Memory
SGG: Chapters 8 & 9

I. Overview & Background

A. One of the resources needed by all processes is a memory

B. What users want: infinite space, private, infinitely fast, and cheap

C. Reality: All memory is limited, only some is fast (and expensive), others are slow (but cheap)

1. Creates a **memory hierarchy**
2. Small, fast,expensive (registers, cache, RAM) -> Large, slow, cheap (ssd, disk, tape)
3. In order to support multi-programming we must share and utilize memory hierarchy intelligently

D. Handling of registers, cache is a hardware problem, so we start with RAM

E. A process's code & data must both be in RAM (memory)

1. CPU must fetch both as part of its instruction-execution cycle

F. Memory is an array of "words" (width of memory), each addressable

G. Main memory (RAM), Cache, and Registers are the only memory a CPU may access directly

1. Therefore, any data or instruction must be in these direct access devices in order to operate on them

H. Registers are accessible in one clock cycle; RAM requires a transfer on the data bus (many cycles)

1. This results in a CPU **stall**; stalls can be mitigated by **cache**

I. Memory must also be protected

1. Process isolation; Kernel isolation; separation of data and code
2. Can be enforced with hardware

J. Memory must be **abstracted**
   1. A non-abstraction would give every process direct access to memory; impossible to have co-resident processes; dangerous
   2. We need to provide an **address space**
   3. E.g. Two CPU registers: **base** and **limit** registers; controlled by the OS via privileged instruction
      a) Base holds the smallest address, whereas limit holds the range
      b) These ranges can help enforce legal memory accesses (for user-space processes); kernel has total access
   4. These registers give each process an abstraction of the address space

K. Memory should be **granular**
   1. If we have a memory abstraction, what unit do we access it?
   2. What are the tradeoffs of being able to access every byte vs large chunks

L. Memory should be **respected** as a resource: Moving Programs into Memory
   1. Bindings: when and what do we move into memory?
      a) Compile time:
      b) Load time:
      c) Run time:

M. In order to support features like: run time binding, swapping
   1. We disconnect physical address and logical (virtual) addresses
   2. Memory-management unit (MMU) which maps virtual to physical addresses
      a) Example: A simple MMU uses a **relocation register** (base register) to create a mapping; user programs never see real addresses

b) Relocation and limit registers are stored and restored as part of context switch
3. This allows processes to have a uniform address space (e.g. starting at 0 and going to “max”); or even overlapping logical address spaces

N. **Dynamic loading**: allows us to move only portions of a program into memory
1. Otherwise, we’d be restricted to running processes strictly smaller than our physical memory
2. Routines may not load until they are called
3. Useful for large, rarely used routines (e.g. error handling)
4. Does not require OS support, but OSes often provide library routines

O. **Dynamic Linking**: allows us to link to shared libraries at run time
1. As opposed to static linking (which includes the library as part of the process)
2. Allows multiple processes to share a single library memory image
3. Allows for libraries to be updated without recompiling processes
4. **Stubs** within a process for each library routine
    a) give instructions on which library to load, and if loaded, replaced with the routine’s address
5. Does requires OS support:
    a) E.g. provides isolation among processes that share the library
    b) We’ll see more of this later in shared paging

II. Swapping

A. Swapping is what allows an entire process to be moved from main memory to a backing store (like disk), which is big enough to store all running processes

B. Q: When do we swap?

1. When we want to run more processes than can fit into physical memory

C. Q: When swapping back in, where does the process go?
   1. Depends on the memory binding
   2. If execution time, then it can go anywhere (maximum flexibility)

D. Remember though: disk is very slow, so we must be careful and clever about who, how, and when we swap
   1. The amount of time we spend swapping is directly proportional to the amount of memory we want to swap
      a) Transfer time dominates (not context switching)
   2. The state of a process matters
      a) E.g. if it's blocked on IO, we may 1) further delay the IO by using the disk to swap, 2) a direct IO request may return to the wrong process
   3. Programs are growing larger, so quite expensive to swap
      a) e.g. 1GB program 10sec per swap
   4. In reality: we don't really swap entire processes; too costly and unnecessary

III. Contiguous Memory Allocation

A. A way of allocating memory to user processes such that the addresses are contiguous
   1. **Note**: this is historical, and other, better solutions exist

B. Memory may be partitioned, where each fixed-sized partition contains one process
   1. A fixed number of partitions limits the number of processes
   2. This is dumb and inflexible; although simple

C. **Variable partition** allows partitions to vary in size and number
   1. Can create holes, which leads to a whole set of interesting allocation algorithms (**dynamic storage allocation**)
      a) We'll see it again with file systems
   2. To begin: we can allocate processes from the scheduler until

none fit in any holes

a) How do we choose to allocate space?

