**Introduction**

OS: Software which makes computer usable. File system, comms system, process mgr, security mgr, memory mgr.

Kernel: program which is always running

Monolithic kernel: all parts of OS in 1 program. Adv: any section can operate with any other. All in same addr space, fast and efficient. Disadv: no protection between systems.

Micro-kernel: Safe, reliable, slower. Carries out bare necessities.

Manager model: OS is collection of managers. Each manager is independent.

Onion model: OS is series of layers. Outer layers can access inner layer resources but not vice versa.

**History**

Off-lining: smaller computers did slow IO, first parallelism

**Resident monitors**: Program always in memory, controlled by Job Control Language (JCL), starting point of OS. Clear memory of last program, load next program, jump to start addr of new program. Maintain standard IO routines in mem. Two programs in memory. Distinction between data and programs.

Interruptible processors: changed how IO was performed, use of a stack instead of return addresses. IO and CPU can execute concurrently. IO goes to local buffer of controller. CPU moves data to main mem from local buffer. Device controller informs CPU of op finish using interrupt.

**SPOOLing**: Simultaneous Peripheral Operation On-Line. Time waiting for IO can be used.

**Multiprogramming:** Have several programs running in a machine.

**Memory protection:** Prevent unauthorized programs from accessing memory outside assigned range. Need operating modes and privileged instructions + limited address range

User mode: execution done on behalf of user.

Kernel mode: execution done on behalf of OS

Mode bit: In processor status register to indicate which mode the processor is operating in. Interrupts, syscalls, faults cause processor to change mode and jump to a location. Privileged instructions cannot be executed in user mode.

**Batch systems**: Each job has own mem. Scheduling automated. Includes: multiprogramming, interrupts, scheduling, file system, mem mgmt.

**Time-sharing systems**: Multiple people working at machine, CPU time is split amongst them. Requires more security and admin.

Network OS: file sharing and comm scheme, runs independently from other computers on network.

Distributed OS: less autonomy between comps, impression of single OS controlling network.

Loosely couple systems as opposed to tightly coupled.

**Multiprocessor Systems**

Tightly coupled system: processors share mem and clock

Parallel systems increase throughput. Increase reliability, graceful degradation.

Symmetric multiprocessing (SMP): Each processor runs identical copy of OS. Many processes run at once w/o perf degrading.

Asymmetric multiprocessing: Each processor assigned task; master processor schedules and allocates work to servant processors.

**Real-time systems**

Hard real-time: must satisfy reqs within definite time periods or system fails.

Soft real-time: not too impactful if time constraint not met exactly.

**Virtual Machines**

Often a layer between hardware and OS

Adv: Can choose OS. Modify/develop OS w/o crashing machines having to reboot. More safety, isolation between user’s VMs. Virtual servers.

Disadv: Some resources allocated to one VM can’t be shared. Extra layer of complexity. Trying to reduce perf hit

Virtual servers: many apps assume they are the only thing running on a machine. Using virtual servers, multiple servers can be running simultaneously on 1 machine. Errors/security problems do not affect other servers. Cost effective. Flexibility – VMs can be migrated between machines w/o rebooting. Easy to copy VM

**Virtualization**

Fidelity: software should run identically

Performance: most instructions in VM run on hardware (so similar speed)

Safety: VM Manager (VMM) in control of resources and must be safe from any VM actions. VM’s must also be safe from each other.

Host OS: OS which VMM runs on.

Guest OS: OS running in VM

**Hypervisor Types**

Type 0: implemented in hardware and firmware, loads at boot time. Guest OSs load into partitions separated by hardware. Allocated dedicated resources. Guest OSs are native w/ subset of the hardware

Type 1: Special purpose OSs, load at boot time, provide optimised enviro to run guest OSs. Run in kernel mode. Implement devices drivers and guests access devices through them. Provide services to manage guests (backup, monitoring).

Type 2: Run as apps on host OS (e.g. VMWare Workstation and VirtualBox). In modern type 2 VMs part of the VMM runs in kernel mode.

**Trap and Emulate**: privileged instructions cause exception, VMM detects and emulates instruction for guest OS then returns back to VM.

**Virtualization problems**: sensitive instructions ran in both user and kernel modes. So not causing a trap in VMM. Some instructions allowed program to determine if it was running in privileged mode. Breaking fidelity requirement. Problems protecting page table info and consistency.

Binary translation: Code running in kernel mode translated at run-time into code which causes exception passed to VMM. Very simple and efficient. Most code is same as original. Translated code is cached and reused. V good perf v.s. true hardware virtualization.

**Hardware virtualization**: Extra high privilege area for VMM, OS runs in ring 0 (kernel mode). Processor state is maintained for each guest OS and VMM in separate address spaces. Context switching between guests done in hardware. Second Level Address Translation (SLAT) – determine guest physical address from guest virtual address, determine host physical address from guest physical address.

