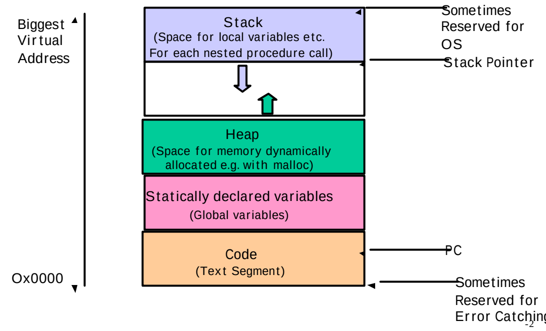
13: Memory Management

Last Modified:
6/21/2004 9:51:29 AM

-1

Recall: Address Space Map



-2

Processes Address Space

- Logically all of this address space should be resident in physical memory when the process is running
- How many machines do you use that have 2^{32}= 4 GB of DRAM? Let alone 4 GB for *each* process!!

-3

Let's be reasonable

- Does each process really need all of this space in memory at all times?
  - First has it even used it all? lots of room in the middle between the heap growing up and the stack growing down
  - Second even it has actively used a chunk of the address space is it using it actively right now
    - May be lots of code that is rarely used (initialization code used only at beginning, error handling code, etc.)
    - Allocate space on heap then deallocate
    - Stack grows big once but then normally small

-4

Freeing up System Memory

- What do we do with portions of address space never used?
  - Don't allocate them until touched!
- What do we do with rarely used portions of the address space?
  - This isn't so easy
  - Just because a variable rarely used doesn't mean that we don't need to store its value in memory
  - Still it's a shame to take up precious system memory with things we are rarely using! (The FS could sure use that space to do caching remember?)
  - What could we do with it?

-5

Send it to disk

- Why couldn't we send it to disk to get it out of our way?
  - In this case, the disk is not really being used for non-volatile storage but simply as temporary staging area
  - What would it take to restore running processes after a crash? (Maybe restore to a consistent checkpoint in the past?) Would you want that functionality?
- We'd have to remember where we wrote it so that if we need it again we can read it back in

-6

Logistics

- How will we keep track of which regions are paged out and where we put them?
- What will happen when a process tries to access a region that has been paged to disk?
- How will we share DRAM and disk with the FS?
- Will we have a minimum size region that can be sent to disk?
  - Like in FS, a fixed size block or page is useful for reducing fragmentation and for efficient disk access

Virtual Memory

- Virtual Memory = basic OS memory management abstraction/technique
- Processes use virtual addresses
  - Every time a process fetches an instruction or loads a value into a register it refers to virtual memory address
- OS (with help from hardware) translates virtual addresses to physical addresses
  - Translation must be fast!
- OS manages sending some portions of virtual address space to disk when needed
  - Sometime translation will involve stalling to fetch page from disk

Virtual Memory provides…

- Protection/isolation among processes
- Illusion of more available system memory

Virtual Memory: Isolation Among Processes

- Protection (Data Isolation)
  - Processes use virtual memory addresses
  - These must be converted to physical memory addresses in order to access the physical memory in the system
  - Gives protection because processes unable even to address (talk about) another processes address space
- Performance Isolation
  - OS also tries to share limited memory resources fairly among processes
  - Can one process use so much of the memory that other processes forced to page heavily?
  - Can one process use so much of the backing store that other processes get out of memory errors?

Virtual Memory: Illusion of Full Address Space

- We've seen that it makes sense for processes not to have their entire address space resident in memory but rather to move it in and out as needed
  - Programmers used to manage this themselves
- One service of virtual memory is to provide an convenient abstraction for programmers ("Your whole working set is available and if necessary I will bring it to and from disk for you")
- Breaks in this illusion?
  - When you are "paging" heavily you know it!
  - Out of memory errors - what do they mean?

