the signal caller continue and start the awakened process after the caller has exited the monitor
- important note: condition variables do not count wakeups / signals; if a signal is done without any process waiting on it, the signal is lost; therefore wait always have to be before signal
  ```

2.3.6 Message Passing

- problem in distributed systems: most of above primitives are too low-level and monitors only usable in a few programming languages
- message passing uses send(destination, &message) and receive(source, &message) which are system calls again, not language constructs
- if no message is available, the receiver
  - can block until one arrives
  - can return with an error code
- more problems when processes communicate over a network: messages can be lost
  - possible solution: receiver of a message sends back a special acknowledgement message
  - if sender does not receive this acknowledgement in a certain time interval, it retransmits the message
  - when acknowledgement message is lost, sender retransmits the message and the receiver has a duplicate of the message
  - this can be discovered using sequence numbers

2.3.7 Barriers

- intended for groups of processes and applications which are divided into phases
- when a process reaches a barrier it is blocked until all processes have reached the barrier

2.4 Scheduling

2.4.1 Introduction

- with multiprogramming, multiple processes competing for the CPU at the same time
- a choice has to be made by the scheduler which process to run next
- context switches have to be efficient since they are expensive
  - switch from user mode to kernel mode
  - state of the current process must be saved, including storing its registers in the process table
  - memory map must be saved in most systems as well
  - a new process must be selected by the scheduling algorithm
  - MMU must be reloaded with the memory map of the new process
  - finally, the new process must be started
  - normally, the process switch invalidates the entire memory cache, forcing it to be reloaded twice (upon entering the kernel and upon leaving it)

Process behavior

- processes alternate bursts of computing with (disk) I/O requests
  - compute-bound processes
  - I/O-bound processes

When to schedule

- when a new process is created a decision has to be made whether to run the child or the parent process
- when a process exits
- when a process blocks on I/O, semaphore or other reasons another process has to be selected
- when an I/O interrupt occurs, a scheduling decision may be made
- when a clock interrupt occurs, a scheduling decision can be made
  - nonpreemptive scheduling algorithms
    - picks a process to run and then lets it run until it blocks or until it voluntarily releases the CPU
    - it will not be forceably suspended
    - in effect, no scheduling decisions are made during clock interrupts
  - preemptive scheduling algorithms:
    - picks a process and lets it run for a maximum of some fixed time
    - if it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run
    - clock interrupt is required

Categories of Scheduling Algorithms

- there are three environments where different scheduling algorithms are used
- Batch
  - no users impatiently waiting for a quick response
  - thus nonpreemptive or preemptive scheduling algorithms with long time intervals are acceptable
  - less context switches, better performance
- Interactive
  - preemption is essential so that users get answers quickly
- Real time
  - preemption is sometimes not needed since the processes know that they may not run for long periods of time and usually do their work and block quickly
  - difference to interactive systems is that only programs run that are intended to further the application at hand

Goals of scheduling algorithms

- all systems
  - fairness - giving each process a fair share of the CPU
  - policy enforcement - seeing that stated policy is carried out
  - balance - keeping all parts of the system busy
- batch systems
  - throughput - maximize jobs per hour
  - turnaround time - minimize time between submission and termination
  - CPU utilization - keep the CPU busy all the time

- interactive systems
  - response time - respond to requests quickly
  - proportionality - meet user's expectations
- real-time systems
  - meeting deadlines - avoid losing data
  - predictability - acoid quality degradation in multimedia systems

2.4.2 Scheduling in Batch Systems

First-Come First-Serverd

- nonpreemptive
- processes are assigned the CPU in the order they request it
- blocked processes are put on the end of the queue like a newly arrived job
- advantages
  - easy to understand
  - easy to program
  - fair
  - easy to get (just a dequeue operation)
- disadvantage
  - e.g. one compute-bound process which runs for 1sec until requesting a disk block
  - there are many I/O-bound processes with little CPU time but every process needs 1000 disk blocks
  - approximately, FCFS needs 1000 seconds to finish, instead of 10 seconds (optimal way, preemptive)

Shortest Job First

- nonpreemptive
- run times are known in advance
- all processes have to be available simultaneously
- pick the shortest jobs first

Shortest Remaining Time Next

- preemptive version of Shortest Job First
- choose that process whose remaining run time is the shortest
- new jobs get good service

Three-Level Scheduling

- three levels with batch systems
  - jobs arrive andthey are put in the input queue where an admission scheduler decides which jobs are granted to the CPU
  - since there is the possibility that not all processes fit in the main memory, a memory scheduler is needed which swaps the processes in and out
  - actually picking one of the ready processes is done by the CPU scheduler (any scheduling algorithm can be used here)

2.4.3 Scheduling in Interactive Systems

- two-level scheduling (memory and CPU scheduler) is possible and common

Round-Robin Scheduling

- each process is assigned a time interval (quantum) which it is allowed to run
- if the process still runs at the end of its quantum, the CPU is preempted and given to another process
- advantages
  - fair
  - easy to implement (list of runnable processes; when a process uses up its quantum it is put on the end of the list)
  - simple
- disadvantage
  - how high should be the quantum?
    - setting the quantum too short causes too many process switches and lowers the CPU efficiency
    - setting the quantum too long may cause poor response to short interactive requests
    - quantum between 20 - 50 msec is often a reasonable compromise
  - makes the implicit assumption that all processes are equally important

Priority Scheduling

- each process is assigned a priority, and the runnable process with the highest priority is allowed to run
- problem of processes running forever can be erased
  - by decreasing the priority on each clock interrupt
  - by assigning a maximum time quantum
- priorities can be assigned statically or dynamically (using a function to compute the priority, e.g. 1/f)
- processes are often grouped into priority classes and use priority scheduling among the classes but round-robin scheduling within the classes

Multiple Queues

- build up priority classes: processes in the first class were run for one quantum, processes in the next class for two quanta, etc.