**OS level virtualization:** When virtualizing servers, we can often use same OS. Containers look like servers but all use underlying kernel and are separate from each other.

WSLv1**:** User level pico processes and kernel drivers. I/O slow, no Linux kernel.

WSLv2: Revert to more traditional Linux running in VM on Hyper-V

**C**

Low-level access to mem, easy to map to machine instr, easy to inline assembly language code, small runtime reqs.

pointers: Can have integers assigned, which repr actual addresses.

Volatile: do not optimise accesses to this var

Memory mgmt: local vars disappear when functions return. Space is allocated on stack for vars in each function invoc. Areas of static mem, global vars, static vars are stored here. Static mem alloc at compile time, cannot be easily released.

Dynamic mem: requires explicit control. No garbage collection available. Malloc and free

**Processes and Threads**

**Process**: An instance of a program in execution AKA heavyweight process. Made up of resources (things process owns + info about self) and streams of execution. Resource is called task/job, location called thread.

**Thread**: lightweight process. A sequence of instructions being executed when there is no external intervention. Easier to create than process, easier to switch between. Encapsulate a problem within process rather than multiple processes. Share all resources, except call stack and other local storage.

Uses: splitting work across processors, add responsiveness, controlling and monitoring threads, server apps, program abstraction.

**User-level threads**

OS only sees one thread per process, constructs other threads by user-level lib calls or by hand. User-level control over starting/stopping. Usually req made to OS to interrupt process regularly to schedule another thread.

Works w/o OS support for threads. Easier to create (no syscall), no switch to kernel mode. Control can be app specific. Easier to switch between. When one thread blocks, all threads in that process block.

Jacketing: A blocking syscall has user-level jacket. Jacket checks if resource is available, if not another thread is started.

**System-level threads**

OS knows about multiple threads per process. Threads constructed and controlled by syscalls. System knows state of each thread. pthread\_create, pthread\_join.

Each thread can be treated separately. No blocking of all threads when thread in kernel mode. Diff threads can be scheduled on diff processors.

**Process Control Blocks**

Process Control Blocks (PCBs) – where OS can find info about a process, in kernel space and user space(only needed when process is resident)

|  |  |
| --- | --- |
| * Memory | * Owner |
| * Open streams/files | * Which processor |
| * Devices | * Links to other processes |
| * Links to condition handlers | * Process group |
| * Processor registers | * Resource limits/usage |
| * Process identification | * Access rights |
| * Process state | * Process result |
| * Priority |  |

**Windows NT PCBs**

Executive Process Block (EPROCESS): KPROCESS and PEB, pid and ppid, file name of program, window station, exit status, create and exit times, links to next process, memory quotas and management info, ports for exceptions and debugging, security info

Kernel Process Block (KPROCESS): kernel and user times, pointers to threads, priority info, process state, processor affinity

Process Environment Block (PEB): includes info writable in user mode, image info: base address version numbers module list, heaps

**Process Table:** collection of PCBs, originally array, now dynamic collection of pointers to PCBs

**Thread Structures:** like PCBs, private memory, processor registers, thread identification, thread state

**Process Creation**

Find a spare or create new PCB; Mark as created; Generate unique identifier/id; Get some memory; Fill page table entries; Sets up PCB values; Set priority and resource limits; Change state to runnable

**UNIX Fork**: duplicates currently running process, memory duplicated, share open files, open file info blocks increate count of processes accessing them, shared memory regions

**UNIX Exec:** checks if executable, save parameters to system memory, release held memory, load program, moves parameters to stack space of program, then run, bad with fork as copied memory not used

* Copy on write: no copy made first, data pages made read only, if write, then exception occurs and data duplicated for child
* vfork: parent blocks until child finishes or calls exec, programmers must know what to do

NT Process Creation

|  |  |
| --- | --- |
| * Open exe * Create NT process object * Setup EPROCESS block * Create initial address space * Create KPROCESS block * Finish setting up address space | * Add process block to end of active processes * Set up PEB * Create initial thread * Notify Win32 subsystem * Start initial thread * Load and initialize DLL |

**UNIX Pipes**: data in one side out the other, reading blocked when pipes empty, writing blocked when pipes full

* Broken Pipes: reading when no writer gets EOF, writing with no reader gets signal
* Limitation: can only communicate between related processes, file handles are just low integers which index file table, same numbers only make sense in same process

Runnable: single core only 1 process/thread can run at a time, but many may be runnable (running or ready to run)

**Preemptive Multitasking:** a clock interrupt causes OS to check if current thread should continue, under control from OS

* Adv: control, predictability
* Disadv: critical sections, efficiency

Cooperative Multitasking: either process yields its right to run, or it makes a system call thus stopping process

**Context Switch**: change from one process running to another running on same processor, context involves registers, memory, files, resources and caches, state saved in PCB

* State Transition: restore properties to restart, page table may need altering, environment needs restoring

