

CSCE 410

history (1-1)

- 1st gen
 - single user writes program that operates entire computer and manages all hardware
 - * TODO? no development except right at the computer?
 - no need for relocatable code
 - * drivers and such were per-machine? TODO
- 2nd gen
 - move programming off-line (no longer programming in production?)
 - automated program loading by special monitor program
 - * batch programming, but only one program kept in memory at a time
 - misbehaving programs could mess with monitor, or other programs though
- 3rd gen
 - scheduling of user programs to allow other stuff to happen while one program is waiting for IO
 - * time sharing
 - created need for time sharing
 - time sharing brought need for and development of
 - * passwords
 - * filesystems
 - * interleaved execution of multiple programs (process scheduling)
 - * remote access (computing as a utility)
 - * virtual memory
- what is an OS
 - OS controls and coordinates physical resources
 - abstracts various hardware details
 - * IO, networking, processes
 - keeps computer efficient
 - * scheduling

architectural support (1-2)

- OSes need hardware to do: asynchronous events, hardware protection (of other processes), address spaces, timers
- asynchronous events
 - (nearly) all asynchronous events are handled using interrupts
 - * IO, user input, timers, etc. . .
 - when an interrupt happens the CPU does:

- * stop current execution
 - * save state (registers and flags and stuff)
 - * changes to supervisor mode (privileged mode)
 - * branches to predefined location (interrupt handler corresponding to interrupt, in interrupt vector table (IVT))
- return from interrupt (**rti**) instruction automatically restores state
- interrupt can be caused by
 - * an asynchronous event (hardware, timer, error)
 - * software (system call)
- hardware protection
 - some instructions are marked as supervisor-mode-only, so regular user programs can't use them
 - * OS then exposes that functionality using system calls
 - memory is segmented (base+limit), so user programs can't access out of their bounds
 - * also virtual memory allows multiple programs to have the same virtual addresses, but be in different physical memory locations
 - hardware-provided timers allow OS to regain control from user programs at a regular interval
 - * needed for reliable scheduling

OS structure (1-3)

- onion view of OS
 - view kernel as layered like an onion
 - innermost layer is lowest level hardware handling code
 - outermost layer is user programs
 - each layer can't be any closer to core than it is, due to dependencies on lower layers
 - from inside to out: hardware, memory, process manger, process coordinator, IPC, RTC, device manager, networking, filesystem, user programs
 - monolithic
- layering, monolithic kernels
 - monolithic kernel has layers that are all in kernel space and all equally trusted and important
 - layers are hierarchical
- microkernel
 - kernel has only very most core parts of OS
 - other stuff (like filesystems, scheduling...) is in user space server processes
 - more overhead because of context switching
 - benefits
 - * server processes could be in different machines, communicating

- over a network (distributed system)
 - * portable: kernel is easier to port because kernel core is smaller
 - * robust: server process can fail without crashing the whole kernel
- exokernel
 - do not abstract/emulate hardware, instead, export hardware securely
 - library operating system
 - * used by apps to abstract the hardware

system calls (1-4)

- skipped

interrupts and exceptions (9, 1-5)

- hardware interrupts
 - usually some kind of I/O: keyboard input, HDD ready, etc...
- there's an interrupt for periodic hardware timer
 - allows kernel to recover control from user program periodically
- exception: interrupt due to an error
 - may have an error code associated with it
- interrupt descriptor table (IDT)
 - 256 elements large
 - first 32 entries are for exceptions
 - hardware interrupts can then map to any other interrupt using PIC
- handling interrupt
 - save state of the calling program
 - handle interrupt
 - restore calling program state, return with `iret`
- typical interrupt handling is to have the IDT just be stubs that just push to stack the type of interrupt and call common handling code
 - in our MPs, that common handling code then calls virtual methods of proper handling class
 - reduces code bloat?
- interrupts are prioritized (somehow)
 - if a higher priority interrupt comes in while handling a lower priority one, the lower priority interrupt is interrupted by the higher priority one
 - for equal priority, it's FIFO
- 8259 programmable interrupt controller (PIC)
 - just need to know that this exists, and is necessary for handling interrupts

allocation (2-1)

- need to allocate space for:
 - new process (`fork()`)
 - new program (`execve()`)
 - process stack grows
 - process expands heap (`malloc()`)
 - process creates (attaches to) shared memory(`shmat()`)
- external vs internal fragmentation
 - external: OS needs to allocate a contiguous block of frames but there is no single contiguous space big enough
 - internal: you need a specific amount of memory, but maybe you have to round up to a multiple of page size. The difference between needed and what it takes up is internal fragmentation?
- naive allocator
 - TODO is a slab allocator a naive allocator?
 - TODO
- buddy allocator
 - maintain free lists in power-of-2 sizes
 - allocate:
 - * round up size to next power of 2
 - * lookup in free list of that size
 - * if available, return
 - * if that free list is empty, split a block from the next size up into 2 blocks and repeat
 - may need to split more than one size larger
 - deallocation
 - * put back into free list
 - * check buddy (flip bit in offset). if buddy is also free, join the blocks
 - may need to join more than one size larger
- different allocators:
 - `malloc()`: virtually contiguous, size in bytes, implemented in user level library
 - `kmalloc()`: physically contiguous, bytes, kernel, slab allocator
 - `vmalloc()`: virtually contiguous, bytes, kernel, slab allocator
 - `alloc_pages()`, `__get_free_pages()`: contiguous frames/pages, kernel, buddy allocator

paging (7, 2-3) (8, 2-4)

- page size:
 - large pages => more fragmentation
 - small pages => more overhead