- when one process uses up all its quantum it is moved to the next class to receive a higher quantum for the next time

Shortest Process Next

- minimize overall response time by running the shortest job first
- problem is to figure out which of the currently runnable processes is the shortest one
  - use aging - technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate

Guarantee Scheduling

- make real promises to the users about performance, e.g. if there are n users logged in, you will receive about 1/n of the CPU power
- difficult to implement

Lottery Scheduling

- simpler implemantation as with Guarantee Scheduling
- each process gets lottery tickets for various system resources (e.g. CPU time)
- whenever a scheduling decision has to be made, a lottery ticket is chosen and the process holding the ticket gets the resource
- more important processes can be given extra tickets to increase their odds of winning
- highly responsive scheduling algorithm
- tickets can be interchanged between cooperating processes to increase / decrease the odds

Fair-Share Scheduling

- take into account who owns a process before scheduling (e.g. one user starting 9 processes and another user starting 1 process only)
- relation between users can be expressed

2.4.4 Policy versus Mechanism

- separate the scheduling mechanism from the scheduling policy
- that means that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes
- e.g. consider a database management system with a parent and many children; now the parent can control exactly how the children are scheduled without doing the scheduling itself

3 Tanenbaum chapter 3: Deadlocks

- OS has to able to grant temporarily exclusive access to certain resources
- deadlocks are situations where two or more processes are blocked because of requesting a resource gained by another process
- deadlocks can occur on hardware and on software resources

3.1 Resources

- the objects granted are resources (devices, files, printers, records in a database, ...)
  - preemptable resources
    - one that can be taken away from the process owning it with no ill effects
    - e.g. memory
  - nonpreemptable resources
    - one that cannot be taken away from its current owner without causing the computation to fail
    - e.g. CD-Burner
- normal procedure for acquiring a resource

  - request the resource
  - use the resource
  - release the resource

- associate a semaphore with each resource, initialized to 1
  - request means down on the semaphore
  - use the resource
  - release means up on the semaphore

3.2 Introduction to Deadlocks

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

3.2.1 Conditions for a deadlock

- four conditions must hold for there to be a deadlock
  1. mutual exclusion condition: each resource is either currently assigned to exactly one process or is available
  2. hold and wait condition: processes currently holding resources granted earlier can request new resources
  3. no preemption condition: resources previously granted cannot be forcibly taken away from the process, they must be released by the process holding them
  4. circular wait condition: there must be a circular chain of two or more processes, each of which is waiting for a resource held by the next nmember of the chain

- four possibilities to attack deadlocks
  1. just ignore the problem at all
  2. detection and recovery
  3. dynamic avoidance by careful resource allocation
  4. prevention by structurally negating one of the four conditions necessary to cause a deadlock

3.3 Ostrich Algorithm

- "stick your head in the sand and pretend there is no problem at all"
- since deadlock elimination is expensive, a trade-off between correctness and convenience has to be made (since deadlock elimination leads to inconvenient restrictions for the users)

3.4 Deadlock Detection and Recovery

- let deadlocks occur, try to detect when this happens and then take some action to recover after the fact

3.4.1 Deadlock Detection with One Resource of Each Type

- only one resource of each type means one cd recorder, one scanner, etc.
- create the resource graph and detect cycles using a nice AlgoTech algorithm :D

3.4.2 Deadlock Detection with Multiple Resource of Each Type

- matrix based algorithm for n processes with m resource classes
- the algorithm marks all processes which are not deadlocked

3.4.3 Recovery from Deadlock

- what to do when a deadlock is detected? None of the following methods is really nice...

Recovery through Preemption

- sometimes it is possible to take away the resource from a process and give it to another one
- manuel intervention is often needed, especially in batch systems
- e.g. laser printer; take out the printed pages and stop the current process; another process can now print its data; afterwards, the old process can print its lacking pages
- this method is often impossible (since one cannot preempt a CD recorder for example)

Recovery through Rollback

- processes checkpointing periodically (write the current state of the process into a file so that it can be restarted later with that state)
- when a deadlock occurs, one process goes back to its checkpointed state, givesg away his resource shortly to another process requesting it and waits until this process has finished using the resource

Recovery through Killing Processes

- kill one or more processes in the deadlock cycle
- it is best to kill a process that can be rerun from the beginning without ill effects

3.5 Deadlock Avoidance

- is there an algorithm that can always avoid deadlock by making the right choice all the time? only if certain information is known in advance!
- the banker's algorithm avoids deadlocks by checking if granting a request leads to an unsafe state
- in practice useless, because processes rarely know in advance what their maximum resource needs will be
- also the number of processes is adjusting dynamically since new users log in and out
- furthermore, resources can suddenly vanish (e.g. tape drives can break)

3.6 Deadlock Prevention

- try to attack the four conditions for deadlocks

Attacking the Mutual Exclusion Condition

- one possible but not everywhere applicable solution is to introduce spooling
- problem that not every resource can be spooled
- furthermore, space can be scarce if spooling files get too big which can lead to another deadlock (considering disk space this time)

Attacking the Hold and Wait Condition

- one possible solution: require all processes to request their resources before starting execution, else just wait
  - but: many processes do not know in advance which resources they need (otherwise, the banker's algorithm could be used)
  - resources are not used optimally with this approach
- another approach: when requesting a resource, all resources held by the process have to be released

Attacking the No Preemption Condition

- this does not work for many resources as printers, scanner, CD recorders etc.

