User-Mode Thread implementation

1. Introduction
    a. If process is the only unit of computation:
        i. Processes are expensive to create and dispatch, since they have own virtual address space and ownership of numerous system resources
        ii. Each process operates in its own address space and cannot share im-memory resources with parallel processes
    b. Threads
        i. Independent schedulable unit of execution
        ii. Runs within the address space of a process
        iii. Has access to all of the system resources owned by that process
        iv. Has its own general registers
        v. Has its own stack, within the owning process's address space
    c. UNIX:
        i. Process can create thread for additional parallelism
    d. Windows NT:
        i. Designed with threads from the start
            1. A process is a container for an address space and resources
            2. A thread the THE unit of scheduled execution
2. A simple threads library
    a. Pure user-mode library, initially
        i. Basic model
            1. Each time a new thread is created
                a. We allocate memory for a fixed size thread private stack from heap
                b. Create a new thread descriptor that contains identification, scheduling info and ptr to the stack
                c. Add the new thread to a ready queue
            2. When a thread calls yield() or sleep(), we save its general registers and select the next thread on the ready queue
            3. To dispatch a new thread: restore its saved registers including rsp and return from the call that caused it to yield
            4. If a thread called sleep() we should remove it from the ready queue. When re-awaken, put it back onto the ready queue
            5. When a thread exited, we would free its stack and thread descriptor
        ii. Eventually want preemptive scheduling
            1. Ensure good interactive response, prevent buggy thread freeze up application
            2. How
                a. Before dispatching a thread, we can schedule a SIGALARM that will interrupt the thread running too long: yield, save state, move on
3. Kernel implemented threads
    a. Fundamental problems with user-mode thread library
        i. What happens when a system call blocks
            1. If a user-mode thread issues a system call (read or open), the process is blocked until the operation completes.
            2. When a thread blocks, all threads within the process stop execution.
            3. Since the thread were implemented in user-mode, the OS has no idea that other threads may still be runnable
        ii. Exploiting multi-processors

1. If CPU has multiple execution cores, the OS can schedule proceses in parallel. But If OS is not aware that a process is composed of multiple threads, those threads cannot execute in parallel on the available resources

4. Performance implications
    a. If non-preemptive:
        i. User-mode sleep/yield model is more efficient than doing context switches
    b. If preemptive
        i. The cost of setting alarm and servicing signals > allow OS to do the scheduling
    c. If threads can run in parallel on multi-processor
        i. Added thru put from true parallelism >> overhead of context switches
        ii. Maximize cache-line sharing: OS knows which threads are part of the same process
    d. Signal disabling and reenabling for a user-mode mutex or condition variable can me more expensive than doing context switches

Deadlock Avoidance

1. Introduction
    a. Example
        i. Main memory is exhausted
        ii. We need to swap some processes out to the secondary storage to free up memory
        iii. Swapping processes involves the creation of new I/O request descriptors, which must be allocated from main memory
    b. Example
        i. Some process will free up resources when it completes
        ii. But the process needs more resources in order to complete
    c. Def: keep track of resources and reject requests that would put system into a dangerously resource depleted state to prevent deadlock
2. Reservations
    a. Failure of random allocation request in mid-operation can be difficult to handle gracefully
        i. Sol: ask process to reserve resources before they actually needs them
    b. Example: sbrk(2)
        i. Does NOT actually allocate any more memory
        ii. Requests OS to change the size of the data segment in the process's virtual address space
        iii. Actual assignments will not happen until the process begins referencing those newly authorized pages
        iv. If we decide a memory request will exhaust the resource, we can return an error from sbrk(2) and then the process decide how to handle. But if we wait until the paged was referenced to reject, we MUST kill the process
3. Over-booking
    a. It is unlikely that all client will simultaneously request their maximum resource reservations
        i. Relatively safe to grant more than we have
    b. +: get more works done with same resources
    c. -: might have demand that cannot gracefully handle
    d. In OS
        i. Notion of killing random processes is so abhorrent so often refuse over-book. In fact, often under-book: reserve 10% for emergency
4. Dealing with Rejection
    a. Simple program might log an error message
    b. Stubborn program might continue retrying the request

    c. Robust program might return errors for request that cannot be processes but continue trying to serve new requests
    d. A more civic-minded program might attempt to reduce its resource use and therefore the number of requests it can serve

