

Notes

1. Compare multiprogramming with fixed memory partitions (MFT) with variable size memory partitions (MVT). What are the advantages and disadvantages of each scheme?

Ans:

Fixed - Internal fragmentation. Not very flexible! (What if you had partitions of size n , but needed $n + 1$ memory)

Variable - Flexible, but external fragmentations. More overhead.

2. What kind of inter-process protection does Segmentation provide? Is it possible for one process to read/modify the segments of another process without permission? Why/why not?

Ans:

If the OS is responsible and segments are implemented correctly, then no.

- *Each process has its own segment table, meaning it can only find its own segments.*
- *If we can't address it through our table, we can't address it. Period.*
- *If there is segment sharing enabled, then sharing information is passed by the OS, and the protection bits on segment must be set carefully by the OS.*

3. Assuming that each process maintains its own address space, what does adding segmentation/paging to a multi-programmed system add to context switches?

Ans:

Now also need to switch out segment/page table base register so we can find the segment/page table! Alternate, for segments, if we have one register for each segment need to save/switch out these.

4. Say you have a segment-based memory system, and the current process performs a 'new' operation that requires its stack segment to grow. What are the possible sequences of events that could happen before the 'new' call returns?

Ans:

OS is invoked.

- *Try to increase segment.*
- *If space, increase, update table, and return.*

- *If no space, need to either rearrange segments or swap out a segment.*
- *Copy to disk if dirty.*
- *Increase segment space in table and return*

5. Say you have a segment-based memory system, and the current process performs a 'read' operation from a segment that is not in memory. Starting immediately after the 'read' command, describe what happens before the 'read' call can return with the data.

Ans:

- *Seg. fault generated from segment table read.*
- *OS gets disk address of segment from either seg. table or another swap file table.*
- *OS initiates IO to read in segment. (Process will block)*
- *IO finishes, segment table for process updated with new address of segment.*
- *Read commences, reads in from memory. (Next time process is scheduled)*
- *Program continues.*

6. What are some advantages and disadvantages of paging vs. segmenting?

Ans:

- *Segmentation pros:*
 - *Easy to implement*
 - *Each segment requires a base and a limit register and some protection bits.*
 - *Divides up program memory space into nice logical partitions*
 - *Can have your code segment, your stack segment, etc.*
 - *Easy to share segments to avoid wasting memory*
 - *i.e. if a library is used by more than one program, load it into its own segment and share it!*
 - *Can do this with pages too, but not as nice.*
- *Segmentation Cons:*
 - *External fragmentation in physical memory*
 - *This can be a real drag and make things REALLY inefficient.*
- *Paging pros:*
 - *Easy to allocate physical memory*
 - *physical memory is allocated from free list of frames*
 - *to allocate a frame, just remove it from its free list*
 - *external fragmentation is not a problem!*
 - *complication for kernel contiguous physical memory allocation*
 - *many lists, each keeps track of free regions of particular size*
 - *regions' sizes are multiples of page sizes*

- “buddy algorithm”
- Easy to “page out” chunks of programs
 - all chunks are the same size (page size)
 - use valid bit to detect references to “paged-out” pages
 - also, page sizes are usually chosen to be convenient multiples of disk block sizes
- Paging cons:
 - Can still have internal fragmentation
 - process may not use memory in exact multiples of pages
 - Memory reference overhead
 - 2 references per address lookup (page table, then memory)
 - solution: use a hardware cache to absorb page table lookups
 - Memory required to hold page tables can be large
 - need one PTE per page in virtual address space
 - 32 bit AS with 4KB pages = 220 PTEs = 1,048,576 PTEs
 - 4 bytes/PTE = 4MB per page table
 - OS's typically have separate page tables per process
 - 25 processes = 100MB of page tables