3. Once there is no hole big enough to hold next process, then how do we choose?

D. Based on scheduler?

E. Based on fit?

F. Can we rearrange memory to consolidate holes?

G. Some common solutions to dynamic storage allocation

1. **First fit**: Allocate the first hole that is big enough (search 1/2 the space on average)
2. **Best fit**: Allocate the smallest hole (requires searching entire space)
3. **Worst fit**: Allocate the largest hole, (also full search) which may provide more flexibility for the process to grow

H. External Fragmentation

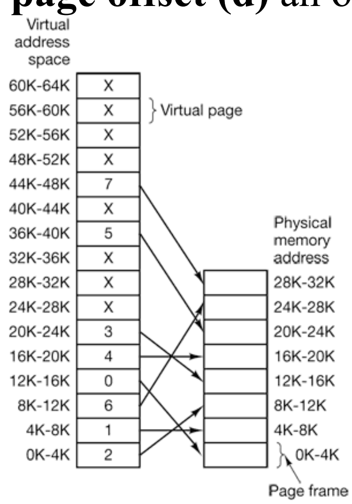
1. Ideally, we want all memory to be allocated, but
2. In reality, we're left with many holes that are not one is big enough to allocate anything into
   a) Although, combined they may be
3. This is known as **external fragmentation**
4. All the allocation strategies above can suffer from external fragmentation
5. Swap suffers the same problem
6. "External" as opposed to "internal," which we'll discuss later
7. **Compaction**: allows for memory to be reallocated to consolidate holes
   a) Requires execution time binding
   b) Suffers overhead (1GB of RAM, 1 word copy in 20ns, is 5 seconds)
   c) Processes can grow, requiring moving
8. External fragmentation is an artifact of contiguous allocation; why be contiguous?
   a) We'll two/three solution: paging, segmentation and a combination of the two

