Chapter 15: Mechanism: Address Translation

In developing the virtualization of the CPU, we focused on a general mechanism known as limited direct execution (LDE)

- Remembering LDE
    - let the program run directly on the hardware -> context switching from user mode to kernel mode
    - limited direct execution includes hardware to execute trap handlers to switch contexts between user and kernel
    - At boot time, the kernel initializes the trap table and initializes elements such as the CPU and the process list, then returns from trap in order to go back to user mode
    - The OS is not running when the CPU is executing a process!
    - Two approaches
        - Cooperative Approach: The OS simply waits for a system call to be made to switch processes/contexts, or for a process to end or be terminated
        - Uncooperative Approach: The OS takes a more aggressive approach to regaining control over the CPU, setting up timer interrupts so that every time increment, the OS takes control of the CPU. These timer interrupts essentially include trap handlers and system calls so that they can execute these context switches. The hardware that handles the interrupt must have the responsibility of saving and storing the data somewhere, as well as returning the correct information when the CPU regains control
- Efficiency and control together are two of the main goals of modern operating systems
- In virtual memory, we will keep the same goals, as we want to achieve efficiency with the help of certain hardware, and control so that no application is allowed to access any memory that is not their own
- We also want to add more flexibility, meaning that we'd like programs to use their address space however they want
- **Hardware-based address translation:** (address translation) - the hardware transforms each memory access (e.g. an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address where the desired information is actually located. (essentially performs a translation from virtual to physical address)
- The OS must keep track/ manage memory to see which locations are free and which locations are in use

15.2: An Example

- Imagine there is a process whose address space is as indicated below

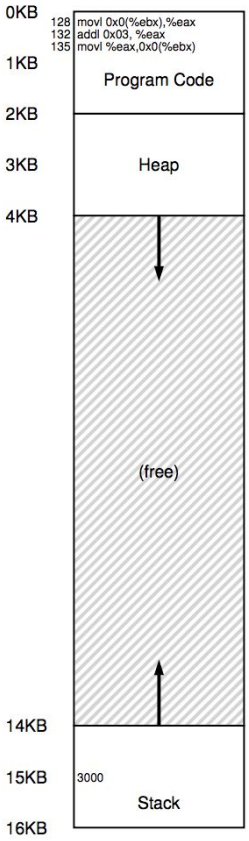


Figure 15.1: **A Process And Its Address Space**

- Tip: Interposition is powerful
    - Interposition is a generic and powerful technique that is often used to great effect in computer systems
    - Achieves transparency; the interposition often is done without changing the client of the interface, thus requiring no changes to said client
- From the program's perspective, its address space starts at address 0 and grows to be a maximum of 16KB; all memory references it generates should be within these bounds
    - However, to virtualize memory, the OS needs to place this process somewhere in the physical address space, but not necessarily at address 0

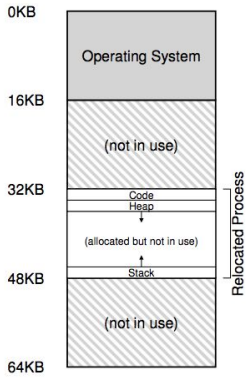


Figure 15.2: **Physical Memory with a Single Relocated Process**

- Above is what physical memory may look like after process 1 has been placed in it
    - not necessarily stored at address space 0

15.3: Dynamic (Hardware-based) Relocation

- Need two hardware registers within each CPU; one is called the **base** register, and the other the **bounds** (sometimes called the limit register)
- This base and bounds pair is going to allow us to place the address space anywhere we'd like in physical memory, and doing so while ensuring that the process can only access its own address space
- Each program is written and compiled as if it is loaded at address zero. However, when a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.
    - Now when any memory reference is generated by the process, it is translated by the processor: physical address = virtual address + base
    - Each memory reference generated by the process is a virtual address; the hardware in turn adds the contents of the base register to this address and the result is a physical address that can be issued to the memory system
- Example: 128: mov1 0x0 (%ebx), %eax
    - The program counter (PC) is set to 128; when the hardware needs to fetch this instruction, if first adds the value to the base register value of 32KB (32768) to get a physical address of 32896.
    - Next the processor begins executing the instruction. At some point, the process then issues the load from virtual address 15KB, which the processor takes and again adds to the base register (32KB) -> 47KB as the final physical address
- **Dynamic relocation:** The relocation of addresses happens at runtime, and can also be moved address spaces even after the process has started running.
- Bounds register helps ensure protection because the processor will check that the memory reference is within bounds to make sure it is legal;
- Bounds and Base registers are hardware stored on the chip and address translation is sometimes called the **Memory Management Unit (MMU)**

15.4: Hardware Support: A Summary

- The OS runs in privileged mode (or kernel mode), where it has access to the entire machine
- Applications run in user mode, where they are limited in what they can do
- A single bit, perhaps stored in some kind of processor status word, indicates which mode the CPU is currently running in
- The hardware should provide instructions to change the values of the base and bound registers through privileged instructions
- The CPU must be able to generate exceptions in situations where a user program tries to access memory illegally

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figure 15.3: **Dynamic Relocation: Hardware Requirements**

-

15.5: Operating System Issues

- The OS must take action when a process is created, finding space for its address space in memory.
- When a new process is created, the OS will have to search a data structure like a free list to find room for the new address space and then mark it used
- The OS must do some work when a process is terminated, such as clean up all memory and put it back on the free list
- The OS must save and restore the base-and-bounds pair when it switches between processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the process structure or process control block (PCB)
- To move a process' address space, the OS first deschedules the process, then the OS copies the address space from the current location to a new location. The OS then updates the saved base register to point to the new location
- The OS must provide exception handlers, or functions to be called when an exception is raised such as accessing memory outside its bounds
- The OS still follows the basic principles of limited direct execution. In most cases the OS just sets up the hardware appropriately and lets the process run directly on the CPU

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Figure 15.4: **Dynamic Relocation: Operating System Responsibilities**

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of... system call handler timer handler illegal mem-access handler illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table allocate memory for process set base/bounds registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs Fetch instruction
	Translate virtual address and perform fetch	
		Execute instruction
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
		...
	Timer interrupt move to **kernel mode** Jump to interrupt handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) (including base/bounds) restore regs(B) from proc-struct(B) (including base/bounds) **return-from-trap** (into B)		
	restore registers of B move to **user mode** jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to **kernel mode** jump to trap handler	
Handle the trap Decide to terminate process B de-allocate B's memory free B's entry in process table		

Figure 15.5: **Limited Direct Execution Protocol (Dynamic Relocation)**

- All of these things must be performed in a way that is transparent to the process so that the process does not know what is going on when this occurs

15.6: Summary
- Advantages and Disadvantages of Base and Bounds
    - Advantages: Very simple hardware implementation and circuity with pretty good efficiency. Base and Bounds also offers protection to ensure that no process can generate memory references outside its own address space
    - Disadvantages: Internal fragmentation: If the stack and heap are not too big, there is space that is unused that is wasted by the process. Restricted to placing an address space in a fixed-sized slot, leading to internal fragmentation

Chapter 16: Segmentation
- So far we have been putting the entire address space of each process in memory.
- Even though the space between the stack and heap is not being used, it is still being stored in memory
- The simple approach to using a base and bounds register can lead to internal fragmentation and inefficient use of memory

16.1: Segmentation: Generalized Base/Bounds
- Idea: Instead of having just one base and bounds pair in our MMU, why not have a base and bounds pair per logical segment of the address space? A segment is just a contiguous portion of the address space of a particular length

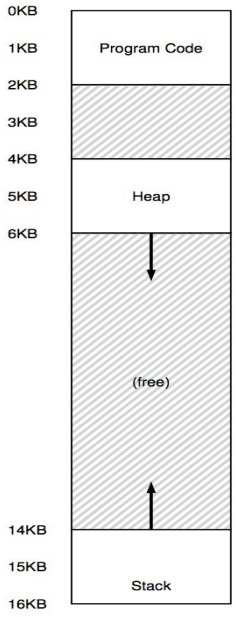


Figure 16.1: **An Address Space (Again)**

- Three logically-different segments: the code, the stack, and the heap (also remember data that stores initialized variables)

- What segmentation allows the OS to do is to place each one of those segments in different parts of physical memory, and thus avoid filling physical memory with unused virtual address space
- Example: With a base and bounds pair per segment, we can place each segment independently in physical memory
- Only used memory is allocated space in physical memory, and thus large address space with large amounts of unused address space can be accommodated
- Segmentation Fault: The term segmentation fault or violation arises from a memory access on a segmented machine to an illegal address.

