Slide 1

Administrivia

- Quiz 3 Wednesday?
- Next homework to be on the Web soon (tomorrow I hope). Due a week from Wednesday.

Slide 2

Minute Essay From Last Lecture

- (Almost no one came very close.)
- Not using a disk for paging? (How else . . .)
- Disk is full? (But then how could you even do paging?)
- Keeping everything on disk rather than paging? (But then how do you do loads/stores?)
- Same disk for paging and other stuff? (Semi-plausible.)

Slide 3

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 means fewer page faults).

Slide 4

Page Replacement Algorithms — Recap

- Nice summary in textbook (table at end of section 3.4).
- Tanenbaum says best choices are aging, WSClock.
- Now move on to other issues to consider . . .

Demand Paging Versus Prepaging

- The purest form of paging is “demand paging” — processes are started with no pages in memory, and pages are loaded into memory on demand only.
- An alternative is “prepaging” — try to load pages in advance of demand. How?

Slide 5

Global Versus Local Allocation

- In deciding which page to replace, consider all pages (“global allocation”), or just those that belong to the current process (“local allocation”)?
- Generally, global approach works better, but not all page replacement algorithms can work that way (e.g., WSClock). Hybrid strategy — combine local approach with some way to vary processes’ allocations.

Slide 6

Thrashing and Load Control

- What happens if combined working sets of all processes don't fit into memory? "Thrashing". (See minute essay from last time!)
- What to do? temporarily "swap out" some processes, or other forms of "load control".

Slide 7

One More Design Issue

- Page replacement algorithms as discussed all seem based on the idea that we let memory fill up, and then "steal" page frames as needed. Is that really the best way . . .
- An alternative — background process ("paging daemon") that tries to keep a supply of free page frames, or at least ones that can be stolen without needing to write out their contents. Can use algorithms similar to page replacement algorithms to do 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

Slide 11

Processing Memory References — MMU

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

Slide 12

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 — terminate process.
- Is there a free page frame? If not, choose one to steal. 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 it at instruction that generated page fault.

Sharing Pages

Slide 13

- 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?

Sharing Pages and `fork`

Slide 14

- 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 15

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)? Seems attractive; are there potential problems?

Slide 16

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. (Is there a downside to that?)
- How to make this happen . . . At link time, programs get “stub” versions of functions. References to real versions resolved at load time. Does this remind you of anything? and suggest a possible problem? how to fix?

Shared Libraries, Continued

- Downside of replacing shared libraries — may break applications that call their function. UNIX provides a way around this.
- 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 17

Memory-Mapped File I/O

- Worth mentioning here that some systems also provide a mechanism (e.g., via system calls) to allow reading/writing whole files into/from memory. If there’s enough memory, this could improve performance.
- Example of how this works in Linux — `man` page for `mmap`.

Slide 18

Paging — One More Hardware Issue

- What if page to be replaced is waiting for I/O? probably trouble if we replace it anyway.
- One solution — allow pages to be "locked".
- Another solution — do all I/O to o/s pages, then move to user pages.

Slide 19

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 20

Slide 21

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?
- 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 — processes must know where “their” pages are (via page table and some other data structure), operating system must know where free slots are (in memory and in swap space).

Slide 22

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 23

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.

Slide 24

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 25

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 26

Minute Essay

- Is the material on memory management making sense? Questions? Things you’d like to hear more about?