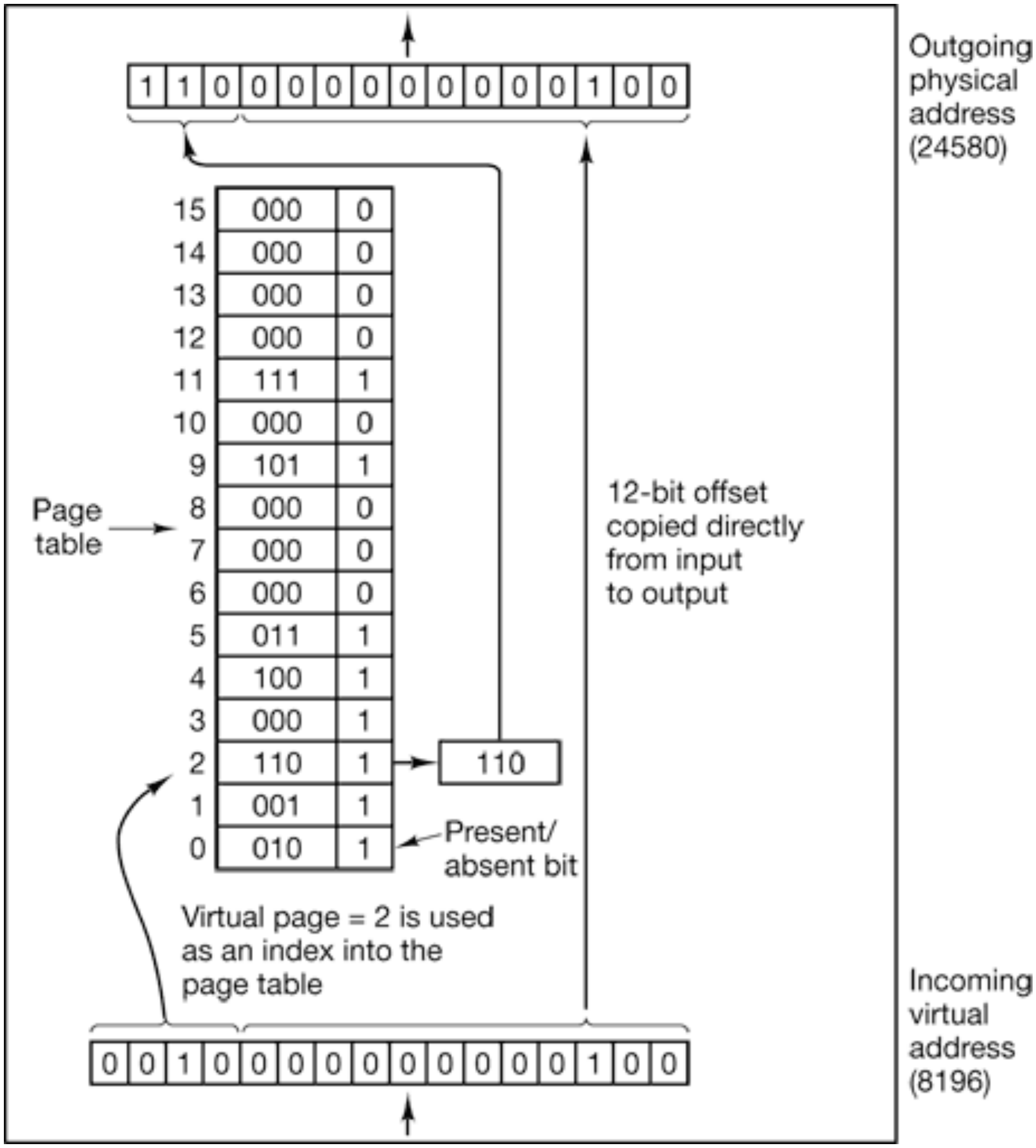
IV. Paging

A. A memory-management scheme that enables noncontiguous allocation of process memory
   1. Part of a larger idea known as **virtual memory** (which we'll see in total, later) [Fotheringham61]
   2. Requires hardware support (which can get complicated)

B. Physical memory is broken into fixed-sized regions: **frames**; logical memory is broken into fixed-sized regions: **pages**; backing store is also addressed as fixed-sized **blocks**;
   1. The paging system handles the mapping of pages to frames
   2. Pages may be logically contiguous, whereas the backing frames and blocks are not
   3. typically |frames| = |pages| = |blocks|; typically between 512-16MB (we'll discuss this later)

C. Page table (simplified!): is the hardware component that enables the translation from logical addresses (**virtual addresses**) to physical frames
   1. Mapping is totally hidden from process/user (no way to map outside of the page table); controlled by OS, who manages the page table
   2. **Page number (p)** being an offset into the page table; and a **page offset (d)** an offset into physical memory





D. Example (above): 2^m is the size of the address space (in bytes), and page size is 2^n bytes, then the higher *m-n* bits designate page number; *n* low-order bits are page offset

1. m = 16; n = 12; 32K of physical memory; 64K virtual address space; page size is 4K; 2^4 = 16 pages;
2. every address gets mapped to some (non-contiguous) frame
3. Using n bits, we can address all 4096 bytes within the page

E. An analogy: each memory frame has its own base/relocation register for each frame

F. No external fragmentation, as any frame may be allocated

G. Page tables allow addressing into greater than 2^n addresses (for n-bit words); another layer of abstraction

H. All this metadata (page table, free frames (**frame table**), etc) must be managed by OS (with hardware support)

   1. What's typical: a page table per process (increases context switching time)
   2. Requires hardware support

I. Page Size

   1. Depends greatly on system usage
   2. Smaller provides granularity, but a larger page table, more overhead
   3. Larger increases throughput and minimizes disk IO
   4. Internal Fragmentation
      a) Managing each and every byte of memory could be expensive
      b) Memory is often broken into blocks
      c) If a process's size isn't an integral number of the block size, it will experience **internal fragmentation**
      d) On average: 1/2 page size per process

J. Structure of Page Table Entries

   1. What constitutes a page frame?
      a) Page frame number (obvi)
      b) Present/absent bit: is the page frame number valid
      c) Protection bits: What kinds of access are permitted (read, write, exec)
      d) Modified/Dirty bit: Has this page been modified, perhaps needing to get written to disk before eviction
      e) Referenced bit: Has the page been referenced; used for eviction
      f) Caching disabled bit: Is cache disabled for this page (useful for direct IO)
   2. What's not in a page frame?

a) We only store things necessary for hardware to make the address translations
b) Backing store addresses: managed by the file system

K. In summary: Paging allows us to
1. load processes at a finer granularity
2. not experience external fragmentation
3. swap at page granularity
4. But:
   a) how do I know which pages are in memory, and which are not?
   b) when I need to swap page, which go and which stay?
   c) how do I know which pages contain modified data, and I need to write to disk?