Attacking the Circular Wait Condition

- simple solution: a process can acquire only one resource at a time
  - this is not always possible, consider copying a huge file from tape to printer
- global numbering of all resources: processes can request resources whenever they want to, but all requests must be made in numerical order

4 Tanenbaum chapter 4: Memory Management

- memory manager is that part of the OS which manages the memory hierarchy
- jobs
  - keep track of which parts of memory are in use and which parts are not in use
  - allocate memory to processes when they need it
  - deallocate memory when processes are done
  - manage swapping between main memory and disk when main memory is too small to hold all the processes

4.1 Basic Memory Management

- there are two types of memory management systems
  - those that move processes back and forth between main memory and disk during execution (swapping and paging)
  - those who don't

4.1.1 Monoprogramming without Swapping or Paging

- just run one program at a time
- share memory only between that single program and OS
- three ways of organizing this scheme in memory
  - OS at the bottom of memory in RAM (mainframes, minicomputers)
  - OS at the top of memory in ROM (palmtop, embedded systems)
  - drivers in ROM at the top and rest of OS in RAM at the bottom of memory (early PCs, portion in ROM is the BIOS)
- OS presents a command line where user can type in the program name he wishes to run
- OS copies the requestes program from disk to memory and executes it until the process is finished and waits then for a new command

4.1.2 Multiprogramming with Fixed Partitions

- MFT (Multiprogramming with a Fixed number of Tasks)
- divide memory up into n (possibly unequal) partitions which can be done for example at system startup
- arriving jobs are put in the input queue of the smallest partition large enough to hold the job
- since partitions are of fixed size, any space in a partition not used is lost
- disadvantage of sorting the incoming jobs (input queue of large partition could be empty while another one is full)
  - several strategies available to resolve this problem

4.1.3 Relocation and Protection

- two essential problems which have to be solved
- relocation problem: when a program is linked (main program, procedures and libraries are combined into a single address space) the linker must now at what address the program will begin in memory
- possible solution: relocation during loading
  - actually modify the instructions as the program is loaded into memory (add stard address of partition the program is put in)
  - needs information about which program words are addresses to be relocated in the binary file
  - does not solve the protection problem (program can read any memory word since absolute addresses are used instead of relative ones)
- solution for the protection problem by IBM
  - divide memory into blocks of 2KB and assign a 4-bit protection code to each block
- alternative solution (for both problems): using base and limit registers
  - when process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition
  - every address within the program is now added to the base register and checked against the limit register, if it is in valid range
  - need for an extra addition and comparison and additionally extra hardware registers

4.2 More complex Memory Management - Swapping

- fixed partitions are only good for batch systems
- in interactive systems, there is not always enough memory to hold all programs in memory
- swapping
  - bring each process in its entirety into memory, run it for a while, then put it back on the disk
- virtual memory
  - allows programs to run even when they are only partially in main memory
- deal with swapping here
- difference to fixed partitions: the number, location and size of the partitions vary dynamically as processes come and go
  - improves memory utilization

  - complicates allocating and deallocating memory (since programs have to relocated every time they are swapped in

- swapping can create memory holes
  - memory compaction to move all the processes downwards as far as possible to erase the memory holes
  - very time consuming

- how much memory should be allocated for a process when it is created or swapped in
  - fixed size process: just allocate the size it wants
  - variable size: problem when there is no adjacent hole
    * move growing process to a memory hole large enough
    * or swap out other processes to disk
    * or allocate little extra memory above the partition while swapping in
    * or allocate little extra memory between stack (growing downwards) and heap (growing upwards)
    * or just kill the process

4.2.1 Memory Management with Bitmaps

- when memory is assigned dynamically, the OS must manage it using bitmaps or free lists
- with a bitmap, memory is divided up into allocation units
- corresponding to each allocation is a bit in the bitmap (0 if it is free, 1 if it is occupied)
- size of allocation unit is an important design issue
  - the smaller the allocation, the larger the bitmap
  - if the allocation is too large, bitmap will be smaller, but memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit
- when process wants k unit, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map (which is inefficient for big k's)

4.2.2 Memory Management with Linked Lists

- use linked list of allocated and free memory segments where a segment is either a process or a hole between two processes
- format: P|H, address at which it starts, length, pointer to the next entry (P is for process and H for hole)
- several algorithms possible for allocating memory for a newly created process (or a process swapped in)
  - first fit
    * scan from the beginning until it finds a hole large enough
  - next fit
    * variant of first fit
    * keeps track of where it is and starts the search from this point on
  - best fit
    * searches the entire list and takes the smallest hole that is adequate
  - worst fit
    * variant of best fit, but take the largest hole so that the hole broken off will be big enough to be useful
  - quick fit
    * maintains separate lists for some of the more common sizes requested

4.3 Virtual Memory

- solution in early days: programm split programs into pieces, called overlays
  - program started with overlay 0 and could swap in and swap out other overlays
  - biggest disadvantage: the splitting has be done manually
- automatically way of doing this: virtual memory
  - takes into account that amount of physical memory is not enough for all programs
  - OS keeps those parts of the programs in memory which are currently used and the rest on disk

4.3.1 Paging

- most virtual memory systems use paging technique
  - program-generated addresses are called virtual address and form the virtual address space
  - on computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written
  - if virtual memory is used, the virtual address goes to the MMU where it is mapped onto the physical memory address
  - virtual addresss space is divided into units called pages
  - the corresponding units in the physical memory are called page frames
  - page size = page frame size
  - a present / absent bit keeps track of which pages are physically present in memory (of which are mapped to physical memory)
  - page faults
    - if the program tries to use an unmapped page, the MMU causes the CPU to trap to the OS (page fault)
    - the OS picks a little-used page frame and writes its content back to the disk
    - then it picks the page just referenced into the page frame just freed, changes the map and restarts the trapped instruction
  - inside the MMU
    - e.g. incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset
    - 4 bits for page number means 16 pages, 12 bits for offset means 4096 bytes addresses within each page
    - page number is used as an index into the page table
    - if the present / absent bit is 0, a trap to the OS is caused (page fault)