Waiting: processes seldom have the necessary data or memory to start, thus waiting processes state changed to waiting and removed from ready queue, when available changed to running and switches to runnable queue

Suspended: OS stops a process to allow others to run to completion, preserve work done if fault, or restart process in background, commonly swapped out of real memory

Java Suspend: can enter deadlock as suspend keeps threads

Java Stop: stop kills threads to release locks, but may end up in inconsistent state

UNIX Waiting: process in queue associated with hash value of kernel address, first come first serve

Finishing: reduce count on shared resources, update accounting, remove associated processes, remove user from system, account for all resources

Stopping Normally: call exit routine, tidy up

Forced Stops: only by parents or processes with same owner/group, cascading termination

UNIX Stopping – call exit, memory freed, accounting updated, become zombie, get init as stepparent, parent signaled and PCB freed

**Scheduling Processes**

* Batch Systems: keep machine going
* Time-Sharing Systems: keep user going
* Real Time Systems: priorities

Batch System Levels of Scheduling

* Very Long Term: before submission can user afford, administrative decisions
* Long Term: admin decisions, which jobs, resource requirements, CPU seconds
* Medium Term: if fuckup suspend and swap
* Short Term: which runnable goes next
* Dispatcher: code that context switches

**Scheduling Algorithms**

Explicit Priorities – unchanging, possible to get starvation

Variable Priorities – vary over life of process, prevents starvation

* First Come First Served (FCFS) – no time needed to determine what next, little overhead as context switch only when required
* Shortest Job First (SJF), need to know CPU burst time (usually don’t)
* Round-robin, using a time-slice
* First In First Out (FIFO)
* Linux Credit Algorithm – rerunning: credits=credits/2 + priority, when reaching 0 another process chosen, every tick credit lost, waiting processes got extra credits

(c, p, d) – (c)omputation time, (p)eriod time, (d)eadline

Computation time found through analysis or simulation

In sporadic processes, p is the minimum time between events

In aperiodic process p is 0

Cyclic Executives (CE) – handles periodic processes, or converted sporadic

Major Cycle Time – LCM of periods

**Priority Allocation**

Fixed

* Rate Monotonic (RM) – shorter period higher priority, OPT
* Least Compute Time (LCT) – shorter execution, higher priority

Dynamic

* Shortest Completion Time (SCT) – shortest job first with preemption
* Earliest Deadline First (EDF) – process with closest deadline highest priority, OPT
* Least Slack Time (LST) – least slack time highest priority where slack=deadline-time\_to\_complete, OPT

Optimal only true for single processors, for multiprocessors, pre-assign to CPU with heuristic, then schedule independently

**Critical Sections** – area of code in which we want only 1 thread at a time, mutual exclusion

Spin Locks/Busy Waits

|  |  |
| --- | --- |
| while locked  end  locked = true | * Bad because while waiting, thread wasting processor time * Between lines 2 and 3, multiple people can get lock * Not fair as FCFS |

Priority Inversion – priorities on processes and locking mechanism, lower priorities with lock can force high priorities to wait, but as low priority not run frequently

Priority Inheritance – higher priority process gives process with resource the priority of blocked process, thus high priority will only wait during critical section, thus solving priority inversion

Thread waiting is put on queue and stopped, thus fairness and wasting processor cycles

**Semaphore**, S – integer count, two atomic operations and an initialization

|  |  |
| --- | --- |
| V(S): # signal  S=S+1 | P(S): # wait  wait until S>0  S=S-1 |

Semaphore is initialized to 1, to get resource thread calls P, to return the resource, thread calls V

|  |  |
| --- | --- |
| signal(S):  if anything waiting on S then  start first on S queue  else  S=S+1 | wait(S):  if S<1 then  put process on S queue  else  S=S-1 |

**Readers/Writers Problem:** to ensure integrity of data: 1 writer access data at any time, if writer no readers, multiple readers allowed

* Writer Preferred – waiting writers before waiting readers
* Reader Preferred – readers before writers
* Neither Preferred – treat fairly with a queue
* Both 1 and 2 can lead to indefinite postponement

Monitors – an object which only allows 1 thread to be executing inside it, includes: shared resource, procedures, queue, scheduler, local state, initialization code, condition variables

Condition Variable – queue to hold threads, can call wait and signal operations

* Wait – puts current thread to sleep on queue, signal with nothing does nothing
* Signal – wakes up 1 thread from queue, wait always sleeps thread
* No internal state, simpler than semaphores

If this is the case, two threads running in monitor, thus big shit

* Stop thread that called signal
* Don’t start thread until current has left (preferred)

Java Monitors: synchronized, count with each lock, signal called notify, generally use wait with while

Semaphores equivalent to monitors and vice versa

**Deadlock** – circle of processes holding a resource wanted by another, resources cannot be shared, only owner can release, process can hold resource while holding another, cycle in resource graph