16.2: Which Segment Are We Referring To?

- The hardware uses segment registers during translation
- Q: How does it know the offset into a segment, and to which segment an address refers to?
    - **Explicit Approach:** Chop up the address space into segments based on the top few bits of the virtual address (used in VAX/VMS system)
        - Set the top few bits in the virtual address to represent which segment you are referring to (e.g. 00 represents the code, 01 the heap)
        - Hardware takes the remaining as the offset into the segment
        - By adding the base register to the offset, the hardware arrives at the final physical address
        - The often eases the bounds as well; we can simply check if the offset is less than the bounds, if not it is illegal
        - Because one bit goes unused if we used two bits, some systems put the stack and heap in the same segment and use only one bit to select a segment
    - **Implicit Approach:** the hardware determines the segment by noticing how the address was formed.
        - If, for example, the address was generated from the program counter (i.e. from an instruction fetch), then the address is within the code segment.
        - If the address is based off of the stack or base pointer, it must be in the stack segment, any other address must be in the heap

16.3: What about the stack?

- The stack grows backwards (ex: starting at 28KB and going to 26KB)
- We need extra hardware to show which way the stack/heap grows (ex: the code and heap grows positive while the stack grows negative) -> keep a bit that keeps track of that

16.4: Support for Sharing

- To save memory, it is useful to share certain memory segments between address spaces. In particular, **code sharing** is common and still used in systems today
- To support sharing, we need more support from the hardware -> protection bits

- **Protection bits:** Adds a few bits per segment indicating whether or not a program can read or write a segment, or perhaps even execute code that lies within a segment
- If you set a code segment to read only (i.e. in shared libraries) you are able to share it across multiple processes without having to worry about isolation

16.5: Fine-grained vs. Coarse-grained Segmentation

- Think of segmentation as **coarse-grained**, as it chops up the address space into relatively large, coarse chunks
    - Some early systems allowed for more flexible address spaces that consisted of smaller **fine-grained** segmentation
- Supporting many segments requires even further hardware support with a **segment table** of some kind stored in memory
- Some early systems expected to chop code and data into separate segments which the OS and hardware would then support

16.6: OS Support

- Segmentation: Pieces of address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved through avoiding internal fragmentation that is caused by a simple base/bounds implementation
    - all the space between the stack and heap that isn't used will not be allocated in physical memory
- Issues: What should the OS do on a context switch?
    - The segment registers must be saved and restored -> OS must make sure that the registers are properly restored before returning
- Issue: Managing free space in physical memory
    - External fragmentation between used and free spaces of physical memory is caused by variable sizes in segments.
- One solution to external fragmentation in physical memory: **Compaction (Defragmentation)**
    - The OS could stop whichever processes are running, copy their data to one contiguous region of memory, change their segment register values to point to new physical locations, and thus have a large free extent of memory to work with
    - Disadvantages: Compaction is expensive and copying segments is very memory-intensive and uses lots of processor time
    - Compaction -> similar to defragmentation -> copies pieces of memory into a contiguous region, then rearranges so that coalescing can occur
- Another solution: Free list Management Algorithms
    - algorithms that we learned in the previous lecture such as best-fit, worst-fit, next-fit, first-fit, buddy allocation, slab allocation, etc
    - No matter how smart the algorithm -> "there ain't no such thing as free lunch" -> external fragmentation occurs

Chapter 18: Paging

- Chopping space into fixed-sized spaces -> **paging**
- Instead of splitting up a process' address space into some number of variable sized segments such as stack, heap, and code, we divide them into fixed-sized units called pages
    - Correspondingly, we view physical memory as an array of fixed-sized slots called page frames
    - We want to virtualize memory so that we can avoid the problems of segmentation

18.1: Simple Example and Overview

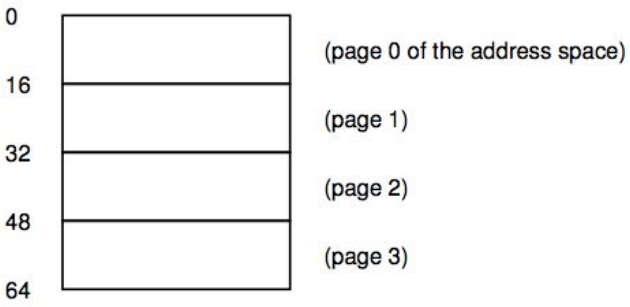


Figure 18.1: **A Simple 64-byte Address Space**

- Using above example, we have 4 16-byte virtual pages
- Physical memory consists of a number of fixed-sized slots (frames) and the OS uses some of the physical memory for itself
- Advantages over previous approaches
    - **Flexibility**: the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space
        - ex: make assumptions about the direction the heap and stack grow
    - **Simplicity of free space management:** The OS can keep a free list of all free pages for this and just grab the first free pages -> increased efficiency
- To record where each virtual page of the address space is placed in physical memory, the OS usually keeps a per-process data structure called a **page table**
    - The page table stores address translations for each of the virtual pages of the address space -> to physical address
- Remember: The page table is a per-process data structure. If another process were to run, the OS would have to manage a different page table for it to map to different parts of physical memory
- Translation from Virtual to Physical (Page Tables)
    - Split the page into two components; the **virtual page number (VPN)** and the **offset** within the page
    - ex: if your virtual address space is 64 bytes, we need 6 bits total for our virtual address (2^6 = 64)
    - In this case, we can say that the top two bits can represent the virtual page numbers and the remaining four can represent the offset

- Example: Assume that a load was to virtual address 21
    - Turn 21 into binary form 010101 and examine that the VPN is 01 and the offset is 0101 -> this means that the address “21” is on the 5th byte of virtual page 1
    - Then, because we know the VPN, we can index our page table and find which physical frame virtual page 1 resides within
    - Ex: for virtual page 1, our **physical page number/physical frame number (PPN or PFN)** is 7 (binary 111)

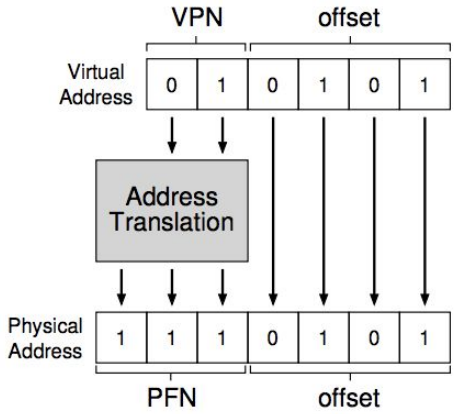


Figure 18.3: **The Address Translation Process**

    -
    - Note that the offset remains the same and it does not get translated like the VPN does
        - this is because offset just tells us which byte within the page we want

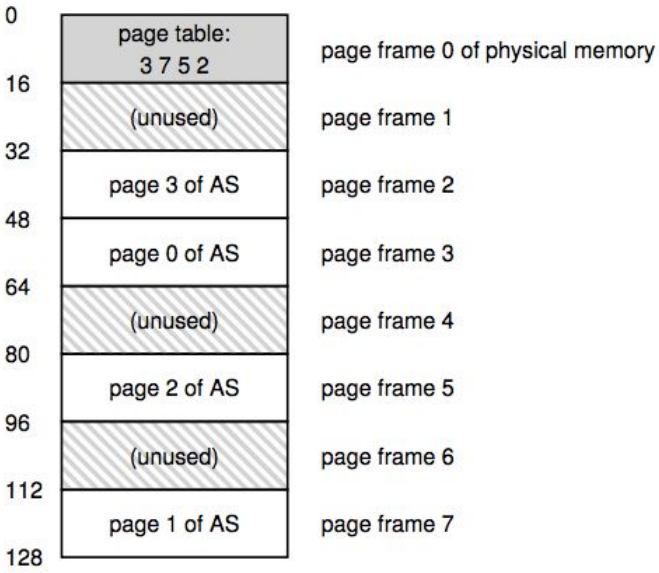


Figure 18.4: **Example: Page Table in Kernel Physical Memory**

18.2: Where are Page Tables Stored?