    - if the present / absent bit is 1, the page frame number is copied to the high-order 3 bits of the output register, along with 12-bit offset (15-bit physical address)
    - output register is put onto the memory bus

4.3.2 Page Tables

- purpose of the page table is to map virtual pages onto page frames
- page table can be extremely large
  - virtual addresses of at least 32 bits are common
  - with 4-KB page size, you have 1 million pages
  - remember that each process needs its own page table (because it has its own virtual address space)
- mapping must be fast

- mapping must be done on every memory reference
- e.g. typical instruction has an instruction word and often a memory operand as well
- it is necessary to make 1, 2 or more page table references per instruction
- the proportion between instruction duration and page table lookup duration must be appropriate

simple design: single page table out of hardware registers

- single page table consisting of an array of fast hardware registers with one entry for each virtual page
  - when process is started, OS loads the registers with the process' page table, taken from a copy kept in main memory
  - during process execution, no more memory references are needed for the page table
  - very expensive if page table is large
  - furthermore, having to load the full page table at every context switch hurts performance

simple design: page table in main memory

- page table can be entirely in main memory
  - only one single hardware register is needed which points to the beginning of the page table
  - context switch is done by reloading one register
  - disadvantage of requiring one or more memory references to read page table entries during execution of each instruction

Multilevel Page Tables

- they avoid to store huge page tables in memory all the time (do not store all page tables in memory)
- e.g. virtual address is divided into a 10-bit PT1 field, a 10-bit PT2 field and a 12-bit Offset field (thus, page size is 4 KB, 1 mio. pages)
  - top-level page table with 1024 entries corresponding to the 10-bit PT1 field
    - the PT1 field is extracted and the value is used as an index into the top-level page table
    - each of these entries in the top-level page table represent 4 MB
  - the entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table
    - the PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself
  - only a few page tables are really present in memory, those who are needed

Structure of a Page Table Entry

- exact layout is machine-dependent, with 32 bit as a common size
- most important fields in a page table entry
  - page frame number
  - present / absent bit (1 means valid, 0 means not in memory currently)
  - protection bits (read only or read/write, read or write or execute)
  - modified bit / dirty bit (when a page is written to)
  - referenced bit (set whenever a page is read or written)
  - caching disabled bit (important for devices that map onto device registers rather than memory)

4.3.3 Translation Lookaside Buffers

- most programs tend to make a large number of references to a small number of pages and not the other way around
- TLB is a device which maps virtual addresses to physical addresses without looking into the page table
- sits usually inside the MMU and consists of only few entries
- each entry has information about one page: virtual page number, modified bit, protection bits, valid bit, physical page frame
- how does it work
  - when the virtual address is present to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries in parallel
  - if a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB
  - if the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated (the same way as it would be from the page table itself)
  - if the virtual page number is not in the TLB, the MMU does an ordinary page table lookup
    - it then evicts one of the entries from the TLB and replaces it with the page table entry just looked up (taking all fields from memory)
    - when an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory (the other values are already there)

4.3.4 Inverted Page Tables

- one entry per virtual page, since they are indexed by virtual page number
- that's okay for 32-bit and 4 KB pages, page table will be at least 4 MB, but not okay for 64-bit
- one entry per page frame in real memory
- e.g. 64-bit with 4 KB pages and 256 MB RAM leads to 65,536 entries
- virtual-to-physical translation becomes much harder because the entire inverted page table has to be searched for the entry (n, p) (process n which wants virtual page p)
- this search has to be done on each memory reference, not each page fault
- way-out: using TLB

4.4 Page Replacement Algorithms

- when a page fault occurs, the OS has to choose a page to remove from memory to make room for the new page requested
- if the page to be removed has been modified while in memory, it must be written back to disk to bring the disk copy up to date
- the page to be read in just overwrites the page being evicted (security ?!?!)

4.4.1 Optimal Page Replacement Algorithm

- easy to describe but impossible to implement
- pages in memory are labeled with number of instructions that will be executed before that page is first referenced
- choose the page with the highest label for removing
- problem is that at the time of the page fault the OS does not know when each of the pages will be referenced next
- used to compare other replacement algorithms with this one

4.4.2 The Not Recently Used Page Replacement Algorithm

- regard the referenced and the modified bit which are stored in each page table entry and set by the hardware on each memory reference
- once it is set 1, it stays 1 until the OS sets it back to 0
- when a process starts up, both page bits for all its pages are set to 0 by the OS
- periodically on each clock interrupt the referenced bit is set to 0 indicating that the page has not been referenced recently
- when a page fault occurs, the OS inspects all the pages and divides them into four categories
  - class 0: not referenced, not modified
  - class 1: not referenced, modified
  - class 2: referenced, not modified
  - class 3: referenced, modified
- the NRU algorithm removes a page at random from the lowest numbered nonempty class
- easy to understand, moderately efficient to implement, gives a good performance

4.4.3 The First-In, First-Out Page Replacement Algorithm

- the OS maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival
- on a page fault, the page at the head is removed and the new page is added to the tail
- rarely used

4.4.4 The Second Chance Page Replacement Algorithm

- simple modification of FIFO avoids the problem of throwing out a heavily used page
- inspect the referenced bit of the oldest page
- if it is 0 it is old and unused, thus it can be replaced
- if it is 1 it is cleared and the page is put onto the end of the list of pages
- inefficient because it is moving pages around in the list

4.4.5 The Clock Page Replacement Algorithm

- keep pages on a circular list in the form of a clock with a hand pointing to the oldest page
- on a page fault, the page being pointed by the hand is inspected
- if the R bit is 0, the page is evicted and the new page is inserted into the clock in its place, the hand is advanced by one position
- if R is 1, it is cleared and the hand is advanced to the next page
- just a better implementation of second chance

4.4.6 The Least Recently Used Page Replacement Algorithm

- observation: pages that have been used recently will be used again in the next few instructions (probably)
- when a page fault occurs, throw out the page that has been unused for the longest time (LRU paging)
- can be implemented with a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear
- difficulty is that the list must be updated on every memory reference
- hardware solution with a 64-bit counter which is increased with every instruction
  - each page table entry needs a field for a counter value
  - after each memory reference, the current value of the counter is stored in the page table entry for the page just referenced