* Remove a process using age or priority
* Kill everything
* Force restart process or rollback
* To prevent, remove requirements

**Bankers Algorithm** – request granted repeat until no more processes can be finished (search for process which can be given all its resources, return all that processes to the system), if all processes can be removed then state is safe and allocation can go ahead

**Distributed Deadlock** – resource ordering, process ordering, Bankers algorithm (expensive)

* Wait-Die – A requests resource from B, if A older then it waits, otherwise A restarts, old privileges
* Wound-Wait – A requests resource from B, if A older then it takes resource and B restarts, otherwise A waits, big old privileges

**Distributed deadlock detection:** Each processor keeps track of the resource allocation graph to do with its local resources. May include remote processes. Cycles don't just occur locally. Need to check the union of resource allocation graphs.

Avoiding False Cycle Detection – A->B inserted into graph a, only inserted into graph b if b cannot immediately grant resource to a, ask all other sites for wait-for graphs, only insert edge if appears in more than one local graph with the same timestamp

* Distributed Approach – all local processes waiting external point to extra node Pex, if cycle with Pex, then possible deadlock and info communicated, if deadlock found in other site repeat, until deadlock detected or there is no cycle

**Centralized deadlock detection process.**

• Info may have changed by the time the data from the last machine is gathered, the data from the first machine is probably out of date.

• Graph is only an approximation of the real allocation of resources and requests. If there is deadlock it will be detected, but it is possible to detect deadlock when it doesn't exist.

• Timestamps can be used to avoid false deadlock detection. Central system can correct its clock and know if messages arriving are outdated

**Direct Communication** – process to process, address name/id, link, receiver doesn’t need to know id of sender, but can’t change name of processes easily

Indirect Communication – mailbox/ports, owned by system (survives without processes) or owned by process (created process, mailbox removed when finished)

Sockets – inter-process connection in distributed environment

* Stream – bidirectional, reliable, sequenced, unduplicated
* Datagram – bidirectional, but not reliable, sequenced or unduplicated
* Raw – access to underlying protocols

Socket Setup

|  |  |
| --- | --- |
| * Ssocket: make socket, specify domain and protocol * Sbind: associate name with socket * Slisten: ready to get connections | * Saccept: gets a connection and returns new socket * Csocket: make socket * Sconnect: makes connection between sockets |

Distributed Shared Memory – copy resource, but mark it read only, if process writes throw fault and send write request to original machine, to broadcast changes

Distributed Concurrency – locks semaphores and monitors all require shared memory, but doesn’t exist

* Centralized Method – have a process on one machine to coordinate requests, called the server or coordinator process, Request -> Reply -> Release
* Fully Distributed Method – request must be broadcast to all other processes in case resource is currently being used, process continues with secure access after hearing back from all processes, if inside critical section replies after, if not in critical section and does not want to enter replies immediately, if it wants to enter critical section, check to see which request happened earlier, cant use synchronized clocks need to use logical ones as could be slower, but fully distributed assumes all know each other and processors assumed not to fail
* Token-Passing Method – token passed around, can’t enter critical section until it gets and holds token, pass token when done, but tokens can get lost and processes can die, thus rings die

**File Systems**

File: named collection of related info that is recorded on secondary storage, related in that used by a program or is said program

Secondary Storage – disk drive/SSD, non-volatile memory

File System Requirements – storing info independent from program and non-volatile, infinite variety of data, naming the data such that it makes retrieval easy

Hard Disk Drive

* Overhead – time taken for CPU to start disk operation
* Positioning Time – time to initiate a disk transfer of 1 byte to memory
* Seek Time – time to position head over correct cylinder
* Rotational Time – time for correct sector to rotate under the head
* Transfer Rate – once initiated, rate of IO transfer/bandwidth

Access Methods – sequential or direct

File System Operations

* Create – space needs to be found for file, specify name and type, file descriptor for metadata, size of file
* Copy – creating new file then reading from old and writing to new, file info needs to point to original
* Change Attributes – some should be secure
* Delete – find dir entry release all file space so it can be reused, erase/mark as free
* Move – if on same device just change file info, otherwise will need to copy and delete
* Seek – repositioning within a file, change file position pointer, no need for IO
* Read – need to specify what data to read, how much where to put it, sequential: data is retrieved in same order as stored and read.next(), direct: easy on disk device specify exact data wanted with byte offset
* Write – similar to read but requires allocation of extra space, sequential: write pointer and write.next() every write, direct: seek to new position and write but can create holes

File Attributes

|  |  |
| --- | --- |
| * File name * Identifile – number to identify * Location – where stored with pointer * Size – bytes, blocks, records | * Owner info * Access info – who can do what * Dates and times * Type – e.g. txt exe docx |

