Demand Paging

A processes' page table at any time has two types of mappings, one to the physical memory and the other to the disk. The physical memory mappings are translated by the hardware. On access to a disk mapping, a page fault is generated, and handled by the operating system. The OS brings the page into the physical memory, and creates a physical memory mapping for the faulting address in the page table.

Physical memory is roughly 100 times costlier than magnetic disk storage, i.e., one byte of physical memory costs 100 times more than one byte of magnetic disk storage. Also, magnetic disks can be very large (100GBs-TBs); physical memories are smaller (10-100GBs). On the other hand, disk access is much slower than physical memory access (milliseconds vs. nanoseconds).

Another way to think is to consider the physical memory as a cache to the disk which maintains all the virtual memory contents. A cache hit costs around 10-100 nanoseconds, while a cache miss costs around 10 milliseconds (around million times slower). Thus, having a high hit-rate in the cache is crucial to the success of this demand paging system. As an example, if the hit rate was 90\%, then the average memory access time would be:

```
10% * 10ms + 90% * 100ns = 1ms
```

In other words, on average, every memory access would take 1ms, which is 10,000 times slower than the physical memory access time of 100ns. Thus, this hit rate is unacceptable. A hit rate of around 99.99% would be more realistic.

Fortunately, because most workloads exhibit a significant degree of spatial and temporal locality in their memory access patterns, high hit rates are usually possible. Thus through demand paging, the OS is able to provide disk-sized address space, that is as fast as physical memory.

The 90/10 rule: Spatial and temporal locality can also be articulated using the 90/10 rule, i.e., 10% of the memory addresses get 90% of the memory references. Given this, it would suffice if the OS ensures that the 10% *hot* memory addresses are usually present in physical memory, while the 90% *cold* memory addresses may be present on disk.

Figure with memory address on the X axis and number of references in the Y axis.

Cache characteristics:

- block size: one page. A very large block size may result in unnecessary data to be read in, and thus cause cache pollution. A very small block size may not be able to exploit spatial locality. More importantly, a small block size will not be able to amortize the cost of a disk seek/rotation over the amount of data read.
- organization or associativity: direct mapped/set-associative/fully-associative? Given that hit-rate is of prime importance, and that miss costs are very high, fully associative seems best. This is because fully-associative caches are likely to have the best possible hit-rates (no conflict misses). The drawback of implementing a fully-associative cache is that identifying the best candidate to replace is costlier (in direct-mapped there is only one choice). However, this cost at replacement time is relatively small as it only involves memory accesses --- recall that a disk access is necessary during replacement, and that dominates the replacement time overwhelmingly.
- hit or miss decision? check the present bit in page table entry in hardware. <1ns if TLB hit, 10-100ns if TLB miss (page table walk)
- hit path? if L1 cache hit, 1-2ns, else 10-100ns to access L2 cache/main memory (both in hardware).
- miss path? page fault handler, disk access (in software)
- replacement policy? sort-of LRU, needs more discussion.
- what happens on write? definitely write-back! (write-through would require a disk access on every write). How does the OS know at replacement time if the page needs to be written back to disk:
    - Option 1: always write back to disk on replacement. con: unnecessary disk accesses
    - Option 2: have a *dirty bit* in the page table entry. This bit is set by the hardware on a write access (notice that this needs to be set by hardware as it is on the hit path and needs to be fast). When this page is first loaded by the OS, the corresponding dirty bit is set to zero, indicating that the page is clean, i.e., its contents are identical to the corresponding disk contents. At replacement time, the page needs to be written back to disk only if its dirty bit is set. The x86 architecture supports the dirty bit, and this mechanism is used for implementing write-back.

What to fetch initially and on-demand? Some options:

- Ask the user: not reliable
- Load some initial pages (based on common-case), e.g., first instruction, stack, and then do demand paging. Can do read-ahead (e.g., if fetch X, then also fetch X+1) or other forms of prefetching (e.g., if fetch X, then also fetch Y based on previous observations). Why is read-ahead useful? spatial locality and that the cost to read two pages from disk is not very different from the cost to read one page (dominated by seek time and rotational latency).

What to eject and when? Cache replacement policy

- Random: pro- easiest/fastest to implement, con- may replace a hot page
- FIFO: throw out oldest page (the page that was brought into the cache earliest).

  pro- easy/cheap to implement (just use an in-memory queue and insert/replace in page-fault handler), fair.

  con- ignores usage. A page that is being used often may get replaced just because it was brought in earlier, while a cold page that was

brought-in later may keep occupying cache space.