- ex: 32-bit address space with 4KB pages: 20-bit VPN with 12-bit offset
    - 10 bits will be needed to make 1KB two more bits to get 4KB
    - A 20 bit VPN implies that there are 2^20 to manage for each process
    - Assuming we need 4 bytes per **page table entry (PTE)** to hold the physical translation -> we get 4MB of memory needed per page
- This means that page tables can get extremely big
- Because page tables are so big, we don't keep any special on-chip hardware in the MMU to store the page table of the currently-running process
- OS memory itself can be virtualized and so the page table is located in the virtualized memory of the OS but for now just assume that the page table lives in the physical memory

18.3: What's Actually in the Page Table?

- The page table is essentially just a data structure to map virtual addresses/virtual page numbers to physical addresses/physical frame numbers
    - Simplest data structure is called a **linear page table**, which is a simple array
    - The OS indexes the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at the index in order to find the desired physical frame number (PFN)
- In the PTE
    - **Valid bit:** Indicate whether the particular translation is valid
        - ex: the unused space between the stack and the heap will be marked as invalid and if other processes try to access that memory it will throw a trap
    - **Protection Bits:** Indicates whether the page could be read from, written to, or executed from
    - **Present bit:** Indicates whether the page is in physical memory or on a disk (i.e., it has been swapped out)
    - **Dirty bit:** Indicates whether the page has been modified since it was brought into memory
    - **Reference bit:** used to track whether a page has been accessed and is crucial in determining whether a page is popular and should be kept in memory

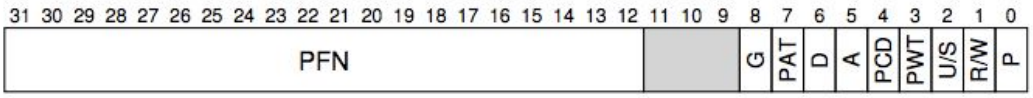


Figure 18.5: **An x86 Page Table Entry (PTE)**

- Above contains a present bit (P), a read/write bit (RW), a user/supervisor bit (U/S), an accessed bit (A) and a dirty bit (D) as well as a few bits (PWT, PCD, PAT, and G) that determines how hardware caching works for these pages

- Things to note about PTE: PTE's store the physical frame number because the virtual address number is usually used as the index to the array. They do not store the size of the pages because the pages need to be a fixed size in order to avoid external fragmentation

18.4: Paging: Also Too Slow

- To fetch desired data, the system must first translate the virtual address into the correct physical address
- To do so, the hardware must know where the page table is for the currently running process
- Assuming that the page table base register contains the physical location of the starting location of the page table, the hardware must first pick out the VPN bits and then to shift to set the offset.
    - ex: for virtual address 21 (010101) -> the shift turns the bits into 01, or virtual page 1 -> use this value as an index to the array of PTEs jointed to the page table base register
    - Once the physical address is known, the hardware can fetch the PTE from memory and get the PFN, and concatenate if from the offset from the virtual address
    - Then the hardware can fetch the desire data from memory and put it into register eax.
    - Because there are extra memory references, paging can be very expensive

Chapter 19: Paging: Faster Translations (TLBs)

- Because the mapping from virtual memory to physical memory is very expensive, paging can be very slow
- **Translation Lookaside Buffer (TLB):** A TLB is part of the MUMU and is simply a hardware cache of popular virtual-to-physical address translations (a better name would be an address-translation cache)
- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein. If so, the translation is performed (quickly) without having to consult the page table

19.1: TLB Basic Algorithm

- Assume a linear page table (the page table is an array) and a hardware-managed TLB

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)    // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                    // TLB Miss
11      PTEAddr = PTBR + (VPN * sizeof(PTE))
12      PTE = AccessMemory(PTEAddr)
13      if (PTE.Valid == False)
14          RaiseException(SEGMENTATION_FAULT)
15      else if (CanAccess(PTE.ProtectBits) == False)
16          RaiseException(PROTECTION_FAULT)
17      else
18          TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19          RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

-
- Algorithm that the hardware follows
    - 1) Extract the virtual page number (VPN) from the virtual address (line 1) and check if the TLB holds the translation for this VPN
        - if it does, we have a **TLB Hit**
        - extract the PFN from the relevant TLB entry and concatenate the offset from the virtual address to form the physical address and access memory
    - 2) If the CPU does not achieve a TLB Hit, (TLB miss)
        - the hardware accesses the page table to find the translation (lines 11 and 12)
        - If the virtual address is valid and accessible, it updates the TLB with the translation from the table
        - then, the hardware retries the instruction and looks for it in the TLB
- The TLB, like all caches, is built on the premise that in the common case, translations are found in the cache. If so, little overhead is added
    - However, TLB misses lead to more memory access to the page table and therefore becomes slow and expensive

19.2: Example: Accessing an Array

- TLB improves performance due to **spatial locality**: the elements of the array are packed tightly into pages and thus only the first access of an element of a page yields a TLB miss
- **Spatial locality:** the likelihood of using addresses that are near other recently-referenced addresses
- The bigger the page size, the less misses and the better hit rate you have because of temporal and spatial locality
- Tip: Using caching when possible
    - **temporal locality:** the idea is that an instruction or data item that has been recently accessed will likely be re-accessed in the future

        - ex: loop variables or instructions in a loop are accessed repeatedly
    - **spatial locality:** the idea is that if a program accesses memory at address x, it will likely soon access memory near x
- Hardware caches take advantages of locality by keeping copies of memory in small, fast on-chip memory.

19.3: Who Handles the TLB Miss?

- Either the hardware or the software
- Hardware
    - In the olden days, the hardware had complex instruction sets such as CISC and the people who built the hardware didn't much trust OS people
    - To do this, the hardware must know where the page tables are located in memory via the page-table base register, as well as the exact format
    - If a TLB miss occurs, the hardware walks through the page table and finds the correct page table entry and extracts the desired translation then updates the TLB
    - The Intel x86 architecture uses a fixed multi-level page table
- Software
    - More modern architectures (MIPS R10K or Sun's SPARC both RISC (reduced-instruction set computers) have what is known as a **software-managed TLB**
    - The hardware simply raises an exception, which pauses the current instruction stream, and jumps to a trap handler
    - The trap handler is code within the OS that is written with the purpose of handling TLB misses
    - When run, the code will lookup the translation in the page table and use privileged instructions to update the TLB
    - Returns from the trap handler then tries again, and will result in a TLB hit
- When returning from a TLB-miss handling trap, the hardware must resume execution at the instruction that caused the trap, so that it can retry the execution
- Thus, the hardware may have to save a different program counter when trapping into the OS, in order to resume properly
- Must be sure to not cause an infinite chain of TLB misses
    - Solutions: Keep TLB Miss handlers in physical memory, where they are unmapped and not subject to address translation
    - Reserve some entries in the TLB for permanently-valid translations and use some of those translations -> **wire translations** always hit TLB
- Advantage of Software-managed TLB
    - flexibility: the OS can use any data structure it wants to implement the page table without changing the hardware
    - Simplicity: the hardware doesn't have to do much on a miss, as all it does is handle the miss and then the OS does the TLB miss handling

19.4: TLB Contents: What's in there?