**Directory Structure**

* **Single Level**: all files in single dir. Adv: works well with simple systems, Disadv: finding files as number of files grows and requires long filenames
* **Two Level**: create directories for each user with top level (MFD) with users and second level (UFD) for systems for each user, adv: allows same file name of diff user and more efficient searching, disadv: no growing capability
* **Tree**: as many as required through subdirectories, adv: efficient searching and grouping
* **Acyclic Graph**: any changes in subdir will affect both subdirectories, adv: allows file sharing and searching is easy due to diff paths, disadv: sharing files via linking and deleting may be a problem; if soft link and deleted then dangling pointer; if hard link and deleted need to delete all references associated

**Hard Links:** separate table with file info, dir entries just point to info in this table, each dir stores pointer to files inode, which holds rea info about the file

**Symbolic links**: only one true file entry in dir, others just reference this entry

**Per-file File Control Block (FCB)**: contains details about file such as inode number, ownership, permissions, size, dates

* UNIX is inode (info node) – count of hard links to file, contained in an inode table stored in multiple locations, UNIX dir is just table of names and inode numbers
* NTFS is MFT, each file has file record with attributes, folders have indexed table of file info (file name pointer to MGT, commonly ref file attributes), much is duplicated in dir entry
* MS-DOS is dir entry

**Allocation Methods**

* Contiguous Allocation – contiguous blocks, dir contains start block and blocks needed, table of block usages for each file, adv: simple and great for sequential and random access, disadv: external fragmentation
* Linked Allocation – each block contains pointer to next block, dir entry contains pointer to first and last block, adv: simple no fragmentation and good for sequential, disadv: bad for random access (improved by clustering but can cause internal fragmentation) reliability as damage to one block can propagate and takes up space due to the pointers taking up data block space (only a problem with small blocks
* File Allocation Table (FAT) – collection of linked list, each entry for each block, each dir holds block number for files first block which also indexes FAT.Adv: fewer disk accesses than linked access FAT can hold several blocks and can cache entire FAT to determine block numbers with no disk access: as FAT is bigger cannot cache whole thing
* Indexed Allocation – each file has own index block of pointers, block numbers in single contiguous table. Adv: good for direct access and no external fragmentation. Disadv: file size limited by indices in index table reliability as can lose whole index block and wasted space for index block

**NTFS Extents** – stores extents which is a start cluster (blocks) number and a length for the number of clusters

**Free Space Management**

* Bitmap – each bit represents a used or free block. Disadv: space
* Linked List (free list) – trivial to find first block. Adv: no waste of space no need to traverse entire list. Disadv: hard to get contiguous space and insufficient if we want several blocks at a time
* Grouping – modify linked list to store address of next n-1 free blocks, and a pointer that contains the next one
* Counting – keep address of first free block and count length, free space list then has addresses and counts

System Wide Open File Table – copy of FCB of each file and other info as system must keep track of all open files

Process Open File Table – contains pointers to entries to system-wide open file table as well as other info such as file position and pointer to buffer being used

File Buffer – holds data blocks from secondary storage

**In Memory File System Structure**

* Open() – searches file with name, verifies process has access rights, records file is open in system-wide open file table, construct entry in the process open file table, allocate buffer and return pointer/file handle for future access
* Open() and Read() – FCB and data blocks, system wide open file table and process open file table
* UNIX Runtime File Structure – read() -> process -> file -> inode -> data
* UNIX Open() – convert filename to inode, lock for exclusive access return error if no permissions, allocate system wide file table entry to point to inode fill process file table entry with pointer to system wide, unlock inode and return index into process file table entry (fd)
* UNIX Write() – get file table entry from fd check accessibility lock inode and start writing (request new block if no block, update inode), update file size and unlock the inode

Delay Write – buffers shared by system, write doesn’t occur until another process is to use buffer for diff block or daemon process flushes it. Adv: process wants to access info already in memory. Disadv: info not written immediately

UNIX Append – need atomicity as length of file might be changed, thus append mode when opening file

**File Versioning:** recover from mistakes, restore damaged files, compare versions

* Log Version – 11i54 54 chars inserted at position 11, to get previous versions go backwards through the log, very compact, access to current version is same as without versioning, slow to revert but can use checkpoints
* Multi-version B-Tree – A1 1-? Show which version the leaf is valid for where ? is up to the present, very compact, quick to revert, if many versions tree can be big and access to current version can be slow
* No method works if large changes/completely diff
* Windows XP checkpointing

Pruning – fixed number of versions, some changes more important, observe user behavior e.g. most accessed

Self-Securing Storage – metadata directories and critical files kept on versioning system, any intrusions tracked as intruder cannot erase changes they made, maintain all versions between checks for intrusion, system unable to keep enough versions signal alarm: not enough space allocated for normal usage or someone forced prune

**Distributed File System (DFS)** – file system with data stored in several sites/hosts, should be transparent and not distinguish between local and remote files