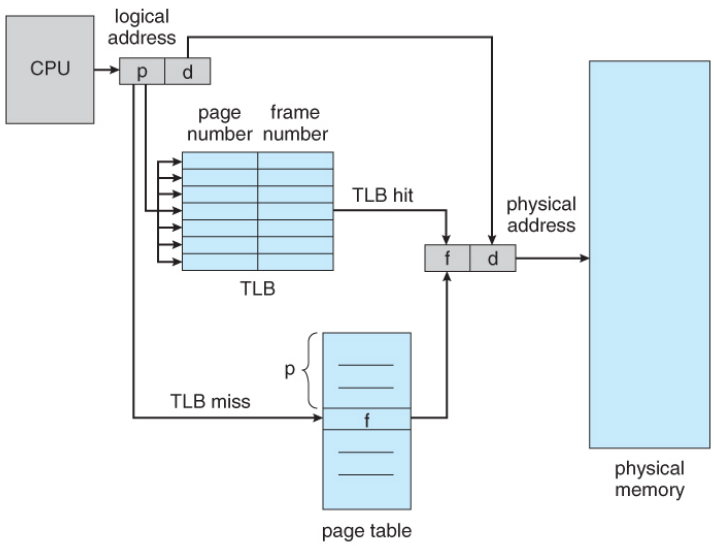
V. Page Table Structures

A. To be efficient: requires hardware support and clever structures.
1. Reason 1 (Speed): Translations must be done on every memory reference
2. Reason 2 (Cost): memory is getting larger: 32-bit words, 4K page: 1 million entries at 4 bytes each = 4MB per process; 64 bit words and 4K page: 2^{52} entires at 8 bytes each = 30M GB (30 PB)

B. Historical solution 1: Page Tables can be built with registers, and specialized translation hardware
1. Registers are reloaded for each process (part of context switching time)
2. not scalable for millions of entries (too expensive)

C. Historical solution 2: Page tables can also be stored in main memory, with a only one register to point to the table **(page-table base pointer)**
1. All process page tables can be co-resident, requiring only a single resister to be updated
2. Requires two memory accesses for each actual memory

access (doubles stall time)
I.e. CPU-bound jobs would take *twice as long* to complete

