Introduction to Operating Systems EECS 482

hcentner@umich.edu

Harrison Centner
Prof. Brian Noble

VIRTUAL MEMORY

Overview

Virtualizing memory is useful because it provides

(a) Process isolation. Protection from malicious code. Sandboxing.

(b) Interprocess communication.

(c) Shared code segments. Common libraries. Reduce memory footprint.

(d) Program initialization. A program can begin before all of its code has been laoded form disk.

(e) Efficient dynamic memory allocation. As the heap or stack grows, we only need to trap to the kernel to allocate more memory.

(f) Page coloring allows for cache management.

(g) Debugging. Catching pointer errors (using small physical address numbers for pointers).

(h) Efficient I/O.

(i) Memory mapped files. Allows programs to directly reference files.

(j) Checkpoints and restarting programs in case of a crash.

(k) Persistent memory.

(l) Process migration. Moving an executing program transparently from one server to another, possibly for load balancing.

(m) Information flow control. Provides added security.

(n) Distributed shared memory. Transparently turn a network of servers into a large-scale shared-memory parallel computer.

Strategies

Base and Bound provides a process with a base register and a bound register. Any reads to memory outside this range results in a segmentation fault.

Advantages

- Fast.
- Easy to implement.

Disadvantages

- Difficult to share resources between processes.
- Memory must be contiguous.
- Difficult to support dynamic memory like growing heaps, memory mapped files, and thread stacks.
- No way to prevent program from overwriting its code.

Segmented Memory provides a process with an array of base and bound registers. Any reads to memory outside these ranges results in a segmentation fault.

Advantages

- Fast.
- Easy to implement and add features:
  read-only memory
  shared memory.
  inter process communication.
- Efficient management of dynamically allocated memory. Requires zero-on-reference.

Disadvantages

- Large overhead of managing dynamically growing memory segments.
- External fragmentation. Memory is available but not contiguous.
- Compactification can solve fragmentation, but is expensive.
- Lot's of memory copying during program run.

Paged Memory provides a process with a page table containing page frames. Page frames are pointers to segments of fixed-size memory.

Advantages

- Little external fragmentation.
- Straightforward free space allocation.
- Easy to implement and add features:
  read-only memory
  shared memory.
  inter process communication.
- Efficient program startup, execution and loading occurs simultaneously.
- Page-granularity protection/sharing.

Disadvantages

- Requires virtual address translation.
- Internal fragmentation, large page size results in unused memory within pages.
- Compilers, runtime dynamic memory libraries expect contiguous memory.
- Might require multi-level translation.

Paging

Operating systems need to keep track of the hardware memory and the virtual memory. Hardware memory consists of (i) segments, (ii) multi-level page tables. These allow the OS to securely execute, load, and store data.

The software memory management data structures include:

(a) List of memory objects (local segments). Here the kernel keeps track of what memory regions represent underlying data, program code, library code, shared data, COW region, or memory mapped file. This allows the kernel to check for already linked libraries and keep reference counts to shared data.

(b) Virtual to physical translation. The kernel keeps track of whether a page is invalid, unreferenced, read-only, or simulating a breakpoint.

(c) Core map. The OS keeps track of the processes that map to physical memory locations, ensuring references are consistent when virtual addresses are updated.

An **inverted page table** is the name for a page table that uses hashing instead of linked trees. Standard techniques like chaining or rehashing are used to handle collisions.

The **Translation Lookaside Buffer** is a hardware table containing recent address translation (virtual page $\mapsto$ physical page). A TLB hit still needs to check for permissions[1] Some OS use *superpages* (contiguous pages) this reduces TLB miss rate because TLB entries can have a flag to indicate superpage status. Superpages are often used for the computer screen display and large matrices in scientific code.

As with any cache, precautions must be taken to ensure consistency of the TLB and memory:

(a) On a context switch the TLB can be flushed (incurring a performance penalty) or tagged (incurring complexity penalty) i.e. associated with given process.

(b) When the OS reduces permissions of a page, the TLB entry either needs to be removed or the TLB flushed.

(c) A **TLB shootdown** occurs when a PTE is modified. Every processor's corresponding entry in its TLB needs to remove the old entry. TLB shootdown requests are heavyweight and consequently often batched to reduce frequency of interrupts at a cost in latency.

[1] Even in the best case a trap to OS will take hundreds or thousands of instructions to process.

Cache Concepts

Locality is king.

Caches are ubiquitous and the TLB is just one example. *Virtually addressed caches* suffer the same consistency issues as the TLB. Oftentimes, chips also have a *physically addressed cache* which reduces costs of memory references and TLB misses. *Just in time compilation* and *conditional branch prediction* are prototypical examples where the cache is used.

Computations that have a *working set* gain strong benefits from caching. Computations with heavy-tailed distributions gain only logarithmic improvements from better caching.

Some types of caching policies are:

(a) Random. Never the worst possible choice (in expectation).

(b) FIFO. The worst possible choice for programs that scan memory.

(c) MIN. Evict the block that will be used farthest in the future. MIN is optimal, but uncomputable.

(d) LRU. Evict the least recently used. Optimal for programs that adhere to temporal locality. Pessimal for programs that scan memory.

