

Slide 1

Administrivia

- (None.)

Slide 2

Sharing Pages, Continued

- Shared pages can be useful, but can also present problems.
- Examples discussed last time: processes sharing all of “code” part of address space, optimization for `fork` system call.
- Also mentioned: sharing code at a level below whole programs (UNIX “shared libraries”, Windows DLLs)? Seems attractive; are there potential problems?

Shared Libraries

Slide 3

- 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

Slide 4

- 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”.

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 5

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 6

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 7

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 8

Slide 9

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 10

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.

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 11

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 12

Keeping Track of Pages on Disk

Slide 13

- 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).

One More Memory Management Strategy — Segmentation

Slide 14

- 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.)

Segmentation, Continued

Slide 15

- 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.

Memory Management in Windows

Slide 16

- 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.

Memory Management in UNIX/Linux

Slide 17

- 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.

Minute Essay

Slide 18

- At least one early mainframe operating system supported virtual memory (i.e., keeping some pages on disk), but provided only a single address space that all processes shared. How does this compare with giving each process its own address space — with respect to simplicity, efficiency, anything else you can think of? (There appears to be active research into reviving this idea. Hm!)
- Anything about memory management you'd like to hear more about / have clarified?

Minute Essay Answer

- A single address space seems like it might be simpler and more efficient — could have a single page table. Also it would be easier for processes to share memory. But it seems potentially less secure — how to protect one process's memory for another? — and probably would make it very difficult to implement fork. (You may think of other considerations!)

Slide 19