Slide 1

Administrivia

- Reading quizzes and homework posted. Due week after holiday.
- One more round of quizzes and homeworks, but homeworks will be short.
- Final will be worth 150 points not 200. (Possibly even just 100?)
- Code examples added to course Web site, including synchronization examples.

Slide 2

Page Replacement Algorithms — Recap

- If there are always free frames to bring in pages from disk, no need to make decisions. Not always guaranteed, hence the need to choose.
- Many many ways to choose (no surprise!). Goal is to reduce number of page faults. Often based on observation that recent past predicts near future — notion of “working set”.

Slide 3

“Thrashing”

- Recall 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.
- Unlikely these days, but still possible! (I have a home system where Eclipse seems to have this problem! Runs, but so slowly it’s not usable.)

Slide 4

Modeling Page Replacement Algorithms

- Intuitively obvious that more memory leads to fewer page faults, right? Not always!
- Counterexample — “Belady’s anomaly”, sparked interest in modeling page replacement algorithms.
- Modeling based on simplified version of reality — one process only, known inputs. Can then record “reference string” of pages referenced.
- Given reference string, p.r.a., and number of page frames, we can calculate number of page faults.
- How is this useful? can compare different algorithms, and also determine if a given algorithm is a “stack algorithm” (more memory always means fewer page faults).

Slide 5

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 6

Sidebar: "Smashing the Stack"

- Usual scheme for memory use within a process puts a stack at high addresses, used in function calls (for parameters and return address) and also for local variables. What happens if an attempt is made to store more data in a local-variable array than will fit? (And in C this is all too easy, no?)
- Well, you know from CSCI 1120, no? Whatever is after the array is overwritten . . .

Slide 7

“Smashing the Stack”, Continued

- . . . possibly including the function’s return address!
- This is an example of deliberately “smashing the stack”, and if the input is very carefully crafted non-text, can be used to invoke attacker’s code. (Full details in a very old paper referenced in “Useful links” on course Web site. Uses x86 assembly language but I think is fairly readable even if you don’t know that, and has a useful overview of various things relevant to this course.)
- Relies on being able to transfer control to any memory location user has access to, including writable locations.
- Including “executable” bit in page table can help defeat this!

Slide 8

Paging — Operating System Versus MMU

- Some aspects of paging are dealt with by hardware (MMU) — translation of program addresses to physical addresses, generation of page faults, setting of R and M bits.
- Other aspects need O/S involvement. What/when?

Slide 9

Paging — Operating System Involvement

- Process creation requires setting up page tables and other data structures. Process termination requires freeing them.
- Context switches require changing whatever the MMU uses to find the current page table.
- And of course it's the operating system that handles page faults!
- Some details . . .

Slide 10

Processing Memory References — MMU

- Does cache contain data for (virtual) address? If so, use it.
- If not, does TLB contain matching page table entry? If so, generate physical address and send to memory bus.
- If not, does page table entry (in memory) say page is present? If so, put PTE in TLB and continue as above.
- If not, generate page fault interrupt. Transfers control to interrupt handler.

Slide 11

Processing Memory References — Page Fault Interrupt Handler

- Is page on disk or invalid (based on entry in process table, or other O/S data structure)? If invalid, error — signal process (possibly terminate).
- Is there a free page frame? If not, choose one to steal (using page replacement algorithm). If it needs to be saved to disk, start I/O to do that. Update process table, PTE, etc., for "victim" process. Block process until I/O done.
- Start I/O to bring needed page in from swap space (or zero out new page). If I/O needed, block process until done.
- Update process table, etc., for process that caused the page fault, and restart at instruction that generated page fault.

Slide 12

Processing Memory References — Details Still To Fill In

- How to keep track of pages on disk.
- How to keep track of which page frames are free.
- How to "schedule I/O" (but that's later).

Slide 13

Keeping Track of Pages on Disk

- To implement virtual memory, need space on disk to keep pages not in main memory. Reserve part of disk for this purpose ("swap space"); (conceptually) divide it into page-sized chunks. How to keep track of which pages are where? (Figure 3-28 in textbook.)
- One approach — give each process a contiguous piece of swap space. Advantages/disadvantages?
- Another approach — assign chunks of swap space individually. Advantages/disadvantages?
- Either way — each process must know where "its" pages are (via page table and some other data structure), operating system must know where free slots are (in memory and in swap space).
- ("Swap space"? Region on disk for holding out-of-memory pages. Can be flat file or separate partition. A.k.a. "backing store".)

Slide 14

Sharing Pages

- (Pause first to try to combat Zoom fatigue . . .)
- Shared pages can be useful, but can also present problems.
- Multiple processes running the same program is relatively easy (why?) but has one potential downside (what?)
- UNIX `fork` system call is — interesting? — in this context. POSIX definition says that child process's address space is basically a copy of the parent's address space. What's the easy-to-implement way to do this? What downside does that have in current systems? Is there a way to reduce its impact? And why duplicate in the first place?

Slide 15

Sharing Pages and `fork`

- Duplicating pages is easy but inefficient, especially if the child process is going to call `execve` or something similar right away. Some systems use “copy-on-write” to improve efficiency.
- Why did the people who designed UNIX require this duplication . . . Possibly because it makes some things easy (such as setting up parent/child pipes) and wasn’t very costly when designed. Windows’s system call for creating processes takes a different approach. Maybe that’s better!

Slide 16

Sharing Pages, Continued

- One use for shared pages is multiple processes running the same program.
- What about sharing code at a level below whole programs (UNIX “shared libraries”, Windows DLLs)?

Slide 17

Shared Libraries

- One attraction is somewhat obvious: If code for library functions (e.g., `printf`) is statically linked into every program that uses it, programs need more memory — seems wasteful if processes can share one copy of code in memory.
- Another attraction is that library code can be updated independently of programs that use it. (But is there a downside to that?)
- How to make this happen . . .

Slide 18

Shared Libraries, Continued

- A good-and-bad aspect is that if the shared code is updated, all programs that use it are affected.
- How to make this happen . . . At link time, programs get "stub" versions of functions. References to real versions resolved at load time.
- Resolving references to shared code at load time — finer-grained version of "relocation problem", no? and fixable by making sure library contains only "position-independent code".

Slide 19

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 20

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 on course Web site.)

Memory-Mapped File I/O

- Worth mentioning here that some systems also provide a mechanism (e.g., via system calls) to allow “mapping” whole files into/from memory. Reading/writing file is done using paging mechanism.
- If there’s enough memory, this could improve performance.

Slide 21

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 on course Web site.)

Slide 22

Slide 23

One More Memory Management Strategy — Segmentation

- Idea — make program address "two-dimensional" / separate address space into logical parts. So a virtual address has two parts, a segment and an offset.
- To map virtual address to memory location, need "segment table", like page table except each entry also requires a length/limit field. (So this is like a cross between contiguous-allocation schemes and paging.)

Slide 24

Segmentation, Continued

- Benefits?
  - Nice abstraction; nice way to share memory.
  - Flexible use of memory — can have many areas that grow/shrink as required, not just heap and stack — especially if we combine with paging.
- Drawbacks?
  - External fragmentation possible (can offset by also paging).
  - More complex.
  - "Paging" in/out more complex — issues similar to with contiguous-allocation.

Minute Essay

Slide 25

- That wraps up what I have to say about memory management. Anything you really want to know about that I didn't mention?