  - when a page fault occurs, the OS examines all the counters in the page table to find the lowest one; this page is the least recently used
- another hardware LRU algorithm using a n x n matrix for n page frames possible

4.4.7 Simulating LRU in Software with Not Frequently Used

- since the above mentioned hardware is mostly not given, a software solution is needed
- it requires a software counter associated with each page
- at each clock interrupt the OS scans all the pages in memory and for each page, the R bit is added to the counter
- so the counters keep track how often the pages have been referenced
- on a page fault, the page with the lowest number is evicted
- problem: NFU never forgets anything (consider pages which have been used often at the beginning, but rarely used at the end; they still have a high number and are not evicted, misleadingly)
- two modifications to simulate LRU adequately (called aging)
  - the counter is shifted right 1 bit before the R bit is added
  - the R bit is added to the left most, not the right most bit

4.4.8 The Working Set Page Replacement Algorithm

- demand paging loads pages on demand, not in advance
- most programs exhibit locality of reference meaning that during any phase of execution, the process references only a relatively small fraction of its pages
- the set of pages that a process is currently using is called the working set
- a program causing page faults every few instructions is said to be thrashing
- working set model: try to keep track of each process' working set and make sure that it is in memory before letting the process run (also called prepaging)
- page replacement: when a page fault occurs, find a page not in the working set and evict it (the information which pages in the working set are and which not is crucial)
- approximation to find the working set: define the working set as the set of pages used during the past 100 msec of (its) execution time
- hardware sets R and M bits as mentioned before
- R bit is cleared on every clock interrupt
- on every page fault the page table is scanned to look for a suitable candidate
- if R is 1, the current virtual time is written into the Time of last use field in the page table
- if R is 0, its age is computed (age = current virtual time - time of last use field) and compared to tau
  - if the age is greater tau, the page is no longer in the working set and is evicted
  - if not, the page is still in the working set and temporarily spared and marked (if no other candidate to evict is found, this one will be evicted9

4.4.9 The WSClock Page Replacement Algorithm

- variant of working set and clock algorithm
- good performance and easy to implement
- pages form a circle, with a time of last use field, as well as the R and M bits
- if R is 1, the page is no candidate
- if R is 0, check the age
  - if age greater than tau and the page is clean, the page is evicted and the new page loaded in its place
  - if age greater than tau but the page is dirty, the hand is only advanced

4.5 Design Issues for Paging Systems

4.5.1 Local versus Global Allocation Policy

- local algorithms effectively correspond to allocating every process a fixed fraction of the memory
  - when working set grows, thrashing will result even if there are plenty of free page frames
  - when working set shrinks, local algorithms waste memory
- global algorithms dynamically allocate page frames among the runnable processes
  - system must decide continually how many page frames to assign to each process
  - one could start each process up with some number of pages proportional to the process size, but the allocation has to be updated dynamically as the processes run (use page fault frequency algorithm to control size of the allocation set)
- keep in mind that some page replacement algorithms can be run in local and global way (e.g. FIFO), others not (e.g. WSClock)

4.5.2 Load Control

- despite optimal global allocation and best replacement algorithm, the system can thrash (whenever the combined working sets of all processes exceed the capacity of memory)
- solution is to swap out processes and free up all the pages they are holding
  - thus swapping is still needed even with paging, only now swapping is used to reduce potential demand for memory, rather than to reclaim blocks of it for immediate use

4.5.3 Page size

- can be chosen by the OS
- small page sizes because
  - internal fragmentation: memory that is wasted in the last page of a process
  - a large page size will cause more unused program to be in memory than a small page size
- big page sizes because
  - because the page table is then smaller
  - on machines with page tables in hardware it is better to have smaller page tables thus greater page size

4.6 Segmentation

- provide the machine with many completely independent address spaces, called segments
  - each segment consists of a linear sequence of addresses, from 0 to some maximum
  - different segments have usually different lengths which may change during execution
  - different segments do not affect other ones
  - segments rarely fill up since they are usually very large
  - to specify an address in this segmented memory, the program must supply two-part addresses, a segment number and an address within the segment
  - segment is a logical entity which the programmer is aware of and uses it as a logical entity
- advantages of segmentation
  - great advantage for data structures which are steadily growing and shrinking
  - linking is simplified when only a few segments have been changed since only these have to be changed
  - also good for shared libraries or shared data (since these libraries for example are in one segment and not in each process' address space)
  - specific protection schemes (read only etc.) can be assigned to different segments
- fix me figure 4-37

4.6.1 Implementation of Pure Segmentation

- pages are fixed size, segments are not
- external fragmentation happens with segmentation (can be dealt with compaction)

4.6.2 Segmentation with Paging: Multics

- if segments are large, it is sometimes impossible to keep them in their entity in main memory
- idea of paging them, so that only those pages that are actually needed have to be around
- combine advantages of paging and segmentation
  - paging
    - uniform page size
    - not having to keep the whole segment in memory if only part of it is being used
  - segmentation
    - ease of programming
    - modularity
    - protection
    - sharing
- each Multics program has a segment table with one descriptor per segment (the segment table itself is a segment and is paged)
- segment descriptor is an indicator whether the segment is in main memory or not
- if any part of the segment is in memory, the segment is considered to be in memory and its page table will be in memory
- if the segment is in memory, its descriptor contains an 18-bit pointer to its page table, segment size, protection bits and few other items
- the address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler
- each segment is an ordinary virtual space and is paged in the same way
- page size is normally 1024 words
- address in Multics consists of two parts: segment and address within the segment
- the address within the segment is further divided into a page number and a word within the page
- now this is what happens when a memory reference occurs
  - segment number is used to find the segment descriptor
  - check is made to see if the segments' page table is in memory; if the page table is in memory, it is located; if not, a segment fault occurs, if there is a protection violation, a fault (trap) occurs
  - the page table entry for the requested virtual page is examined, if the page is not in memory, a page fault occurs; if it is, the main memory address of the start of the page is extracted from the page table entry
  - offset is added to the page origin to give the main memory address where the word is located
  - read or write takes place
- only efficient with the TLB

4.7 Segmentation with Paging: Intel Pentium

still to come ...

