

CSE 451: Operating Systems

Section 5

Midterm review

Kernel/userspace separation

- * Userspace processes cannot interact directly with hardware (non-privileged mode)
- * Attempting to execute a system call instruction causes a trap to the kernel (privileged mode), which handles the request
- * Why is it necessary to have both privileged and non-privileged mode?
- * How is privileged mode enforced, and how do virtual machine monitors work inside this model?

IO from userspace

- * Userspace processes interact with disks and other devices via `open()`, `read()`, `write()`, and other system calls
- * Multiple levels of abstraction: kernel surfaces filesystem to userspace, and device drivers surface (mostly) unified interface to reading and writing data to kernel
 - * What are the benefits and drawbacks of designing a system in this way?

Monolithic and microkernels

- * Monolithic kernels encapsulate all aspects of functionality aside from hardware and user programs
 - * Pro: Low communication cost, since everything is in the same address space
 - * Cons: No isolation between modules, not easy to tack on new features
- * Microkernels separate functionality into separate modules that each expose an API
 - * Pros and cons are opposite those of monolithic kernels
 - * Amazon's internal reorganization of services a few years ago had very much a microkernel vibe

Processes versus threads

- * Processes have multiple pieces of state associated with them
 - * Program counter, registers, virtual memory, open file handles, mutexes, registered signal handlers, the text and data segment of the program, and so on
- * Threads are “lightweight” versions of processes
 - * Which pieces of state listed above do threads not maintain individually?

Process creation

- * `fork()`: create and initialize a new process control block
 - * Copy resources of current process but assign a new address space
 - * Calls to `fork()` return twice—once to parent (with pid of child process) and once to child
 - * What makes this system call fast even for large processes?
`vfork()` versus copy-on-write
- * `exec()`: stop the current process and begin execution of a new one
 - * Existing process image is overwritten
 - * No new process is created
 - * Is there a reason why `fork()` and `exec()` are separate system calls?

Threads

- * How is a kernel thread different from a userspace thread?
 - * Kernel thread: managed by OS, can run on a different CPU core than parent process
 - * Userspace thread: managed by process/thread library, provides concurrency but no parallelism (can't have two userspace threads within a process executing instructions at the same time)
- * CPU sharing
 - * Threads share CPU either implicitly (via preemption) or explicitly via calls to `yield()`
 - * What happens when a userspace thread blocks on IO?

Synchronization

- * Critical sections are sequences of instructions that may produce incorrect behavior if two threads interleave or execute them at the same time
 - * E.g. the banking example that everyone loves to use
- * Mutexes are constructs that enforce mutual exclusion
 - * `mutex.lock() / acquire()`: wait until no other thread holds the lock and then acquire it
 - * `mutex.unlock() / release()`: release the Kraken! Er, lock
 - * Mutexes rely on hardware support such as an atomic test-and-set instruction or being able to disable interrupts (why?)

Synchronization constructs

- * Spinlocks are mutexes where `lock()` spins in a loop until the lock can be acquired
 - * High CPU overhead, but no expensive context switches are necessary
 - * In what type of scenario are spinlocks useful?
- * Semaphores are counters that support atomic increments and decrements
 - * `P(sem)`: block until semaphore count is positive, then decrement and continue
 - * `V(sem)`: increment semaphore count
 - * How are semaphores different from spinlocks?

Synchronization constructs

- * Condition variables associated with mutexes allow threads to wait for events and to signal when they have occurred
 - * `cv.wait(mutex* m)`: release mutex `m` and block until the condition variable `cv` is signaled. `m` will be held when `wait()` returns
 - * `cv.signal()`: unblock one of the waiting threads. `m` must be held during the call but released sometime afterward
 - * Why is it necessary to associate a mutex with a condition variable?
 - * What happens if `signal()` is invoked before a call to `wait()`?

