Administrivia

Slide 1

- (None?)

Homework 1 Programming Problem, Revisited

Slide 2

- What most people turned in was not bad — most (but not all!) of you figured out what information to pass to the two system-call functions. (Review briefly.)
- What almost no one got, though, was what happens if `execve` fails!

Slide 3

Homework 1 Programming Problem, Revisited

- The simple shell you wrote in this assignment created a new process for each command, using `fork()`, which creates *a full copy of the calling process*, including its program counter, with the intent of using this process to run the desired command. So now you have two processes, a "parent" and a "child" . . .
- The parent process should then wait for the child to complete (successfully or not) and then continue with the next command.
- Meanwhile, the child process should use `execve` to — what? If it succeeds, it discards the running program (a copy of the parent process) and executes the program from the specified file, terminating when it's done. What if it doesn't succeed? *The existing program keeps running.* "Oops"? Does this explain behavior that — were you puzzled?

Slide 4

Memory Management — Review

- The problem we're solving: Partition physical memory among processes. Two related issues (program relocation and memory protection) both nicely solved by defining "address space" abstraction and implementing with help from hardware (MMU).
- Contiguous-allocation schemes are simple but not very flexible.
- Paging is more flexible but more complex.

Slide 5

Paging — Recap

- Idea — divide both address spaces and memory into fixed-size blocks (“pages” and “page frames”), allow non-contiguous allocation.
- Makes for a much more flexible system but at a cost in complexity — keeping track of a process’s memory requires a “page table” to be used by both hardware (MMU) and software (O/S).

Slide 6

Page Tables — Performance Issues

- One possibility is to keep the whole page table for the current process in registers. Could possibly use general-purpose registers for this but likely would not. Should make for fast translation of addresses, but — is this really feasible for a large table? and what about context switches?
- Another possibility is to keep the process table in memory and just have one register (probably a special-purpose one) point to it. Cost/benefit tradeoffs here seem like the opposite of the first scheme, no?

  The big downside is slow lookup. Can be mitigated with a “translation lookaside buffer” (TLB) — special-purpose cache.

Slide 7

Paging — Feasibility Issues

- Clearly page tables can be big, if we want them all to be the same size (probably) and big enough to represent the system's maximum address space (also probably).
- How to make this feasible? some possibilities, based on the observation that the number of valid page table entries (ones that point to a page frame) is manageable (in contrast to the number of total potential page table entries).

Slide 8

Multi-Level Page Tables

- Idea here is make page tables hierarchical in a sense:
- Each entry in the top-level table represents a range of pages. If no valid pages in that range, entry is "invalid"; else it points to a lower-level table. Only lowest-level tables reference actual page frames.
- In principle, can have arbitrarily many levels, though in practice it depends on what MMU allows.
- Lookup is slower than with a single level (think about why), but again the TLB idea should help.

Slide 9

Inverted Page Tables

- Idea here is to map not from page number to page frame number but the other way around.
- So, in this scheme there's one combined table (rather than one per process), indexed by *page frame number*, with entries containing a process ID and a page number.
- Seems like then lookups would be quite slow — potentially have to search the whole table — but a clever implementation could/would have some way to make it fast.
- Potentially more difficult to implement efficiently, so at one time not used much. Coming back with 64-bit addressing?

Slide 10

Paging and Virtual Memory

- Idea — if we don't have room for all pages of all processes in main memory, keep some on disk ("pretend we have more memory than we really do").
- Or a simpler view: All address spaces live in secondary memory / swap space / backing store, and we "page in" as needed (demand paging).
- (Aside: Why are we even bothering? Can't the processor(s) access disk? Yes, but . . .)
- Making this work requires help from both hardware (MMU) and software (operating system).

Page Fault Interrupts

- We said MMU should generate a “page fault” interrupt for a page that’s not present in real memory. What happens then? It’s an interrupt, so . . .
- Control goes to an interrupt handler. What should it do? (Are there different possibilities for what caused the page faults?)

Slide 11

Page Fault Interrupts, Continued

- One possible cause — an address that’s not valid. You know (sort of) what happens then . . .
- Another cause — an address that’s valid, but the page is on disk rather than in real memory. So — do I/O to read it in. Where to put it? If there’s a free page frame, choice is easy. What if there’s not?

Slide 12

Slide 13

Finding A Free Frame — Page Replacement Algorithms

- Processing a page fault can involve finding a free page frame. Would be easy if the current set of processes aren't taking up all of main memory, but what if they are? Must steal a page frame from someone. How to choose one?
- Several ways to make choice (as with CPU scheduling) — "page replacement algorithms".
- "Good" algorithms are those that result in few page faults. (What happens if there are many page faults?)
- Choice usually constrained by what MMU provides (though that is influenced by what would help O/S designers).
- Many choices (no surprise, right?) . . . (To be continued.)

Slide 14

Minute Essay

- If you even remember doing Homework 1 — did you notice the somewhat-strange behavior when a command wasn't found?