- paging allows for address spaces
- page table lookup in hardware
 - MMU hardware
 - hardware uses top few bits of address as index into page table
 - then get that entry from the page table, add the lower bits of the virtual address (the offset), and go there in memory
- multi-level
 - split the virtual address into multiple indexes
 - this way you have a hierarchy of page tables, so all page tables below the top level can be lazily allocated
 - * saves memory (reduces overhead)
 - page table level n at index contains physical address of the page table $n + 1$ for use with $n + 1$ th index
- inverted
 - linear array indexed by frame number
 - one table per entire system
 - maps frame number to (PID, page number)
 - saves space because there is always exactly one entry per physical frame
 - very slow because you have to search the entire thing for the PID and virtual address you're looking for
 - also you cannot have shared memory between processes
- hash page tables
 - one per system
 - indexed by (PID, page number)
 - typically collisions are handled with chaining
 - scales well with physical memory
 - does not allow for shared memory?

segmentation (10, 2-6)

- segments are logical for programming (code, data, stack, heap)
 - stuff in segment is semantically related
- segmentation allows for easy data sharing between processes
 - e.g. forked process or shared dynamic library
- can lead to more external fragmentation because segments are larger than pages
- segmented paging:
 - split virtual address into segment number, page number, and offset
 - reduces fragmentation

TLB

- translation look-aside buffer
- caches page table lookups
- you must clear entries when
 - pages are freed/unmapped
 - context switch happens
- in x86, it is managed by hardware
- indexed by page table number and page table index
 - e.g. just 20 bits, not by the full address
- parameters
 - size
 - lookup latency
 - miss penalty (depends on MMU and memory speed)
 - target miss rate (depends on application)
- MIPS software managed TLB
 - TLB is the only cache
 - filling the TLB is done in software
 - TLB miss triggers special TLB refill exception

page replacement policies (15, 2-10)

- cost of page fault, or effective memory access time (ema)
 - $ema = (1-p) * ma + p * \text{PageFaultTime}$
 - p = probability of a page fault
 - ma = memory access time
- locality of reference
 - if a program references location n , it is likely to reference n and locations near n in the near future
- when you need more memory, how do you decide which pages to evict (and possibly writ to disk)?
- FIFO
 - just evict the page that has been in memory the longest
 - pro: simple
 - con: does not exploit principle of locality of reference (assumes that pages resident in memory longer are more likely to continue to be referenced)
- ideal
 - evict page that will be next used farthest in the future (if at all)
 - pro: proven lowest number of page faults
 - con: impossible to implement in real life because we cannot see the future
- least recently used (LRU)

- evict the page that has not been accessed in the longest period of time
- pro: good performance
- con: difficult to implement (must keep track of all references)
- must keep chronological history of page references
 - * software: use a stack
 - * hardware: charge a capacitor and let it slowly drain
- 2nd chance
 - approximation to LRU
 - have a use bit for each page, and keep a pointer to the next candidate victim page
 - * use bit is set every time frame is referenced
 - * when a frame is newly loaded, use bit starts at 1
 - when reclaiming memory:
 - * if pointed-to page has use bit 0: use that frame as victim, and increment pointer to next frame
 - * if pointed-to page has use bit 1: set that use bit to 0, increment pointer, and start over
 - when victim pointer reaches end of memory, wrap it around to the beginning
 - if all frames have use bit 1, you will end up looping through all of them (and setting use bit 0 each time), and finally selecting the one that was pointed to at first
- 2nd chance enhanced
 - also track dirty bit
 - * dirty bit is only set when frame is written to (and thus the frame is dirty)
 - also keep track of dirty frames separately, because we sometimes clear dirty bit in algorithm
 - choice at each step (bits are u,d)

use, dirty	next
1,1	0,1
1,0	0,0
0,1	0,0*
0,0	select as victim

- *
 - * when going from 0,1 \Rightarrow 0,0* , you must list that page in the external dirty frames list, because it's still dirty, but it is now not indicated in bits
- the choice steps (above) mean that a dirty page will not be selected as victim until passed over twice
 - * this is desirable because dirty frames incur a higher eviction penalty (they must be written to disk)

working set (2-11)

- AKA resident set? TODO I think it's a little different
- ideal resident set size changes dynamically as program runs
 - but we assume working set is constant
- all pages referenced within a specified time delta
 - because the process doesn't need all of its pages at once
- rule
 1. at each reference, working set is determined, and only pages in virtual set are kept in memory
 2. a program can only run if its entire current resident set is in memory
- note: if all you reference is one page, working set eventually becomes only that page
- removing frames
 - remove any frames last referenced longer than (time delta) time ago
 - victims are not overwritten immediately, instead are put into one of two lists:
 - free frame list
 - * for clean frames (non-dirty) (frames that are ok to overwrite, as they are in sync with the swap)
 - * OS can freely pick frames from here and use them
 - modified frame list
 - * dirty frames
 - * periodically write these frames to disk, and then move them to the free frame list
- when a frame not in the current working set is referenced, first check the free frame list and modified frame list
 - if the frame exists in either of those, you can simply reclaim it
- victims:
 - when looking for a victim, first get from free frame list
 - * if free frame list is non-empty, just pick one and use it
 - if free frame list is empty
 - * pick from modified list, **but write it to disk first**
- case study: solaris page buffering
 - TODO do we need to know this?
- demand paging: TODO is this the same as working set?
- demand paging on less sophisticated hardware
 - for when you don't have a hardware-managed valid bit?

recursive paging (17, 2-12)

- in x86
- allows you to modify page directory and page table entries without mapping them into virtual memory

- set the last entry of the page directory to the physical address of the page directory
- access **n**th entry of page directory: use address | 1023 | 1023 | **n** | 00 |
 - 1023 and **n** are both 10 bits
- access **a**th entry of **b**th page table: use address | 1023 | **b** | **a** | 00 |
 - 1023, **a**, and **b** are all 10 bits