Monitors

- * Monitors are souped-up condition variables that support `enter()`, `exit()`, `wait()`, and `signal()` routines
- * When one thread enters a monitor, no other thread can enter until the first thread exits
- * The exception is that a thread can wait on a condition after entering a monitor, permitting another thread to enter (which will potentially signal and unblock the first thread)
 - * Hoare monitors: `signal()` causes a waiting thread to run immediately
 - * Mesa monitors: `signal()` returns to the caller and a waiting thread will unblock some time later

Deadlock

- * What is an example of deadlock?
- * Methods for preventing and avoiding deadlock
 - * Have threads block until all required locks are available
 - * Have all threads acquire locks in the same global ordering
 - * Run banker's algorithm to simulate what would happen if this thread and others made maximum requests: no deadlock = continue, deadlock = block and check again later
- * Can resolve deadlock by breaking cycles in the dependency graph: choose a thread, kill it, and release its locks
 - * What are the potential problems related to doing this?

Scheduling

- * Operating systems share CPU time between processes by context-switching between them
 - * In systems that support preemption, each process runs for a certain quantum (time slice) before the OS switches contexts to another process
 - * Which process runs next depends on the scheduling policy
- * Scheduling policies can attempt to maximize CPU utilization or throughput or minimize response time, for example
 - * There are always tradeoffs between performance and fairness

Scheduling laws

- * Utilization law: utilization is constant regardless of scheduling policy as long as the workload can be processed
- * Little's law: the better the average response time, the fewer processes there will be in the scheduling system
- * Kleinrock's conservation law: improving the response time of one class of task by increasing its priority hurts the response time of at least one other class of task

Scheduling policies

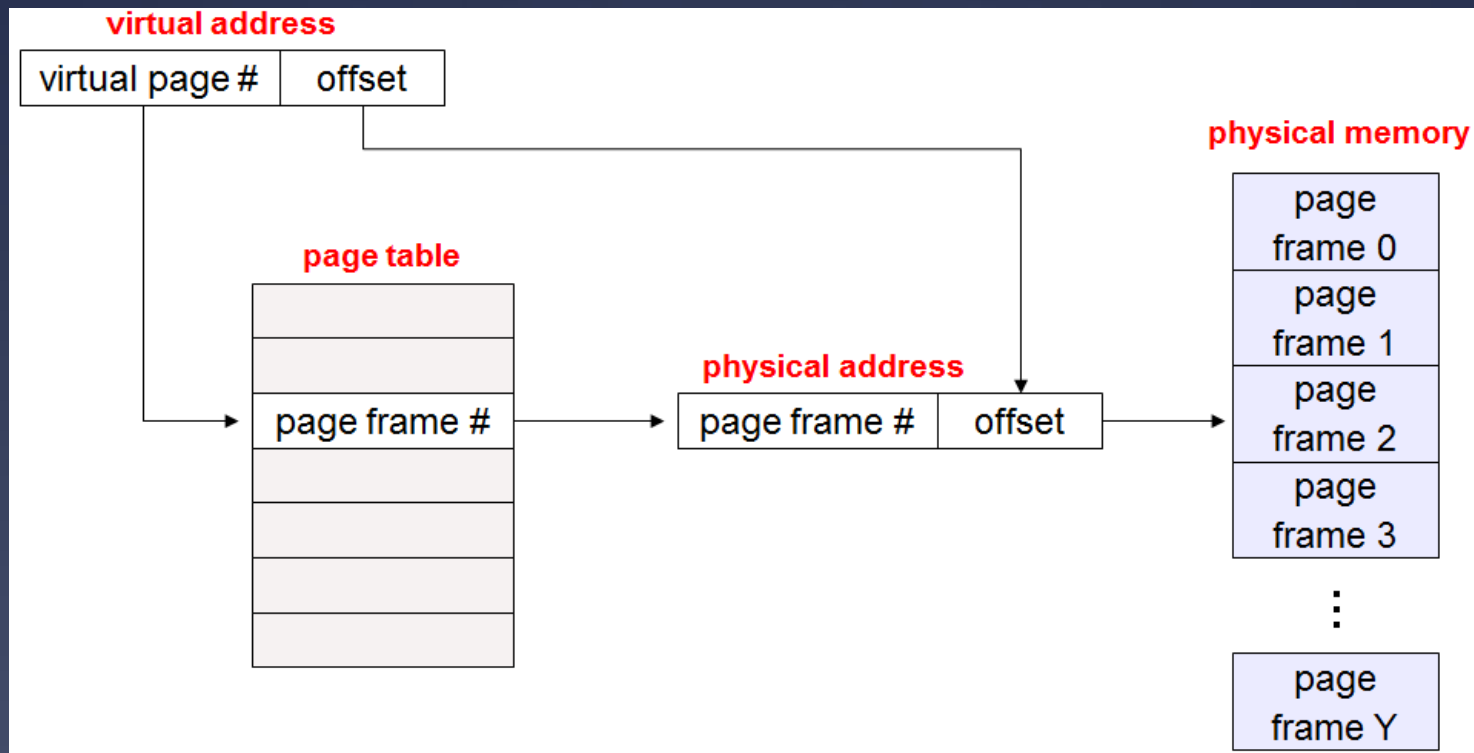
- * FIFO: first in first out
 - * Schedule processes in the order they arrive
- * SPT: shortest processing time first
 - * Schedule process with smallest time requirement
- * RR: round robin
 - * Cycle through processes, executing each for a fixed amount of time
- * Priority
 - * Assign a priority to each process and execute higher-priority processes first
- * What are the benefits and drawbacks of each type of scheduling policy?