* Service – software entity running on one or more machines and proving function to prior unknown clients
* Server – service software running on single machine
* Client – process that can invoke service using operations form client interface (primitive file operations read etc)
* Adv: greater storage, administration flexibility, replicate info for reliability
* Client-Server Model: single client to single server, but ID can be spoofed, and encryption can be perf expensive
* Cluster Based Model: multiple data servers, master metadata server maps which data servers hold what, replication of data chunks, more fault tolerant and scalable
* Naming: location transparency, migration transparency which requires location transparency
* Multilevel Mapping: abstraction of file hides how and where file is stored

**Remote File Access**

* Remote-Service Mechanism: requests for access delivered to server, machine does access and forward to user, simple implementation as no consistency problem, less local memory matches local file access
* Caching: reduces network flow, but cache consistency problem, faster efficient and scales

**Caching Update Policy**

Write-through**:** every write requires block to be sent back to server. Reliable, poor perf

Delayed-write**:** send block to server at a later time (every 30s or on file close).

**Consistency Semantics:** the way changes in data get distributed between processes accessing the same files.

UNIX Semantics: Any change made by any process is immediately visible to any other. File is associated w/ single exclusive physical resource, may delay accesses,

Session Semantics (AFS): Writes to open file not visible during session, only at close. Can be several copies each changed independently. No user delay in reading/writing.

**Remote Service**

* Stateful – knows who has file open, what type of access etc. but if server crashes all state info lost thus difficult to restart, problems with processes which die
* Stateless – server maintains no info, merely responds to requests, server doesn’t have to worry about processes stopping as no records, no complicated recovery process

**Network File System (NFS):** establish connection between specific local dir (mount point) and a specific device, from a remote system. Stateless. Example of client-server model. Based off UNIX mounting system, no need for dedicated servers, mount servers on each machine, maintains list of machines mounted to dir, tries to follow UNIX system calls as closely as possible, admin difficult as anything can mount

* Automounter: mounts and unmounts remote directories on demand through maps, timeout after 5 minutes to unmount when unused
* VFS (Virtual File System) determines local or remote, MFS service layer makes remote procedure call (RPC) to machine, pumped to VFS on remote machine, carried out locally then result returned
* *mount srcIP:/user/share/dir1/ /user/local*

**Andrew File System (AFS):** like NFS local name space, shared name space of /afs, servers dedicated this time, files grouped into volumes, files identified by volume:vnode\_number:uniquefier, client machines run Cache Manager to find and retrieve files from host, cached in large chunks to reduce network traffic

* Changes not seen until shared files closed
* Callback is promise from server that cache is up to date, before file changed server breaks promises, then Cache Managers detect promise and refreshes thus not completely stateless

**Distributed System:** collection of loosely coupled nodes interconnected by a communications network

* Site – location of machine
* Node – specific system
* Server – has a resource
* Client – wants to use resource at diff site
* Adv – more work done, share devices, reliability, easier to expand

**Network Attached Storage (NAS)** – file storage over network, clients to servers e.g. NFS over network

**Storage Area Networks (SAN)** – deals with blocks rather than file systems, client deals with file system SAN provides block storage

Transaction – done successfully or not at all

**Two Phase Commit (2PC)**

* Commit request phase: <prepare> started protocol sent to all sites <ready> recorded returned if ok or <abort>
* Commit phase: <commit> if replied in time send to all sites <abort> sent by coordinator to rollback if went wrong

**Network OS**: communications on top of OS, user aware of diff machines

**Distributed OS:** aim to look like 1 machine, no difference except speed accessing local and remote resources

* Data Migration – transfer entire file or transfer portions necessary for immediate task
* Computation Migration – transfer computation rather than data across system through RPC
* Process Migration – execute an entire process at diff process, allows load balancing to even workload and computation speedup through concurrency on diff sites, move process closer to resource/user, backup
  + Not done for load balancing as too expensive

**Remote Procedure Calls (RPC):** hide message system so it looks like procedure calls, client calls stub, stub marshals and sends request to server stub, server demarshals and makes procedure call to server, vice versa

* Messages include params, version number, timestamps and source address

Protection – process can only access addresses in address space, with base and limit registers

* CPU must check every memory access to make sure it is within limits, otherwise throw error

**Memory Management**

Primary/Main: RAM, Secondary: disks, Tertiary: archival, Cache: one memory pretending to be another

Address Binding – make the connection between code and data and their addresses at diff times

* Compile Time – fixed addresses, not very flexible
* Load Time – more flexible but can’t change after loading, dynamic loading is don’t load unless called
* Run Time – As you go by the hardware

To make addresses of all processes start at 0

* Logical address coming out of process created by CPU
* Physical address used on address bus to memory

**Memory Management Unit (MMU)** – maps logical to physical address, base register called relocation register, the value of which is added to every address generated by user process when sent to memory.

