Memory Management: Virtual Memory

- So far, we have been loading entire processes into memory: process gets started, process asks for whatever memory it needs, then memory is allocated
- *Paging* and *segmentation* break up a process's memory space so that it doesn't have to be physically contiguous — this addresses external fragmentation plus adds other benefits (sharing, protection)
- But does an entire process's address space *have* to be in memory all the time? The answer is no, and as a result, we can have *virtual memory*

Demand Paging

- Note how a process's logical address space (the addresses that it sees and uses) is already cleanly separated from its physical address space
- Instead of loading an entire process from secondary storage into physical memory, we can load it only *as pages are needed* by the code — this is *demand paging*
- Pages of the logical address space that are never accessed during a process's run (rarely used routines, overallocated data structures, etc.) are never loaded — we save memory *and* I/O

Demand Paging Basics

- The *valid/invalid bit* in a process's page table gains new meaning: an *invalid* value may now also indicate that a page is not yet in main memory, in addition to possibly being outside of the process's memory space
- Start with a certain number of pages loaded into memory (or none if we are doing *pure demand paging*)
- When the CPU tries to touch a memory address belonging to an invalid page (including the address of the instruction itself), a *page-fault trap* is triggered

- A typical page-fault handler then follows this routine:
  - Verify the validity of the memory address
  - Terminate the process if the address was invalid
  - Allocate a free physical memory frame
  - Schedule a disk read — theoretically, the CPU can do other things at this point while the I/O does its job
  - Update the page table when the page arrives
  - Restart the interrupted instruction
- The principle of *locality*, specifically *locality of reference*, means that for certain periods, a process will have all of the pages that it needs and thus stays completely *memory resident* for a while

Demand Paging Performance

- With demand paging, average memory access times are now modified by how frequently we get page faults
- If *ma* is pure main memory access time, *pf* is access time with a page fault, and *p* is the probability of a memory access resulting in a page fault, we get:

$$(1 - p)ma + (p)pf = \textit{effective access time}$$

- If you consider that page fault time, which involves secondary storage I/O, can be 100–1000x longer than main memory access, you see how *p* is a *huge* deal!

Copy-on-Write

- One technique that decreases the page-fault rate is *copy-on-write*, and it takes advantage of how child process frequently start out as copies of their parents
- When a child process is forked from a parent, it can *continue to use the same frames* to which the parent's pages are mapped
- However, these pages are marked as "copy-on-write," meaning that, once the child process tries to modify its "copy" of the page, *that's* when the page is duplicated

Page Replacement

- Note how demand paging helps increase the degree of multiprogramming — since we no longer allocate a process's entire address space at a single time, we can potentially run more processes concurrently
- Thus, an OS may work like an airline — in a way, it "overbooks" the available memory on the assumption that the running processes won't want *all* of their possible memory at a single moment
- However, this *may* happen — and so we need to figure out an approach for *page replacement*

Page Replacement Basics

- Page replacement is needed when we get a page fault but don't have a free frame for the incoming page; we therefore choose a *victim frame* to overwrite
- Of course, the victim frame may contain changed memory, so we need to write that to disk (a *page-out*)
- We may therefore have not one but *two* I/O operations during page replacement — a page-out of the victim frame and a page-in of the demanded page
- A *modify* or *dirty bit* may save us a page-out, since we won't need to write a frame that hasn't been changed

Page-Replacement Algorithms

- A *page-replacement algorithm* determines how we decide on the victim frame
- To compare them, we use one or more *reference strings* — sequences of memory accesses that represent addresses as they are needed by processes
- We also need an initial available frame count — how much physical memory do we have in the first place
- When we need to page-in and don't have a free frame, we use the page-replacement algorithm to pick the victim, perform the replacement, then move on

- Like process scheduling, page-replacement algorithms range from very straightforward to a theoretical ideal…which can only be approximated in real systems
- We start with *first-in, first-out* (FIFO) page replacement — or, "always replace the oldest page"
    - Conceptually simple, easy to code
    - But it leads to *Belady's anomaly* — with certain reference strings, we may have *more* page faults as we increase the number of available frames!
- There is a theoretically optimal page-replacement algorithm (a.k.a. OPT): "Replace the page that won't be used for the longest period of time" — but, as with SJF, this involves future knowledge that we can't have

Least-Recently-Used (LRU) Page Replacement

- LRU tries to approximate OPT by replacing the page that *hasn't been* used for the longest time, instead of the page that *won't be* used
- With LRU, we thus need to store a value for each page indicating when it was last used — two options:
  - *Counters*: Increment for every memory reference, and write the current value into each page; requires a search for the page with the lowest value
  - *Stack*: Push pages as they are referenced (possibly moving them from the middle of the stack); no search required, just grab from the bottom of the stack
- LRU does not exhibit Belady's anomaly

