COMP 3000
Lecture 11, October 14th, 2004
Notes by Geoffrey Holtby

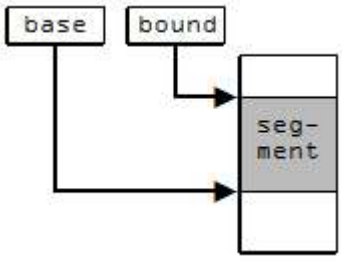
Memory Segmentation

- A segment is just a variably-sized region of memory, defined by a **base** and a **bound**.
- The number and size of segments depends upon the microprocessor architecture
- Before the advent of reduced instruction set (RISC) architectures, microprocessor designers endeavored to create "High Level Language Computer Architecture" systems, or processors that used instructions that were closer to those in high-level languages. The theory was that by creating a higher-level machine language, it would be easier to compile code in FORTRAN, COBOL, or PL/1 into machine code.
- The problem is that complex high-level instructions waste resources, because they tend to be over-general. Example: the VAX instruction for calling functions saves all registers always, even though some functions may not care if some registers get overwritten.
- Segments can thus be seen in two ways: as a way to better align processor-level memory management with the needs of higher-level languages/assembly language programmers, or as a hack to increase address space size while maintaining backwards compatibility.
- Example: The Intel 80286 had a 16 Mbyte (24-bit) address space, but a segment could only be 64K in size to maintain compatibility with 8086/8088 code. Thus, if you wanted an array larger than 64K, you had to divide it up into multiple pieces. A major pain, as you might imagine. The 80386 and later processors allow for full 32-bit segments in a 32-bit address space.

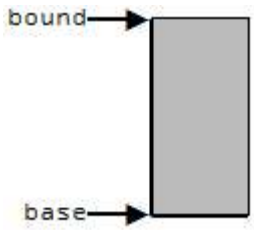
- Segmentation is good for:
  - logical program organization
  - in VM context: semantic swapping
  - extending address space (kind of a hack)

- Modern OSes are moving towards a flat (non-segmented) memory model (i.e. segments are set to 0 so all segments overlap and occupy the entire address space). Thus, processor-level segmentation is mostly a legacy feature at this point.

Segmented Memory Model:



Flat Memory Model:



- Segments can be used for swapping in virtual memory schemes.
  - Downside: lots of external fragmentation! (since the segments are all different sizes)
  - No internal fragmentation, though.

- Paging is used instead of segmentation to get around the problem of external fragmentation.
  - It has some internal fragmentation, but this is minor compared to the massive external fragmentation that can occur with segmentation

- overlapping segments + virtual memory swapping = BAD

- bottom line: paging is a lot nicer than segmentation

OSes and Virtual Memory

- While the hardware can take care of translating individual logical memory references to their corresponding physical address, the OS must manage the set of mappings in the form of page tables.
- Each process has its own page table, because typically each process lives in its own logical address space. The kernel also has its own page table, but it only maps part of the logical address space (e.g. the upper 1G of memory). The kernel's address space is then mapped into the same portion of every process's address space. While a process cannot directly access kernel memory, this arrangement makes it easier for the kernel to access process code and data. (Note: while some OSs don't do this, both Windows NT/2000/XP, Linux, and UNIX use this arrangement.)
- A process's page table must be initialized when it is created; however, this page table need not refer to any physical pages – with virtual memory, the program code & data does not have to be loaded into physical memory on process creation
- So, when does the OS load the pages of a program from disk?
    - lazy model: *demand paging* (only load in code when it is needed)
    - *prepaging*: try to predict what code will be needed, before it's actually asked for.
- In practice, we want some mix of the above two methods.
- i.e. do demand paging, but load in more than one page at a time (this works because of the principle of locality)

Working Set vs. Resident Set:

- *Working Set:* pages being used by program in its current state
- *Resident Set:* pages actually present in physical memory
- Because disks are slow, it is most efficient to try to keep the right pages (working set) in memory
- We ideally want Working Set = Resident Set.
- This can be hard to do! This kind of optimization is an evolving field.

