CS201 W'00 Lecture Notes
Lecture 11

Memory Hierarchy

As with most computer hardware, different memory types offer a tradeoff: high speed vs. large capacity and low price.

This tradeoff extends to secondary storage as well.

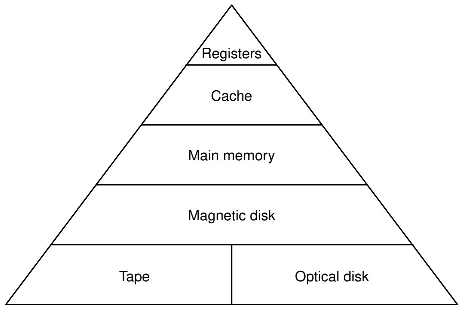

Figure 2-18. A five-level memory hierarchy.

As we descend hierarchy, access times, capacity, and bits/$ all increase.

Disks

Most important form of secondary storage is **magnetic disk**.

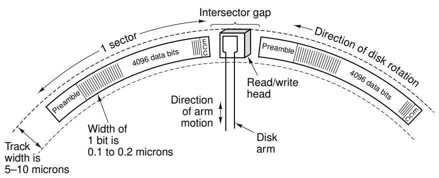

Figure 2-19. A portion of a disk track. Two sectors are illustrated.

Stores bits by selectively magnetizing spots on rotating aluminum platter.

– Current widths typically 3-12 cm.

– Current rotation speeds 3600 to 10,800 revolutions per minute (RPM).

Bits are read/written underneath **head**, attached to **arm**, which moves in and out between center and edge of disk.

Circular path under one head position is a **track**. Current disks support about 1-2K tracks/cm along each radius.

Track is divided into fixed-length **sectors** (e.g., 512 bytes) together with synchronization and error correction fields.

– Current disks support 50-100K bits/cm along circumference.

– Outer tracks can store more sectors than inner ones.

Disk Performance and Organization

Access times:

• **Seek time** to get arm to desired track takes about 5-15msec.

• **Rotational latency** to get desired sector under head: average is 1/2 rotation, i.e. 2-8 msec.

• Peak **transfer rate** is typically 5-20 MB/sec, i.e., 25-100μsec/sector.

Common use of disk storage is to hold named **files**, which might be organized for **sequential** or **random access**, and are read/written using a procedural interface (usually O/S calls). Files can hold either data or programs.

Access time moral: Avoid random access to small amounts of data! (Hence importance of keeping disks organized.)

Virtual Memory

File abstraction is very useful, but not well-suited for

• text of **programs** that are too large to fit in physical memory

• contents of dynamic data regions (stack, heap)

Alternative: view disk memory as **transparent** extension of physical main memory, called **virtual memory**.

• Programs fetch instructions and perform memory load/store operations using **virtual addresses**.

• Range of virtual addresses can be much larger than actual physical memory (though still must fit in available bits in instructions).

• If program accesses a virtual address that isn't currently in physical memory, hardware and O/S cooperate to load the accessed value transparently.

• If a virtual addresses isn't accessed often, its corresponding physical memory is eventually reclaimed for use by other virtual addresses.

• Programmers can write code under illusion that entire VM exists.

VM is very similar to caching!

Physical memory acts as fast "cache" for slow secondary memory.

To get reasonable efficiency, VM relies on **spatial/temporal locality** of memory references.

Difference: VM requires (O/S) software support to manage "misses."

VM is such a convenient abstraction that it is widely used even on systems with plenty of physical memory.

(Some systems even allow ordinary data files to be "mapped" to a range of virtual memory addresses, and accessed using ordinary loads/stores.)

Address Spaces and Paging

The **virtual address space** of a machine is just the set of numeric addresses that can expressed by the bits in the address fields of instructions.

Examples: SPARC V8: $0, 1, 2, \ldots, 2^{32} - 1$; SPARC V9: $0, 1, 2, \ldots, 2^{64} - 1$.

The **physical address space** is a the set of physical memory locations that actually exist.

Example: a machine with 64MB of byte-addressable memory would have physical addresses $0, 1, 2, \ldots, 2^{26} - 1$.

If there is no VM system, (low-end) virtual addresses typically just correspond directly to physical addresses; virtual addresses above the top of physical memory are simply illegal.

VM systems include a hardware **memory management unit (MMU)** that maintains a flexible **mapping** from virtual addrs to physical addrs.

Memory mapping is specified in terms of **pages** of contiguous addresses, typically 512 to 64K bytes.

For each page of virtual address space, memory map tells which page of physical address space (if any) holds values.

MMU Operation

On each memory access, MMU interprets virtual address and consults memory map.

• If page containing address is in physical memory, MMU computes the physical memory address and accesses the memory.

• If page is not in physical memory (**page fault**), MMU causes a hardware trap, which is handled by O/S, which:

– finds an empty slot in the memory map,

– reads the contents of the page from disk into the corresponding physical memory,

– updates the memory map, and

– restarts the program just before the memory access instruction that faulted.

