

Slide 1

### Administrivia

- Reminder: Homework 4 due today.
- Homework 5 on the Web. Written problems due next Wednesday, programming problems the following Monday.

Slide 2

### Paging and Virtual Memory — Recap/Review

- Basic idea is fairly simple: If there are more pages in the union of all process's address spaces than will fit into main memory, keep some (we hope the less-active ones) on disk.
- With this addition, page faults now either mean "invalid address" or "page not in memory but on disk". Page-fault interrupt handler must decide which, and if it's the latter, arrange to bring it in. Similar processing if we want to give a process a new page.
- If memory is not full, not too hard, but if it is? "Steal" a frame from its current owner (write contents to disk first if need be). Choice of page to steal determined by "page replacement algorithm".

Many such algorithms possible, as discussed last time. (Nice summary in textbook as Figure 3-21.)

### 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 3

### 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 4

Slide 5

### 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 6

### 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.

### 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 7

### 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 — 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 8

### Paging — Other Design Issues/Choices

Slide 9

- Demand paging versus prepaging.
- Global versus local allocation.
- “Paging daemon” that tries to keep a supply of free page frames.
- What to do if page to be replaced is waiting for I/O — probably trouble if we replace it anyway, since the pending I/O, when it completes, may write to a physical address. Solutions include “locking” pages, or doing all I/O to O/S pages and then moving data to user pages.

### Modeling Page Replacement Algorithms

Slide 10

- 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. (One of the programming problems will ask you to do this.)
- 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).

## Sharing Pages

Slide 11

- 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 12

- 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!

### 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 13

### 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 14

### Shared Libraries, Continued

Slide 15

- 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”.
- (Some details of how this plays out in Linux next time?)

### Memory-Mapped File I/O

Slide 16

- 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.
- (Example in Linux later?)



### One More Memory Management Strategy — Segmentation

Slide 17

- 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 18

- 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 19

- Another story from long ago: Once upon a time, a mainframe computer was running very slowly. The sysadmins were puzzled, until one of them noticed that one of the disk drives seemed to be very busy and asked “which disk are you using for paging?” The answer made everyone say “aha!” What was wrong (to make the system so slow)?
- This just about wraps up what I have to say about memory management, though Monday I'm planning to show some examples illustrating use of shared libraries and memory-mapped I/O in Linux. Anything else you want to hear about?

### Minute Essay Answer

Slide 20

- The disk being used for paging was the one that was very busy. So, mostly likely the system was spending so much time paging (“thrashing”) that it wasn’t able to get anything else done. Usually this means that the system isn’t able to keep up with active processes’ demand for memory.  
(Memory sizes have increased to a point where this isn’t as likely as it once was. Several years ago we did run into problems with the machines in one of the classrooms trying to run both Eclipse and a Lewis simulation, and then more recently with some of them attempting to run a background program that asked for more memory than its author intended.)