(e) LFU. Evict the lest frequently used. Optimal for heavy tailed distributions.

Poor eviction policies can lead to *thrashing*: a constant state of paging and page faults.

Evicting a dirty page is more expensive than evicting a clean page, so the OS can make a thread proactively clean pages.

Memory-mapped files are segments of virtual memory that are assigned byte-by-byte correlation with a file. This benefits I/O performance, especially for large files, because memory because file read/write operate on a copy whereas load/store do not. Memory mapped files treat memory as a write-back cache for disk. In essence, allowing applications to treat files like virtual address space.

PERSISTENT STORAGE

Storage Devices

Memory is stored on physical devices the most common are magnetic disks (used in servers and older computer) and flash or solid state drives (used on laptops, phones, etc.).

Magnetic Disks have a *write head* located on an arm that levitates over a rapidly spinning disk. Memory writes occur over *sectors* of fixed sizes which lie along concentric circles within the disk called *tracks*. Disk drives often include a few MB of buffer memory. Buffer memory is used for *track buffering* which reads sectors that have not yet been read, but are anticipated to be read. *Write acceleration* is a technique to acknowledge (to the OS) writes that have not actually occurred, risks lost data on crashes.

To memory read/write a magnetic disk must:

(1) Seek the logical block address to read/write to.

(2) Rotate the disk under the head so that the read/write can occur.

(3) Transfer data to or from buffer memory.

Consequently, sequential disk writes are orders of magnitude faster than random disk writes. To minimize the ping-ponging of disk writes OSes need to be smart about scheduling. FIFO scheduling is slow but is safe. *Shortest seek time first* scheduling writes the pending request that can be serviced quickly, unfortunately this can lead to starvation.

Solutions include Elevator, SCAN, and CSCAN. Algorithms that implement directional SSTF.

Flash Drive stores bits in transistors, contains no moving parts, and is less vulnerable to physical damage. Flash storage can be implemented with NAND or NOR gates; NAND gates allow denser memory storage. Flash memory is accessed with three operations:

(a) Erasing an erasure block. Erase occurs by writing a logical 1 and can only occur in large units called erasure blocks.

(b) Once erased, NAND flash memroy can be written on a page-by-page basis.

(c) NAND flash memroy can be read on a page-by-page basis.

Internally, flash devices can have independent, parallel data paths, therefore, OSes can issue multiple concurrent requests to flash.

Flash drives are vulnerable to *Row Hammer attacks*: sequential reads to memory without nearby writes causing electrical disturbances and allowing privilege escalation. Flash drives require *wear-leveling* to distribute writes uniformly so that portions of memory with "hot pages" do not prematurely fail. Often manufactures ship flash drives with spare pages and erasure blocks so that a "full" flash drive can still copy out live pages, allowing wear leveling.

The *TRIM* command was introduced to inform underlying storage systems that certain pages are not needed, thus reducing garbage collection overhead. Previously OS file systems would just update the file metadata and the file system's free space map. Hardware demands affecting OS interfaces!

RELIABLE STORAGE

Physical devices do not last forever. As operating systems architects we should build memory systems that are robust to spontaneous combustion of the physical storage devices.

Recall that a system is *reliable* is it performs its intended function.
A system is *available* if it can respond to a request.

The two main ways a file system becomes *corrupted* are

(1) loss of data and

(2) crash or power failure during an update.

We will attempt to solve these problems via (a) *transactions* and (b) *redundancy*.

Transactions

Transactions provide the illusion of atomicity for non-atomic memory operations, so that file systems are robust to crashes during non-atomic operations. Transactions provide a way to perform a set of updates while maintaining the *ACID properties*:

(a) *Atomicity*. A transaction either *commits* or *rolls back*, meaning all updates take effect or none.

(b) *Consistency*. Invariants are preserved under transaction commits.

(c) *Isolation*. Each transaction executes unaffected by other in-progress transactions.

(d) *Durability*. A committed transaction survives crashes.

Two ways these properties are accomplished are *write-ahead logging* and *shadowing*.

Shadowing keeps two copies of the file system. Updates are made to the "shadow" copy then a persistent pointer is atomically updated to point to the shadow copy which becomes current. Updates are replayed onto the old current copy (now the shadow copy). Optimizing, we see that a shadow copy is only needed for some portion of the system affected by the update.

Write-ahead logging writes updated to an append-only log before applying to file system. Upon appending a set of updates to the log, a commit is added. Now, the OS updates the file system, marks the commit completed, and clears the log. Updates must be *idempotent*.

Upon a crash and reboot and logs with a commit are replayed and any without are discarded.

Advantages	Disadvantages
• Logging is easier to implement.	• Logging writes all updates twice.
• Shadowing is more efficient.	• Shadowing must copy, modify, replay.

Most OSes log the meta-data, but not the data. Not all data needs ACID guarantees and data is opaque to the OS. **sync()** and **fsync()** ensure the durability of updates by writing to persistent storage (updates might be cached in volatile memory).

Redundancy

Storing redundant information on inexpensive disks is not a bad idea because engineering perfect memory systems is not possible! So, storing "copies" of data on cheap disks inexpensively makes a memory system robust to partial failure.