• If no empty slot is available, MMU must free a slot by writing the corresponding physical memory contents back to disk (if changed).

Example VM Addressing Scheme

Assume 4K pages; 32-bit virtual addresses; 32K physical memory (8 pages).

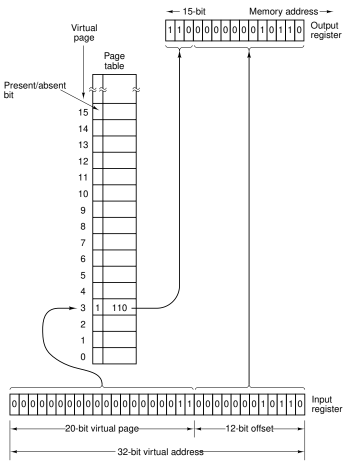

Figure 6-4. Formation of a main memory address from a virtual address.

Example Page Table

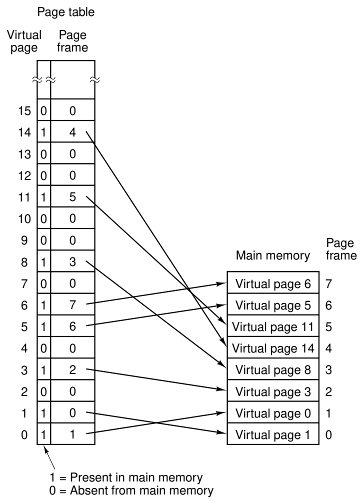

Figure 6-5. A possible mapping of the first 16 virtual pages onto a main memory with eight page frames.

Working Sets and Page Replacement

Under **demand-driven** paging, when program is initially started, its memory map will be empty; programs will cause lots of page faults as they start up.

But if programs continue to page-fault heavily as they run, performance will be bad.

Fortunately, temporal and spatial locality of access patterns mean that most programs touch only a small number of pages (the program's **working set**) within any given interval during execution.

The goal of page-map management is to keep the program's working set in physical memory, by replacing pages that won't be used again soon.

Perfect page replacement would require knowing the program's future access behavior, so must use an approximate **heuristic** instead.

The most common heuristic is to replace the **least-recently used (LRU)** page, on the grounds that this is least likely to be used again soon.

LRU works fairly well for most programs, but there are pathological examples where it is the **worst** policy!

If the working set is larger than available physical memory, no page replacement policy can work well; the program is said to **thrash**.

Page Characteristics

In addition to mapping virtual to physical addresses, most MMU's support **memory protection** by associating a set of **permissions** with each virtual page.

• E.g., pages can be marked as **read-only**, **read-write**, **executable**, completely **invalid**, etc.

• Code and data are allocated to different pages with appropriate permissions; can make page eviction more efficient, and helps catch bugs.

In a multi-processing system like UNIX, several different programs share the MMU, which therefore needs additional fields to indicate which process owns each physical page.

• Code can be **shared** between several processes by having their memory maps point to the **same** physical (read-only) pages (**shared libraries**).

Choosing **page size** is a compromise among several goals:

– maximizing disk performance (favors large pages)

– minimizing MMU hardware (favors large pages)

– avoiding wasting parts of pages (favors small pages)

– avoiding unnecessary thrashing (favors small pages)

VM Storage Organization

• Subdivide virtual address space by function, access, allocation.

• Typical organization (Unix)

```
∞ ---------------
  |             |
  |   STACK     |
  |             |
  ---------------  "top" of stack
  |      |      |      (but stacks nearly always
  |      |      |            grow DOWN!)
  |      ↓      |
  |             |  (in VM systems, these pages
  |             |     don't actually exist)
  |      ↑      |
  |      |      |
  |      |      |
  ---------------  (controlled by  sbrk system call)
  |             |
  |    HEAP     |  Managed by allocator/collector.
  |             |
  ---------------------
  |             |  Uninitialized ("b[lock] s[tatic] s[torage]")
  | STATIC DATA |------
  |             |  Initialized (in object file)
  ---------------------
  |             |
  |   CODE      |  Read-only (in object-file)
  |             |
0 ---------------
```


Storage Classes Static Data : Permanent Lifetimes

• Global variables and constants.

• Allows fixed address to be compiled into code.

• No runtime management costs.

Stack Data : Nested Lifetimes

• Allocation/deallocation is cheap (just adjust stack pointer).

• Most architectures support cheap `sp`-based addressing.

• Good **locality** for VM systems, caches.

• Most languages use stack for activation records.

Heap Data : Arbitrary Lifetimes

• Requires explicit allocation and (dangerous) explicit deallocation or garbage collection.

• Some need heap for activation records, which have non-nested lifetimes.

By allocating different classes of data into different regions in a (very large) address space, we can use VM to get most of the advantages of **segmented** memory models, without the complexity.