Placement Strategy: Which frame gets which page?

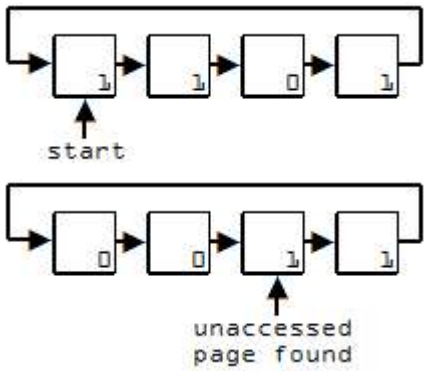
- With paging, this is a very easy problem – It doesn't really matter which page gets which frame.
- It matters a lot in a segmented memory model!
- It also matters in distributed systems using *NUMA (Non Uniform Memory Access)* – in such systems you don't want a process running on one machine to be using memory on a different machine that is has to access over a network, because that would be incredibly slow.

Replacement Policies: We can always use more physical memory, e.g. to buffer I/O – so when you run out, what pages do you overwrite, and when?

- (See Fig 8.15 to see an example of these policies in action)
- *Optimal*: predict the future! ("Crystal Ball" method)
    - Very difficult to do in practice, obviously
    - More useful as a point of comparison for other replacement policies
- *LRU: Least Recently Used*
    - Requires each page to have a time-stamp that is updated on each page access – this is costly!
- *FIFO: First In First Out*
- *Clock Algorithm*: an approximation of LRU
    - Most modern OSes actually use some form of Clock Algorithm.

The Clock Algorithm:

- Makes use of the "accessed" bit that every page has – this bit is set to 1 whenever the page is read or written to. Note this can be a bit complicated – remember, part or all of the page may be cached. Aren't you glad you don't design hardware? :-)
- Cycle through the pages in memory, performing the following steps at each page:
  - Check the access bit.
  - If the access bit is "0", that page can be replaced, and we don't need to look any further
  - If the access bit "1", set it to "0" and proceed to the next page.



The 2-bit Clock Algorithm:

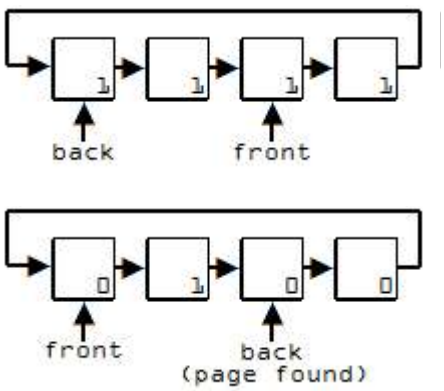
(Makes use of the "modified" bit as well as the "accessed" bit)

- Tries to first find a page that is both un-accessed AND unmodified. If none of these can be found, grab one that has been modified but not accessed (but first, write it to disk!)
- Only zero out the accessed bits after one complete pass through the pages finds no page with both bits zero
- See figure 8.18 and related text in the book..

The 2-handed Clock Algorithm:

(Only uses the "accessed" bit.)

- Use two pointers -- "fronthand" and "backhand".
- Front hand zeros out access bits
- Back hand chooses which page to replace, looking for one with zero access bit
- Both fronthand and backhand continuously scan pages – but their rate (the scanrate) is affected by memory pressure (how many free pages available vs. how many the OS wants to keep in reserve)
- The handspread parameter controls the distance between the two hands
- The scanrate and handspread parameters can be tuned at runtime to meet changing conditions



See Figure 8.23 and related text.

Another strategy: Page Buffering

- FIFO, but with an "elimination queue" -- a buffer where pages wait to be kicked. They can be "rescued" if they are used again before being eliminated.

Replacement Scope:

- *Local:* within the pages already allocated to the process.
- *Global:* all processes.
- Typically, a global scope is used, because this is much easier to implement and it makes best use of system resources.
- # of pages allocated per process – fixed vs. variable allocation
- variable is more typical.