RAID (AKA redundant array of inexpensive disks) spreads data redundantly over multiple disks. This is achieved through *mirroring* or *rotating parity* methods.

(i) *Striping* (or RAID 0) stores sequential blocks on multiple disks. This provides no redundancy but allows reads to execute in parallel.

(ii) *Mirroring* (or RAID 1) stores exact copies on two disks. Reads are fast, they can be executed in parallel. Writes are slow because all disks must be updated, the slowest disk determines the write speed.

(iii) *Rotating parity* (or RAID 5) stores blocks on $n + 1$ disks where n blocks contain storage information and the $(n + 1)$-th block stores a parity bit. If one disk fails the information can be reconstructed with an XOR of the remaining disks. For load balancing parity blocks are rotated among the disks. Striping is used to balance parallelism and sequential access efficiency: several blocks are stored on a disk before rotating to the next.

To further improve reliability with RAID we can (1) increase redundancy, (2) reduce nonrecoverable read error rates, and (3) reduce mean time to repair. Reed Solomon error correcting codes can accomplish (1), detection and scrubbing can solve (2), and having "hot spares" suffices for (3).

Backups with physical (another building) and logical (disabling writes) separation should be used to further protect systems. Software integrity checks and error correcting codes are a good idea too.

Some final methods to ensure atomicity with RAID are using a battery-backed nonvolatile write buffer, transactional update systems, and recovery scans.

FILE SYSTEMS

Memory is stored in filesystems. We should keep in mind that (i) most file accesses are reads, (ii) most processes access files sequentially and entirely, and (iii) most files are small, but most memory belong to large files.

One important command is **fsync()**. The **fsync()** command flushes cached writes in volatile memory and ensures their durability by writing to persistent memory.

NETWORKING

OS architects want to provide a uniform abstraction for applications given the heterogeneous interfaces for networking: WiFi, Ethernet, Bluetooth. The OS multiplexes sockets (virtualized ports) to provide each process the illusion of multiple private Network Interface Cards.

Networking protocols include

(a) *IP*. Packet-based, best effort routing, and numeric naming. No security besides a non-cryptographic checksum.

(b) *UDP*. Built on top of IP so that ports, not just IP addresses, can be used.

(c) *TCP*. Provides reliability and ordering of messages with sequence numbers. Syn-ACKs are used by parties to complete TCP handshake and exchange messages.

(d) *TLS*. Cryptographic protocol built on TCP.

Remote Procedure Calls (RPCs) give clients the illusion of function calls that execute locally, but work is done by a server. Failures, performance issues, and service discovery break the local function call illusion.

SCHEDULING

Careful scheduling of jobs can improve response time, predictability, through, overhead, fairness, and likelihood of starvation.

Some scheduling policies are:

(i) *FIFO*. A good policy when all jobs are small.

(ii) *Shortest Job First*. Can lead to starvation of long jobs.

(iii) *Round Robin*[2]. Each job is prescribed a time quantum on the processor. Too short of a time quantum means context switching will be the brunt of work for the processor. Round Robin can add overhead without any benefit. Results in poor performance for disk bound jobs.

(iv) *Multi-level Feedback Queue*. Uses multiple round robin queues, each given a priority level and time quantum. I/O-bound tasks with modest compute requirements will be scheduled quickly, keeping disk busy. Compute-bound tasks will be deprioritized and given longer time quantum to minimize context switching. Priority inversion raises priority of jobs holding locks.

[2]Hyperthreading uses this strategy.

DISTRIBUTED SYSTEMS

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
>
> -Leslie Lamport.

Absolute temporal ordering is not possible in a distributed system. Lamport Clocks are functions LC such that $p \to q \implies LC(p) < LC(q)$ where $p \to q$ means there is a path p to q. A Lamport Clock generates a partial order on the space of events, a total order can be induced by placing a total order on the computers in the system.

Beware causal channels exist that are not recognizable by the system (phone call, carrier pigeon).

Theorem. (CAP Theorem)
Any distributed system can have at most two of the following desirable properties:
(1) Consistency. All users see the same data.
(2) Availability. All requests are satisfied.
(3) Partition Tolerance. Failures never prevent work.
CP (resp. AP) Systems block (resp. allow) potentially inconsistent updates.
CA systems do not work with any failure and are hence useless for large systems.

Client server systems can choose to *migrate* or *replicate* data. Migration is easier to implement, but concurrent reads lead to ping-ponging. Replication allows simultaneous reads, but creates potential for inconsistency.

Systems that use replication require a mechanism for concurrency control. The general principle used is reader/writer locks. Replicas can be (a) invalid, (b) shared, or (3) exclusive. Heartbeat server messages are used to determine if a replica is stale.

Pessimistic (CP) systems prevent disconnected clients from working and enforce agreement from connected clients.

Optimistic (AP) systems use version control to detect and resolve conflicts. Upon reconnection clients can send replay logs to server.

Aggressive revalidation often isn't the best idea and its need can be reduced by caching files on client stations and using "invalidation" or callbacks instead of revalidation.