5 Input / Output

5.1 Principles of I/O Hardware

5.1.1 I/O Devices

- division of the I/O devices into two categories (not a perfect classification)
  - block devices
    - stores information in fixed-size blocks, each one with its own address
    - it is possible to read or write each block independently of all the other ones
  - character devices
    - delivers or accepts a stream of characters, without regard to any block structure
    - no seek operation, not addressable

5.1.2 Device Controllers

- the electronic component of the I/O unit is called the device controller
- jobs (of a disk controller)
  - convert the serial bit stream into a block of bytes
  - perform any error correction if necessary
  - bits are assembled in a buffer inside the controller, then the checksum is computed and verified, then copied to main memory

5.1.3 Memory Mapped I/O

- OS can command the device to do some job by writing and reading the device registers
- additionally, OS can read the internal device buffers
- how does the OS reads from and writes to the device registers
  - using an I/O port number and special I/O instructions (IN and OUT)
  - using memory-mapped I/O (map all the control registers into the memory space)
    - can be used with normal C, no assembler necessary (since no special instruction necessary)
    - no special protection necessary that users use these control registers directly
    - caching is not good for control registers (can be resolved with caching disabled bit in page table entries)
    - problems with multiple buses (PCI bridge chip decides whether address goes to PCI bus instead of memory)
  - hybrid scheme also possible (Intel architecture)

5.1.4 Direct Memory Access

- needs a special DMA controller (which is on the parentboard or integrated into the disk controllers for example)
- it has access to the system bus independent of the CPU
- contains several registers that can be read and written by the CPU
- what happens without DMA...
  - controller reads the block from the drive serially, until the entire block is in the controller's internal buffer

  - compute checksum to verify that no read errors have occured
  - controller causes an interrupt
  - the OS reads the block from the controller buffer by looping over and reading a word on each iteration, stores it in main memory

- what happens with DMA...
  - CPU programs the DMA controller by setting the DMA registers (what and in which direction)
  - CPU issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum
  - when valid data is in the controller buffer, DMA can begin
  - DMA controller issues a read request over the bus to the disk controller
  - write to memory is another standard bus cycle (from disk controller to main memory)
  - the disk controller acknowledges the disk controller over the bus
  - DMA controller then increments the memory address to use and decrements the byte count
  - repeat the last 4 steps until byte count is 0
  - DMA controller interrupts the CPU to let it know that the transfer is now complete
  - when the OS starts up it does not have to copy the disk block to memory, it is already there

- since buses can operate in two modes, DMA controllers also can operate in either mode
  - word-at-a-time mode: leading to cycle stealing: CPU has to wait, if it wants the bus
  - block mode: DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus (burst mode)
    * more efficient since acquisition of the bus also takes time
    * blocks CPU and other devices longer

- fly-by mode: DMA controller tells the device controller to transfer the data directly to main memory
- alternative mode: send the word from the device controller to the DMA controller which then issues a second bus request to write the word to wherever it is supposed to go
  - more flexible
  - less efficient

- memory addresses in DMA controller
  - normally physical addresses (CPU converts virtual addresses into physical ones and writes those into the DMA address registers)
  - using virtual addresses in DMA controller only possible, if MMU is part of memory instead of CPU

- disadvantages of DMA
  - CPU can do the same job in less time
  - more hardware

5.1.5 Interrupts revisited

- how does interrupts work...
  - when an I/O device has finished, it causes an interrupt
  - this interrupt is caught by the interrupt controller which decides what to do next
  - if no other interrupts are pending, the interrupt controller processes the interrupt immediately
  - if not or if there is another interrupt with higher priority, it is just ignored for the moment (the interrupt is then send again until it is serviced by the CPU)
  - CPU stops what it is doing, takes the number on the address line as an index into the interrupt vector to fetch a new program counter
  - the one points to the start of the corresponding interrupt service procedure
  - some information like program counter and registers have to saved now (mostly on the stack)
  - the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports (controller is free to issue another interrupt now)
  - ...

5.2 Principles of I/O Software

5.2.1 Goals of the I/O Software

- device independence
  - write programs without having to know the device in advance
- uniform naming
- error handling
  - normally, errors should be detected and corrected by hardware, but sometims software is also used (if the lower layers are not able to correct the error)
- synchronous (blocking) versus asynchronous (interrupt-driven) transfers
  - it is up to the OS to make operations that are actually interrupt-driven look blocking to the user programs
- buffering
- sharable versus dedicated devices
  - devices which are used by several users at the same time

5.2.2 Programmed I/O

- have the CPU do all the work
- e.g. program wants to print a string
  - first, data are copied to the kernel
  - OS enters a tight loop outputting the characters one at a time (when the device indicates that it is ready)
  - thus, the CPU continuously polls the device to see if it is ready to accept another one (polling, busy waiting)
- very simple
- wastes CPU time

5.2.3 Interrupt-Driven I/O

- e.g. program wants to print a string
  - buffer is copied to kernel space
  - the first character is transferred to the printer
  - CPU calls the scheduler and some other process runs
  - the user program is blocked until the entire string is printed
  - when the printer is finished it generates an interrupt which stops the current process and saves its state
  - the interrupt service procedure runs: it outputs the next character if there are any characters left or the user program is unblocked
- on each printed character an interrupt occurs, no good performance

5.2.4 I/O using DMA

- let DMA controller feed the characters to the printer one at time, without the CPU being bothered
- reduced number of interrupts from one per character to one per buffer
- normally slower than CPU

5.3 I/O Software Layers

- I/O layers: hardware, interrupt handlers, device drivers, device-independent OS software, user-level I/O software

5.3.1 Interrupt Handlers

- when a driver starts an I/O operation, the best is to block until the I/O has completed and the interrupt occurs
- the interrupt handler has then to unblock the driver that started it
- what has to be done...
