CS3070 – Exploration 5 -- Memory Management
8/6/20

Memory is as important to computing as processing on the CPU. It appears in many forms in our systems:

1. *Registers* – These directly feed the instruction pipelines of the CPU. They are the fastest addressable memory in the entire system and they operate at the same clock speed as the rest of the CPU.
2. *Cache* -- The caches keep recently used data and instructions close to the CPU so that the CPU can reuse data rapidly rather than at the much slower main memory speed. Referred to by level, L1 (level 1) is the fastest and most expensive, positioned "closest" to the CPU registers. Levels L2 and L3 are progressively cheaper, larger, and farther away. Cache is generally completely automatic; system and application programmers generally ignore its existence – it just makes the machine faster without special handling.
3. *Main Memory* (RAM) -- the very fast memory holding program code and data accessed by the CPU. Main memory is usually volatile, meaning that its contents disappear if the power goes off.
4. *Secondary Memory* (Disk or SSD) -- Slower memory that holds any data. CPU cannot access secondary memory directly; it must first copy the data into main memory. Secondary memory is persistent and will not erase data until explicitly ordered to do so. Though slower, it is considerably less expensive than main memory.
5. *Cloud* -- storage servers distributed and replicated across the network hold many digital objects outside a local computer that can be moved to the local computer when needed.

These technologies are organized into a memory hierarchy with the fastest (and most expensive at the top closest to CPU) and the slowest (and least expensive) at the bottom. Memory hierarchy is unavoidable because of significant differences between speed and costs of various technologies. The OS keeps track of where data are in the hierarchy and arranges for them to move up (when needed) or down (when no longer needed). The performance of the system depends critically on how well the OS manages the contents of the memory hierarchy.

Virtual memory is an important part of the hierarchy. It simulates a main memory sufficiently large to hold all the code and data of a process, even if the simulated main memory is smaller than the physically installed RAM. The simulated main memory is called the *address space* of a process. From now on we will use the term main memory for the physically installed RAM of the machine. The address space is divided into equal size *pages* and RAM into *frames* that hold pages. As it executes programs, the CPU generates linear addresses, which can be seen as offsets from the start of the address space. A memory mapping unit (MMU) in the

CPU converts the linear addresses into page-line pairs, finds the frame holding the page, and generates the physical address within that frame. If the MMU encounters a page number not in RAM, it generates a *page fault* interrupt that requests the OS to load the missing page. The MMU contains a small cache, translation-lookaside buffer (TLB), that bypasses page table lookups for recent page-frame pairs.

The performance of virtual memory is very sensitive to the *replacement policy*. This policy that decides which page to evict from main memory. Every eviction becomes a future page fault when the missing page is referenced again. Therefore, we seek replacement policies that minimize total page faults. Because the latency of the secondary memory is so high – for example, disks are 10^6 slower than RAM – poor replacements can quickly accumulate huge delays for processes.

With multiple concurrent processes, the OS partitions the memory among the active processes. There are two ways to partition. In a *fixed partitio*n each process gets a fixed amount of memory. The replacement policy synchronizes evictions with page faults – each fault triggers an eviction – so that the fixed space stays full. Policies commonly used for fixed allocation are FIFO (first in first out), LRU (least recently used), and MIN (minimum possible, that is, optimal).

In a *variable partition*, each process gets a space that varies over time. Fixed space policies can be extended to variable allocation by "globalizing" them. For example, the global LRU evicts the page in main memory that has not been used for the longest time. Thus a page fault in one process can evict a page from another process. This is called "stealing a page". If too many processes are active at once, they all steal from each other so often that the policy introduces a catastrophic collapse of processing efficiency called *thrashing*.

A variable space policy that avoids thrashing is called *working set*. A working set measures the demand of a process for memory, independent of what any other process is doing. It does this with a parameter *T*, called *window size*: whenever a page of the working set has not been used among the last *T* memory accesses, it is automatically evicted. Because evictions do not need to be synchronized with faults, the working set varies in size, rising and falling in accord with the process's demand for memory space. In a working set partition, the portion of memory not in any working set is called *free memory*. A page fault takes a page from the free space and assigns it to the working set; an eviction takes a page from the working set and returns it to free space. Because processes cannot steal from each other's working sets, this form of partitioning cannot thrash.

A working set partition is often close to optimal because of the *principle of locality*: processes refer to small subsets of their address space over extended intervals. These small subsets (locality sets) are strikingly apparent in empirical reference maps (*McMenamin*). The ideal policy (perfectly optimal) would set working sets exactly equal to locality sets. The window *T* usually does an excellent job of discriminating between pages in current locality set because they are used more often than *T*, and pages outside, which are used less often than *T*. Thus, the working set is a close approximation of the optimum.