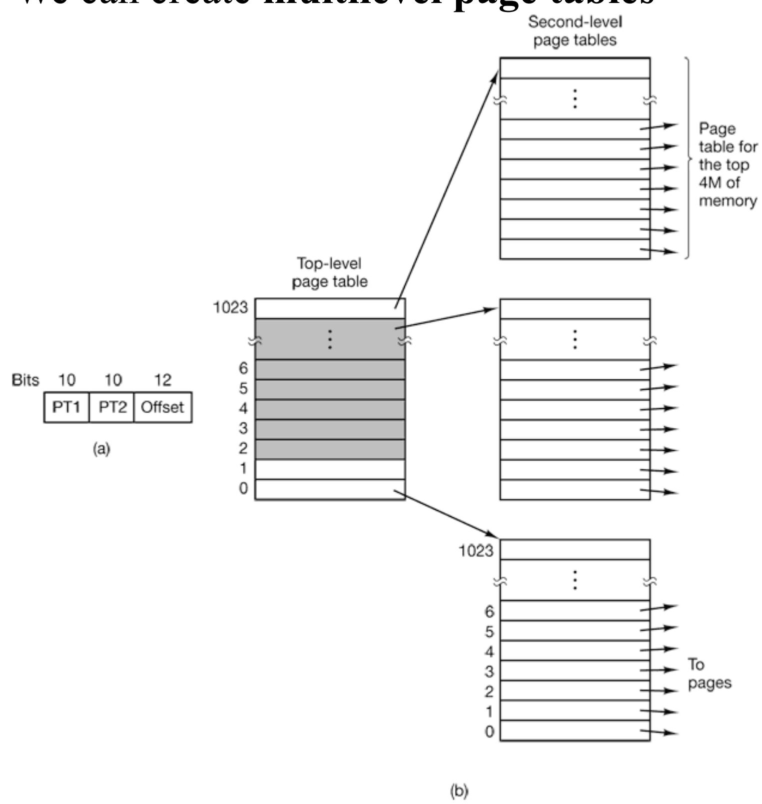


D. **Translation look-aside buffer** (TLB)

1. Sometimes called an **associative memory**,
2. Compromised of key(tag)-value pairs
   a) A cache for the page table
3. All key values can be queried at once; if found, value is returned
4. Expensive, so TLB only contains a small number of entries
5. If it's a **miss** or **page fault**; we must fetch the entry from the page table, perhaps replacing an existing entry
   a) **soft miss**: page is in main memory
   b) **hard miss**: page is on disk
   c) Using which algorithm do we use to replace? We'll see some later.
6. Some entries can be **wired down**, such that can't ever be replaces (like kernel pages)
7. **Address-space identifiers** (ASIDs) will bind TLB entries to a particular process
   a) Preventing a process from access a page that doesn't belong to it
   b) Allows multiple processes to share the same TLB state; otherwise we must **flush** on each context switch

E. Multilevel Page Tables (Hierarchical Paging)

1. What if our virtual address space is quite large?
2. We can create **multilevel page tables**

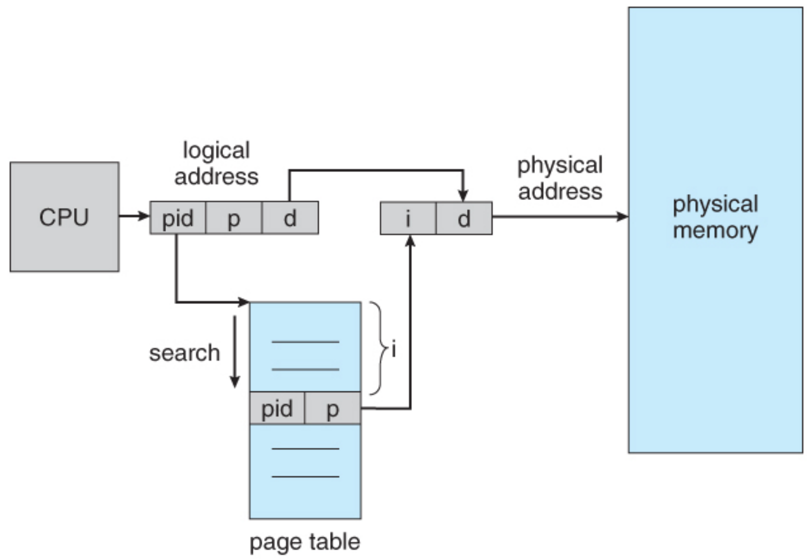


3. Example (above): Two-level page table: Spilt the page table address into two
   a) 10-bit PT1, 10-bit PT2, and 12-bit offset field
   b) Top level page table references 1024 2nd tier page tables
   c) Each 2nd tier page table references 1024, 4K pages
   d) Here 2^{20} pages addressable, with only 4 page tables resident
4. **Idea:** Second-tier page tables need not be resident in main memory
5. Expandable to three or four levels
6. An aside: some file systems take this approach to block allocation

F. **Inverted Page Tables**

1. As mentioned, as word sizes grow from 32 to 64 bits, page table structures can go unwieldy large; even multi-leveled

2. Idea: One entry per frame (of real memory); not one entry per page of virtual address space
   a) The entry stores which process holds that frame, and its virtual address



   b) When a pid/virtual address are found, the offset into the page table is used as the real memory address
   c) Example: 1 GB of physical memory, 4K page, 256K entries
3. Reduces size, but increases lookup complexity
   a) can’t lookup pages by reference any more; must do a full search of the page table (slow)
   b) We can use a **hash table** to index into page table, or
   c) Combine with a TLB to speed up frequent lookups
4. An aside: inverted indexes are what used by Google to make search over a lot of data fast

