15 Sharing Main Memory – Segmentation and Paging

Readings for this topic: Anderson/Dahlin Chapter 8–9; Siberschatz/Galvin Chapter 8–9

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. First generation personal computers also operated 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.

- 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.
- Problem: once a program runs, cannot relocated it again to compact memory (don't know where the pointers are in the heap or stack).

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

- Imagine 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. Note importance of learning to think about things at different levels.

Base & bounds relocation:

- Two hardware registers: base address for process, bounds register that indicates the last valid address the process may generate.

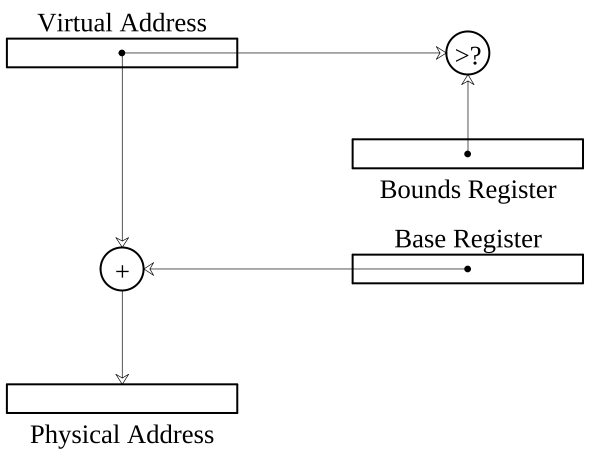
- Each process must be allocated contiguously in real memory.
- On each memory reference, the virtual address is compared to the bounds register, then added to the base register. A bounds violation results in an error trap.
- Each process appears to have a completely private memory of size equal to the 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.
- Examples: CRAY-1.

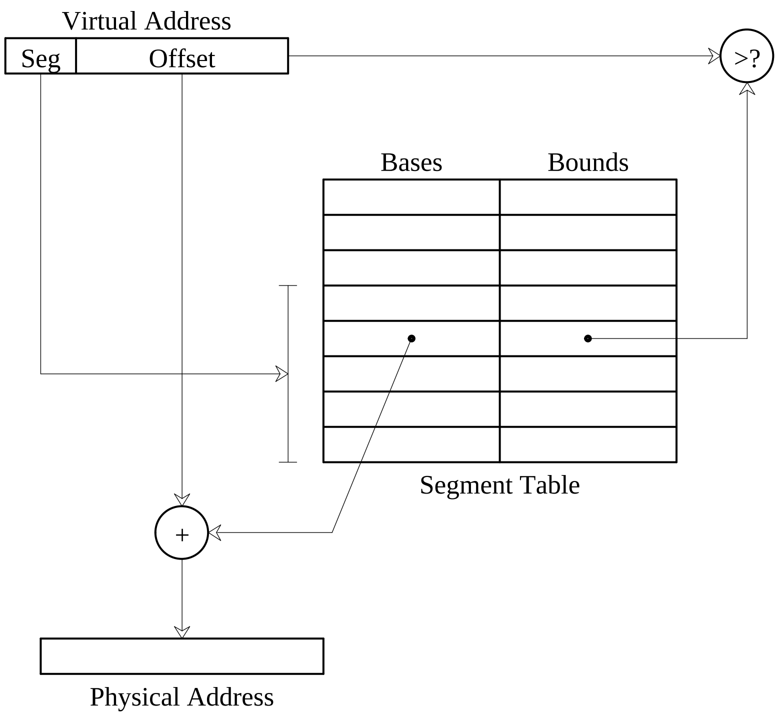
Problem with base&bound relocation:

- Only one segment. How can two processes share code while keeping private data areas (e.g. shared editor)? Can't be done safely with a single-segment scheme.

