UNIVERSITY OF CALIFORNIA AT BERKELEY
College of Engineering
Department of Electrical Engineering and Computer Sciences
Computer Science Division

CS 162
Fall 1988

John Ousterhout

# Topic 13: Sharing Main Memory -- Segmentation and Paging

- Readings for this topic:
  - Silberschatz/Peterson: Chapter 6.

- Simple uniprogramming with a single segment per process:
  - Highest memory holds OS.
  - Process is allocated memory starting at 0, up to the OS area.
  - When loading a process, just bring it in at 0.
  - Examples: early batch monitors where only one job ran at a time and all it could do was wreck the OS, which would be rebooted by an operator. Many of today's personal computers also operate in a similar fashion.

- Issues:
  - Transparency:
    - Want to let several processes coexist in main memory.
    - No process should need to be aware of the fact that memory is shared. Each must run regardless of the number and/or locations of processes.
  - Safety: processes mustn't be able to corrupt each other.
  - Efficiency (both of CPU and memory) shouldn't be degraded badly by sharing. After all, the purpose of sharing is to increase overall efficiency.

- Relocation. *Draw a simple picture of memory with some processes in it.*
  - Because several processes share memory, we can't predict in advance where a process will be loaded in memory. This is similar to a compiler's inability to predict where a subroutine will be after linking.
  - Relocation adjusts a program to run in a different area of memory. Linker is an example of static relocation used to combine modules into programs. Relocation techniques can also be used to allow several programs to share one main memory.

- Simple multiprogramming with static software relocation, no protection, one segment per process:
  - Highest memory holds OS.
  - Processes allocated memory starting at 0, up to the OS area.
  - When a process is loaded, relocate it so that it can run in its allocated memory area (just like linker: linker combines several modules into one program, OS loader combines several processes to fit into one memory; only difference is that there are no cross-references between processes).
  - Problem: any process can destroy any other process and/or the operating system.
  - Problem: only one segment per process.

- Dynamic memory relocation: instead of changing the addresses of a program before it's loaded, change the address dynamically *during every reference*.

  - *Draw a picture of a processor and a memory box, with a memory relocation box in between.*

- Under dynamic relocation, each program-generated address (called a *logical* or *virtual* address) is translated in hardware to a *physical* or *real* address. This happens as part of each memory reference.
- Dynamic relocation leads to two views of memory, called *address spaces*. With static relocation we force the views to coincide. In some systems, there are several levels of mapping.
- *Draw physical view above logical view, show correspondences. Emphasize importance of learning to think about things at different levels.*

- *Lecture 22 ended here.*

- Base & bounds relocation:
  - Two hardware registers: base address for process, bounds register that indicates the last valid address the process may generate.
  - Each process must be allocated contiguously in real memory.
  - On each memory reference, the virtual address is compared to the bounds register, then added to the base register. A bounds violation results in an error trap.
  - Each process appears to have a completely private memory of size equal to the bounds register plus 1. Processes are protected from each other. No address relocation is necessary when a process is loaded.
  - OS runs with relocation turned off (a bit in the processor status word controls relocation). But, must prevent users from turning off relocation or modifying the base and bound registers (another bit in PSW for user/kernel mode).
  - Problem: how does OS regain control once it has given it up? *We must do two things simultaneously:*

- *Branch into or out of the OS.*
- *Change protection (turn relocation on or off).*

  *Special instructions control this. Note that the branch address must be carefully controlled. This is like a new form of procedure call that saves and restores more than just the PC.*
- Base & bounds is cheap -- only 2 registers -- and fast -- the add and compare can be done in parallel.
- *Explain how swapping can work.*
- Examples: CRAY-1.

- Problem with base&bound relocation:
  - Only one segment. How can two processes share code while keeping private data areas (e.g. shared editor)? *Draw a picture to show that it can't be done safely with a single-segment scheme.*

- Multiple segments.
  - Permit process to be split between several areas of memory.
  - Use a separate base and bound for each segment, and also add two protection bits (read and write).
  - Each memory reference indicates a segment and offset in one or more of three ways:
    - Top bits of address select segment, low bits the offset. This is the most common, and the best.
    - Or, segment is selected implicitly by the instruction (e.g. code vs. data, stack vs. data, or 8086 prefixes).

    (The last alternative is a kludge for machines with such small addresses that there isn't room for both a segment number and an offset)
  - Segment table holds the bases and bounds for all the segments of a process.

- Memory mapping procedure consists of table lookup + add + compare.
- Example: PDP-10 with high and low segments selected by high-order address bit.

- Segmentation example: 2-bit segment number, 12-bit offset.
  - Segment table (all numbers in hexadecimal):

| Segment | Base | Bounds | RW |
|---|---|---|---|
| 0 | 4000 | 6FF | 10 |
| 1 | 0 | 4FF | 11 |
| 2 | 3000 | FFF | 11 |
| 3 | ---- | --- | 00 |

  - Main program (addresses are virtual, octal):

```
240:    PUSH #1108
244:    CALL SIN
258:    ...
```

  - SIN procedure (addresses are virtual):

```
360:    MOVE 2(SP),R2
364:    MOVE @R2,R3
        ...
        RETURN
```

  - Where is 240 in physical memory?
  - Where is 1108 in physical memory?
  - Suppose the SP is initially 265C. Where is it in physical memory?