* Rather than moving memory around for contiguous, have multiple memory sections for single process and have multiple base limit registers
* Pages – same size chunks, page number (p) is the index into page table which contains base address of each page in physical memory, page offset (d) is added to base address to get physical address
* Segments – variable size chunks, memory addresses become: <segment\_name, displacement>, but problems with external fragmentation
* Frame is a page sized chunk of physical memory allocated to a page from a process
* Page is a frame sized chunk of logical memory that fits in a frame

**Page Allocator** – first fit, next fit, best fit, worst fit

* Buddy Algorithm – leads to internal fragmentation, if 65k needed you get 128k
* Slab Algorithm – allocates data structures of same size, good for say PCB stuff, used in conjunction

Half Speed Memory – in all logical memory accesses at least 2 memory accesses are required, one for page/segment table and one for actual data

* MMU caches recent page table info in hardware cache called associative registers or translation look-aside buffers (TLBs)

**Effective Access Time (EAT):**

Beta: mem cycle time; Alpha: hit ratio, Epsilon: TLB lookup time

Programs Larger than Memory

* Handled early on by overlays, but programmer needed to split program into sections
* Multiprogramming nowadays allows swapping entire process to disk to provide space for others
* Swapping is slow so not done all the time
* Instead execute program without whole being in program. Adv: unused code doesn’t waste physical memory more memory for processes and speeding up responsiveness
* Allocate contiguous space for entire process in swap space – slows startup, speeds up later ops
* Allocate when page accessed first time – quick startup, new accesses slowed down
* Allocate when swapped out – don’t use swap space unless necessary
* Allocate space for changed data

**90/10 Rule:** almost all memory accesses within short span of time are in a small address space

* Temporal Locality – if process accesses item in memory, tend to reference same item soon
* Spatial Locality – if process accesses item in memory, will access adjacent item soon

**Demand Paging** – only brings page into real memory when page is used by process

* When starting process, all memory can be allocated, if not enough taken away from current pages, if still not enough, go into swap space

Page Faults – if memory access finds the valid bit of page table entry not set we get page fault, jumps to handling routine, checks if page is allocated an d if so find a free frame and read page from swap space, then fix page table entry, restart instruction

**Page Replacement** – when no free frames, need to find frame to replace

* Global – any frame can be chosen, number of frames depends on behavior of other processes
* Local – chose frames must come from own allocated frames, less frames to choose from

**Page Replacement Algorithm**

* Random – fair, easy to implement, with enough pages wont replace pages used too frequently
* FIFO – no priority depending on use, Belady’s anomaly
* Furthest From The Future – optimal but requires future knowledge, FIFO on tie break
* Least Recently Used – expensive to have hardware to keep track of past (counters or stacks). Can also use LRU-approximation, use reference bit for page table entry.
* Least Frequently Used – expensive adding every access
* Most Frequently Used
* Death Row – put in replacement pool depending on FIFO, keep track of which page in each frame

**Thrashing** – a process is busy swapping pages in and out, if sum of pages exceeds number of frames then thrashing. CPU utilization drops after certain point

**Working Set** – set of pages that a process referenced in the past T seconds

**Page Fault Frequency** – used to control number of frames allocated to a process

**Need to Know Principle/Principle of Least Privilege:** limit access to minimum required to achieve the allowed goals

**Protection Domains** – many processes can have the same rights if they execute in the same domain, collection of ordered pairs <object, rights>

* Multics Protection Ring Structure – inner domain rings (lower numbers) have privileges of outer and more

Set-User Identification (SUID) – UID becomes owner file, first part of UNIX -l

Set-Group Identification (SGID) – runs like member of same group of which file is a member of, second part

**Access Matrix:** rows represent domains and columns represent objects, Access(D, O) where D is the domain and O is the object, also can have domains as objects to allow switching to a diff domain if needed

* “\*” indicates that the permission can be copied in the same column
* “owner” indicates that any value on the object/column can be changed
* “control” allows removal of rights from another domain

Too much info in access matrix as bitmaps are shit, thus use capability list

**Confused Deputy Problem** – deputy has access to object, another process uses deputy should it be able to as it means it has access to said object, use capabilities to solve the problem

**Capability** – permission to access an object in certain ways, stored with domains and refer to objects and the access rights <f1, “read, write”>, the list itself is a protected object and should be encrypted/stored on hardware as in UNIX

When process needs to access a protected resource, capability is passed with request and checked with reference monitor to make sure the access is permitted

* Difficult to determine which domains have access rights to given object as they can be passed on as well

Capability security: use hardware or encryption. Store all capabilities in protected kernel mem. Extra bits for each word in mem, distinguish capabilities from ints/strs, only OS can create/change capabilities.

Uses: must only be created by OS. OS should construct owner capability for creator process when new object created.

Problems: difficult to revoke access perms. Keep track of all domains with capability; indirection, capability is to an intermediate obj which points to obj; reacquisition, have expiry time and then req capability again.

**Access Control Lists:** Each obj has list of domains and their access rights. e.g. <d1, “read, write”>. When req comes to obj from specific domain, reference monitor checks ACL to see if access is allowed. No revocation problems just change ACL.