- A typical TLB might have 32, 64, or 128 entries and be **fully associative.** Any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation
    - TLB entry may look like this: VPN | PFN | other bits
- The hardware searches the entries in parallel to see if there is a match
- Aside: TLB valid bit ≠ Page table valid bit
    - If a page table entry is marked invalid, it means that the page has not been allocated by the process, and should not be accessed by a currently working program
        - usual response is to trap the program and respond by killing the process
    - TLB valid bit refers to whether a valid translation exists
- Other bits
    - The TLB commonly has a **valid bit,** which says whether the entry has a valid translation or not
    - **Protection bits**: determine how a page can be accessed (as in the page table) -> can be marked as read and execute or read and write etc.
    - **Address-space identifier, dirty bit**

19.5: TLB Issue: Context Switches

- The TLB only contains virtual-to-physical translations that are only valid for the currently running processes so these translations are not meaningful for other processes
- The hardware or OS must be careful to not accidentally use translations from some previously run process
- The hardware or OS must be able to distinguish translations for each process that is run
- Solutions
    - **Flush**: Flush the TLB on context switches, thus emptying it before you run the next process
        - On a software based system, this can be accomplished with an explicit and privileged hardware instruction
        - On a hardware managed TLB, the flush could be enacted when the page-table register is changed
        - Sets all valid bits to 0, clearing the contents of the TLB
        - Disadvantages: Frequent TLB misses may occur on process switches
    - **Address Space Identifier (ASID)**: a field in the TLB that is kind of like a process identifier that is needed to differentiate otherwise identical translations

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

        -

        - Of course, the hardware must know which processes are running in order to perform translations and so the OS, on a context switch, must set some privileged register to the ASID of the current process
- Two processes can share a page that is mapped to the same physical frame number (PFN)
    - This is done in sharing code pages such as binaries or shared libraries
    - reduces the overhead of memory

19.6: Issue: Replacement Policy

- When we install a new entry in the TLB, we have to replace an old one: which one do we replace?
- **Least Recently Used (LRU):** LRU tries to take advantage of locality in the memory reference stream, assuming it is likely that an entry that has not been recently been used is a good candidate for eviction
- **Random policy:** Evicts a TLB mapping at random

19.7: A Real TLB Entry

Figure 19.4: **A MIPS TLB Entry**

- MIPS TLB is used in the MIPS R4000
- Supports 32-bit address space with 4KB pages. Thus, we would expect 20-bit VPN and 12-bit offset (Since 4k = 2^12) -> 12 bits of offset
- However, only 19 bits of VPN is actually used the user address will only come from half the address space because the rest is reserved for the kernel
- The VPN translates up to a 24-bit physical frame number and hence can support systems with up to 64GB of physical memory (2^24 pages)
- Global bit: Used for pages that are globally shared among processes -> when set the ASID can be ignored -> ASID used to distinguish between address bits
- If there are more than 2^8 processes running at a time, the coherence bits determine how a page is cached by the hardware
- **dirty bit**: marked when the page has been written to
- **valid bit:** tells the hardware if there is a valid translation present in the entry
- page mask field: supports multiple page sizes
- **Wired register:** can be set by the OS to tell the hardware how many slots of the TLB to reserve for the OS
- MIPS TLB is software managed, so there needs to be instructions to update the TLB
- If the number of pages that a program accesses exceeds the number of pages fit into that TLB, the program will generate a large number of TLB misses

Chapter 21: Beyond Physical Memory: Mechanisms

- So far we've assumed that an address space is unrealistically small fits into physical memory
- Additional level in the memory hierarchy: We have assumed that all pages reside in physical memory
    - OS will need a place to stash away portions of address spaces that currently aren't in great demand
    - In modern systems, this problem is solved by a hard disk drive. Thus to our memory hierarchy, big and slow hard drives reside at the bottom, with memory on top of it

21.1: Swap Space

- Reserve some space **on the disk** for moving pages back and forth (**swap space)**
    - because we swap pages out of memory and swap them back in
    - the OS needs to remember the disk address os a given page
- Swap space is not the only on-disk location for swapping traffic

21.2: The Present Bit

- Assume a system with a hardware managed TLB (hardware scans the page table and picks out the physical address associated with the virtual address, then updates the TLB)
- Memory reference: The running process generates virtual memory references for instruction fetches or data accesses, then the hardware translates them into virtual addresses before fetching desired data from memory
- Hardware first extracts the VPN from the virtual address -> checks the TLB for a TLB hit, produces the resulting physical address and fetches it from memory
    - TLB miss -> locates the page table in memory using the base register and looks up the corresponding PTE with the index as the virtual address -> if it finds it then the hardware puts the physical address in the TLB and then retry again so that it gets a TLB hit
- If the hardware looks in the TLB, it may find that the page is not present in physical memory.
    - The hardware determines this through using a present bit in the TLB
    - If the present bit is set to 1, then everything proceeds accordingly
    - If the page is not in memory but rather on the disk somewhere -> accessing a page that is not in physical memory is commonly referred to as a **page fault**
- Upon a page fault, the OS is invoked to service the page fault through a page-fault handler

21.3: The Page Fault

- Virtually all systems have their page-fault handlers in the software
- If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault.

- Q: How will the OS know where to find the desired page?
    - A: In many systems, the page table is a natural place to store information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address
    - The OS looks in the PTE to find the address, and issues the request to disk to fetch the page into memory
- When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the PTE to record the in-memory location of the new page, and retry the instruction
- This next attempt may result in a TLB miss but will still be in physical memory somewhere
- While the I/O is in flight, the process will be in the blocked state so the OS is free to run any other ready processes while the page fault is being serviced.
- Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another is yet another way to handle multiprogrammed systems -> effective use

21.4: What if Memory is Full?

- If memory is full, then we need to have to have a page replacement policy to know which pages to kick out when necessary

21.5: Page Fault Control Flow

- “What happens when a program fetches some data from memory?”
    - VPN from virtual address is used to search the TLB for a page. If there is a page hit, then the TLB returns the proper physical address by concatenating the virtual address’ offset with the physical frame number and then access memory
    - if there is a TLB miss, depending on whether your TLB is hardware managed or software managed, there is a different process that occurs. If your TLB is hardware managed, the hardware scans the page table in memory to see if a valid translation exists. If there is, then it loads that data onto the TLB and then retries, resulting in a TLB hit. If the TLB is software-managed, then all the TLB does is issue a trap handler to the OS where the OS looks through the page tables to and uses privileged instructions to set the TLB -> retries -> TLB Hit
- Hardware control flow
    - **If the page is not valid, it raises segmentation fault exception**
    - If the page is both valid and present, then the TLB can scan the the PTE to find a physical frame number and then store in TLB and try again
    - If the page is valid but not present, then it must run the OS-controlled page fault handler
- Software Control Flow (Page-fault handling)
    - The OS must find a physical frame for the soon-to-be-faulted page to reside within; if there is no such page, we’ll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use

    - With the physical frame in hand, the handler then issues the I/O request to read in the pages from swap space. -> the OS updates the page table and retries the instruction. This will then result in a TLB miss and the process is continued
    - You can see that when page-faulting occurs, swapping becomes extremely expensive

21.6: When Replacements Really Occur

- To keep a small amount of memory free, most OS have some kind of **high watermark (HW) and low watermark (LW)** to help decide when to start evicting pages from memory
- When the OS notices that there are fewer than the LW pages available, a background thread that is responsible for freeing memory runs
    - The thread then evicts until there are HW pages available
    - The background thread is called the swap daemon or the page daemon -> then goes to sleep
- Many systems will cluster or group a number of pages and write them out at once to the swap partition, increasing the efficiency of the disk
- Instead of performing a replacement directly, the algorithm would instead simply check if there are any free pages available
    - If not, it would inform the background paging thread that free pages are needed
- These actions all take transparently to the process! We don't want the process to know about this

21.7: Summary

- More complexity in page-table structures, as a present bit must be included to tell us whether the page is present in memory or not

Chapter 22: Beyond Physical Memory: Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages -> deciding which pages to evict is part of the memory replacement policy

22.1: Cache Management

- Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a **cache** for virtual memory pages in the system.
    - Thus our goal in picking a replacement policy for this cache is to minimize the number of **cache misses** -> number of times we have to call pages from the disk
    - Alternatively, we want to maximize cache hits
- Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT)** for a program
    - AMAT = TM + (Pmiss x TD)
        - TM is the cost of accessing memory, TD is the cost of accessing the disk, Pmiss it the probability of not finding the data in the cache (a miss)