- *Lecture 23 ended here.*
  - Which portions of the virtual and physical address spaces are used by this process?
  - Simulate the call to SIN; make sure that the data value really gets

picked up from the right place and that the procedure returns correctly.

- Do segments have to start on even 000 physical addresses?
- How can the stack be allowed to grow downward instead of upward?
- What if hardware provides eight segments but your operating system (e.g. UNIX) only uses two or three?

*Lecture 24 ended here.*

- Managing segments:
  - Keep copy of segment table in process control block.
  - When creating process, allocate space for segment, fill in PCB bases and bounds.
  - When switching contexts, save segment table in old process's PCB, reload it from new process's PCB.
  - When process dies, return segments to free pool.
  - When there's no space to allocate a new segment:
    - Compact memory (move all segments, update bases) to get all free space together.
    - Or, swap one or more segments to disks to make space (must then check during context switching and bring segments back in before letting process run).
  - To enlarge segment:
    - See if space above segment is free. If so, just update the bound and use that space.
    - Or, move the segment above this one to disk, in order to make the memory free.
    - Or, move this segment to disk and bring it back into a larger hole (or, maybe just copy it to a larger hole).

- Advantage of segmentation: segments can be swapped and assigned to storage independently.

- Problems:
  - External fragmentation: segments of many different sizes, have to be allocated contiguously.

  (These problems also apply to base and bound schemes)

- Paging: goal is to make allocation and swapping easier, and to reduce memory fragmentation.
  - Make all chunks of memory the same size, call them *pages*. Typical sizes range from 512-8k bytes.
  - For each process, a *page table* defines the base address of each of that process' pages along with read-only and existence bits.
  - Translation process: page number always comes directly from the address. Since page size is a power of two, no comparison or addition is necessary. Just do table lookup and bit substitution.
  - Easy to allocate: keep a free list of available pages and grab the first one. Easy to swap since everything is the same size, which is usually the same size as disk blocks to and from which pages are swapped.
  - Problems:
    - Efficiency of access: even small page tables are generally too large to load into fast memory in the relocation box. Instead, page tables are kept in main memory and the relocation box only has the page table's base address. It thus takes one overhead reference for every real memory reference. to make tables manageable.
    - Table space: if pages are small, the table space could be substantial. In fact, this is a problem even for normal page

sizes: consider a 32-bit addresss space with 1k pages. What if the whole table has to be present at once? Partial solution: keep base and bounds for page table, so only large processes have to have large tables.

- Internal fragmentation: page size doesn't match up with information size. The larger the page, the worse this is.

- Paging and segmentation combined: use two levels of mapping to make tables manageable.
  - Each segment contains one or more pages.
  - Segments correspond to logical units: code, data, stack. Segments vary in size and are often large. Pages are for the use of the OS; they are fixed-size to make it easy to manage memory.
  - Going from paging to P+S is like going from single segment to multiple segments, except at a higher level. Instead of having a single page table, have many page tables with a base and bound for each. Call the stuff associated with each page table a segment.

- Show System 370 example: 24-bit virtual address space, 4 bits of segment number, 8 bits of page number, and 12 bits of offset. Segment table contains real address of page table along with the length of the page table (a sort of bounds register for the segment). Page table entries are only 12 bits, real addresses are 24 bits.

- Example of S/370 paging (all numbers in hexadecimal):
  - Segment table (base|bounds|protection): 2000|14|R, 0|0|0, 1000|D|RW. Assume page table entries are each 2 bytes long.
  - At location 1000: 6, B, 4 (each value is 2 bytes long).
  - At location 1020: 7, C (each value is 2 bytes long).

    - At location 2000: 13, 2A, 3 (each value is 2 bytes long).
    - At location 2020: 11, 1F (each value is 2 bytes long).
    - Map the following addresses from virtual to physical:
        - 2070 read *(3070)*
        - 202016 read *(4016)*
        - 104C84 *(protection error)*
        - 11424 read *(1F424)*
        - 210014 write *(bounds violation)*

- *Lecture 25 ended here.*

- If a segment isn't used, then there's no need to even have a page table for it.

- Can share at two levels: single page, or single segment (whole page table).

- Pages eliminate external fragmentation, and make it possible for segments to grow without any reshuffling.

- If page size is small compared to most segments, then internal fragmentation is not too bad.

- The user is not given access to the paging tables.

- If translation tables are kept in main memory, overheads could be very high: 1 or 2 overhead references for every real reference.

- *The difficult thing in the VAX example below is to be able to view the same thing at different levels, and to selectively ignore detail. Once*

*understand something, must trust that it works without thinking about it anymore.*.