Health Monitoring and Recovery

1. Introduction
    a. Determine if the system is deadlocked
        i. Identify all the blocked processes
        ii. Identify the resource on which each process is blocked
        iii. Identify the owner of each blocking resource
        iv. Determine whether or not the implied dependency graph contains any loops
    b. Formal deadlock detection in real system
        i. Difficult to perform
        ii. Inadequate to diagnose most hangs
        iii. Does not enable us to fix the problem
        iv. Sol: health monitoring and managed recovery
2. Health monitoring
    a. Device if the system is making any progress
        i. Have a internal monitoring agent watch message traffic or a transaction log to determine whether or not work is continuing
            1. -: can determine service is processing request at reasonable rate, but if internal monitoring agent fails, it may not be able to detect and report errors
        ii. Ask clients to submit failure reports to a central monitoring service when a server appears to become unresponsive
        iii. Have server send periodic heart-beat messages to a central health monitoring service
            1. -: can only tell the app is running, but have no idea on if it serving
        iv. Have external health-monitoring service send periodic test requests to the service that being monitored, ascertain that they are being responded to correctly and within timely
            1. -: can assure responding to requests, but have no idea on other requests
    b. General model:
        i. internal monitoring agent closely watches key applications to detect failures and hangs.
        ii. if the internal monitoring agent is responsible for sending heart-beats (or health status reports) to a central monitoring agent, a failure of the internal monitoring agent will be noticed by the central monitoring agent.
        iii. an external test service that periodically generates test transactions provides an independent assessment that might include external factors (e.g. switches, load balancers, network connectivity) that would not be tested by the internal and central monitoring service
3. Managed recovery
    a. For a failed/hung service: highly available service must restart/recovery/fail over
        i. Software should be designed so that any process in the system can be killed/restart at any time
        ii. Software should be designed to support multiple levels of restart
            1. Warm-start: restore the last saved state and resume service where we left out
            2. Cold-start: ignore any saved state (e.g. corrupted) and restart from scratch
            3. Reset and reboot: reboot the entire system and then cold-start
        iii. Software might also designed be for progressively escalating scope of restarts

            1. Restart only a single process, and expect it to resync with the other process when it comes back
            2. Maintain a list of all the processes involved in the delivery and restart all processes in that group
            3. Restart all the software on a single node
            4. Restart a group of nodes or the entire system
        iv. +: only restart processes that most likely would fail
        v. -:
            1. Process A failed as a result of an incorrect request received from process B
            2. But operation that caused A to fail is still there, so restart – fail – restart -…
            3. The operation that fails A may have been mirrored into other systems will also cause additional failures

4. False reports
    a. Ideal: problem would be found by the internal monitoring agent on the affected node
        i. Trigger restart of affected software on that node
        ii. Fix the problem even before other nodes notice it
    b. Declaring a process fail is very expensive:
        i. Ex: cancel/transmission of requests, other services affected
        ii. We don't want to cancel unless the process truly failed
            1. Best option: the failing system detect its own problem, inform partners, shutdown cleanly
            2. The failure is detected by missing heartbeat, wait for multiple misses before declare failure
            3. to distinguish a problem with a monitored system from a problem in the monitoring infrastructure, we might want to wait for multiple other processes/nodes to notice and report the problem.
        iii. Trade-off: "mark-off threshold"
            1. Time to confirm vs. un-necessary service disruption
            2. If we mis-diagnose the cause of the problem and restart, worse
            3. If we wait too long before initiating fail-overs, prolong the outage
5. Other managed restarts
    a. Non-disruptive rolling upgrades
        i. Possible when the system can operate without some of its nodes
        ii. We take nodes down, one-at-a-time, upgrade each to a new software release, and then reintegrate them into the service
        iii. Note
            1. the new software must be up-wards compatible with the old software, so that new nodes can interoperate with old ones
            2. if the rolling upgrade does not seem to be working, there needs to be an automatic *fall-back* option to return to the previous (working) release
    b. Prophylactic reboots
        i. Observation: the longer sw run, the slower
            1. Common cause: memory leak
            2. Solution #1: fix the bug => hard
            3. Solution #2: automatically restart every system at a regular interval (h/day)
                a. If a system can continue operating when node failures, it should be fairly simple to shut-down and restart nodes one at a time, on a regular schedule