HW Support for Virtual Memory

- Fast translation => hardware support
  - Or OS would have to be involved on every instruction execution
- OS initializes hardware properly on context switch and then hardware supplies translation and protection while

Technique 1: Fixed Partitions

- OS could divide physical memory into fixed sized regions that are available to hold portions of the address spaces of processes
- Each process gets a partition and so the number of partitions => max runnable processes

Translation/Protection With Fixed Sized Partitions

- Hardware support
  - Base register
  - Physical address = Virtual Address + base Register
  - If Physical address > partition size then hardware can generate a "fault"
- During context switch, OS will set base register to the beginning of the new processes partition

Paging to Disk with Fixed Sized Partitions?

- Hardware could have another register that says the base virtual address in the partition
- Then translation/protection would go like this:
  - If virtual address generated by the process is between the base virtual address and base virtual address + length then access is ok and physical address is Virtual Address – Base Virtual Address Register + Base Register
  - Otherwise OS must write out the current contents of the partition and read in the section of the address space being accessed now
  - OS must record location on disk where all non resident regions are written (or record that no space has been allocated on disk or in memory if a region has never been accessed)

Problems With Fixed Sized Partitions

- Must access contiguous portion of address space
  - Using both code and stack could mean a lot of paging!!!
- What is the best fixed size?
  - If try to keep everything a process needs partition might need to be very big (or we would need to change how compiler lays out code)
  - Paging in such a big thing could take a long time (especially if only using a small portion)
  - Also would "best" size vary per process?
    - Some processes might not need all of the "fixed" size while others need more than the "fixed" size
    - Internal fragmentation
    - One fixed sized partition = heavy paging for all processes why?

Technique 2: Variable Sized Partitions

- Very similar to fixed sized partitions
- Add a length register (no longer fixed size for each process) that hardware uses in translation/protection calculations and that OS saves/restores on context switch
- No longer have problem with internal fragmentation

Variable Partitions (con't)

- May have external fragmentation
  - As processes are created and complete, free space in memory is likely to be divided into small pieces
  - Could relocate processes to coalesce the free space?
- How does OS know how big to make each processes partition? Also how does OS decide what is a fair amount to give each process?
- Still have problem of only using only contiguous regions

Paging

- Could solve the external fragmentation problem, minimize the internal fragmentation problem and allow non-contiguous regions of address space to be resident by..
- Breaking both physical and virtual memory up into fixed sized units
  - Smaller than a partition but big enough to make read/write to disk efficient often 4K/8K
  - Often match FS – why?

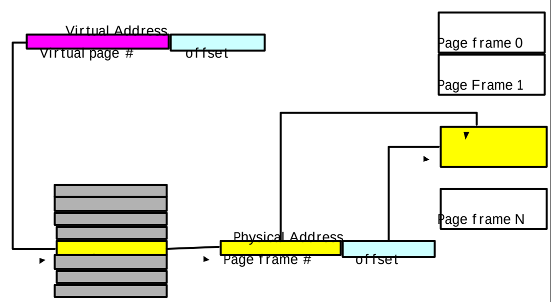
Finding pages?

- Any page of physical memory can hold any page of virtual memory from any process
  - How are we going to keep track of this?
  - How are we going to do translation?
- Need to map virtual memory pages to physical memory pages (or to disk locations or that no space is yet allocated)
- Such maps called Page tables
  - One for each process (virtual address x will map differently to physcial pages for different processes)

Page Table Entries

- Each entry in a page table maps virtual page numbers (VPNs) to physical page frame numbers (PFNs)
  - Virtual addresses have 2 parts: VPN and offset
  - Physical addresses have 2 parts: PFN and offset
  - Offset stays the same is virtual and physical pages are the same size
  - VPN is index into page table; page table entry tells PFN
  - Are VPN and PFN the same size?

