Slide 1

Administrivia

- (By e-mail.)

Slide 2

Complete Virtual Memory Systems

- Previous chapters talked about parts of virtualizing memory — big picture, many details.
- Putting it all together might be another matter, so two case studies.

Slide 3

Case Study: VAX/VMS

- One is from O/S for DEC's VAX architecture, from 1970s / early 1980s. O/S itself may not be in use, but many ideas are. (And even if not: Times change but sometimes ideas fall out of favor and then come back.)
- DEC major player in the days of "minicomputers" — multi-user computers smaller than mainframes. In those days every company that who made hardware also made an O/S for it (or more than one, for IBM anyway). Meant buying new hardware was more complicated than now.

Slide 4

VAX/VMS — Architecture

- VAX-11 provided 32-bit virtual addresses. Page/segment hybrid, with relatively small page size of 512 bytes (2^9) and 2 bits for segment number.
- Lower half of address space ("process space") unique to each process. Within that, half used for code and heap, other half for stack.
- Upper half of address space ("system space") used for O/S code and data, protected (of course) but shared by all processes.

Slide 5

VAX/VMS — Managing Page Tables

- Small pages plus large address spaces means big page tables — trouble. What to do?
- First, allow page tables to be just as big as needed — size of code/heap segment plus size of stack segment.
- Also, keep page tables in kernel-managed memory, where they can be swapped out to disk if need be. Complicates address translation.

Slide 6

VAX/VMS — More About Address Space

- Figure 23.1 shows full layout of address space.
- No surprises in process space (except maybe leaving page 0 unused).
- System space . . . Idea is to make some kind of communication between O/S and user programs easier — in textbook's words, makes kernel almost like a library to user programs. Should seem plausible?

  Of course, access to O/S data controlled by memory protection, so user programs can't do things they shouldn't.

  Textbook says this idea is widely used, and it matches what I know about mainframes I worked on.

Slide 7

VAX/VMS — Page Replacement

- PTE has many of usual fields — but no reference bit(!). (But as it turns out, apparently you can emulate it. Details in textbook.)
- Architecture is from a time when memory was a scarce resource. So on multi-user system, concern about programs using more than "their share" of memory. Many page-replacement algorithms don't try to balance memory use equally among processes or users.
- What to do . . .

Slide 8

VAX/VMS — Segmented FIFO Replacement Policy

- Each process has a "resident set size" (RSS) — maximum number of pages in memory. Each process has a FIFO queue of pages it's using.
- Simple and requires no support from hardware, but doesn't perform very well, so:
- O/S also keeps two global "second-chance" lists, one for clean (unmodified) pages, one for dirty pages. Pages can be reclaimed from one of these if needed (avoiding one of the worst aspects of FIFO); processes needing (and allowed to have?) more memory pull free frames from the clean-pages list.

  Effect is a sort of hybrid of FIFO and LRU.

Slide 9

VAX/VMS — More Optimizations

- Small pages also mean swapping I/O is inefficient — disks faster at transferring data in big chunks. So VAX/VMS writes “clusters” of pages from global dirty-pages list rather than single pages.
- For security, pages newly added to page’s address space must be filled with zeros. But what if page is never used? Hence “demand zeroing” — initially mark page inaccessible so first use traps to O/S,
- “Copy-on-write” is somewhat similar: When copying a page from one address space to another, don’t actually copy unless one process changes something, and copy then.
- All ideas used in more-recent systems. (Example: UNIX `fork()` followed by `exec*()` sounds like madness, but doesn’t have to be!)

Slide 10

Case Study: Linux

- Linux interesting in that it runs on a very diverse set of platforms. (And as textbook points out, that requires compromises that mean it may not be optimal for any of them.)
- Focus in this chapter on important aspects of virtual memory in Linux, and on implementation for x86 architecture. (I’m *almost* sorry I don’t teach x86 in CSCI 2321 — but I continue to believe it’s horrible as a first assembler language.)

Slide 11

Linux — More About Address Space

- Figure 23.2 shows layout of address space. (Note in passing that this appears to be 32-bit x86 — but textbook says x86-64 follows same plan but with different split.)
- Note similarity to VAX/VMS address space — user portion, kernel portion (though split is 3G/1G rather than 2G/2G), with kernel portion shared among processes.

  (Picture is somewhat at odds with results of experiment, in which highest address on stack looks like it would be `0x7ffffff` not `0xbffffff`. I admit I don't understand this!)
- Kernel portion further divided into "logical" and "virtual" parts.

Slide 12

Linux — Kernel Part of Address Space (Logical)

- Where most O/S data structures live. Allocate with `kmalloc()`.
- *Cannot* be swapped to disk, and maps directly to lowest-numbered physical addresses (`0xC000 0000` to `0x0000 0000`).
- How is this useful? Some operations — such as ones involving I/O — need contiguous physical memory to work. Further, some ("memory-mapped I/O") require referencing specific physical addresses.

