CIS520 – Operating Systems
Handout 15
Segments

- Programs need to share data on a controlled basis. Examples: all processes should use same compiler. Two processes may wish to share part of their data.
- Program needs to treat different pieces of its memory differently. Examples: process should be able to execute its code, but not its data. Process should be able to write its data, but not its code. Process should share part of memory with other processes, but not all of memory. Some memory may need to be exported read-only, other memory exported read/write.
- Mechanism to support treating different pieces of address space separately: segments. Program's memory is structured as a set of segments. Each segment is a variable-sized chunk of memory. An address is a segment,offset pair. Each segment has protection bits that specify which kind of accesses can be performed. Typically will have something like read, write and execute bits.
- Where are segments stored in physical memory? One alternative: each segment is stored contiguously in physical memory. Each segment has a base and a bound. So, each process has a segment table giving the base, bounds and protection bits for each segment.
- How does program generate an address containing a segment identifier? There are several ways:
  - Top bits of address specify segment, low bits specify offset.
  - Instruction implicity specifies segment. I.E. code vs. data vs. stack.
  - Current data segment stored in a register.
  - Store several segment ids in registers; instruction specifies which one.
- What does address translation mechanism look like now?
  - Find base and bound for segment id.
  - Add base to offset.
  - Check that offset $<$ bound.
  - Check that access permissions match actual access.
  - Reference the generated physical address.

  How can this be fast enough? Several parts of strategy:
  - Segment table cache stored in fast memory. Typically fully associative.
  - Full segment table stored in physical memory. If the segment id misses in the cache, get from physical memory and reload the cache.
- OS may need to reference data from any process. How is this done? One way: OS runs with address translation turned off. Reserve certain parts of physical memory to hold OS data structures (buffers, PCB's, etc.).
- How do user and OS communicate? Via shared memory. But, OS must manually apply translation any time user program gives it a pointer to data. Example: Exec(file) system call in nachos.
- What must OS do to manage segments?

  - Keep copy of segment table in PCB.
  - When create process, allocate space for segments, fill in base and bounds registers.
  - When switch contexts, switch segment information state in hardware. Examples: may invalidate segment id cache.

- What about memory management? Segments come in variable sized chunks, so must allocate physical memory in variable sized chunks. Can use a variety of heuristics: first fit, best fit, etc. All suffer from fragmentation (external fragmentation).

- What to do when must allocate a segment and it doesn't fit given segments that are already resident? Have several options:

  - Can compact segments. Copy segments to contiguous physical memory locations so that small holes are collected into one big hole. Notice that this changes the physical memory locations of segment's data. What must OS do to implement new translation?
  - Can push segments out to disk. But, must provide a mechanism to detect a reference to the swapped segment. When the reference happens, will then reload from disk.

- What happens when must enlarge a segment? (This can happen if user needs to dynamically allocate more memory). If lucky, there is a hole above the segment and can just increment bound for that segment. If not, maybe can move to a larger hole where the new size fits. If not, may have to compact or swap out segments.

- Protection: How does one process ensure that no other process can access its memory? Make sure OS never creates a segment table entry that points to same physical memory.

- Sharing: How do processes share memory? Typically at segment level. Segment tables of the two processes point to the same physical memory.

- What about protection for a shared segment? What if one process only wants other processes to read segment? Typically have access bits in segment table and OS can make segment read only in one process.

- Naming: Processes must name segments created and manipulated by other processes. Typically have a name space for segments; processes export segments for other processes to use under given names. In Multics, had a tree structured segment name space, and segments were persistent across process invocations.

- Efficiency: It is efficient. The segment table lookup typically does not impose too much time overhead, and segment tables tend to be small with not much memory overhead.

- Granularity: allows processes to specify which memory they share with other processes. But if whole segment is either resident or not, limits the flexibility of OS memory allocation.

- Advantages of segmentation:

  - Can share data in a controlled way with appropriate protection mechanisms.
  - Can move segments independently.
  - Can put segments on disk independently.
  - Is a nice abstraction for sharing data. In fact, abstraction is often preserved as a software concept in systems that use other hardware mechanisms to share data.

- Problems with segmentation:

  - Fragmentation and complicated memory management.
  - Whole segment must be resident or not. Allocation granularity may be too large for efficient memory utilization. Example: have a big segment but only access a small part of it for a long time. Waste memory used to hold the rest.
  - Potentially bad address space utilization if have fixed size segment id field in addresses. If have few segments, waste bits in segment field. If have small segments, waste bits in offset field.
  - Must be sure to make offset field large enough. See 8086.