        - Pmiss varies from 0.0 to 1.0, and sometimes we refer to a percent miss rate instead of a probability (e.g., a 10% miss rate means Pmiss = 0.1)
        - You must pay the cost of accessing the data in memory + pay the cost of fetching the data from disk
- Example: A machine with an address space: 4KB with 256-byte pages.
    - 4 bit VPN and 8 bit offset
    - Thus a process in this example can access 2^4 or 16 total virtual pages
    - These virtual addresses refer to the first byte of each of the first ten pages of the address space
    - Assume that every page except virtual page 3 is already in memory
    - Thus our, sequence of memory references will encounter the following behavior: hit, hit, hit, miss, hit, hit, hit, hit, hit, hit
    - The hit rate = 90%
    - AMAT = TM+TD*Pmiss
        - Assume TM=100 nanoseconds, and the cost accessing disk (TD) is about 10 milliseconds
        - AMAT: 100ns + 0.1*10ms
        - if our hit rate was 99.9%, the result is quite different as the AMAT is roughly 100 times faster

22.2: The Optimal Replacement Policy

- Belady's Optimal Policy: Leads to the fewest amount of number of misses overall
    - Replaces the page that will be accessed **furthest in the future**
    - By doing this you are essentially assigning priorities to pages to be accessed
- A miss at the beginning of the call is called a cold-start miss
- ex: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1
    - 0, 1, 2 are cold-start misses: 0, 1, the next time around are cache hits, and then when we reach 3, we realize that the cache is full and a replacement is necessary
    - We realize that 2 is accessed furthest in the future, thus we evict page 2 and have 0, 1, and 3 in cache
- Aside: Types of Cache Misses
    - Character misses by type: compulsory, capacity, and conflict misses
    - A compulsory miss -> cold start miss
    - capacity miss-> occurs because the cache ran out of space and had to evict an item to bring a new item into the cache
    - conflict miss -> arises in hardware because of limits on where an item can be placed in a hardware cache -> set associativity. Does not arise in the OS page cache because these caches are always fully associative
    - Very difficult to implement the optimal policy because it is difficult to predict the future

22.3: A Simple Policy: FIFO

- Some systems use FIFO replacement, where pages were simply placed in a queue when they enter the system; when a replacement occurs, the page on the tail of the queue (first-in) is evicted
- Advantages
    - simple to implement
- Disadvantages
    - high miss rate, doesn't know the importance of each block
- In general, you would expect the cache hit rate to increase (get better) when the cache gets larger. However, with FIFO, it gets worse! -> **Belady's Anomaly**
- LRU don't suffer from this problem. LRU has what is known as a stack property
    - For algorithms with this property, a cache size of N+1 naturally includes the contents of a cache of size N
    - Thus, when increasing the cache size, hit rate will either stay the same or improve. FIFO and Random clearly do not obey the stack property and thus are susceptible to anomalous behavior

22.4: Another Simple Policy: Random

- Under memory pressure, a random page is selected to be replaced. It is simple to implement but it doesn't intelligently think about what blocks to choose
- Random does not obey the stack property and therefore increases hit rate when the cache gets larger
- With time, random eventually becomes as good as optimal, but it really depends on the luck of the draw

22.5: Using History: LRU

- Any simple policy is likely to kick out an important page, one that is about to be referenced again
- As we did with scheduling policy, to improve our guess at the future we once again lean on the past and use history as our guide
    - ex: if a program has accessed a page in the near past, it is likely to access it again in the near future -> locality
- One type of historical information: frequency -> if a page has been accessed many times, it should not be replaced because it potentially holds valuable memory that you need
- A more commonly-used property of a page is its **recency** of access; the more recently a page has been accessed, it may be accessed in the future again
- **Principle of Locality:** programs tend to access certain code sequences and data structures quite frequently; therefore, we can keep a history to figure out which pages are important, and keep those pages in memory when it comes to eviction time
- **Least-Frequently-Used (LFU):** replaces the least-frequently-used page, takes advantage of spatial locality
- **Least-Recently-Used (LRU):** replaces the least-recently-used page

- **Spatial Locality:** if a page P is accessed, it is likely the pages around it will be accessed as well
- **Temporal Locality:** If a page is recently accessed in the past, it is likely to be accessed in the near future
- Caches can take advantage of locality to simplify the process of replacement in caches
- Follows the stack property as with time, and as the cache size increases, the better our algorithm works

22.6: Workload Examples

- Ex: No locality, which means that each reference is to a random page within the set of accessed pages.
- The workload accesses 100 unique pages over time, choosing the next page to refer to at random; overall, 10,000 pages are accessed

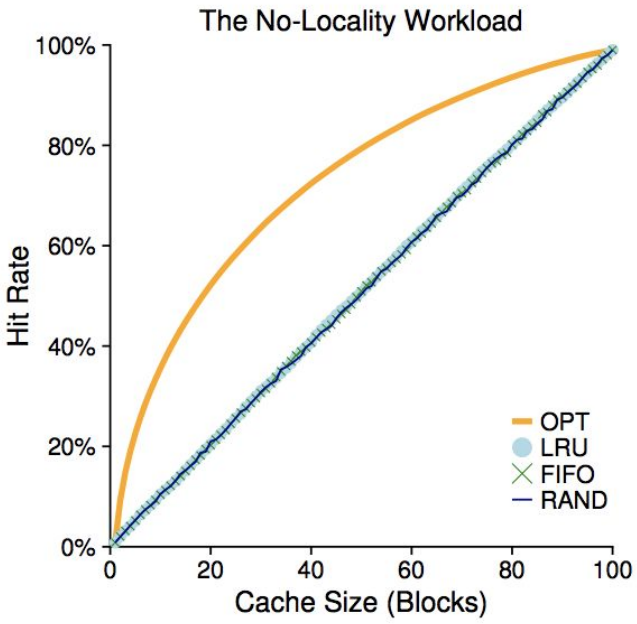


Figure 22.6: **The No-Locality Workload**

- LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache
- When the cache is able to fit the entire workload, all policies will converge to a 100% hit rate
- “80-20”: 80% of the references are made to 20% of the pages (the “hot” pages); the remaining 20% of the references are made to the remaining 80% of the pages (the “cold” pages)
    - Hot pages are referred to most of the time, and the cold pages the remainder

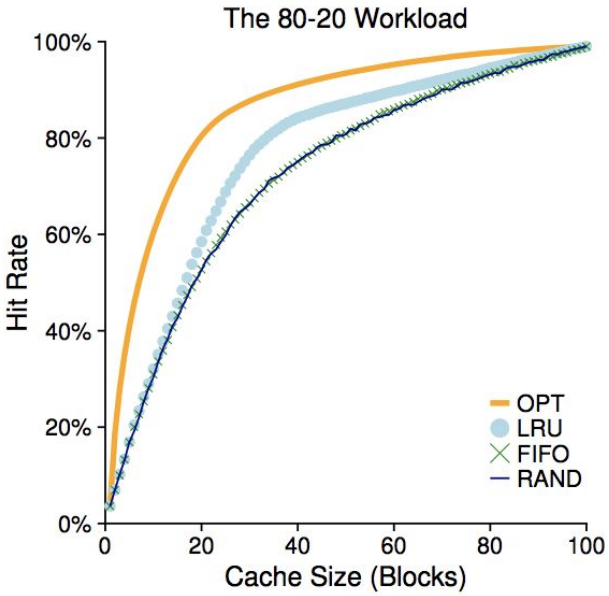


Figure 22.7: **The 80-20 Workload**

- Looping Sequential Workload: Refer to 50 pages in sequence, starting at 0, ... , up to 49, and then we loop, repeating those accesses for a total of 10,000 accesses of 50 unique pages