Virtual memory

- * Each process has the illusion of use of the complete address space
- * Facilitates isolation and also sharing (how?)
- * The OS manages a mapping of virtual address to physical addresses
- * Address translation
 - * High bits of virtual address map to physical page
 - * Low bits of address determine offset within page

Page tables

- * Per-process constructs that map virtual pages to physical pages



Page table entries

- * Page table entries contain bits that describe whether a page is valid, whether it has been referenced, whether it is dirty, its protection flags (read, write, execute), and the physical page number

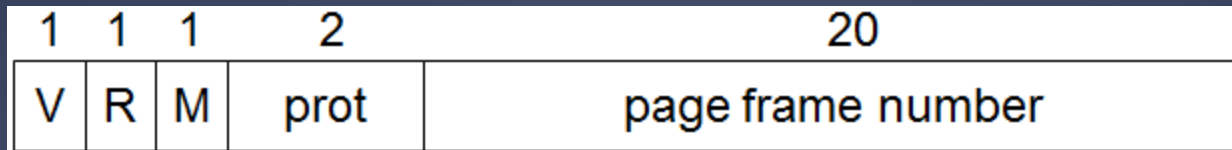


Diagram from Ed Lazowska

Segmentation

- * What are the advantages and disadvantages of using virtual memory?
 - * Pros: isolation, reduced (virtual) memory fragmentation
 - * Cons: additional overhead in address lookups, page tables themselves require memory
- * Segmentation: break up regions of virtual memory by usage
 - * E.g. stack, heap, and text (instructions)
 - * Virtual addresses have the form <segment number, offset> under this scheme
- * Can combine paging and segmentation: keep page table per segment as opposed to per process
 - * Addresses take the form <segment number, virtual page number, offset>

Paged virtual memory

- * Memory is limited, so operating systems will often evict pages to disk to make room for others
 - * Can mark page table entry to indicate that page is actually on disk
 - * If a page that is on disk is referenced, a page fault occurs and the operating system reads the page back into memory

Page replacement algorithms

- * Page replacement algorithms describe which page should be evicted to disk in order to bring another into memory
- * Belady's algorithm: evict the page that won't be used for the longest time in the future
 - * Useful for comparison, but impossible to implement
- * FIFO: evict page brought in the longest time ago
- * LRU: evict the least recently used page

Page replacement algorithms

- * LRU clock: sweep through pages like the hand of a clock
 - * Whenever a page is accessed, set its reference bit
 - * When looking for a page to evict, clear the bit for any page passed
 - * Return the first page seen with an unset bit
- * Working set: keep a certain working set of pages in memory for each process
- * Page fault frequency: attempt to equalize page fault rate between processes