VI. Virtual Memory

A. Virtual Memory is system built around the core ideas supported by paging

B. Allows the system to present a large, logically contiguous memory to every process, while being non-contiguous in physical memory

C. Many advantages
1. Processes see the same logical, contiguous address space, making programing easier
2. Logical memory may be **sparse**; i.e. a hole in the logical space does not (necessarily) mean a hole in the physical space
3. Pages may be shared among processes (shared libraries, shared memory or fork()'d processes)
4. Not all of a process need be in main memory to execute
5. Logical memory (for a single or many processes) may actually be bigger that physical memory

D. Not all process pages can **resident** in memory at once; further, we many not need all pages in memory at once
1. But, we need a page to be in main memory, to be accessed
2. Q: How do we select pages to be in main memory;
3. Q: How do we select which pages we replace;

VII. Demand Paging

A. Demand Paging is scheme whereby only the pages that are needed are moved into main memory

B. Here we see why the valid bit is useful
1. Valid implies "allocated and in memory"
2. Invalid implies "not allocated or not in memory"

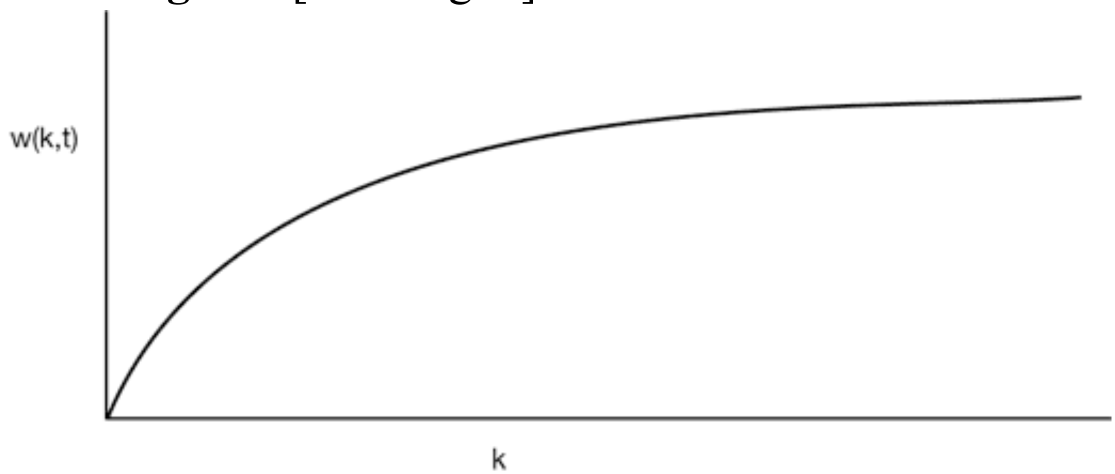
C. If process has a **page miss**, we trap to the OS, which
1. interrupts the process
2. finds a free frame (if one exists)
3. requests the block from the backing store or swap space
4. brings the page into memory
5. updates the page table
6. restarts the processes

D. Non-resident pages may have never been in memory (so in the file system) or were once resident and swapped out (to swap space)

E. **Pure demand paging**: Demanding paging starting from program execution

1. Page faults for the necessary pages to get started (main instructions, stack, globals etc);
2. Most programs settle down, without many more page faults
3. This is because: **locality of reference**: processes only access a small fraction of their pages (of course, systematic pathologies are possible)

F. **Working Set**: [Denning68]



G. Example: k is the most recent memory reference; w(k,t) is the size of the working set at time t

H. The active pages of a running process are called its **working set**;

1. If the physical memory is too small to hold a process's working set, it will cause many page faults, reducing performance
2. When the page fault time exceeds the time spent executing, we are said to be **thrashing**
3. **Working set model:** Keep track of a process's working set
   a) **Prepaging:** Loading working set pages before they are accessed
   b) Avoid the costs associated with loading the initial pages in a working set

I. **Read-modify-write**: Consider the case where we write to a non-resident page; we must page it in, modify it, and write it back out again

VIII. Copy on Write

A. Copy-on-write is an allocation strategy where two initially duplicate objects (processes, files, etc.) can share a single memory image; only when one object writes is a new allocation made

B. Example: A forked process may share the same pages as its parent; only when it modifies a page will a new page be allocated

