Slide 1

Administrivia

- Reminder: Homework 5 written problems due Wednesday. (Or should it be next Monday?) Programming problems scaled back just a bit.

Slide 2

Minute Essay From Last Lecture

- Most people pretty much got the point, which was . . .

Slide 3

“Thrashing”

- Recall the notion of a process’s “working set” — portion of its address space currently in use.
- Q: What happens if the combined sizes of all active processes’ working sets is too big for RAM?
- A: Pretty much what the sysadmins in my minute-essay story observed — system will spend so much time paging it can’t do much else.

Slide 4

Memory Protection, Revisited

- Paging provides one form of memory protection: If a given page in memory isn’t mapped to some page in a process’s address space via its page table, the process can’t access the page at all.
- But that’s “all or nothing”, and sometimes it would be useful to have more control. Some MMU hardware supports page table entries that in addition to R and M bits have . . .
- A “read-only” bit that’s what its name suggests. So for example there might be a page that’s accessible (for reading) to all processes but is writeable only by the O/S.
- An “execution allowed” bit that means it’s okay for the processor to fetch instructions from this page. Very useful in defending against classic buffer-overflow attacks (by not setting this bit for stack pages)!

Slide 5

Memory Management in Windows

- Apparently very complex, but basic idea is paging.
- Intraprocess memory management is in terms of code regions (some shared — DLLs), data regions, stack, and area for O/S. “Virtual Address Descriptor” for each contiguous group of pages tracks location on disk, etc.
- Memory-mapped files can make I/O faster and allow processes to (in effect) share memory.
- Demand-paged, with six (!) background threads that try to maintain a store of free page frames. Page replacement algorithm is based on idea of working set.

Slide 6

Memory Management in UNIX/Linux

- Very early UNIX used contiguous-allocation or segmentation with swapping. Later versions use paging. Linux uses multi-level page tables; details depend on architecture (e.g., three levels for Alpha, two for Pentium).
- Intraprocess memory management is in terms of text (code) segment, data segment, and stack segment. Linux reserves part of address space for O/S. For each contiguous group of pages, “vm_area_struct” tracks location on disk, etc.
- Memory-mapped files can make I/O faster and allow processes to (in effect) share memory.
- Demand-paged, with background process (“page daemon”) that tries to maintain a store of free page frames. Page replacement algorithms are mostly variants of clock algorithm.

Slide 7

Page Faults — A Little More

- Last time I said if an address is invalid the process referencing the page is terminated? Not strictly true: It's notified that the address was invalid and can then take appropriate action — which might well be terminating.

  ("Notified"? In UNIX/Linux, there's a `signal` mechanism, about which more shortly. In Windows, I'm not sure, but apparently it's rather different.)
- One student asked in class about what happens if the handler for page-fault interrupts is itself interrupted by a context switch. Clearly(?) this could lead to chaos if more than one process at a time is trying to obtain a free page frame. I'm not clear on details, but some things I learned in trying to find out . . .

Slide 8

Interrupt Handlers in Linux

- Problem: Interrupt handlers normally execute with interrupts disabled. This means they should be very fast and should not do anything that would cause them to wait. But sometimes completely processing an interrupt requires non-trivial work.
- Linux solution: Split interrupt-handling code into "top half" and "bottom half".
- "Top half" is the real interrupt handler. Executes with interrupts disabled. If significant processing, or anything involving a wait, is needed, schedules "bottom half".
- "Bottom half" can do more, including waits. Several mechanisms for scheduling these, differing in "context" (interrupt, kernel, process) and whether waits are allowed.

Slide 9

Signals in UNIX/Linux

- Signals are a limited form of IPC: A process can signal another process or itself. Signal is itself just an integer value. POSIX defines meanings for several (`man 7 signal` for a complete list).
- For each defined signal, there's a default action (ignore, stop/suspend process, terminate, with or without "core dump"). Or a process can install its own "signal handler".
- Somewhat akin to exception processing in languages that support that, though not entirely.

Slide 10

Memory-Mapped I/O in Linux

- System calls `mmap`, etc., allow whole or partial files to be "mapped" to memory. Map can be private to process (essentially a copy of the file, with changes not saved back) or shared among processes.
- Actual file reads happen only as locations are referenced, using more or less the same mechanism as paging. Actual file writes happen only with shared maps, either as pages are swapped in and out of memory or via `msync` system call.
- (Example program.)

Slide 11

Linking, Revisited

- Traditional method of getting from source code to something the processor can execute involves compiling (to object code) and linking.
- Linking combines object code and (references to) libraries to produce an executable.
- Libraries can be static (code merged into executable at link time) or dynamic (code loaded at runtime, potentially shared among processes).

Slide 12

Libraries in Linux

- You may remember that (sometimes?) when you call math-library functions in C you have to compile with the extra flag `-lm`? Actually a flag to the linker `ld`. What it means . . .
- `-l`*foobar* tells the linker to try to find functions in library file `lib`*foobar*`.a` (for static linking) or `lib`*foobar*`.so` (for dynamic linking — “shared library”).
- Somewhat elaborate scheme for naming shared libraries allows multiple versions to coexist. Programs that use them can reference latest version (default) or specify particular version.
- References to functions in shared libraries resolved when program is loaded into memory. Can also dynamically load functions at runtime. Both depend on system being able to find shared libraries.
- Standard places to find library code, or you can explicitly specify alternate places.

Slide 13

Libraries in Linux, Continued

- Creating a static library is relatively straightforward:

  Compile code as usual and then use `ar` to combine object code files into library.

- Creating a shared library is less so:

  Compile code with flag to generate “position-independent code” (why? to avoid “relocation problem” previously discussed).

  Generate shared library and set up symbolic links following naming conventions (in which a library has a “real name”, an “soname”, and a name by which the linker normally finds it).

  At runtime, must be sure system knows where to find library. Either “hardcode” in executable or use environment variable `LD_LIBRARY_PATH`.

- (Example.)

Slide 14

Minute Essay

- Questions? otherwise just sign in.