Measuring OS performance

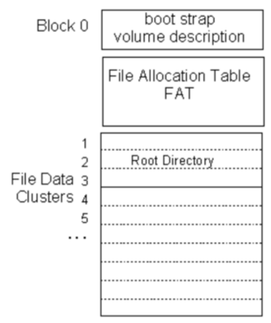
Load and Stress Testing

1. Introduction
    a. The number of enumerated test cases is relatively small, and the particulars of each test case is particularly important
    b. Run the test cases in pseudo-random orders for unspecified periods of times
    c. No expectation that results will be repeatable
    d. In many cases, there is no definitive pass indication. Rather, we can only say that the test has run for a period of time
    e. We may not take the trouble to define a complete set of assertions to determine the correctness of any particular operation has been performed. We may even look at the returned results
2. Load testing
    a. Purpose: measure the ability of a system to provide service at a specific load level
    b. A test harness issues requests to the component under test at a specified rate (the offered load), and collects performance data. The collected performance data depends on what is to be measured, but typically includes things like
        i. Response time for each request
        ii. Aggregate thru put
        iii. CPU time and utilization
        iv. Disk I/O operation and utilization
        v. Network packet and utilization
    c. Resulting info: measure system speed & capacity, analyze bottlenecks -> improvements
    d. Load generation
        i. Load generator: a system that can generate test traffic corresponding to a specific profile at a calibrated rate, to drive the system that is being measured
        ii. Testing thru "whole system" 's primary service interfaces
            1. If system is network server: load generator will pretend to be enormous clients
            2. If system is application server: LG create multiple test task to be run
            3. If system is I/O device: LG generates I/O requests
        iii. Test load characterized by
            1. Request rate (op/sec)
            2. Request mix: different types of clients use the system in a different way
            3. Sequence fidelity: sometimes sufficient to negate right mix, sometimes simulate particular access patterns, sometimes simulate realistic patterns
    e. Performance measurement
        i. Typical ways of using LG for performance measurement
            1. Deliver requests at a specified rate and measure the response time
            2. Deliver the requests at a increasing rate until the max thruput is reached
            3. Deliver the requests at a rate, use this as a calibrated background for measuring performance other system services
            4. Deliver requests at a rate, use this as test load for detailed study of bottlenecks
    f. Accelerated aging
        i. Many errors (e.g., mem leak) will have significant effect only as time goes
        ii. LG can be used to create realistic traffic to simulate accelerated aging
3. Stress testing
    a. Residual error tend to involve combinations of unlikely circumstances: stress test
    b. Robustness and stability of the program

- c. Few software product survive such testing
- d. Stress test
    - i. Use randomly generated complex usage scenarios to increase the likelihood of encountering unlikely combinations of operations
    - ii. Deliberately generate large numbers of conflicting requests (e.g., multiple clients try to update the same file at the same time)
    - iii. Introduce a wide range randomly generated errors and simulated resource exhuasions
        1. System continuously experience & recover errors
    - iv. Introduce wide swings in load and regular overload situations

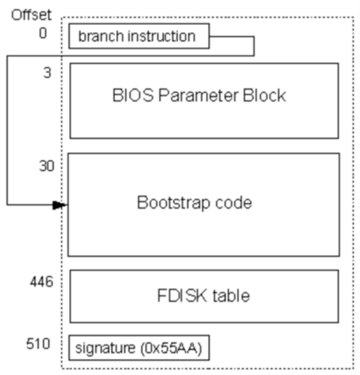
Introduction to DOS FAT Volume and File Structure

1. Introduction
    - a. BIOS ROM: basic I/O Subsystem
        - i. Provides run-time support for BASIC interpreter
    - b. DOS FAT us worth studying for because:
        - i. Heavily used all over the world
        - ii. Basis for more modern file systems
        - iii. Provides reasonable performance (large transfers and well clustered allocation) with simple implementations
        - iv. Very successful example of “linked list” space allocation
2. Structured overview
    - a. All file systems include a few basic types of data structures
        - i. Bootstrap: code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first trach of the first cylinder for the boot strap
        - ii. Volume descriptors: info describing the size, type, layout of the file system, and how to find other key metadata descriptors
        - iii. File descriptors: info that describes a file (ownership, protection, time of last update, etc.) and points to where actual data is stored on the disk
        - iv. Free space descriptors: list of blocks of currently unused space can be allocated to files
        - v. File name descriptors: data structures that user-chosen names with each file
    - b. DOS FAT file system divide the volume into fixed sized (physical) blocks, which are grouped into larger fixed sized (logical) block clusters
    - c. The first block of DOS FAT contains bootstrap along with volume descriptor info
    - d. After which comes File Allocation Table (FAT)
        - i. Keep track which block has been allocated to which file
    - e. Remainder of the volume: data clusters
        - i. Can be allocated to files/directories
        - ii. First file on the volume: root directory



3. Boot block BIOS parameter block and FDISK table
    - a. Boot
        - i. DOS: combine bootstrap block and volume description info into 1 block
        - ii. Boot record:
            1. Begins with a branch instruction (to the start of the real bootstrap code)
            2. Followed by a volume description (BIOS parameter block)

            3. Followed by the real bootstrap code
            4. Followed by an optional disk partitioning table
            5. Followed by a signature (for error checking)