C. Significantly speeds up fork operation; particularly if they will soon exec

IX. Page Replacement

A. When a running process needs a page, but physical memory is full, we must evict a frame to make room; this is **page replacement**

B. Which frame do we evict?

C. The frame to evict is called the **victim** (morose!)

1. Victims may be overwritten (because they are read only)
2. Or may need to be written to swap (because they have been modified)

D. Page replacement is a costly operation: at least two I/O operations, so its important to minimize

1. If we're not careful in how we select our victims, we may further delay the system

E. Dirty pages may be good victims, because it has to be written anyways

1. However, it forces the write which may have better optimized with other dirty blocks
2. Dirty blocks also are indicative of activity

F. Optimal Page Replacement (OPT)

1. We really want to evict the least used page (now and into the future)
2. Each page could be labeled with the number of instructions that will be executed BEFORE page is referenced
3. Of course, this is impossible (akin to SJF as optimal), but useful as a benchmark

G. First-In, First-Out (FIFO)

1. Simple: first page in (the oldest page), is the first page evicted
2. Indiscriminate of use: might throw out important pages
3. Stands to reason that the first thing in, may actually be very important

3 frames

0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	3	3	3	4	4	4	4	4	4
	1	1	1	0	0	0	0	0	2	2	2
		2	2	2	1	1	1	1	1	3	3

page fault=9 miss ratio=9/12=75% hit ratio = 25%

4 frames

0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	0	0	4	4	4	4	3	3
	1	1	1	1	1	1	0	0	0	0	4
		2	2	2	2	2	2	1	1	1	1
			3	3	3	3	3	3	2	2	2

page fault=10 miss ratio=10/12=83.3% hit ratio = 16.7%

4. Exhibits **Belady's anomaly**
   - H. The number of page faults v. the number of frames is not monotonic
   - I. i.e. Increasing the number of frames does not necessarily decrease the number of page faults

J. Least Recently Used (LRU)

1. An approximation of OPT
2. Idea: Each page has some age with respect to use; replace page that hasn't been used in the longest time
3. Like OPT, but only looking backwards

a) Still the chance that the page you evicted will get accessed immediately
4. Q: How do we implement such a thing?
5. Counters: each page table entry has an associated logical clock
   a) Takes space (per entry, per process)
   b) Take time (must search the list on each eviction)
6. Ordered Queue: keep pages in an ordered queue
   a) Less space than a counter, but requires many memory accesses (pointers) to keep up-to-date
7. Both implementations impractical without specialized hardware
8. Does not exhibit Belady's anomaly
   a) A set of pages in memory with n frames is always a subset of the set of pages in memory with n+1 frames
9. We can approximate LRU with a single bit in the page table entry: **reference bit**
   a) Bit set when page is referenced
   b) Bits set to zero initially and on context switch

K. **Second Chance**

1. Use a FIFO queue and a reference bit
   a) If bit is 0, replace the page
   b) If bit is 1, set to 0 and move to the end of the queue
2. Moving pages around is expensive

L. Clock (Second Chance Variant)

1. Page frames in a circular list, and have a clock "hand" point to the newest page
2. When evicting, look at page pointed to by hand
   a) If reference bit is 0, replace page
   b) If bit is 1, set to 0, and advance hand
3. A nice approximation of LRU

M. Working Set & Working Set Clock (WSClock)

1. Both based on tracking the working set pages
2. Expensive to implement

N. Summary

1. Many different page replacement algorithms (many not covered here), with different performance characteristics and hardware requirements
2. e.g. Linux: LRU 2Q
   a) Two FIFO lists, each simulating the reference bit state
   b) No hardware requirement
3. Note that the concept of page replacement is seen at all interfaces of the memory hierarchy, just a different time scales; and, within certain applications
   a) We focus on the disk-RAM interface because the difference in access time is so great
   b) Web servers and browsers also maintain caches of a fixed size

X. Segmentation

A. Under virtual memory, processes are given a single, contiguous, logical address space

B. It may be beneficial to have many separate virtual address spaces

C. Consider what constitutes a process:

1. Code, globals, symbols, stack, heap
2. Where do each of these go in memory? At fixed offsets?

D. **Segmentation** supports many virtual address spaces per process, addressed by <segment-name, offset>

1. A 2D array of logical space, maps to a 1D array of physical addresses