Base and Bounds Relocation



Segmentation



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):

```
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?
- 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?

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.

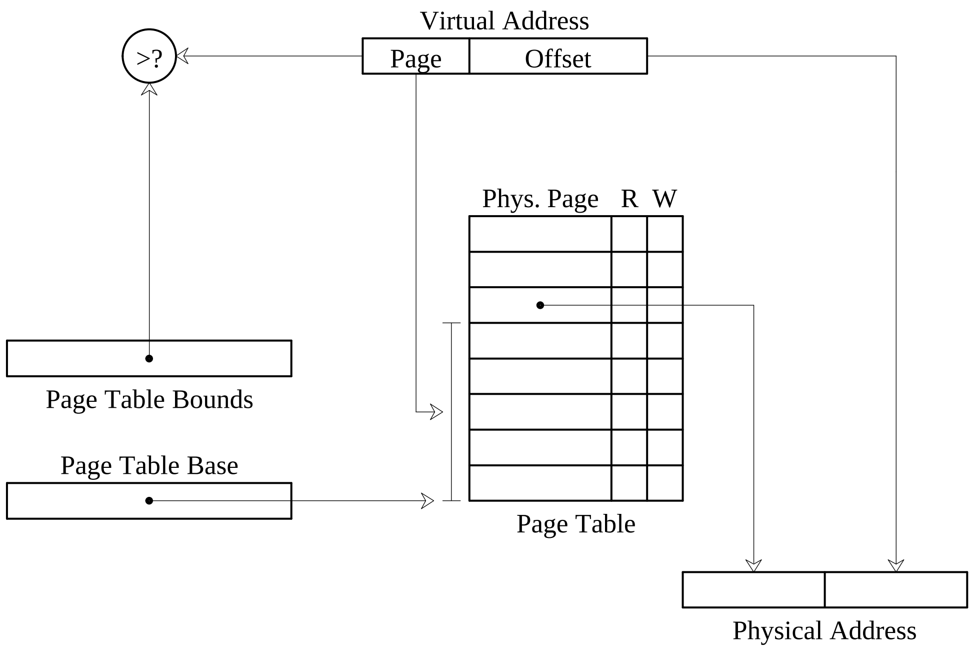
(These problems also apply to base and bound schemes)

Paging: goal is to make allocation and swapping easier, and to reduce memory fragmentation.

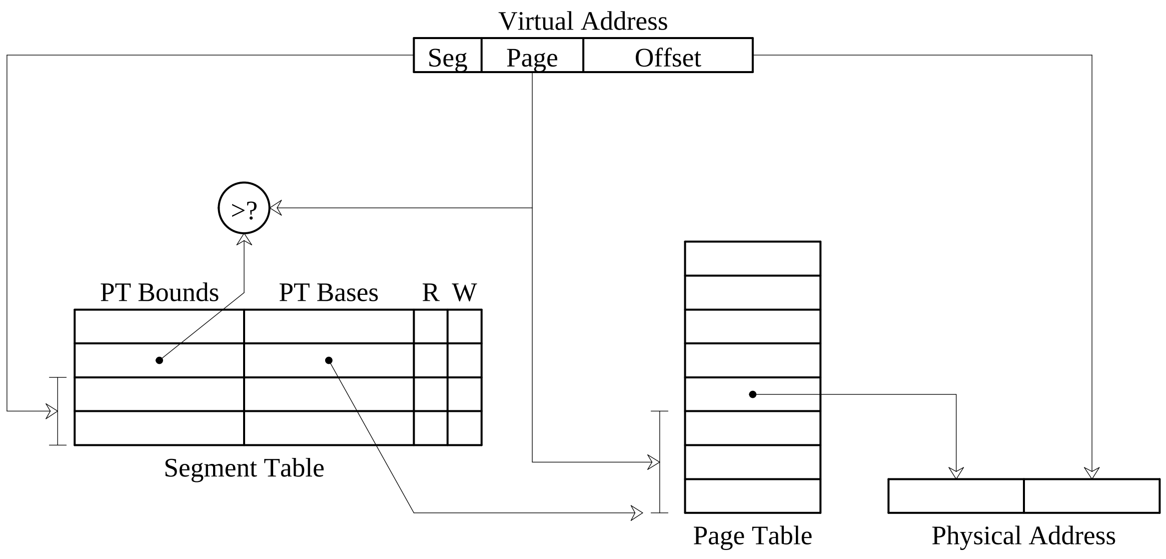
- Make all chunks of memory the same size, call them *pages.* Typical sizes are 4 or 8k bytes.
- For each process, a *page table* defines the base address of each of that process' pages along with read-only and existence bits.
- Translation process: page number always comes directly from the address. 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.
  - 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 address space with 4k 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



Combined Paging and Segmentation



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.

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)

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.

Another example: VAX. 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.

- 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. Think of it 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 Lookaside Buffer (TLB). 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:

- First ask TLB if it knows about the page. If so, the reference proceeds fast.
- If TLB 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 TLB so it will know it for next reference (TLB must forget one of its current entries in order to record new one).

TLB Organization: black box. Virtual page number goes in, physical page location comes out. Similar to a cache, usually fully associative.

What needs to be done to the TLB during a context switch? Make sure cached translations are not used in a different process. Two approaches:

- Invalidate all entries during context switches. Lots of TLB misses after a context switch.
- Tag each entry with a domain identifier. Add a HW register that contains the domain id of the currently executing process (changed during context switch.) TLB hits if an entry's domain id matches the contents of that register.

In traditional memory management units (MMU), TLB entries are loaded from main memory by hardware.

- This is fast.
- OS intervention is required only in case of a page fault.
- Page/Segment table structure is prescribed by MMU hardware.

With TLB-only memory management units (found, for example in the MIPS architecture, and many modern RISC architectures.), TLB misses are handled as follows:

- Trap to the OS.
- OS determines which translation was required and missing from the TLB.
- OS selects a TLB entry for replacement.
- OS computes and loads the requested entry. (This may require lookup of OS-specific page-table-like data structures, and potentially page fault handling.)
- OS returns from trap handler.

With this approach,

- The MMU hardware is very simple, permitting larger, faster TLB.
- TLB misses are somewhat more expensive (SW handling).
- OS designer has complete flexibility in choice of MM data structures.

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.
- 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.

- 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.