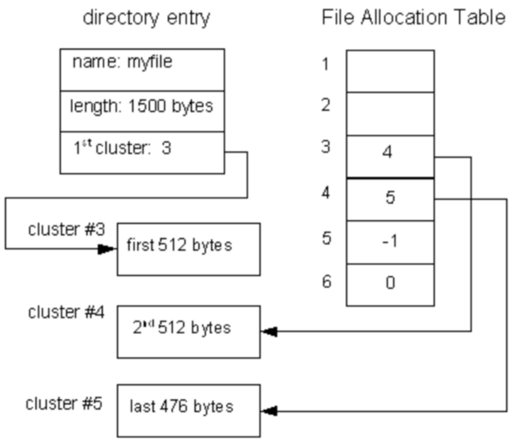


b. BIOS parameter block
    i. After the first few bytes of the bootstrap comes the BIOS parameter block, which contains summary of device and file system
        1. Number of bytes per physical sector
        2. Number of sectors per track
        3. Number of tracks per cylinder
        4. Total number of sectors on the volume
    ii. Describes the way the file system is layed out on the volume
        1. Number of sectors per logical cluster
        2. Number of reserved sectors (not part of file system)
        3. Number of Alternate File Allocation Tables
        4. Number of entries in the root directory
c. FDISK table
    i. As disks get larger: can put multiple file system on each disk
        1. Need to partition the disk into logical sub-disks => FDISK table
    ii. FDISK table has 4 entries, each containing:
        1. A partition type (e.g. Primary DOS partition, UNIX partition)
        2. An ACTIVE indication
        3. The disk address where that partition starts and ends
        4. The number of sectors contained within that partition
    iii. The addition of FDISK changed the structure of bootstrap record
        1. The first sector of a disk contains the Master Boot Record (MBR)
        2. Everyone except Bill Gates make their MBR bootstrap ask which system you wan to boot from, and boot the active one from default after a few seconds
            a. Microsoft make the choice for you: you want Windows
    iv. Structure of partition boot record is entirely OS specific

4. File descriptors (directories)
    a. DOS combine both file description and file naming into one single file descriptor (directory entry)
    b. DOS directory is a file (of special type) that contains a series of fixed sized (32 bytes) directory entries, each entry describes a single file:
        i. An 11-byte name (8 char base name + 3 char extensions)
            1. If first character of file name is NULL (0x00), the directory entry is UNUSED
            2. If first character of file name is 0xE5, the directory entry is a deleted file
        ii. A byte of attribute bits for the file, which include:
            1. File or sub-directory
            2. Has this file changed since the last backup
            3. Is this file hidden
            4. Is this file RDONLY
            5. Is this a system file
            6. Does this entry describe a volume label
        iii. Times and date of creation, last modification, last access
            1. Time and dates stored in 16-bit pattern -> until 2107
                a. 7 bits of year, 4 bits of month, 5 bits of day of month
                b. 5 bits of hour, 6 bits of minutes, 5 bits of seconds

            2. All file system date in DOS is relative to midnight Jan 1, 1980
        iv. A pointer to the first logical block of the file
        v. Length (# of valid data bytes) in the file

5. Links and free space
    a. If a block is allocated, FAT entry gives the logical block number of the next logical block in file
    b. Cluster size and performance
        i. Space is allocated to files not in physical blocks but in logical multi-block clusters
        ii. The number of clusters per block is determined at file system creation time
        iii. Allocating space to files in bigger chunks
            1. Improve I/O performance by decrease r/w operations
            2. Cost: higher internal fragmentation
                a. On average, half of the cluster of each file is unused
                b. BUT as disk grow larger, people worry less on internal frag
        iv. Max number of clusters a volume can support depends on the width of FAT table
    c. Next block pointers
        i. A file's directory entry contains a pointer to the first cluster of that file
        ii. The FAT entry for that file cluster tells the cluster number NEXT cluster in the file
            1. Last FAT entry of file: -1



        iii. The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of ar file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential
        iv. FAT is keep in memory when file system is in use
        v. => successor blocks can be looked up w/o extra I/O
    d. Free space
        i. To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.
    e. Garbage collection
        i. Does not bother to free blocks when file is deleted, but mark it with 0xE5
        ii. File system regularly run out of space
        iii. Garbage collection:
            1. Starting from root directory, find every valid entry
            2. Mark not referenced clusters as free in FAT
        iv. +: files cannot be reallocated until GC => recoverable

6. Descendents of the DOS file system
    a. Long file names
        i. Solution to complaints on time/date filename:
            1. Put the extended filenames in additional directory entries
            2. Each file would be described by an old-format directory entry, but supplementary directory entries could provide extensions to the file name
            3. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label)
        ii. Problem to old filenames:

        1.
    b. Alternate/back-up FATs
    c. ISO 9660
7. Summary