Translation

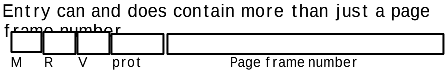


Example

- Assume a 32 bit address space and 4K page size
  - 32 bit address space => virtual addresses have 32 bits and full address space is 4 GB
  - 4K page means offset is 12 bits (2^{12} = 4K)
  - 32-12 = 20 so VPN is 20 bits
  - How many bits in PFN? Often 20 bits as well but wouldn't have to be (enough just to cover physical memory)
- Suppose virtual address 00000000000000011000000000000111 or Ox18007
  - Offset is Ox7, VPN is 0x18
  - Suppose page table says VPN 0x18 translates to PFN 0x148 or 101001000
- So physical address is 00000000000101001000000000000111 or 0x148007

Page Table Entries Revisited

- Entry can and does contain more than just a page frame number



- (M)odify bit – whether or not the page is dirty
- (R)eference bit – whether of not the page page has been read/written
- (V)alid bit – whether or not the page table entry contains valid translation
- (prot)ection bits say which operations are valid on this page (Read/Write/Execute)
- Page frame number

Processes' View of Paging

- Processes view memory as a contiguous address space from bytes 0 through N
  - OS may reserve some of this address space for its own use (map OS into all processes address space is a certain range or declare some addresses invalid)
- In reality, virtual pages are scattered across physical memory frames (and possibly paged out to disk)
  - Mapping is invisible to the program and beyond its control
- Programs cannot reference memory outside its virtual address space because virtual address X will map to different physical addresses for different processes!

-25

Advantages of Paging

- Avoid external fragmentation
  - Any physical page can be used for any virtual page
  - OS maintains list of free physical frames
- Minimize internal fragmentation (pages are much smaller than partitions)
- Easy to send pages to disk
  - Don't need to send a huge region at once
  - Use valid bit to detect reference to paged out regions
- Can have non-contiguous regions of the address space resident in memory

-26

Disadvantage of Paging

- Memory to hold page tables can be large
  - One PTE per virtual page
  - 32 bit address space with 4KB pages and 4 bytes/PTE = 4 MB per page table per process!!!
  - 25 processes = 100 MB of page tables!!!!!
  - Can we reduce this size?
- Memory reference overhead
  - Memory reference means 1 memory access for the page table entry, doing the translation then 1 memory access for the actual memory reference
  - Caching translations?
- Still some internal fragmentation
  - Process may not be using memory in exact multiples of page size
  - Pages big enough to amortize disk latency

-27

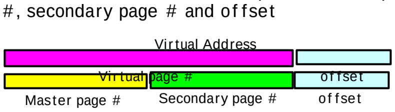
Reduce the Size of the Page Tables?

- Play same trick we did with address space – why have a PTE for virtual pages never touched?
  - Add a level of indirection ☺
  - Two level page tables

-28

Two level Page Table

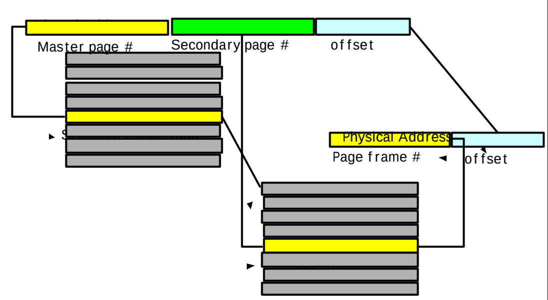
- Add a level of indirection
- Virtual addresses now have 3 parts: master page #, secondary page # and offset



- Make Master page table fit in one page
  - 4K page = 1024 4 byte PTEs
  - So 1024 secondary page tables = 10 bits for master, still 12 for offset so 10 left for secondary
- Invalid MPTE means whole chunk of address space not there

-29

Translation



-30

Page the page tables

- In addition to allowing MPTE's to say invalid could also say this secondary page table is on disk
- Master PTE for each process must stay in memory
  - Or maybe add another level of indirection?
  - Table mapping Master PTEs for each process to DRAM location of disk LBA

-31

Too much of a good thing?

- Each level of indirection adds to access cost
- Original page table scheme doubled the cost of memory access (one for page table entry and one for real memory location)
- Two level page tables triple the cost

- Solve problem with our other favorite CS technique....caching!

-32

TLB

- Add a hardware cache inside the CPU to store recent virtual page to page table entries
  - Fast! One machine cycle for a hit
- OS doesn't even have to get involved when hit in the TLB
- TLB = translation lookaside buffer

-33

TLB

- Usually a fully associative cache
- Cache tags are virtual page numbers
  - FAST! All entries are searched/compared in parallel
  - SMALL! Usually only 16-48 entries (64-192KB)
  - In hardware, SMALL often equals FAST
- Cache values are PTEs
- TLB is managed by the memory management unit or MMU
  - With PTE + offset, MMU can directly calculate the physical address

-34

How effective are TLBs?