- Another example: VAX.
  - Address is 32 bits, top two select segment. Four base-bound pairs define page tables (system, P0, P1, unused).
  - Pages are 512 bytes long.
  - Read-write protection information is contained in the page table entries, not in the segment table.
  - One segment contains operating system stuff, two contain stuff of current user process.
  - Potential problem: page tables can get big. Don't want to have to allocate them contiguously, especially for large user processes. Solution is to use the system page table to map the user page tables so the user page tables can be scattered:
    - System base-bounds pairs are physical addresses, system tables must be contiguous.
    - User base-bounds pairs are virtual addresses in the system space. This allows the user page tables to be scattered in non-contiguous pages of physical memory. *Explain it as like recursion*.
    - The result is a two-level scheme.

- Problem with segmentation and paging: extra memory references to access translation tables can slow programs down by a factor of two or three. Too many entries in translation tables to keep them all loaded in fast processor memory.

- Remember notion of locality: at any given time a process is only using a few pages or segments.

- Solution: Translation Buffer (TB). A translation buffer is used to store a few of the translation table entries. It's very fast, but only remembers a small number of entries. On each memory reference: *(draw picture, explain name)*
  - First ask TB if it knows about the page. If so, the reference proceeds fast.
  - If TB has no info for page, must go through page and segment tables to get info. Reference takes a long time, but give the info for this page to TB so it will know it for next reference (TB must forget one of its current entries in order to record new one).

- TB Organization: Show picture of black box. Virtual page number goes in, physical page location comes out. Similar to a cache, usually direct-mapped.

- TB is just a memory with some comparators. Typical size of memory: 2k entries. Each entry holds a virtual page number and the corresponding physical page number. How can memory be organized to find an entry quickly?
  - One possibility: search whole table from start on every reference.
  - A better possibility: restrict the info for any given virtual page to fall in exactly one location in the memory. Then only need to check that one location. E.g. use the low-order bits of the virtual page number as the index into the memory. This is the way real TB's work.

- Disadvantage of TB scheme: if two pages use the same entry of the memory, only one of them can be remembered at once. If process is referencing both pages at same time, TB doesn't work very well.

- *Lecture 26 ended here.*

- Example: TB with 128 (80 hexidecimal) slots. Suppose the following virtual pages are referenced (hex): 62A, 21C5, 62A, 21C5, ... 3AA, 21C5, 3AA, 62A.

- TBs are a lot like hash tables except simpler (must be to be implemented in hardware). Some hash functions are better than others.
  - Is it better to use low page number bits than high ones? *Low ones are best: if a large contiguous chunk of memory is being used, all pages will fall in different slots*.
  - Is there any way to improve on the TB hashing function? *Use some of the segment bits too: that way code and data can't interfere with each other.*

- Another approach: let any given virtual page use either of *two* slots in the TB. Make memory wider, use two comparators to check both slots at once.
  - This is about as fast as the simple scheme, but a bit more expensive (two comparators instead of one, also have to decide which old entry to replace when bringing in a new entry).
  - Advantage: less likely that there will be conflicts that degrade performance (takes three pages falling in the same place, instead of two).
  - Explain names:
    - Direct mapped.
    - Set associative.
    - Fully associative.

- Must be careful to flush TB during each context swap. Why?

- In practice, TB's have been extremely successful: 98% hit ratio is typical for 128 entries.

- Memory mapping for Sun workstations: unusual hybrid that combines TB and two-level mapping into a single structure.
  - Two tables, each with 16k entries: segment table, page table. Both implemented in hardware.
  - 8-kbyte pages (13 bits).
  - Each segment contains 32 pages (5 bits).
  - First table indexed by 3-bit PID plus 11 bits of segment number (total virtual address size: 13+5+11 = 2**29 = 512 Mbytes).
  - Output of first table: 6 bits of id for Page Map Entry Group (PMEG) in second table.
  - Second table indexed by first table output + page number within segment.
  - Output of second table: Physical page number plus protection bits.

- Advantages of Sun scheme:
  - Take advantage of technology (large fast RAM chips).
  - Eliminate complexity of TB: reloads are handled by software.
  - Keeps several processes' addressing information loaded simultaneously: faster context switches.

- Problem: how does the operating system get information from user memory? E.g. I/O buffers, parameter blocks. Note that the user passes the OS a *virtual address*.
  - In some cases the OS just runs unmapped. Then all it has to do is read the tables and translate user addresses in software. However, addresses that are contiguous in the virtual address space may

not be contiguous physically. Thus I/O operations may have to be split up into multiple blocks. *Draw an example.*

- A few machines, most notably the VAX, make both system information and user information visible at once (but can't touch system stuff unless running with special kernel protection bit set). This makes life easy for the kernel, although it doesn't solve the I/O problem.

- *Lecture 27 ended here.*
  - Suppose the operating system also runs mapped in a different address space than the user. Then it must generate a page table entry for the user area. Some machines provide special instructions to get at user stuff. Note that under no circumstances should users be given access to mapping tables.