Slide 13

Linux — Kernel Part of Address Space (Virtual)

- Useful for large data structures.
- *Can* be swapped to disk, and need not be contiguous.

Slide 14

Linux — x86 Specifics

- Textbook says x86 architecture provides hardware-managed multi-level page tables.
- Based on a fairly quick Web search, things are more complicated, and 32-bit version of architecture is a segmentation/paging hybrid, though some O/S's don't really make use of the segmentation aspects. As with so much about this architecture, hardware has evolved, but architecture retains traces of its earliest ancestors. (Link to Intel's manual on course Web site under "Links". Interesting reading if you want to know more.)
- x86-64 architecture cleans that up some — paging only, for now 48-bit addresses.
- Standard page size 4K (2^{12}) bytes. Additional page sizes (2M and even 1G — "huge pages") also supported. (Apparently done by merging some levels of the page table, e.g., combining lowest levels into bigger offsets).

Slide 15

Linux — “Page Cache”

- Linux apparently incorporates swapping into a larger strategy for keeping frequently-used data in physical memory, in page-size chunks:
  - Data read from files via `mmap()`.
  - Other data from filesystems (files and metadata).
  - Heap and storage pages (“anonymous memory” — also obtained from `mmap()`).

  (What about code pages? Can be handled via `mmap()`.)
- All kept in “page cache hash table” for quicker lookup.
- Page cache tracks which pages are “dirty” (have been modified); periodically background threads write out dirty pages, to files for file-backed data, swap space for “anonymous data”. Done periodically or when too many pages dirty. “Voo-doo constants”, maybe, so configurable.
- When number of free pages runs low, must evict some pages . . .

Slide 16

Linux — Page Replacement Algorithm

- Modified form of “2Q” (described in referenced paper?).
- Basic idea: LRU effective but can be subverted by common scenarios, such as accessing all of file that fills up all or most of physical memory. What to do?
- Set up two lists: Page goes on *inactive list* on first access, promoted to *active list* on next access. Periodically less-recently used pages moved from active to inactive list. (Both lists ideally kept in LRU order, but impractical, so some approximation (e.g., clock) used.) Pages on inactive list are candidates for replacement.
- Combines best features of LRU with a way to avoid a common bad scenario.

Slide 17

Linux — Buffer Overflows and Security

- From CSCI 1120: C doesn't help you avoid out-of-bounds array accesses or use of invalid pointers. And these can even lead to "security problems".
- How so? Local variables for function typically go on stack, as does return address (where to return from function). Stack grows upward, so local variables at smaller addresses — so possible to overwrite return address.
- At one time, attacker who knew enough about a program could craft non-text input that would put its own code in the input field plus data that would overwrite return address with code that would return not to caller but to attacker code just read in — !!
- Classic paper ("Smashing the Stack for Fun and Profit", referenced under "Links" on course Web site) goes through full details of example. Cool if scary, but . . .

Slide 18

Linux — Buffer Overflows and Security, Continued

- Attacks in classic paper don't work any more, partly because conventions for calling functions in x86 assembler have changed, but also because of explicit changes:
- Some architectures now have "execute" bit on pages; if not set, code from page can't be executed. Idea is for it to be set for code pages, not for stack pages.
- So attackers can't make function return to code they just inserted — but they can make it return somewhere else.
- "Return-oriented programming" does just that, returning to library code in memory. (I haven't read up, but sounds cool if scary!)
- That too can be thwarted, by not always placing parts of address space (e.g., code) at same location ("address space layout randomization").

Linux — Meltdown and Spectre

Slide 19

- Not long ago “Meltdown” and “Spectre’ bugs made the news. They describe ways for attackers to get access to data they shouldn’t be able to access — and at first reading, sounds impossible, based on what you now know about address translation.
- *But* trying for always more speed, chip designers make use of “speculative execution”: E.g., loading from memory is comparatively slow, so start early to load what you think you’ll need, and if that’s wrong then throw result away and try again.
- Sounds good, *but* wrong path not taken may leave data in caches etc., and this can expose data thought to be protected.

Linux — Meltdown and Spectre, Continued

Slide 20

- Sounds like part of the problem is having kernel data in page table for all processes — even if not marked readable, it can be *found*. Could be addressed by going back to putting kernel data in separate address space — at performance cost.
- “Interesting time to be alive” the textbook authors say. Indeed.

Slide 21

Linux — Summary

- Memory management in Linux big and complicated, and many details still murky. (I question authors' claim about details being easy to follow, though general ideas are manageable.)
- Interesting to note that while many details don't exactly match more-general discussion earlier, general ideas and principles should seem familiar.
- In fact in general my guess is that real systems start from known ideas and then combine them and tweak them.

Slide 22

Minute Essay

- Questions?