[08] SEGMENTATION

OUTLINE

- Segmentation
  - An Alternative to Paging
- Implementing Segments
  - Segment Table
  - Lookup Algorithm
- Protection and Sharing
  - Sharing Subtleties
  - External Fragmentation
- Segmentation vs Paging
  - Comparison
  - Combination
- Summary
- Extras
  - Dynamic Linking & Loading

SEGMENTATION

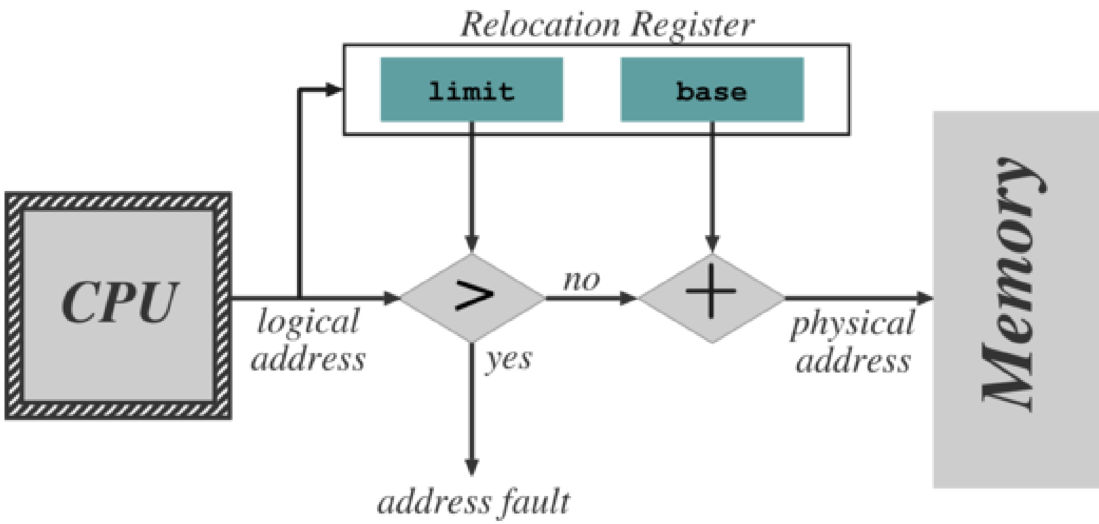
- **Segmentation**
  - **An Alternative to Paging**
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- Summary
- Extras

AN ALTERNATIVE TO PAGING

View memory as a set of segments of no particular size, with no particular ordering

This corresponds to typical modular approaches taken to program development

The length of a segment depends on the complexity of the function (e.g., `sqrt`)



WHAT IS A SEGMENT?

Segmentation supports the user-view of memory that the logical address space becomes a collection of (typically disjoint) segments

Segments have a **name** (or a **number**) and a **length**. Addresses specify **segment**, and **offset** within segment

To access memory, user program specifies *segment* + *offset*, and the compiler (or, as in MULTICS, the OS) translates

This contrasts with paging where the user is unaware of memory structure — everything is managed invisibly by the OS

IMPLEMENTING SEGMENTS

- Segmentation
- **Implementing Segments**
  - **Segment Table**
  - **Lookup Algorithm**
- Protection and Sharing
- Segmentation vs Paging
- Summary
- Extras

IMPLEMENTING SEGMENTS

Logical addresses are pairs, (`segment`, `offset`)

For example, the compiler might construct distinct segments for global variables, procedure call stack, code for each procedure/function, local variables for each procedure/function

Finally the loader takes each segment and maps it to a physical segment number

IMPLEMENTING SEGMENTS

Segment	Access	Base	Size	Others!

Maintain a **Segment Table** for each process:

- If there are too many segments then the table is kept in memory, pointed to by **ST Base Register** (STBR)
- Also have an **ST Length Register** (STLR) since the number of segments used by diferent programs will diverge widely
- ST is part of the process context and hence is changed on each process switch
- ST logically accessed on each memory reference, so speed is critical

IMPLEMENTING SEGMENTS: ALGORITHM

1. Program presents address (s, d).
2. If $s \geq$ STLR then give up
3. Obtain table entry at reference s+STBR, a tuple of form (b_s, l_s)
4. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d), else fault

---

- The two operations b_s, d (concatenation) and $0 \leq d < l_s$ can be done simultaneously to save time
- Still requires 2 memory references per lookup though, so care needed
- E.g., Use a set of associative registers to hold most recently used ST entries
- Similar performance gains to the TLB description earlier

PROTECTION AND SHARING

- Segmentation
- Implementing Segments
- **Protection and Sharing**
  - **Sharing Subtleties**
  - **External Fragmentation**
- Segmentation vs Paging
- Summary
- Extras

PROTECTION

Segmentation's big advantage is to provide protection between components

That protection is provided *per segment*; i.e. it corresponds to the logical view

Protection bits associated with each ST entry checked in usual way, e.g., instruction segments should not be self-modifying, so are protected against writes