  - save any registers that have not already been saved by the interrupt hardware
  - set up a context for the interrupt service procedure
  - set up a stack for the interrupt service procedure
  - acknowledge the interrupt controller
  - copy the registers from where they were saved to the process table
  - run the interrupt service procedure (which will extract information from the interrupting device controller's registers)
  - choose which process to run next
  - set up the MMU context for the new process, also TLB
  - load the new process' registers
  - start running the new process

5.3.2 Device Drivers

- each device attached to a computer needs some device-specific code for controlling it, the device driver
- most kernels expect drivers to run in the kernel
- most OS define a standard interface that all block drivers for example must support
- jobs of device drivers
  - accept abstract read and write requests from the device-independent software above it and see that they are carried out
  - initialize the device
  - log events
  - manage power requirements
- how it works
  - check input parameters if they are valid
  - check if device is ready or not (then the request is queued for later processing)
  - write actual concrete command into the controller's device registers
  - check whether the device has accepted the command
  - if the device has done the task without delay, the driver needn't be blocked; otherwise, it has to be blocked until the device has finished the task
  - after the operation has been completed, the driver must check for errors
  - perhaps the device has data for the driver, then the device driver has to pass the data to the device-independent software
  - return some status information for error reporting back to the caller
  - do the next request or block itself until next request is coming in

5.3.3 Device-Independent Software

- basic function is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software

Uniform Interfacing for Device Drivers

- important that all I/O devices and drivers look more-or-less the same
- all drivers should have the same interface, so that it becomes easier to plug in new drivers
- a device name specifies the i-node for a special file, and this i-node contains the major device number (used to locate the appropriate driver) and the minor device number (used to specify the unit to be read or written)

Buffering

- unbuffered input
  - interrupt for each character coming in
  - user process has to be started up for every character
- buffering in user-space
  - only one interrupt for the whole buffer (n characters)
  - what if buffer is paged out

- buffering in kernel-space followed by copying to user space
  - what is with characters arriving while the page with the user buffer is being brought in from the disk (buffer is full, no place to store them)
- double buffering in the kernel
- buffering suffers performance if data gets buffered too many times

Error Reporting

- OS must handle errors as best it can
- programming errors (report back an error code to the caller)
- actual I/O errors (up to the driver what to do next)

Allocating and Releasing Dedicated Devices

- OS must decide whether the requested device is available or not (e.g. CD-Rom)
- block caller if device is not available (and put them in a queue)

Device-Independent Block Size

- it is up to the device-independent software to hide the fact that different disks have different sector sizes
- provide uniform block size to higher layers (e.g. treat several sectors as a single logical block)

5.3.4 User-Space I/O Software

- libraries running outside the kernel
- they call the appropriate system call with the parameters
- some do really work like printf
- not all user-space software consists of library functions but of spooling software (generating files in a special spooling directory which is visited by a daemon)

5.4 Disks

- reading and writing is equally fast

5.4.1 Disk Hardware

Magnetic Disks

- organized into cylinders, each one containing as many tracks as there are heads stacked vertically
- tracks are divided into sectors

RAID

- let RAID controller do the work instead of a disk controller
- parallel access to several disks for better performance or higher security
- distributing data over multiple drives is called striping
- redundancy is important
- a little bit more information should be here...

5.4.2 Disk Formatting

- before the disk can be used, each platter must receive a low-level format
  - consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors (
  - sector consists of preamble, data, ECC fields
    - preamble is a bit pattern that allows the hardware to recognize the start of the sector; it contains also the cylinder and sector numbers and some other information
    - size of data portion is regulated by the low-level formatting program
    - ECC field for redundant information to recover from read errors
- position of sector 0 on each track is offset from the previous track when the low-level format is laid down (offset is called cylinder skew)
  - leads to better performance
- final step is the high-level format of each partition
  - lays down the boot block, the free storage administraion (free list or bitmap), root directory and an empty file system

5.4.3 Disk Arm Scheduling Algorithms

- time to read or write a disk block is determined by
  - seek time (time to move the arm to the proper cylinder)
  - rotational delay (time for the proper sector to rotate under the head)
  - actual data transfer time
- see the slides...

5.4.4 Error Handling

- really important???

6 File Systems

- long-term information storage (files)
  - must be possible to store a very large amount of information
  - information must survive the termination of the process using it
  - multiple processes must be able to access the information concurrently
- OS issues (file system): how are files...
  - structured
  - named
  - accessed
  - used
  - protected
  - implemented

6.1 Files (from user's view)

6.1.1 File Naming

- files are an abstraction mechanism
- provide a way to store information on the disk and read it back later
- when a process creates a file, it gives the file a name; this name continues to exist when the process terminates and can be accessed later by other processes under this name

6.1.2 File Structure

- unstructured sequence of bytes
  - OS does not know or care what is in the file
  - meaning must be imposed by user-level programs
  - maximum flexibility
- fixed-length records
  - each record has some internal structure
  - read should return one record and write should overwrite or append one record
- tree of records
  - not necessarily all the same length
  - each contains a key field in a fixed position in the record
  - tree is sorted on the key field for rapid searching for a key

6.1.3 File Types

- regular files
  - contain user information
  - ASCII files
    - can be read and written with every text editor
  - binary files
    - mostly with internal structure
    - executable files, archives, ...
- directories
  - system files for maintaining the structure of the file system
- character special files
  - related to input / output and used to model serial I/O devices
- block special files
  - used to model disks

6.1.4 File Access

- random access files
  - files whose bytes or records can be read in any order
- read and seek operation

6.1.5 File Attributes

- every file has a name and its data plus additional attributes
- list of attributes
  - protection
  - password
  - creator
  - owner
  - read-only flag
  - hidden flag
  - system flag
  - archive flag (0 means has been backed up, 1 means needs to be backed up)
  - ASCII / binary flag
  - random access flags (0 for sequential access only, 1 for random access)
  - temporary flags (0 for normal file, 1 for file to be deleted on process exit)
  - lock flags
  - record length (number of bytes in a record)
  - key position (offset of the key within each record)
  - key length (number of bytes in the key field)
  - creation time
  - time of last access
  - time of last change
  - current size
  - maximum size

6.1.6 File Operations

- list of operations
  - create
  - delete
  - open
  - close
  - read
  - write
  - append
  - seek
  - get attributes
  - set attributes
  - rename

6.1.7 Memory-Mapped Files

- files can be mapped into the address space of a running process using system calls map and unmap
- a read from an address causes a page fault, bringing in page x of the file
- best with segmentation where each file is mapped into its own segment
- eliminates need for I/O
- but OS does not know in advance the length of a file written to in mmap
- what happens when two processes use the file, one with mmap and one in the conventional way, and process one writes into the file (mmap), that change will not be identified by process two.
