

# Memory Management

In multiprogramming systems, processes share a common store. Processes need space for:

- code (instructions)
- static data (compiler initialized variables, strings, etc.)
- global data (global variables)
- stack (local variables)
- heap (*new*, *malloc*)

Memory management is complicated by:

- *unbounded demand* — processes may require more than the total physical storage of the machine
- *protection vs. sharing*
  1. some parts of memory allocated to one process must be protected from another
  2. some memory can be shared among processes for better resource utilization (space and time)
- *dynamic memory requirements* — stacks and heap grow and shrink dynamically. The operating system may need to take memory from one process to give to another

Storage is organized into a hierarchy. In general, the faster the access, the more expensive the store. See Figure 3.1 Finkel.

- punch cards
- magnetic tape
- magnetic disk
- main store
- registers
- on chip cache

## Cache Principle

*Cache Principle*: the more frequently data are accessed, the faster the access should be.  
Goal of caching: keep frequently accessed items in higher levels of hierarchy, less frequently accessed items in lower levels.

A *cache* is a data repository that can be accessed quickly.

An *archive* is a data repository from which access is slower.

## Cache Example

A *cache hit* refers to a data reference that can be satisfied from the cache, while a *cache miss* refers to a data access that must be fetched from the archive.

In many cases, caching significantly improves performance. For example: Assume two level hierarchy having an access time of 50 ns and 500 ns respectively. With a cache *hit rate* of 95% (and *miss rate* of 5%):

$$\text{access time} = .95X50ns + .05X500ns = 72.50ns$$

Note: moving data from one level of the hierarchy to another is relatively expensive. The hysteresis principle implies that data should be left in one place at least long enough to pay back the expense.

## Thrashing

Violating the hysteresis principle leads to *thrashing*, a condition in which the CPU spends most of its time moving data between different levels of storage and little of its time performing useful work.

## No Swapping or Paging

Look at Fig 4.1

Can only execute one program at a time. What is the CPU doing while I/O activity is going on? It's waiting idle! Thus multiprogramming is good (Tanenbaum makes an extended argument to this effect).

## Generic Memory Management

Rather than fix the partitions allow them to be *swapped* in and out of memory. Fig 4.5, 4.6.

Leads to question of how to manage memory, which is the same problem for managing heap space.

Problem: managing a contiguous chunk of memory. Memory manager must support:

- $ptr = getmem(size)$  — get  $size$  bytes of contiguous memory, *new*, *malloc()*.
- $freemem(ptr)$  — return a chunk of memory (previously acquired via *getmem*) to the free pool, *delete*, *free()*.

The storage manager must keep track of which pieces of physical store are in use and which are free. Possibilities:

### Bit Map

*bit map* — Divide memory into fixed size chunks, with one bit (used/unused) devoted to each chunk.

Disadvantages:

- *getmem* requests are rounded up to a multiple of the block size; internal fragmentation results
- tradeoff between size of chunks and size of bit map.

## Boundary Tag (Linked Lists)

*boundary-tag* — keep doubly linked list of all pieces of memory. Reserved fields associated with each piece indicate:

- status of piece (used or unused)
- start and end of piece

*Getmem* searches the list for a piece of adequate size. If an exact size match cannot be found, a large piece is split into two smaller pieces, one returned to the user, the other returned to the free list.

*Freemem* combines adjacent pieces into a single larger piece. See Fig 4-8.

When servicing a *getmem* request, which piece should be chosen?

- *first fit* — find the first piece that is large enough, split it in two, return one piece to user, other to free list
- *best fit* — find smallest piece that satisfies request, split it in two.

Intuitively, *best fit* would appear better. In reality: best fit leaves many tiny pieces too small to use.

*Next Fit*: Variation on first fit: rather than starting search at beginning of list, start where previous search ended. The former tends to increase search time because small pieces reside near start of list.

*Quick Fit*: keep a list sorted by commonly requested sizes. However merging returned space is more expensive.

## Buddy System

Somewhat like boundary tag, but segments are broken into fixed sizes. See Figure 10.18.

## Wasted Space

- *external fragmentation* — wasted space outside of any process. (e.g., small pieces in best fit)

Can analyze how much space is lost to external fragmentation. 50% rule indicates half as many holes as allocations. Can also determine the amount of unused memory.

*Compaction* may help: move chunks around (within physical memory) in order to combine unused pieces into larger pieces. However:

- compaction expensive
  - can only be used in virtual memory systems that bind virtual addresses to physical addresses at access time
- *internal fragmentation* — space allocated to a process that is not used (e.g., if bit maps are used, *getmem* requests are rounded up to a multiple of the chunk size)
  - *overhead space* — space lost to memory management data structures (e.g., bit maps, reserved fields in boundary-tag)