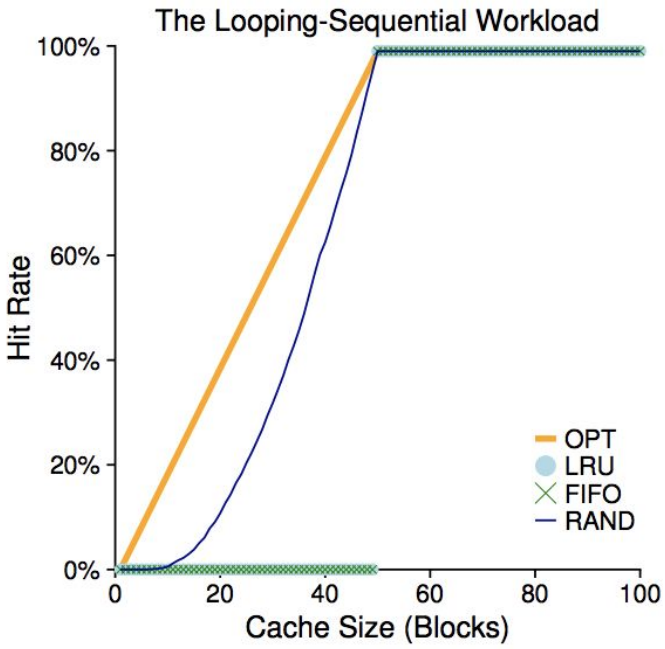


Figure 22.8: **The Looping Workload**

- Worst case scenario for LRU and FIFO because they kick out older pages, but because of their looping nature, meaning the one that was accessed at the end will have to be accessed again

22.7: Implementing Historical Algorithms

- FIFO or Random might throw out important pages that are necessary so the LRU can generally do better than simpler policies
- Implementing LRU: Upon each page access (i.e., each memory access, whether an instruction fetch or a load or store), we must update some data structure to move this page to the front of the list (i.e. the MRU side).
- FIFO list of pages is only accessed when a page is evicted (by removing the first-in page) or when a new page is added to the list.
- Hardware support could greatly help speed the process up
    - Ex: A machine could update, on each page access, a time field in memory
- Thus, when a page is accessed, the time field would be set, by the hardware to the current time. The OS can simply scan all the time fields in the system to find the least-recently-used page
- As the array of pages grows, it becomes difficult and time-consuming to scan through the list of time slots

22.8: Approximating LRU

- Because implementing perfect LRU is very expensive, we must come up with a method to approximate LRU
- **Use bit (reference bit):** There is one use bit per page of the system, and the use bits live in memory somewhere.
    - Whenever a page is referenced, the use bit is set by the hardware to 1. The hardware never clears the bit; only the OS does
- **Clock Algorithm:** Imagine all the pages of the system arranged in a circular list
    - A clock hand points to some particular page to begin with.
    - When a replacement must occur, the OS checks if the currently-pointed to page P has a use bit **(reference bit)** of 1 or 0.
    - If 1, this implies that page P was recently used and thus is not a good candidate for replacement
        - Then, the use bit of P is set to 0 and then the clock hand moves to page (P+1)
        - The algorithm continues until it finds a use bit that is set to 0, implying this page has not recently been used
            - If everything is cleared, that means that all pages have been recently used, clearing all the bits

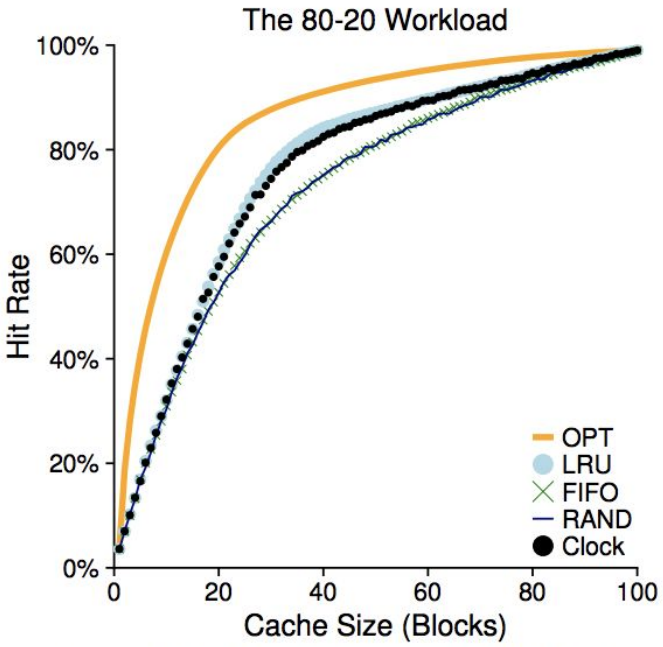


Figure 22.9: **The 80-20 Workload With Clock**

-
- Variant of clock algorithm: Randomly scans pages when doing a replacement, when it encounters a page with a reference bit set to 1, it clears the bit
- It doesn't do quite as well as the perfect LRU, which picks the least recently used page

22.9: Considering Dirty Pages

- Small modification to the clock algorithm: Additional consideration of whether a page has been modified or not while in memory
- If a page has been modified and is thus dirty, it must be written back to disk to evict it, which is expensive
- If it has not been modified, the eviction is free, the physical frame can simply be reused for other purposes without additional I/O. Therefore, some VM systems prefer to evict clean pages over dirty pages
    - dirty pages can be very expensive because if a page has been modified
    - Remember: paging is a memory management scheme by which a computer stores and retrieves data from secondary storage such as a disk
- To support this behavior, the hardware should include a modified bit (dirty bit) where it is set any time a page is written and thus can be incorporated into the page-replacement algorithm
- ex: Clock algorithm could be changed to scan for pages that are both unused and clean to evict first

22.10: Other VM Policies

- Page replacement is not the only policy the VM subsystem employs

- The OS has to decide when to bring a page into memory -> page selection policy
- **Demand paging:** OS brings the page into memory when it is accessed, “on demand” as it were
- **Prefetching:** The OS could guess that a page is about to be used and bring it in ahead of time
    - For example, if a page P is accessed, page P+1 is likely going to be accessed (spatial locality) so the OS could prefetch the page needed from the physical address
- How the OS writes pages out to disk:
    - Of course, they could simply be written out one at a time
    - However, many systems collect a number of pending writes together in memory and write them to disk in one (more efficient) write. -> called clustering or grouping of writes

22.11: Thrashing

- What should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory?
    - The system will constantly be paging, leading to a condition called thrashing
- Earlier OS had a fairly sophisticated set of mechanisms to both detect and cope with thrashing
    - A system could decide not to run a subset of processes, with the hope that the reduced set of processes’ working sets (the pages that are using actively) fit in memory and thus can make progress. **Admission Control**
- Linux runs an out-of-memory killer when memory is oversubscribed; the dameon chooses a memory-intensive process and kills it

Working Sets

- “Annual Income twenty pounds, annual expenditure nineteen, nineteen and six .. result happiness” “Annual income twenty pounds, annual expenditure twenty pounds ought and six … result misery