Could go further — e.g., place every array in its own segment so that array limits can be checked by the hardware

SHARING

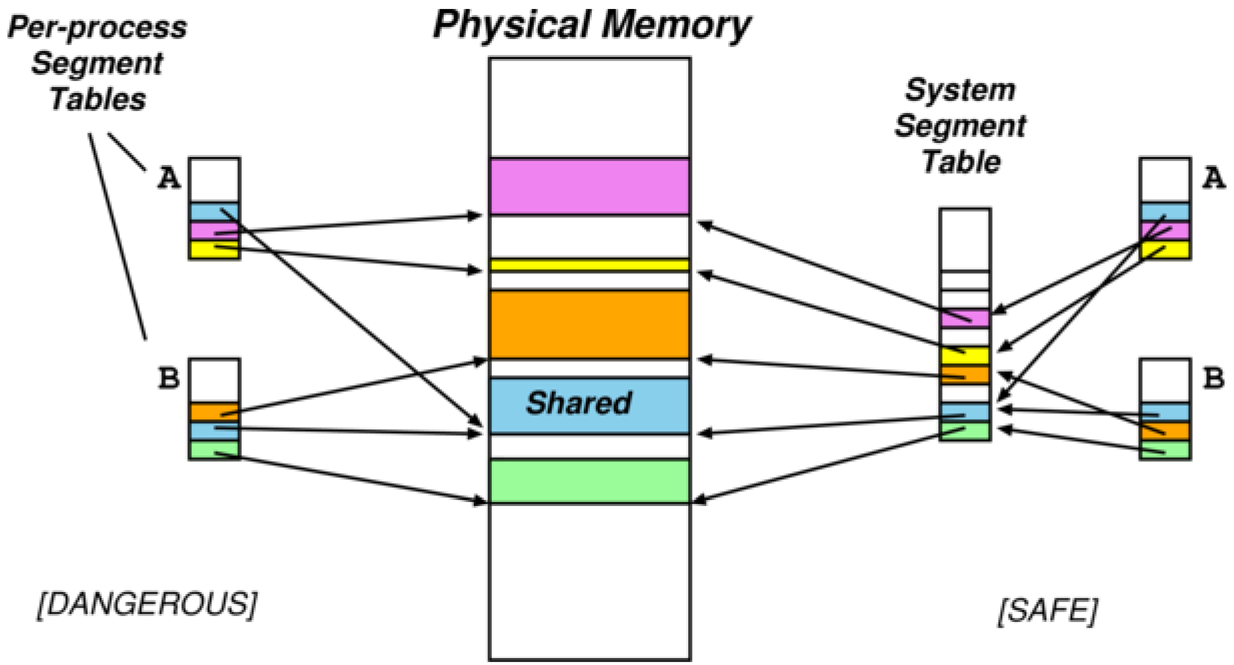
Segmentation also facilitates sharing of code/data:

- Each process has its own STBR/STLR
- Sharing is enabled when two processes have entries for the same physical locations
- Sharing occurs at segment level, with each segment having own protection bits
  - For data segments can use copy-on-write as per paged case
- Can share only parts of programs, e.g., C library but there are subtleties

SHARING: SUBTLETIES

- For example, jumps within shared code
  - Jump specified as a condition + transfer address, i.e., (`segment`, `offset`)
  - Segment is (of course) this one
  - Thus all programs sharing this segment must use the same number to refer to it, else confusion will result
  - As the number of users sharing a segment grows, so does difficulty of finding a common shared segment number
  - Thus, specify branches as PC-relative or relative to a register containing the current segment number
  - (Read only segments containing no pointers may be shared with different seg numbers)

SHARING SEGMENTS



- Wasteful (and dangerous) to store common information on shared segment in each process segment table
- Assign each segment a unique **System Segment Number** (SSN)
- **Process Segment Table** simply maps from a **Process Segment Number** (PSN) to SSN

EXTERNAL FRAGMENTATION RETURNS

Long term scheduler must find spots in memory for all segments of a program. Problem is that segments are variable size – thus, we must handle **fragmentation**

1. Usually resolved with best/first fit algorithm
2. External frag may cause process to have to wait for sufficient space
3. Compaction can be used in cases where a process would be delayed

Tradeoff between compaction/delay depends on average segment size

- Each process has just one segment reduces to variable sized partitions
- Each byte has its own segment separately relocated quadruples memory use!
- Fixed size small segments is equivalent to paging!
- Generally, with small average segment sizes, external fragmentation is small – more likely to make things fit with lots of small ones (box packing)

SEGMENTATION VS PAGING

- Segmentation
- Implementing Segments
- Protection and Sharing
- **Segmentation vs Paging**
  - **Comparison**
  - **Combination**
- Summary
- Extras