- files can be larger than segments and even virtual address space

6.2 Directories

6.2.1 Single-Level Directory Systems

- only one directory called the root directory containing all files
- very simple
- different users could accidently use the same names for their files which does not work

6.2.2 Two-Level Directory Systems

- give each user its private directory
- systems knows where to find a file using owner field
- there is a directory for the utility programs (since otherwise, all binaries have to be in each user directory)

6.2.3 Hierarchical Directory Systems

- each user can have as many directories as are needed so that files can be grouped together in natural way

6.2.4 Directory Operations

- list of directory operations
  - create
  - delete
  - opendir
  - closedir
  - readdir
  - rename
  - link
  - unlink

6.3 File System Implementation

6.3.1 File System Layout

- disks can be divided into partitions with independent file systems on each partition
- sector 0 is the Master Boot Record (MBR) which is used to boot the computer and which contains at the end the partition table
- the partition table gives the start and the end of each partition
- one partition have to be active
- system startup
  - BIOS reads in and executes MBR
  - MBR searches the active partition, reads in its first block (called the boot block) and executes it
  - boot block loads the operating system contained in that partition
- layout of a disk partition varies from file system to file system
- often the file system will contain some of the these items:
  - boot block
  - super block (read in when the FS is touched firstly)
  - free space management (bitmap or list of pointers)
  - i-nodes (array of data structures, one per file, telling all about the file)
  - root dir
  - files and directories
- typical information in the super block
  - magic number to identify the file system type
  - number of blocks in the file system
  - other key administrative information

6.3.2 Implementing Files

- keeping track of which disk block goes with which file

Contiguous Allocation

- simplest allocation scheme is to store each file as a contiguous run of disk blocks
- advantages
  - simple to implement (only necessary information is disk address of the first block and the number of blocks in the file)
  - read performance is excellent (only one seek is needed, the rest with one read operation)
- disadvantages
  - disk becomes fragmented in time

Linked List Allocation

- keep each file as a linked list of disk blocks (first word is used as a pointer to the next one, rest is pure data)
- advantages
  - every disk black can be used, nothing is wasted (except for internal fragmentation in the last block)
  - the directory entry only has to store the disk address of the first block
- disadvantages
  - random access is very slow
  - amount of data storage is no longer a power of two because the pointer needs some space

Linked List Allocation Using a Table in Memory

- put pointer word of each disk block into table residing in memory (FAT - File Allocation Table)
- advantages
  - entire block is available for data
  - random access is much easier (since memory references are faster)
  - the directory entry only has to store the disk address of the first block
- disadvantage
  - table should be all the time in main memory to make it work
  - 20 GB disk and 1 KB block size means 20 million entries with a minimum of 4 bytes; thus the table itself is about 80 MB

I-nodes

- associate each file with a data structure called an i-node (index node) which lists the attributes and disk addresses of the file's blocks
- with the i-node one can find all the blocks of the file
- advantages
  - i-node needs only to be in memory when the corresponding file is open
  - size is proportional to the maximum number of files that may be open at once
- disadvantages
  - each i-node has room for a fixed number of disk addresses, what is when this limit is broken?
    - use last disk address not for a data block, but instead for the address of a block containing more disk block addresses

6.3.3 Implementing Directories

- when a file is opened, the OS uses the path name supplied by the user to locate the directory entry
- the directory entry provides the information needed to find the disk blocks
  - the disk address of the entire file (contiguous allocation)
  - number of the first block (both linked list schemes)
  - number of the i-node (i-nodes)
- map ASCII file name onto the information needed to locate the data
- where are the file attributes stored
  - store them directly into the directory entry
    - list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, addresses where the disk blocks are
  - store attributes into i-nodes, the directory entry only needs information about file name and i-node number
- hash tables are used to minimize look-up of file name in large directories
- another way is to cache the results of searches

6.3.4 Shared Files

- problem: if directory entry contains disk addresses, a copy of the disk addresses will have to be made in the other directory where the link is linked; if one user appends data to the file, the information is missing in the other directory
  - solution 1: do not list disk blocks in directories, use i-nodes instead; the directories would have only point to the same i-node
    - OS has to care about reference count of the i-node to make sure that the file can be first deleted when no one is referenced to it
  - solution 2: let the system create a new file, of type LINK, which only contains the path name of the file to which it is linked; OS is caring about this special file when it is write to (symbolic link)
    - extra overhead: path must be read, parsed and followed, until the i-node is reached
    - extra i-node for each symbolic link, extra disk block to store the path
    - symbolic links can be used to link to files on machines anywhere in the world

6.3.5 Disk Space Management

- two general strategies are possible for storing an n byte file
  - n consecutive bytes of disk space are allocated (bad, because if data is appended to file, it will probably have to be moved on disk)
  - file is split up into a number of (not necessarily) contiguous blocks

Block size

- how big should the block be
- candidates are: sector, track, cylinder, page
- if it is too big, disk space is wasted
- if it is too small, files need more blocks and the more blocks are needed, the more blocks have to be read, the slower is the system
- disk utilization and disk performance compete!

Keeping Track of Free Blocks

- linked list of disk blocks
  - each block holding as many free disk block numbers as will fit
  - e.g. with a 1 KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks (the last one is a pointer to the next block)
  - only one block of pointers need be kept in main memory
- bitmaps
  - disk with n blocks requires a bitmap with n bits
  - free blocks are represented by a 1, allocated blocks by a 0
- bitmap requires less space, since it uses 1 bit per block versus 32 bits in the linked list model
- only when the disk is nearly full, the linked list needs less space