Slide 1

Administrivia

- (None.)

Slide 2

Multiprogramming with Fixed/Variable Partitions — Recap

- Comparing the two schemes:
  - Both based on idea that each process's memory is one contiguous block — simple, works well with the simple base/limit MMU described earlier.
  - Admissions scheduling required with fixed partitions, probably a good idea with variable partitions.
  - Complexity versus flexibility, memory use.
- Either could be adequate for a simple batch system.
- But . . .
  - Can we somehow have more jobs/processes "in the system" than we have memory for? Could be useful if processes sometimes wait a long time.
  - Can we do something so processes can acquire more memory as they run?

Slide 3

Aside — Memory Management Within Processes

- What if we don't know before the program starts how much memory it will want? with very old languages, maybe not an issue, but with more modern ones it is.

  I.e., we might want to manage memory within a process's "address space" (range of possible program/virtual addresses).
- Typical scheme involves
  - Fixed-size allocation for code and any static data.
  - Two variable-size pieces ("heap" and "stack") for dynamically allocated data.

Slide 4

Swapping

- Idea — move processes into / out of main memory (when not in main memory, save on disk).

  (Aside — can we run a program directly from disk?)
- Addresses both questions from previous slide; could also provide a way to "fix" fragmentation.
- Implies another level of scheduling (what to swap in/out).
- Makes non-dynamic solutions to relocation problem unfeasible; MMU-based solution still works, though, and for memory protection.
- Consider tradeoffs again — complexity versus flexibility, efficient use of memory.

Slide 5

Simple Memory Management — Recap

- Contiguous-allocation schemes are simple to understand, implement.
- But they're not very flexible — process's memory must be contiguous, swapping is all-or-nothing.
- Can we do better? yes, by relaxing one or both of those requirements — "paging".

Slide 6

Paging

- Idea — divide both address spaces and memory into fixed-size blocks ("pages" and "page frames"), allow non-contiguous allocation.
- Consider tradeoffs yet again — complexity versus flexibility, efficient use of memory.

Slide 7

Paging — Mapping Program to Physical Addresses

- One consequence — mapping from program addresses to physical addresses is much more complicated.
- How? “page table” for each process maps pages of address space to page frames; use this to calculate physical address from program address.

  (Are there page sizes for which this is easier?)
- As with base/limit scheme, makes more sense to implement this in MMU. (Notice again interaction between hardware design and o/s design.)
- Could let page table size vary, but easier to make them all the same (i.e., each process has the same size address space), have a bit to indicate valid/invalid for each entry. Attempt to access page with invalid bit — “page fault”.

Slide 8

Paging and Virtual Memory

- Idea — extend this scheme to provide “virtual memory” — keep some pages on disk. Allows us to pretend we have more memory than we really do.
- Compare to swapping.

Slide 9

Paging and Memory Protection, Page Sizes

- This scheme also provides memory protection. (How?)
- We could also use it to allow processes to share memory. (How?)
- How big to make pages? compare extreme cases (really big, really small).
- Possibly interesting things to notice:
  - If you know how big addresses are, what does that tell you about (maximum) sizes of physical/virtual memory?
  - If you know that and page size, what do you know about size of page table?

Slide 10

Minute Essay

- Given a page size of 64K (2^{16}), 64-bit addresses, and 4G (2^{32}) of main memory, at least how much space is required for a page table? Assume that you want to allow each process to have the maximum address space possible with 64-bit addresses, i.e., 2^{64} bytes.
- (Hints: How many entries? How much space for each one? and no, this is not a very realistic system.)

Minute Essay Answer

- Number of entries is $2^{64}/2^{16}$, i.e., 2^{48}.
- Size of each entry — at least enough for page frame number. There are 2^{16} of them, so we need 16 bits for that. Probably should also include a valid/invalid bit, for a total of 17 bits. Rounding up to a multiple of 8 bits (one byte), that's 3 bytes per entry.
- Total space is $2^{48} \times 3$ — bigger than main memory!! so, not realistic.