MAIN MEMORY

* Cache sits between main memory and CPU registers
* Program must be brought from disk into memory for it to run.
* Main memory and registers are the only storage the CPU can access directly
* Base and limit registers define logical address space usable by a process – CPU must check every memory access in user to ensure it is within bounds
* Addresses are represented in different ways at different stages of a program’s life.
* Compiled code addresses bind to relocatable addresses at 3 possible stages:

▪ Compile time: If memory location known a priori, absolute code can be generated

▪ Load time: Must generate relocatable code if memory location not known at compile time

▪ Execution time: Binding delayed until run time if the process can be moved during its execution

* Compile-time and load-time are equally efficient with virtual and physical address binding, not the case with execution-time binding.
* User program deals with virtual addresses exclusively
* Unused/uncalled routines are never loaded.
* Static linking = system libraries and program code are combined by the loader into the program binary
* Dynamic linking = linking postponed until execution time
* Memory-Management Unit (MMU) = device that maps virtual to physical address. Methods use can vary
* Simple scheme - uses a relocation register which just adds a base value to address generated by user process
* Swapping - process swapped out temporarily to backing store (fast disk large enough to accommodate copes of all memory images) then brought back in for continued execution; total transfer time is proportional to the amount of memory swapped; allows total physical memory space of processes to exceed physical memory
* Roll out, roll in: swapping variant for priority-based scheduling.
* Contiguous allocation allows kernel code to be transient and variable kernel size - MMU maps logical addresses dynamically
* Solutions to Dynamic Storage-Allocation Problem:
* First-fit: allocate the first hole that is big enough
* Best-fit: allocate the smallest hole that is big enough (must search entire list) → smallest leftover hole
* Worst-fit: allocate the largest hole (search entire list) → largest leftover hole
* Contiguous allocation allows kernel code to be transient and variable kernel size - MMU maps logical addresses dynamically
* External Fragmentation = when total memory space exists to satisfy request, but it is not contiguous - Reduced by compaction: relocate (only if dynamic) free memory to be together in one block
* Internal Fragmentation = when allocated memory may be slightly larger than requested memory
* SEGMENTATION = memory-management scheme that supports user view of memory. A program is a collection of segments. Logical address = tuple <segment\_number, offset>. Segment table maps 2D physical addresses, each entry has: base (starting phys address) & limit (length of segment). Code Sharing occurs at segment level.
* PAGING = Physical memory divided into fixed-sized segments called frames (size is power of 2) ; Logical memory divided into same sized blocks called pages ; Page table (kept in main memory) = translates logical to physical addresses - Page number (p): used as an index into a page table / Page offset (d): combined with base address to define the physical memory address ; Free-frame list is maintained to keep track of which frames can be allocated ; use associative memory or TLBs to reduce overhead.
* Diagram, schematic

  Description automatically generatedTLB (translation look-aside buffer) = a CPU cache that memory management hardware uses to improve virtual address translation speed ; Typically small – 64 to 1024 entries ; On TLB miss, value loaded to TLB for faster access next time
* Associative Bit with each frame to indicate RWX permissions
* Can have multilevel page tables (paged page tables)

Chart, diagram, waterfall chart

Description automatically generatedChart

Description automatically generated

Two-level paging Hashed page tables

* In hashed page tables address spaces > 32 bits. Each element contains: 1. Virtual page number 2. The value of the mapped page frame 3. Pointer to next frame