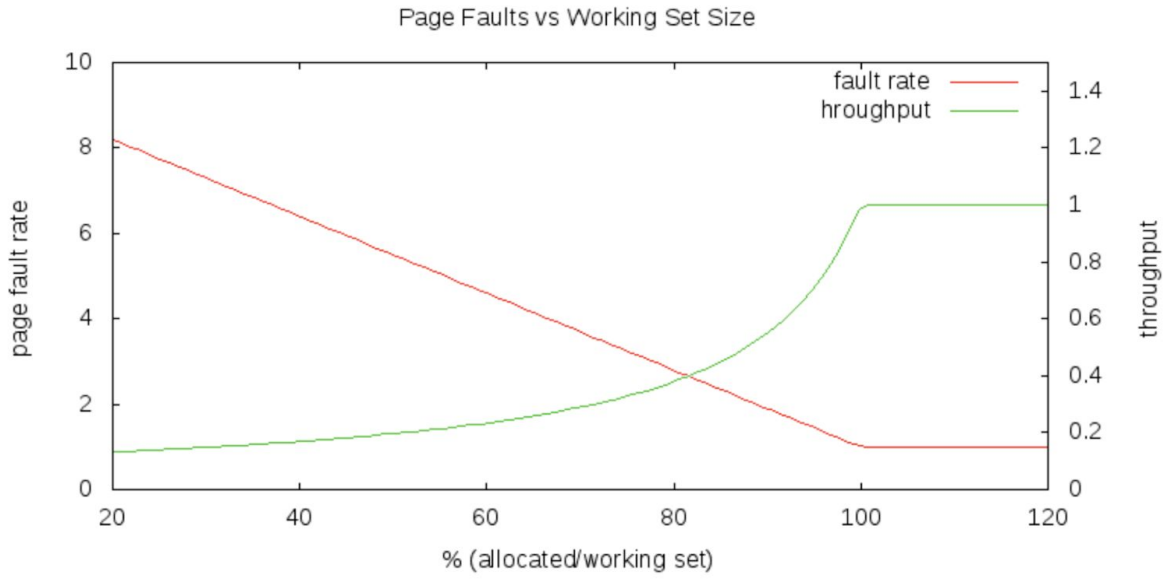
LRU is Not Enough

- The reason why LRU works so well is because it exhibits temporal and spatial locality extremely well
- Spatial Locality: If they access code or data in a particular page, they are likely to reference other code in the same vicinity
- Temporal Locality: If they use some code or data they are likely to come back to the same locations again
- Separate processes seldom access the same pages such as shared text segments (like shared libraries)
- In the absence of temporal and spatial locality, Global LRU will make poor decisions

    - Global LRU and Round-Robin scheduling are both great algorithms but the do not work well as a team
    - Round-Robin destroys temporal and spatial locality because the most recently used pages in memory belong to the process that will not be run for a long time. The least recently used pages in memory belong to the process that is just about to run again.

The Concept of a Working Set

- We want to keep the page-faults down to a manageable rate because page-faults can be slow and expensive to handle
- The improved performance resulting from reducing the number of page faults will more than make up for the delays process experience while being swapped between primary and secondary storage
- There is, for any given computation, at any given time, a number of pages such that
    - If we increase the number of page frames allocated to that process, it makes very little difference in the performance
    - If we reduce the number of page frames allocated to that process, the performance suffers noticeably
- We will call that number the process' working set size (at that particular time)
- Remember throughput is the turnaround time: time of arrival to time of completion of a process



How large is a working set

- Different computations require different amounts of memory
- Requiring more memory is not necessarily a bad thing: it merely reflects the costs of that particular computation

- For scheduling, we saw that different processes had different levels of interactivity and if we could arrange schedules based on this, we should be able to do something similar with working set sizes
- We can infer a process' working set size by observing its behavior
    - If a process is experiencing many page faults, its working set is larger than the memory allocated to it
    - If a process is not experiencing page faults, it may have too much memory allocated to it

Implementing Working Set Replacement

- Regularly scanning all of memory to identify the least recently used page is a very expensive process. With Global LRU, we saw that a clock-like scan was a very inexpensive way to achieve a very similar affect.
- Clock hand was a surrogate for age
    - the most recently examined pages are immediately behind the hand
    - the pages we examined longest ago are immediately in front of the hand
- We can use a similar approach to implement working sets
    - Each page frame is associated with an owning process
    - each process has an accumulated CPU time
    - each page frame will have a last referenced time, value, taken from the accumulated CPU timer of its owning process
    - maintain a target age parameter

```
while ...
{
        // see if this page is old enough to replace
        owningProc = page->owner;
        if (page->referenced) {
                // assume it was just referenced
                page->lastRef = owningProc->accumulatedTime;
                page->referenced = 0;
        } else {
                // has it gone unreferenced long enough?
                age = owningProc->accumulatedTime - page->lastRef;
                if (age > targetAge)
                        return(page);
        }
}
```

-
    - Age decisions are not made on the basis of clock time, but accumulated CPU time in the owning process. Pages only age when their owner runs without referencing them
    - If we find a page that has been referenced since the last scan, we just assumed that it was just referenced
    - If a page is younger than the target age, we do not want to replace it -> recycling young pages may lead to thrashing

    - If a page is older than the target age, we take it away from its current owner and given it to a new process
    - If there are no pages older than the target age, we apparently have too many processes to fit in the available memory
    - If we complete a full scan without finding anything that is older than the target age, we can replace the oldest page in memory -> very expensive complete scan that destroys the purpose of the clock algorithm

Dynamic Equilibrium to the Rescue

- **Page Stealing Algorithm:** A process that needs another page steals it from a process that does not seem to need it as much
    - Every process is contiguously losing pages that it has not recently referenced
    - Every process is continuously stealing pages from other processes
    - Processes that reference more pages more often, will accumulate larger working sets
    - Processes that reference fewer pages less often will find their working sets reduced
    - When programs change their behavior, their allocated working sets adjust promptly and automatically
- Continually opposing processes of stealing and being stolen from will automatically allocate the available memory to the running process in proportion to their working set sizes.
- Manages memory to minimize page faults and avoid thrashing -> does not manage processes to any pre-configured notion of a reasonable working set

Lecture 6

The Need for Relocation

- Memory compaction moves a process
    - from where we originally loaded it to a new place
    - so we can compact the allocated memory
    - coalesce free space to cure external fragmentation
- But a program is full of addresses
- We can’t find/update all of these pointer
    - as we just saw with GC

Why we Swap

- Make best use of a limited amount of memory
    - process can only execute if it is in memory
    - can’t keep all processes in memory all the time
    - if it isn’t READY, it doesn’t need to be in memory
    - swap it out and make room for other processes
- Improve CPU utilization

    - when there are no READY processes, CPU is idle
    - CPU idle time means reduced system throughput
    - more READY processes mean better utilization

Pure Swapping

- Each segment is contiguous
    - in memory, and on secondary storage
    - all in memory, or all on swap device
- *Swap means move everything on disk and place them in memory and vice versa
- Swapping takes a great deal of time
    - transferring entire data (and text) segments
- swapping wastes a great deal of memory
    - processes seldom need the entire segment
- variable length memory/disk allocation
    - complex, expensive, external fragmentation resulting from variable length partitions

The Need for Dynamic Relocation

- there are a few reasons to move a process
    - needs a larger chunk of memory
    - swapped out, swapped back into a new location
    - to compact fragmented free space
- all addresses in the program will be wrong
- it is not feasible to re-linkage edit the program
    - new pointers have been created during run-time

Segment Relocation

- A natural unit of allocation and relocation
    - process address space made up of segments
    - each segment is contiguous w/no holes
- CPU has segment base registers
    - point to (physical memory) base of each segment
    - CPU automatically relocates all references
- OS uses for virtual address translation
    - set base to region where segment is loaded
    - efficient: CPU can relocate every reference
    - transparent: any segment can move anywhere
- If fetching instruction, it's from the code segment, pushing and popping is from the stack, initialized data is from data