- Unfortunately, LRU requires significant hardware support — counter updates or stack manipulations have to take place *on every memory reference*, and so would use too much CPU if done in software
- Thus, we have a family of *LRU-approximation* page replacement algorithms, with simpler hardware needs — specifically, a single *reference bit* set on page access:
  - *Additional-reference-bits* shifts the reference bit on preset timer interrupts, resulting in a usage-per-time-period record…lowest number loses
  - If we can't even afford multiple reference bits, *second-chance* is essentially FIFO with a reference-bit check; if it is set, then we clear the bit and give the page a "second chance"
  - But wait, there's more…there's also *enhanced second-chance*, which adds the *modify* or *dirty bit* (already available to save on page-out costs) to the victim selection criteria — the best victim is the FIFO page that is neither recently used nor modified

Even More Page Replacement Algorithms

- Some page replacement algorithms uses *reference counts* — how many times a page has been accessed
    - *Least-frequently-used* (LFU) replaces the page with the smallest reference count; count can decay over time so that early-activity pages (such as initialization) go away
    - *Most-frequently-used* (MFU) uses a converse premise, that LFU pages may actually be more likely to be used next because "they just got here"
- *Page-buffering algorithms* aren't replacement algorithms in themselves, but try to improve performance
    - *Memory pools* do an intermediate "page-out" to main memory, allowing processes to restart sooner or even re-read that page quickly if it's needed soon enough
    - *Periodic write-out* sends modified pages to disk when there is time, saving on page-out later on if that page is chosen as a victim

---

Application Behavior and Page Replacement

- The wrinkle with page replacement algorithms is that specific performance really depends on the nature of an application — for transparency, we have no choice but to have "one-size-fits-all"
- But sometimes, it's worthwhile to lose the transparency in exchange for better performance — e.g., database management systems, large simulations
- To accommodate this, some OSes provide for a *raw disk*, which an application can read/write directly without OS abstractions

Frame Allocation

- The second major issue in virtual memory is *frame allocation* — how many physical frames should each process get in the first place?
- Some factors to consider:
    - ◇ Obviously the total number of allocated frames cannot exceed physical memory (page sharing helps this a bit, but that hard maximum still exists)
    - ◇ Since an instruction may require multiple accesses (e.g., load an address that is dereferenced from another address — potentially 3 frames, one for the instruction itself and two for the address information), processes must also have a minimum allocation
- Frame allocation algorithms distribute frames in a way that conforms to these constraints

Frame Allocation Algorithms

- *Equal allocation* — Split the available frames evenly across all processes
- *Proportional allocation* — Allocate frames proportionally to a process's size
- With either algorithm, incoming and outgoing processes may dynamically change frame allocations
- There is also an interaction with page replacement: are victim frames chosen from all frames (*global*) vs. just that process (*local*) — allocation may diverge from the specific algorithm with global replacement

Thrashing

- As mentioned, virtual memory allows us to increase multiprogramming by letting us getting away with "overbooking" physical memory…most of the time
- Sometimes, processes *do* exceed their current frame allocation, and so repeatedly page-fault
- Ironically, this may be viewed as a *decrease* in CPU utilization, causing the OS to schedule even *more* processes — making things spiral downward
- This behavior has (IMHO) one of the most aptly-named terms in computer science — *thrashing*

Working-Set Model

- The ultimate cause of thrashing is inadequate frame allocation — a process needs to access memory that occupies more pages than it has frames
- Fortunately, the *principle of locality* presents a possible solution: processes tend to spend blocks of time "within" the same set of pages (a *locality*), moving from one set to another over time
- As long as a process has as many frames as needed by the current locality, then it won't thrash; from this principle, we derive a *working-set model*

Working-Set Model Basics

- We define a parameter, Δ — the *working-set window*
- By looking at the most recent Δ references, we can approximate the current locality; the number of pages in the window sets the process's frame allocation
- Two tricks to this approach:
    - Δ must be the right size, neither too large nor small
    - Implementation — tracking the window for every reference can be unwieldy; we can just approximate using reference bits, *a la* LRU approximation

Page-Fault Frequency

- An alternative to the working-set model is a *page-fault frequency* strategy — just track a process's page-fault rate, and if it gets too high, increase its frame allocation; if it gets too low, then decrease it
- We just need to choose good upper and lower bounds
- Appealing for its directness — in the end, after all, thrashing is all about excessive page faults
- Working sets and page-fault frequency are related: processes page-fault more when changing working sets