VIRTUAL MEMORY

* Desenho todo catita da stack a crescer para baixo e a heap a crescer para cima.
* System libraries shared via mapping into virtual address space.
* Shared memory by mapping pages into virtual address space
* Allows pages to be shared during fork(), speeding process creation
* Demand paging = never swaps a page into memory unless needed (lazy swapper), the pager deals with page swaps. If a page is needed and it is not memory resident, grab page from storage into memory
* Page fault = first time there is a reference to a specific page, it traps the OS. OS must decide to abort if the reference is invalid, or if the desired page is just not in memory yet. If the latter: get empty frame, swap page into frame, reset tables to indicate page now in memory, set validation bit, restart instruction that caused the page fault
* If an instruction accesses multiple pages near each other → less “pain” because of locality of reference
* Pure Demand Paging: process starts with no pages in memory
* Use modify (dirty) bit to know if page has been altered and if it needs to be written on the disk. Reduces the overhead of page transfers.
* Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory. If either process modifies a shared page, only then is the page copied (one copy per child / parent).
* What happens if we have no free frames for a needed page? We select a victim frame and replace it. There are many algorithms to determine what page gets whacked:
* FIFO algorithm – oldest page is replaced
* Optimal algorithm – replace page that will not be used for the longest time (we cant predict the future tho)
* Least-recently-used algorithm
* Enhanced second-chance algorithm
* There are 2 ways for allocating memory for processes:
* Fixed allocation – each process gets the same number of frames
* Priority allocation – if page fault, replace one of its frames (local allocation) or frame from another process (global allocation)
* Thrashing: a process is busy swapping pages in and out → minimal work is actually being performed
* Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
* I/O Interlock: Pages must sometimes be locked into memory

FILE SYSTEM INTERFACE

* File = Uniform logical view of information storage (no matter the medium). An abstract data type. Types: Data (numeric, character, binary), Program, Free form, Structured. Content decided by file’s creator
* File Attributes: Name: Only info in human-readable form ; Identifier: Unique tag ; Type ; Size ; Location: pointer to file location ; Time ; date ; protection (chmod – permissions)
* File operations: create, write, read, reposition within file, delete, truncate
* Open-file table keeps track of all open files, including a pointer to last read/write location per process
* Open file locking: mediates access to a file (shared or exclusive)
* Mandatory – access denied depending on locks held and requested
* Advisory – process can find status of locks and decide what to do
* Line structure:
  + Simple record structure – lines, fixed or variable length
  + Complex structures – formatted document, relocatable load file
* Access Methods:
  + Sequential Access: tape model of a file
  + Direct Access: random access, relative access
* Disk can be subdivided into partitions which can be RAID protected against failure. It can also be used raw – without file system.
* Partitions = mini-disks, slices
* Volume = entity containing file system
* Directory is similar to symbol table – translating file names into their directory entries. Should be efficient, convenient to users, logical grouping.
  + Single-level directory – directory lead straight to files ; naming & grouping problem
  + Two-level directory – separate directory for each user ; grouping problem
  + Tree-structured directory – allows grouping
  + Acyclic graph directories - adds ability to directly share directories between users
    - Acyclic can be guaranteed by: only allowing shared files, not shared sub directories; garbage collection; mechanism to check whether new links are OK
* File system must be mounted before it can be accessed – kernel data structure keeps track of mount points.
* In a file sharing system User IDs and Group IDs help identify a user's permissions
* Controlled Access: when file created, determine r/w/x access for users/groups/others (chmod)

