Swapping

- Segmented paging allows us to have non-contiguous allocations
- But it still limits us to the size of physical RAM
- How can we avoid that?
- By keeping some segments somewhere else
- Where?
- Maybe on a disk

Swapping Segments To Disk

- An obvious strategy to increase effective memory size
- When a process yields, copy its segments to disk
- When it is scheduled, copy them back
- Paged segments mean we need not put any of this data in the same place as before yielding
- Each process could see a memory space as big as the total amount of RAM

Downsides To Segment Swapping

- If we actually move everything out, the costs of a context switch are <u>very</u> high
  - Copy all of RAM out to disk
  - And then copy other stuff from disk to RAM
  - Before the newly scheduled process can do anything
- We're still limiting processes to the amount of RAM we actually have

Demand Paging

- What is paging?
  - What problem does it solve?
  - How does it do so?
- Locality of reference
- Page faults and performance issues

What Is Demand Paging?

- A process doesn't actually need all its pages in memory to run
- It only needs those it actually references
- So, why bother loading up all the pages when a process is scheduled to run?
- And, perhaps, why get rid of all of a process' pages when it yields?
- Move pages onto and off of disk "on demand"

How To Make Demand Paging Work

- The MMU must support “not present” pages
  - Generates a fault/trap when they are referenced
  - OS can bring in page and retry the faulted reference
- Entire process needn’t be in memory to start running
  - Start each process with a subset of its pages
  - Load additional pages as program demands them
- The big challenge will be performance

Achieving Good Performance for Demand Paging

- Demand paging will perform poorly if most memory references require disk access
  - Worse than bringing in all the pages at once, maybe
- So we need to be sure most don't
- How?
- By ensuring that the page holding the next memory reference is already there
  - Almost always

Demand Paging and Locality of Reference

- How can we predict which pages we need in memory?
  - Since they'd better be there when we ask
- Primarily, rely on *locality of reference*
  - Put simply, the next address you ask for is likely to be close to the last address you asked for
- Do programs typically display locality of reference?
- Fortunately, yes!

Instruction Locality of Reference

- Code usually executes sequences of consecutive instructions
- Most branches tend to be relatively short distances (into code in the same routine)
- Even routine calls tend to come in clusters
  - E.g., we'll do a bunch of file I/O, then we'll do a bunch of list operations

Stack Locality of Reference

- Obvious locality here
- We typically need access to things in the current stack frame
  - Either the most recently created one
  - Or one we just returned to from another call
- Since the frames usually aren't huge, obvious locality here

Heap Data Locality of Reference

- Many data references to recently allocated buffers or structures
  - E.g., creating or processing a message
- Also common to do a great deal of processing using one data structure
  - Before using another
- But more chances for non-local behavior than with code or the stack

Page Faults

- Page tables no longer necessarily contain pointers to pages of RAM
- In some cases, the pages are not in RAM, at the moment
  - They're out on disk
- When a program requests an address from such a page, what do we do?
- Generate a *page fault*
  - Which is intended to tell the system to go get it

Handling a Page Fault

- Initialize page table entries to “not present”
- CPU faults if “not present” page is referenced
  - Fault enters kernel, just like any other trap
  - Forwarded to page fault handler
  - Determine which page is required, where it resides
  - Schedule I/O to fetch it, then block the process
  - Make page table point at newly read-in page
  - Back up user-mode PC to retry failed instruction
  - Return to user-mode and try again
- Meanwhile, other processes can run

Pages and Secondary Storage

- When not in memory, pages live on secondary storage
  - Typically a disk
  - In an area called "swap space"
- How do we manage swap space?
  - As a pool of variable length partitions?
    - Allocate a contiguous region for each process
  - As a random collection of pages?
    - Just use a bit-map to keep track of which are free
  - As a file system?
    - Create a file per process (or segment)
    - File offsets correspond to virtual address offsets

Swap Space and Segments

- Should the swap space be organized somehow by segments?
- A paging MMU eliminates need to store consecutive virtual pages in contiguous physical pages
- But locality of reference suggests pages in segments are likely to be used together
- Disk pays a big performance penalty particularly for spreading operations across multiple cylinders
- Well-clustered allocation may lead to more efficient I/O when we are moving pages in and out
- Organizing swap by segments can help

Demand Paging Performance

- Page faults may result in shorter time slices
  - Standard overhead/response-time tradeoff
- Overhead (fault handling, paging-in and out)
  - Process is blocked while we are reading in pages
  - Delaying execution and consuming cycles
  - Directly proportional to the number of page faults
- Key is having the "right" pages in memory
  - Right pages -> few faults, little paging activity
  - Wrong pages -> many faults, much paging
- We can't control what pages we read in
  - Key to performance is choosing which to kick out