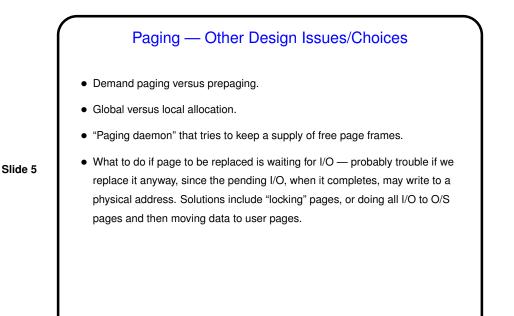


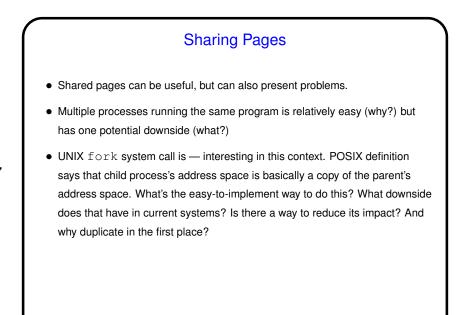
Slide 4

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. (Slides from last time — revised a bit.)

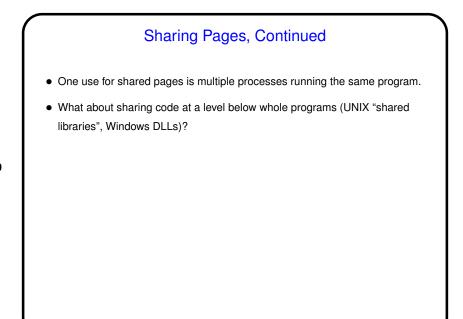


Modeling Page Replacement Algorithms

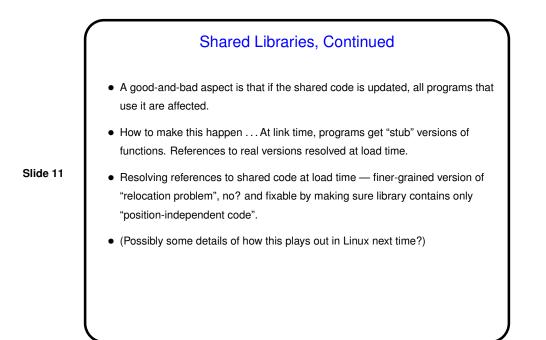
- Intuitively obvious that more memory leads to fewer page faults, right? Not always!
- Counterexample "Belady's anomaly", sparked interest in modeling page replacement algorithms.
- Slide 6
- Modeling based on simplified version of reality one process only, known inputs. Can then record "reference string" 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).

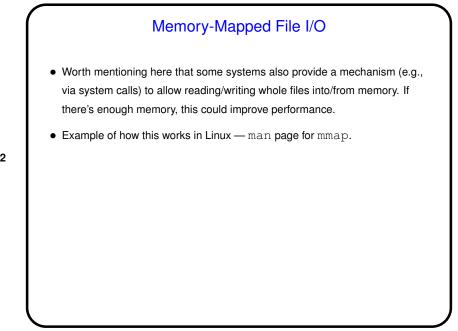


Sharing Pages and fork Duplicating pages is easy but inefficient, especially if the child process is going to call execve or something similar right away. Some systems use "copy-on-write" to improve efficiency. Why did the people who designed UNIX require this duplication ... Possibly because it makes 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!



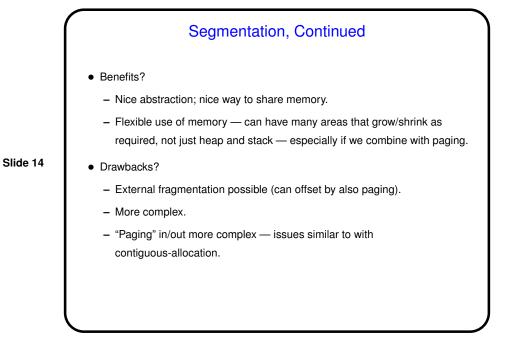
Shared Libraries One attraction is somewhat obvious — if code for library functions (e.g., printf) is statically linked into every program that uses it, programs need more memory — seems wasteful if processes can share one copy of code in memory. Another attraction is that library code can be updated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







- 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.)





• I'm planning one more lecture on memory management, to include some details about shared libraries in UNIXworld and an overview of how memory management is done in a few real-world systems. Anything else you'd like to hear about?