AFS vs NTFS ACLs: AFS uses ACL for access perms only for dirs. NTFS it is possible to access file w/o dir perms.

Problems: Slow down file search ops. Unnecessary complex.

**Meltdown and Spectre Attacks**

Spectre and Meltdown both utilize the same basic idea

**Timing attack**: type of side channel attack, exploiting physical implementation.

**Speculative execution**: CPU performs speculative code while waiting on IO. Rollback everything if speculative code is not executed. There is no mem protection as this is purely a CPU operation (kernel is not involved)!

While speculatively executing, values are entered into RAM cache. The cache is not rolled back. Values can be discovered through a timing attack.

**Kernel page table isolation (KPTI)**: don’t map kernel memory into virtual address space of every process. Keep separate page table used in kernel mode. Process in user space cannot try to access kernel addresses. Decreases efficiency (5-15%) due to loading of new page table and flushing TLB.

**Cryptography**

Means to constrain potential senders (sources) and/or receivers (destinations) of messages. Based on secrets (keys). Enables confirmation of source, receipt only by certain destination. Trust relationship between sender and receiver.

**Encryption:** Constrains set of possible receivers of a message.

Encryption algo: Set K of keys; Set M of messages; Set C of ciphertexts (encrypted messages); A function E: K -> (M->C), for each k in K, Ek is a function for generating ciphertexts from messages; A function D: K -> (C->M), for each k in K, Dk is a function for generating messages from ciphertexts.

Encryption algo must: given ciphertext c, a computer can compute m that Ek(m) = c only if it possesses k.

Symmetric algo: encryption and decryption use same key

Asymmetric algo: diff keys for encryption and decryption. Impossible to produce one from the other.

**IO Systems**

Application IO interface: IO syscalls encapsulate device behaviours in generic classes. Device-driver layer hides differences among IO controllers from kernel. New devices talking already-implemented protocols need no extra work. Each OS has its own IO subsystem structures and device driver frameworks.

**Device driver**: Software which connects OS IO subsystem to underlying hardware of device.

Characteristics of IO devices:

|  |  |  |
| --- | --- | --- |
| Aspect | Variation | Example |
| Data-transfer mode | Character; block | Terminal; disk |
| Access method | Sequential; random | Modem; CD-ROM |
| Transfer schedule | Synchronous; async | Tape; keyboard |
| Sharing | Dedicated; sharable | Tape; keyboard |
| Device speed | Latency; seek time; transfer rate; delay |  |
| IO direction | Read only; write only; read-write | CD-ROM; graphics controller; disk |

Connections: Close connections between device drivers and kernel. Drivers need to reserve mem, setup interrupt handlers, privileged access to IO instructions. Driver are loaded and init at diff times. Some drivers need to be loaded at boot time.

UNIX: Special files created in /dev dir, marked as devices (c or b). Have major (which device driver) and minor (which device) device numbers, this info in the device’s inode.

**Unix IO structure**

**Block devices**: accessed 1 block at a time. Disks, tapes. Transfers buffered through block buffer cache. Blocks are cached as they are read/written. A hashtable connects device and block numbers to corresponding buffers. Reduces IO transfers. Read(), write(), seek()

**Memory-mapped file IO:** Layered on top of block-device drivers. File is mapped to range of mem addrs and paged into mem using virtual mem system.

**Character devices**: accessed 1 byte at a time. Everything which doesn’t use block buffer cache. Keyboards, mice printers… Raw and cooked tty input. Cooked input makes change to data before it is passed to requesting program (backspaces). Raw input is passed on exactly to program. Except for Secure Attention Key handling. Get(), put()

**Network devices**: access is inherently diff from local disk access, most systems provide separate interface for network devices.

**IO request block (IORB):** Data structure containing all info needed to complete IO req. Constructed by IO syscalls and passed to device drivers.

**User level device driver:** A device driver that runs in user mode. Not part of kernel or kernel module. Adv: no special perms needed, crashing doesn’t affect kernel, controlled access to kernel mode and other drivers. Disadv: moving between kernel and user mode has high efficiency loss.

=======================PAST ANSWERS================

**Per process file table**

File tables hold pointers to the global open file table. If the kernel has given a process permission to a file, then its entry will appear in the per process open file table.

If a process is forked, its per process file table is copied to the child process, so any child process has access to the resources of its parent, maintaining consistency. The per-process file table also holds permission info for the process, which would be too complex to handle for the global file table given they are process dependent.

**File system in user space**

Adv: Increases security. Prevents kernel crashes. Increases modifiability of file system functionality

Disadv: Slower, as syscalls need to be made through the kernel to perform operations

**Two problems to solve to implement network-transparent system?**

Making all processors and storage devices seem transparent across network. Distributed file system appears to user as a single file system.

Mobility of users. Allow users to connect to the “system” rather than to a specific machine.