**Administrivia**

- Reminders:
  - Homework 2 final deadline class time today. Turn in at end of class, pick up sample solution. Programs on Web later today.
  - Homework 3 due today by 5pm. Solutions available tomorrow.
  - Review session Wednesday at 5pm (tentative — will confirm by e-mail after 11:20am section).

**Minute Essay From Last Lecture**

- What if anything did you find difficult about the programming part of the homework? What if anything did you find interesting/useful about it? Almost half mentioned rusty programming skills.
- The point of the first problem (together with the first written question) was to provide some exposure to system calls in a real-world context. (Also you probably have a different view now of `bash`?)

**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?

**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.
- Typical scheme involves
  - Fixed-size allocation for code and perhaps static data.
  - Two variable-size pieces (“heap” and “stack”) for dynamically allocated data.
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.

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.

Simple Memory Management — Recap

- Contiguous-allocations 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”.

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”.
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).

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.

Paging — Recap

- Recall idea — divide address space and physical memory into fixed-size blocks. Details follow from this basic idea. More complex, but more flexible.
- Things to look at more:
  - Getting acceptable performance.
  - Dealing with large address spaces.
  - Details of using this idea to provide virtual memory.

Performance / Large Address Spaces

- Even with good choice of page size, serious performance implications — page table can still be big, and every memory reference involves page-table access — how to make this feasible/fast?
- Consider several options — compare access time, cost, context-switch time:
  - Keep page table for current process in registers.
  - Keep whole page table in main memory, pointed to by special register.
  - Use multilevel page tables. (More about this later.)
  - Use inverted page tables (one entry per page frame). (More about this later.)
- If page tables are in memory, performance improves with “translation lookaside buffer” (TLB) — special-purpose cache.
Minute Essay

- Reminder — turn in homework.
- 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?