- Only 16-48 entries
  - Maps only 64-192 KB of address space
  - If process active using more address space than that will get TLB misses
- Amazingly >99% of address translations are hits!!
  - What does that mean?
- Processes have very high degree of locality to their accesses patterns
  - When map a 4K page likely access one memory location, that prefetches the rest and likely to access them next (if so 1 in 1024 4 byte accesses will be hits)

-35

TLB miss

- On a TLB miss, what happens?
- Hardware loaded TLBs
  - Hardware knows where page tables are in memory (stored in register say)
  - Tables must be in HW defined format so that it can parse them
  - X86 works this way
- Software loaded TLB
  - On TLB miss generate an OS fault and OS must find and load the correct PTE and then restart the access
  - OS can define PTE format as long as loads in format HW wants into the TLB
- Either way have to choose one of current entries to kick out – which one?
  - TLB replacement policy usually simple LRU

-36

Other OS responsibility?

- Even if have HW loaded TLB
- What else must the OS do?
  - Hint: context switch

-37

Context Switch

- Contents of TLB reflect mapping from virtual to physical - that applies to only one process
- On context switch must flush the current TLB entries of things from last process
- Could restore entries for new process (preload) or just set them all to invalid and generate faults for first few accesses
- This is a big reason context switches are expensive!!
  - Recall: kernel level thread switch more expensive the user level switch ..now you know even more why!

-38

Segmentation

- Similar technique to paging except partition address space into variable sized segments rather than into fixed sized pages
  - Recall FS blocks vs extents?
  - Variable size = more external fragmentation
- Segments usually correspond to logical units
  - Code segment, Heap segment, Stack segment etc
- Virtual Address = <segment #, offset>
- HW Support? Often multiple base/limit register pairs, one per segment
  - Stored in segment table
  - Segment # used as index into the table

-39

Segment Lookups

-40

Paging Segments?

- Use segments to manage logical units and then divide each segment into fixed sized pages
  - No external fragmentation
  - Segments are pageable so don't need whole segment in memory at a time
- x86 architecture does this

-41

Linux on x86

- 1 kernel code segment and 1 kernel data segment
- 1 user code segment and 1 user data segment
  - Belongs to process currently running
- N task segments (stores registers on context switch)
- All segments paged with three level page tables

-42

Shared Memory

- Exploit level of indirection between virtual address and physical address to allow processes to communicate through memory Shared memory
- Map the same set of physical page frames into different processes virtual address space (maybe at different virtual addresses)
  - Each process has its own PTEs so can give different processes different types of access (read/write/execute)
  - Execute access to same regions good for shared libraries!

Duplicates of large items?

- Suppose two processes each want a private writeable copy of the same data
- If is is small give them there own physical pages
- They want private writeable copies so can't just use normal shared memory
- If it is big painful to duplicate especially if they are each only going to change a little bit

Copy-on-Write

- Instead of copying, make a shared memory region but mark everyone's permissions as read only (even though they really have permission to write)
- Then if they try to write, HW will generate an access violation fault
  - OS invoked on faults and usually end processes
  - In this case, OS will make a copy of just the page written and then set the PTE to point to the new private copy (with write access this time!) and restart
  - Much like servicing a page fault where have to bring data in from disk
- Copy-on-write often used on fork to share a copy of the parent's address space even though logically parent and child each get their own private writeable copy (esp good because often quickly overwritten)

Memory Mapped Files

- Can access files through the virtual memory system as well as through typical open/read/write FS interface
- Map a file into a region of your address space
  - File start = address X
  - Then read file offset Y = look at data a memory location X+Y
  - Write file offset Y = set memory location X+Y equal to new value
- Doesn't read entire file when mapped
  - Initially pages mapped to file are invalid
  - When access the memory mapped region, translated into FS read/write operations by the OS