Slide 1

Administrivia

- Reminder: Reading Quiz 3 due today. (Accepted without penalty through Friday.)
- Next reading quiz coming soon, and possibly one more set of homework, about CPU scheduling. I'll send mail and allow at least a week.
- Like last week, I intend to record a make-up lecture to try to compensate for the missed class Monday, I hope tomorrow rather than late Friday!
- We never decided about a midterm, but: I'm inclined not to do one. Any objections?

Slide 2

Minute Essay From Last Lecture

- Everyone who has responded so far says they learned from the section on virtualizing the CPU.
- Most (but not al!) had heard of Professor Dijkstra.

Slide 3

Virtualizing Memory / Memory Management

- In the discussion of virtualizing the CPU, we mentioned that memory is space-shared, with every process having an “address space”.
- As with processes, basic ideas not so very difficult, but details can be complicated.

Slide 4

Memory Management — Early Days

- Very early on, computers ran one program at a time, so no need to share memory among programs / processes..
- O/S occupied part of memory (at high or low addresses or some of each), current process the rest. Always loaded at the same address, so program could use real memory addresses as absolute addresses (remember those from Computer Design?).

Slide 5

Memory Management — Next Steps

- Recall from overview of history:: Very early advance was “multiprogramming”. How to share memory if multiple applications “open”??
- Simplest way was to basically time-share! keep same model as with one-at-a-time, except swap contents of memory on context switches.
- Simple, but not practical unless memory is small and switches infrequent. For time-sharing, latter is clearly(?) not workable, no?
- So, space-share memory.

Slide 6

Space-Sharing Memory

- Simplest scheme is to just assign each process a contiguous chunk, and that works and was used in some early mainframe systesm. But there are problems . . .
- First, probably we’d also like to protect processes from either other, and this doesn’t do that.
- Second, absolute addresses within the program (e.g., to static data, or for jumps) depend on where it’s loaded in memory, so we’ll need to patch those when starting. Can be done, but.
- Is there another way?

Slide 7

Address Spaces — Virtualizing the CPU

- Instead, define "address space" abstraction:

  Every process has available to it the same large range of addresses.
- The challenge then is to map this onto real memory (one big flat space, space-shared): Provide a nice abstraction, including protection/isolation, and do it efficiently.
- At the heart of it — distinction between "virtual address" (relative to address space — same range for all processes) and physical address (relative to physical memory — distinct).

Slide 8

Layout of Address Space

- Typically divide memory into regions:
- One (or more) for code and "static" / fixed data. Usually at low addresses.
- A "stack" region that can grow as needed. Usually at high addresses, grows toward lower ones. Used for procedure calls and local variables.
- A "heap" region that can also grow as needed. Usually starts just after code / fixed data and grows toward higher addresses. Used for dynamically-allocated variables.

Slide 9

Memory API — User Level

- Stack used for "automatic" variables (local to function); managed as part of procedure-call mechanism (as discussed in Computer Design).
- Heap used for dynamically-allocated variables: `malloc()` and `free()` in C, `new` and `delete` in C++, etc. C (and to some extent C++) put burden of freeing unused storage on programmers; more-modern languages manage it for you, using "garbage collection".

  The C way — as you may remember from Low-Level — is full of pitfalls!

Slide 10

Memory Allocation — System Level

- `malloc()` and `free()` might seem like they'd be system calls, but no! entirely user-level library code, managing memory within that allocated by the O/S to the process.
- System call to get more memory from O/S — `brk`, `sbrk`. Also `mmap`.

Slide 11

Quibbles!

- Casting of value from `malloc()` controversial in some circles. Required in C++; optional in C, and there are those that point out that doing it can mask a failure to include the proper `#include`.
- "Try it out" is not bad advice in general, but in the context of C, I say not so much — some things left up to implementation (e.g., sizes of data types), so results on one system may not apply to all. Just sayin'?

Slide 12

Minute Essay

- Questions?