SEGMENTATION VERSUS PAGING

- Protection, Sharing, Demand etc are all per segment or page, depending on scheme
- For **protection and sharing**, easier to have it per logical entity, i.e., per segment
- For **allocation and demand access** (and, in fact, certain types of sharing such as COW), we prefer paging because:
  - Allocation is easier
  - Cost of sharing/demand loading is minimised

	logical view	**allocation**
segmentation	good	bad
paging	bad	good

COMBINING SEGMENTATION AND PAGING

1. **Paged segments**, used in Multics, OS/2
   - Divide each segment s_i into $k = \lceil (l_i/2^n) \rceil$ pages, where l_i is the limit (length) of the segment
   - Provision one page table per segment
   - Unfortunately: high hardware cost and complexity; not very portable
2. **Software segments**, used in most modern OSs
   - Consider pages $[m, \ldots, m+l]$ to be a segment
   - OS must ensure protection and sharing kept consistent over region
   - Unfortunately, leads to a loss of granularity
   - However, it is relatively simple and portable

Arguably, main reason hardware segments lost is portability: you can do software segments with just paging hardware, but cannot (easily) do software paging with segmentation hardware

SUMMARY

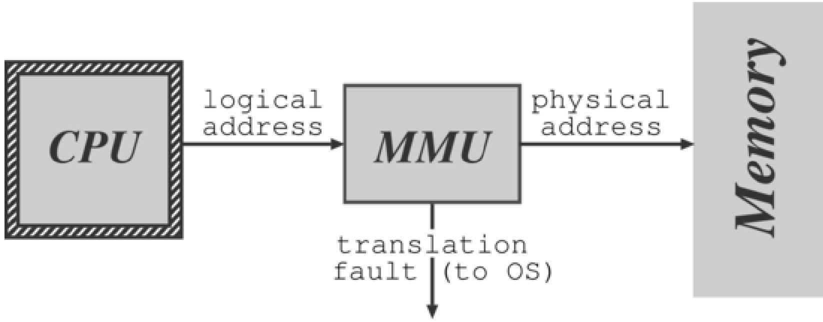
- Segmentation
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- **Summary**
- Extras

SUMMARY: VIRTUAL ADDRESSING

- Direct access to physical memory is not great as have to handle:
  - Contiguous allocation: need a large lump, end up with external fragmentation
  - Address binding: handling absolute addressing
  - Portability: how much memory does a "standard" machine have?
- Avoid problems by separating concepts of virtual (logical) and physical addresses (Atlas computer, 1962)
- Needham's comment

> *"every problem in computer science can be solved by an extra level of indirection"*

SUMMARY: VIRTUAL TO PHYSICAL ADDRESS MAPPING



- Runtime mapping of logical to physical addresses handled by the MMU. Make mapping per-process, then:
  - Allocation problem split:
    - Virtual address allocation easy
    - Allocate physical memory 'behind the scenes'
  - Address binding solved:
    - Bind to logical addresses at compile-time
    - Bind to real addresses at load time/run time
- Modern operating systems use paging hardware and fake out segments in software

SUMMARY: IMPLEMENTATION CONSIDERATIONS

- **Hardware support**
  - Simple base register enough for partitioning
  - Segmentation and paging need large tables
- **Performance**
  - Complex algorithms need more lookups per reference plus hardware support
  - Simple schemes preferred eg., simple addition to base
- **Fragmentation**: internal/external from fixed/variable size allocation units
- **Relocation**: solves external fragmentation, at high cost
  - Logical addresses must be computed dynamically, doesn't work with load time relocation
- **Swapping**: can be added to any algorithm, allowing more processes to access main memory
- **Sharing**: increases multiprogramming but requires paging or segmentation
- **Protection**: always useful, necessary to share code/data, needs a couple of bits

EXTRAS

- Segmentation
- Implementing Segments
- Protection and Sharing
- Segmentation vs Paging
- Summary
- **Extras**
  - **Dynamic Linking & Loading**

DYNAMIC LINKING

Relatively new appearance in OS (early 80's). Uses *shared objects/libraries* (Unix), or *dynamically linked libraries* (DLLs; Windows). Enables a compiled binary to invoke, at runtime, routines which are dynamically linked:

- If a routine is invoked which is part of the dynamically linked code, this will be implemented as a call into a set of stubs
- Stubs check if routine has been loaded
- If not, linker loads routine (if necessary) and replaces stub code by routing
- If sharing a library, the address binding problem must also be solved, requiring OS support: in the system, only the OS knows which libraries are being shared among which processes
- Shared libs must be stateless or concurrency safe or copy on write

Results in smaller binaries (on-disk and in-memory) and increase flexibility (fix a bug without relinking all binaries)

DYNAMIC LOADING

- At runtime a routine is loaded when first invoked
- The dynamic loader performs relocation on the fly
- It is the responsibility of the user to implement loading
- OS may provide library support to assist user

SUMMARY

- Segmentation
  - An Alternative to Paging
- Implementing Segments
  - Segment Table
  - Lookup Algorithm
- Protection and Sharing
  - Sharing Subtleties
  - External Fragmentation
- Segmentation vs Paging
  - Comparison
  - Combination
- Summary
- Extras
  - Dynamic Linking & Loading