FILE SYSTEM IMPLEMENTATION

* File control block = storage structure consisting of information about a file (exist per-file)
* File system resides on secondary storage – disks; file system is organized into layers: application programs -> logical file system -> file-organization module -> basic file system -> I/O control -> devices
  + Device driver = controls the physical device; manage I/O devices
  + File organization module = understands files, logical addresses, and physical blocks ; Translates logical block number to physical block number ; Manages free space, disk allocation
  + Logical file system: manages metadata information – maintains file control blocks
* Boot control block: contains info needed by system to boot OS from volume
* Volume control block: contains volume details; ex: total # blocks, # free blocks, block size, free block pointers
* Directory structure organizes files
* Per-file File Control Block (FCB) contains many details about the file-inode number, permissions, size, dates, etc.
* Root partition: contains OS; mounted at boot time. For all partitions, system is consistency checked at mount time – check metadata for correctness, only allow mount to occur if so.
* Virtual file systems provide object-oriented way of implementing file systems. Linux has 4 object types inode, file, superblock, dentry. VFS defines set of functions objects must implement.
* Directories can be implemented as:
  + Linear Lists of file names with pointer to data blocks – simple but slow
  + Hash Tables – linear list with hash data structure, decreased search time; Good if entries are fixed size; Collisions can occur (two file names hash to same location)
* Allocation Methods:
  + Contiguous allocation: each file occupies set of contiguous blocks – Simple, best performance in most cases; problem – finding space for file, external fragmentation. Extent = contiguous allocation of blocks scheme. File = 1+ extents
  + Linked Allocation: each file is a linked list of blocks – no external fragmentation; Locating a block can take many I/Os and disk seeks: File Alocation Table (FAT) Variation – beginning of volume has table, indexed by block number
  + Indexed Allocation: each file has its own index block(s) of pointers to its data blocks – can be random access, dynamic access without external fragmentation but has overhead
* Best methods: linked good for sequential, not random; contiguous good for sequential and random; Indexed can reduce CPU overhead. Declare access types at creation, either contiguous or linked
* File system maintains free-space list to track available blocks/clusters, a bit vector or bit map (n blocks). It requires additional space
  + Grouping – modify linked list to store address of next n-1 free blocks in first free block
  + Counting – Keep address of first free block and count of following free blocks
  + Space maps – divide device space into metaslab units and manages metaslabs
* Efficiency dependent on: disk allocation and directory algorithms, data type, fixed or varying-size data structures
* Buffer cache – separate section of main memory for frequently used blocks
* Page cache – caches pages rather than disk blocks using virtual memory techniques and addresses. Memory mapped I/O uses a page cache while routine I/O through file system uses buffer (disk) cache
* Unified buffer cache: uses same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
* Consistency checking – compares data in directory structure with datablocks on disk and tries to fix inconsistencies. Can be slow and sometimes fail.
* Log structured file systems – record each metadata update to the file system as a transaction. Faster recovery from crash

I/O SYSTEMS

* Device drivers encapsulate device details and present uniform device access interface to I/O subsystem.
* Port: connection point for device
* Bus: daisy chain or shared direct access
* Controller (host adapter): electronics that operate port, bus, device – sometimes integrated. Contains processor, microcode, bus controller, private memory.
* I/O instructions control devices. Devices usually have registers where device driver places command addresses and data from registers after command execution
* Polling for each byte of data – busy-wait for I/O from device. Reasonable for fast devices, inefficient for slow ones. Can happen in 3 instruction cycles – read status, logical-and to extract status bit, branch if not 0.
* CPU interrupt-request line is triggered by I/O devices, checked by processor after each instruction.
* Interrupt Handler – receives interrupts, maskable to ignore or delay some interrupts
* Interrupt vector – to dispatch interrupt to correct interrupt handler, some are non-maskable, based on priority.
* Interrupt chaining occurs if there is more than one device at the same interrupt number
* Interrupt mechanism is also used for exceptions
* Direct memory access is used to avoid programmed I/O for large data movement. Requires DMA controller. Bypasses CPU to transfer data directly between I/O device and memory.
* OS writes DMA command block into memory. Version that is aware of virtual addresses can be even more efficient (DMVA)
* Device driver layer hides differences among I/O controllers from kernel. Devices vary in many dimensions: character stream/block, sequential/random access, synchronous/asynchronous, sharable/dedicated, speed, rw/ro/wo
* Block devices include disk drives – commands include read, write, seek. Raw I/O, Direct I/O ir file-system access.
* Blocking I/O: process suspended until I/O completed
* Nonblocking I/O: I/O call returns as much as available – implemented via multi-threading, returns quickly
* Asynchronous: process runs while I/O executes – difficult to use, process signaled upon I/O completion.
* Vectored I/O: allows one system call to perform multiple I/O operations. Decreases context switching and overhead.
* Spooling: hold output for a device, if device can only serve one request at a time (ex: printer)
* Device Reservation: provides exclusive access to a device – must be careful of deadlock. System calls for allocation and deallocation.
* User process may attempt to disrupt normal operation via illegal I/O operations. All are privileged and must be performed via system calls.
* Kernel keeps state info for I/O components, including open file tables, network connections, character device states. Complex data structures track buffers, memory allocation, “dirty” blocks
* STREAM: full-duplex communication channel between user-level process and device in UNIX. Each module contains read queue and write queue.