Memory-Mapped Files

- Virtual memory techniques have another related but distinct application — *memory-mapped files*
- When a file is memory-mapped, its disk blocks are associated with pages in a process's address space, thus making file I/O look like memory accesses
- Memory mapping may be explicit (e.g., *mmap()* in BSD, *MapViewOfFile()* in Win32) or implicit (e.g., in Solaris, all file I/O is memory-mapped)
- Can be used for, but not necessarily the only way, to implement shared memory

Memory-Mapped I/O

- The "make-it-look-like-memory-access" approach also applies to other I/O in addition to files — this is *memory-mapped I/O*
- Same strategy: direct certain memory addresses to I/O devices — a process just reads from/writes to these addresses, and the CPU sends that data to the device behind the scenes
- Particularly useful for fast-response devices, such as a graphics card: "drawing" on a screen is actually I/O, but looks like memory transfers in code

Kernel Memory Allocation

- Memory used by the kernel tends to require a different strategy from user process memory:
  - We *really* need to minimize fragmentation, particularly if kernel memory is not paged
  - Because the kernel communicates directly with other hardware, sometimes physical contiguity is required
- A simple approach is the *buddy system* — keep dividing memory by two until you get the largest power-of-2 that can accommodate a memory request…simple, but still prone to internal fragmentation

Slab Allocation

- The *slab allocation* strategy eliminates fragmentation; it takes advantage of the fact that it *knows about* the data structures needed by the kernel
- Contiguous memory is divided into *slabs*, which are then assigned to *caches*
- Caches correspond to and are "sized-to-fit" a specific kernel data structure (e.g., PCBs, semaphores, etc.)
- Memory-efficient and fast: size-to-fit ensures no wasted memory, and preallocation enables rapid reuse

Miscellaneous Virtual Memory Issues

Page replacement and frame allocation are the primary issues in virtual memory management, but there are many others — as always, the devil is in the details

- *Prepaging*: Bring more than one page in at a time, such as at startup and/or resuming after suspension (e.g., page-in the entire working-set of a suspended process)
- *Page size*: We've seen how some CPUs offer a choice of page sizes; so, we can potentially choose between improving on fragmentation and locality (small pages) or minimizing table size and I/O (large pages)

- *TLB reach*: Like a TLB's hit ratio, *TLB reach* is related to how many entries it can hold — it is the amount of memory that can be "seen" by the TLB, or *number of entries * page size* — the larger the TLB reach, the more likely that a process's working set is in the TLB
- *Inverted page tables*: While we can't completely do without external page tables, having an inverted page table may reduce second-order page faults
- *Program structure*: Virtual memory is transparent in principle, but programs may perform very differently with a slight change; good compilers will help here
- *I/O interlock*: We need to make sure that we don't page out frames that are waiting on I/O devices; either never do I/O in user space, or allow *locking* of pages

Real-World Specifics

- Windows XP
  - Demand paging with *clustering* — page-in adjacent pages to the requested page
  - Designated working-set minimum and maximum (typically 50 and 345 pages, respectively) with *automatic working-set trimming* when free memory starts running low
  - *Clock* page-replacement algorithm on single-processor x86, FIFO variant on others
- Solaris
  - Maintains a free-page pool with a threshold, *lotsfree*, usually 1/64 of physical memory
  - When pool goes below *lotsfree*, a *pageout* process runs with a second-chance algorithm variant; starts at 4 times per second, then goes to 100 times a second if free memory falls below another threshold, *desfree*
  - A final threshold cross, *minfree*, results in *pageout* with every memory access

- Linux
  - Four kernel threads — *kscand*, *kswapd*, *kupdated*, and *bdflush* — track the state of a page; counter-based LRU provided by *kscand*
  - Five states: *free*, *active*, *inactive dirty*, *inactive laundered*, *inactive clean*; a page becomes *inactive* if its counter goes to zero, and enters *laundered* state while its contents are still being paged out to disk
  - Virtual memory behavior is highly tunable through assorted parameters (e.g., *bdflush.age_buffer* determines how old a buffer may be before flushing to disk; *vm.max_map_count* sets the maximum number of virtual memory areas a process can have, effectively limiting its memory allocation regardless of overall available memory)
- Mac OS X
  - Implements copy-on-write, locks (“wired” memory in OS X terms), and LRU variant
  - Has a special subsystem called *Task Working Set* (TWS) that tracks per-user, per-process fault behavior in on-disk files (*/var/vm/app_profile*); helps in pre-paging and allocating disk blocks — working sets are kept contiguous on disk to minimize seek time
  - Has a *secure virtual memory* feature that encrypts on-disk swap files
  - 64-bit processes have 18 *exabytes* of address space (1 exabyte ≈ 2^{60})