- What could be the optimal policy? MIN: throw out page not used for longest time in future. A page that is not used for longest time in future is the best candidate for ejection, as all other pages will be used before this page. The problem is of course, that we do not know anything about the future. Hence, this is impractical, but nonetheless a good yardstick.
- Least Recently Used (LRU): throw out page not used for longest time in past. This is just an approximation of MIN, where we assume that past can be used to predict the future (a common theme in several optimizations). If past == future, then LRU=MIN. If past is somewhat equal to future, LRU has roughly the same performance as MIN

Implementing perfect LRU requires that each page is timestamped on every access. This timestamp needs to be saved on every access, and is thus on the hit path and has to be done in hardware. Saving the timestamp has both time and space overheads. Instead, the hardware implements a 1-bit approximation to a timestamp, namely an *accessed bit* in the page table entry. Thus, we approximate LRU using a 1-bit timestamp, namely the accessed-bit.

The accessed-bit in the pagetable entry is set by the hardware on a memory access. This bit distinguishes between pages that have been accessed recently and those that have not been accessed recently. accessed=1 implies "accessed recently", and accessed=0 implies "not accessed recently". The operating system periodically checks and clears the accessed bit, i.e., sets it to zero. If in a consecutive check, the OS finds that the accessed bit is set, it indicates that the page was accessed in the last periodic interval. Otherwise, it was not accessed. The OS only replaces a page that was not accessed. Also, among all the not-accessed pages, the pages are replaced in FIFO order (recall that FIFO orders the pages by their first access time, so it has some merit at-least, if not precise LRU). This 1-bit approximation to LRU is also called the CLOCK algorithm and is commonly used in operating systems.

One way to implement CLOCK is to arrange the pages in a circular list (clock) and have a pointer iterate over this circular list (clock hand). Each page has its accessed bit set to either zero or one. At replacement time, the clock hand looks at the page at which it is pointing, and uses the following logic:

```
if (page->accessed== 1) {
  page->accessed = 0;
   skip this page, i.e., advance the clock hand to next page
} else {
   evict this page, write to disk if needed
   read new page in this position, and set its accessed bit to zero
   advance the clock hand by one
}
```

Notice that because the clock hand is advanced by one after the new page is added, the newly added pages will be examined in FIFO order. If all pages have the same value for accessed bit, then this algorithm is equivalent to FIFO. On the other hand, if the page was accessed during the time period it takes to complete one revolution of the clock-hand, then it will not be replaced (i.e., it will be given priority over pages that were not accessed). However, its accessed bit will be set to zero for the next round.

What does it mean for the clock hand to move too fast? e.g., if it has to make one full revolution on every page fault. This means that all the pages are being accessed frequently, and this indicates that your physical memory is too small to fit the hot pages of the running program (also called its *working set*). This indicates that perhaps you are incurring many capacity misses, and thus need to buy more memory. On the other hand, if the clock hand moves too slow, it shows that most of the pages in physical memory are cold, i.e., they are not being accessed, and the working set of the program is much smaller than the size of the physical memory.

Two-hand clock: In the clock algorithm discussed so far, the notion of "recency" is defined by one clock revolution, i.e., if a page is accessed within one clock revolution, it is considered to be accessed "recently". If the size of the physical memory is too large, this interval of one clock revolution may become too coarse-grained. To deal with this, a variation is a clock with two hands, separated by a constant angle `theta`. The leading hand clears the accessed bit. The trailing hand looks for the victim page with accessed bit still cleared, i.e., it evicts the page if it finds that the accessed bit is still zero. Thus for a page to be considered recent, it needs to be accessed within the angle formed by the two hands.

If theta=0, it means that the accessed bit information is completely ignored, and it defaults to FIFO. If theta=360, it becomes identical to a single-hand clock.

An example of clock algorithm statistics recorded on real machines/OS:

```
bigmachine$ vmstat -s       # on SunOS
pages scanned by clock/second
- 200K pages examined
- 6 revolutions
- 120K pages freed

smallmachine$ vmstat -s
- 15K revolutions
```

The clock hand in the small machine is moving too fast, indicating the need to buy more memory.