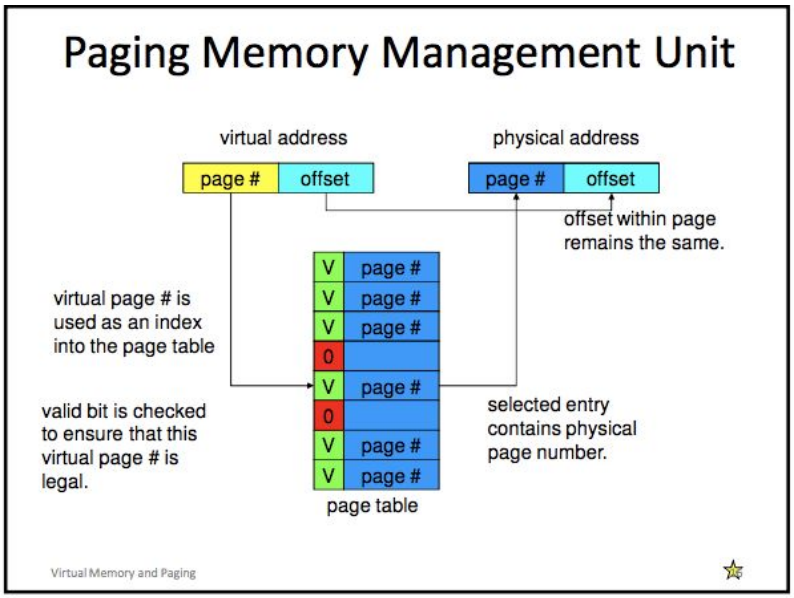
Privacy and Protection

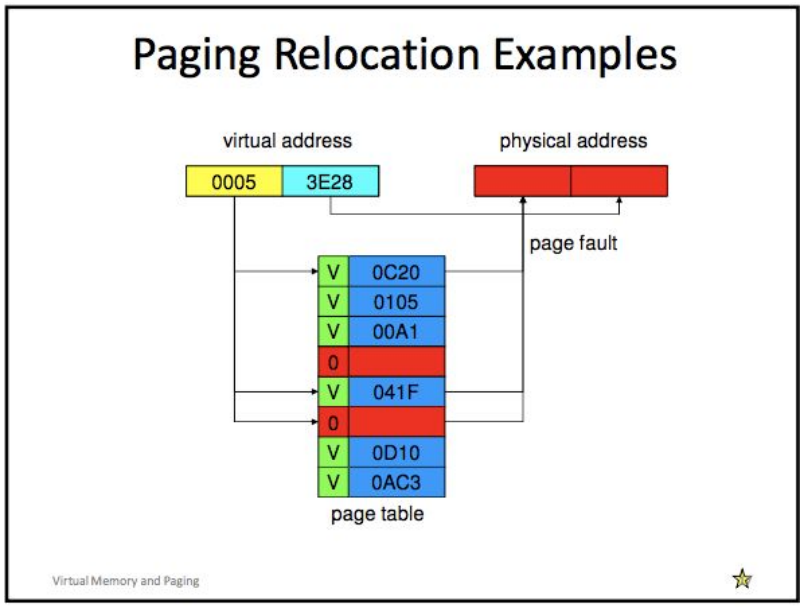
- Confine process to its own address space
    - each segment also has a length/limit register

    - CPU verifies all offsets are within range
    - generates addressing exception if not
- protecting read only segments
    - associate read/write access with each segment
    - CPU ensures integrity of read-only segments
- segmentation register update is privileged

Paging and Fragmentation

- A segment is implemented as a set of virtual pages
- Internal fragmentation
    - averages only ½ page (half of the last one)
- external fragmentation does not exist in paging





Swapping Is Wasteful

- Process does not use all its pages all the time

Loading Pages “On Demand”

- paging MMU supports not present pages
    - CPU access of present pages proceeds normally
- accessing not present page generates a trap
    - operating system can process this page fault
    - recognize that it is a request for another page
    - read that page in and resume process execution

Page Fault Handling Process

- Initialize page table entries to not present
- CPU faults when invalid page is referenced
    - 1) trap forwarded to page fault handler
    - 2) determine which page it resides in
    - 3) Find and allocate a free page frame
    - 4) Block process, schedule I/O to read page in
    - 5) Update page table point at newly read-in page
    - 6) Back up user-mode PC to retry failed instructions
    - 7) Unblock process, return to user mode
- Meanwhile, because the process is blocked, other processes can run -> virtualization

Demand Paging-advantages

- Improved system performance
    - fewer in-memory pages per process
    - more processes in primary memory
        - more parallelism, better throughput
        - better response time for processes already in memory
    - less time required to page processes in and out
    - less disk I/O means reduced queuing delays
- fewer limitations on process size
    - process can be larger than physical memory
    - process can have huge (sparse) virtual space

Are Page Faults a Problem?

- Page faults should not affect correctness
    - after fault is handled, the right page is in RAM
    - process runs again, and now can use that page
- But programs might run very slowly

Minimizing Number of Page Faults

- Two ways
    - keep the “right” pages in memory
    - gives a process more pages of memory
- How do we keep “right” pages in memory?
    - we have no control over what pages we bring in
    - but we can decide which pages to evict
    - this is called “replacement strategy”
- How many pages does a process need?
    - that depends on which process and when
    - this is called the process “working set”

Approximating Optimal Replacement

- Note which pages have recently been used
    - use this data to predict future behavior
- possible replacement algorithms
    - random, FIFO, straw-men … forgot them
- Least Recently Used
    - assert near future will be like recent past
        - programs do exhibit temporal and spatial locality
        - if we haven’t used it recently, we probably won’t use it soon
    - we don’t have to be right 100% of the time
        - the more right we are, the more page faults we save

Why Programs Exhibit Locality

- Code Locality
    - code in the same routine is in same/adjacent page
    - loops iterate over the same code
    - a few routines are called repeatedly
    - intra-module calls are common
- Stack locality
    - activity focuses on this and adjacent call frames
- Data reference locality
    - this is common, but not assured

True LRU is hard to implement

- Too expensive to looks through all of the entries

Practical LRU surrogates

- must be cheap
    - can’t cause additional page faults
    - avoid scanning the whole page table (it is big)
- clock algorithms: a surrogate for LRU
    - organize all pages in a circular list

    - position around the list is a surrogate for age
    - progressive scan whenever we need another page
        - for each page, ask MMU if page has been referenced
        - if so, reset the reference bit in the MMU, skip page
        - if not, consider the page to be the least recently used

Working Sets: Per Process LRU

- Global LRU and Round-Robin contrast each other and eliminates the purpose of LRU
- Global LRU is probably a blunder
    - bad interaction with round-robin scheduling
    - better to give each process it own page pool
    - do LRU replacement within that pool
- Fixed # of pages per process is also bad
    - different processes exhibit different locality
        - which pages are needed changes over time
        - number of pages needed changes over time
    - much like different natural scheduling intervals
- we clearly want dynamic working sets

Implementing Working Sets

- Managed working set size
    - assign page frames to each in-memory process
    - processes page against themselves in working set
    - observe paging behavior (faults/time)
    - adjust number of assigned page frames accordingly
- Paging steal (WS-Clock) algorithms
    - track last use time for each page for owning process
    - find page last recently used (by its owner)
    - processes that need more pages tend to get more
    - processes that don't use their pages tend to lose them

Pre-loading - a page/swap hybrid

- what happens when process swaps in
- pure swapping
    - all pages present before process is run, no page faults
- pure demand paging
    - pages are only brought in when needed
    - fewer pages per process, more processes in memory
- What if we pre-load the working set
    - far fewer pages to be read in than swapping
    - probably the same disk reads as pure demand paging
    - far fewer initial page faults pure demand paging

Clean and Dirty Pages

- consider a page, recently paged in from disk
    - there are two copies, one on disk, one in memory
- If the in-memory copy has not been modified
    - there is still a valid copy on the disk
    - the in memory copy is said to be clean
    - we can replace page without writing it back to disk
- If the in memory copy has been modified
    - the copy on disk is no longer up to date
    - the in-memory copy is said to be “dirty”
    - if we write it out to disk, it becomes “clean again”

Preemptive Page Laundering

- Clean pages can be replaced at any time
    - copy on disk is already up to date
    - clean pages give flexibility to memory scheduler
    - many pages that can, if necessary be replaced
- Ongoing background write-out of dirty pages
    - find and write-out all dirty, non running pages
        - no point in writing out a page that is actively in use
    - on assumption we will eventually have to page out
    - make them clean again, available for replacement
- equivalent of pre-loading

Copy on Write

- fork(2) is a very expensive operation
    - we must copy all private data/stack pages
    - sadly most will be discarded by exec(2)
- assume child will not update most pages
    - share all private pages, mark them copy on write
    - change them to be read-only for parent and child
    - on write-page fault, make a copy of that page
    - on exec, remaining pages become private again
- copy on write is a common optimization

Paging and Segmentation

- pages are a very nice memory allocation unit
    - they eliminate internal and external fragmentation
    - they admit of a very simple and powerful MMU
- they are not a particularly natural unit of data
    - programs are comprised of, and operate on, segments
    - segments are the natural “chunks” of virtual address space
    - each code, data, stack segments contain many pages

- two levels of memory management abstraction
    - a virtual address space is comprised of segments
    - relocation and swapping is done on a page basis
    - segment base addressing, with page based relocation
- user processes see segments, paging is invisible