LECTURE 2.3
VIRTUAL MEMORY

COP4600

Dr. Matthew Gerber

3/14/2015

VIRTUAL MEMORY (CH. 9)

Programs, Processes and Memory (9.1)

- In general, we have assumed that programs need to become processes *fully* present in memory to execute
- But programs – and even processes – don’t necessarily use all parts of themselves all the time
  - Error handling routines
  - Full extents of arrays
  - Seldom-used features
- If we could execute programs that were only *partially* in memory…
  - Address spaces could be essentially arbitrary in size
  - With little-used routines not in memory, we could load more programs at once
  - Not having to load entire programs, load time would be faster

Virtual Memory (9.1)

- Combine paging with swapping
- Extend the page table concept to be able to refer to areas of disk as well as areas of memory
  - We will discuss this more shortly
- **Swap *individual pages* to disk rather than processes**
- Bring pages back into physical memory when they need to be worked with
- Processes can now be in and out of memory in any part – code, data, and other pages can be either active in memory or inactive on disk in any combination
- No reason to keep inactive parts of a process in memory
  - …but remember, nothing is free…

Virtual Addressing (9.1)

- A *virtual address* under virtual memory is a lot like a logical address under paging without virtual memory – conceptually, it’s the page table that’s different
- One way it’s different is the ability to refer to the disk – we’ll get to that in a bit
- The other way – and the one that changes addressing itself – is that we now explicitly use *sparse* addressing
  - We hinted at this before but virtual memory brings the concept to the foreground

Virtual Addressing (9.1)

- We only map addresses that processes actually use, but…
- …instead of pages that aren't in the page table being presumed to be unallocated, they're now simply space for the process to grow
- The heap grows up from the bottom of the address space, the stack grows down from the top
- **Each process now has its own virtual address space that is *completely arbitrary in size!***
- This technique provides a huge number of benefits

Working with Virtual Memory (9.1, 9.3)

- With the truly arbitrary size of the address space, virtual memory unifies and enables a few concepts we’ve talked about before
  - System libraries become even easier to share between processes
    - Just stick the whole library in the process address space
  - Sharing memory between processes becomes trivial
    - Just map some of the processes’ pages to the same frames
  - We can fork processes’ data *without* copying the entire address space
    - Mark the data pages of the parent and child processes as *copy-on-write*
    - Whenever either the parent or child writes to one of them, truly copy only *that* page
  - Also, we can fork processes without ever copying code at all
- And we haven’t even gotten to the actual swapping yet

Demand Paging (9.2)

- Speaking of swapping…
- When we start a process, we can (very) safely assume there's no need to actually load the whole program into physical memory
  - Under *demand paging* we load only pages that are actually *going to be used* into physical memory
  - …or pages that we're pretty sure will be used, at least
- The page table's valid/invalid bit concept is extended
  - If a page is invalid, *either* it is an invalid memory address *or* it is on disk
  - Either way, if an invalid page is accessed, we raise an interrupt and let the OS figure it out
  - Can you guess what this is called?

Page Faults and Swapping (9.2.1)

- A **page fault** is generated whenever an invalid page is accessed
- The program is interrupted and the OS figures out whether the reference was invalid or to a page that's swapped out to disk
- If the reference was invalid, the program was terminated
- If the reference was to a page that was swapped to disk, the page needs to be retrieved *from* disk before the program can continue
- We find a free frame and load the page into it from disk
- We resume the process when the page is finished loading
- We need hardware support for this – but it's actually pretty much the same as hardware support for paging

Demand Paging Performance (9.2.1-9.2.2)

- Theoretically every memory access could result in several new pages needing to be swapped in
- This isn't actually going to happen because of *locality of reference* (more on this later)
- Demand paging is still *really, really slow* when it actually has to be done
  - Memory access is measured in a moderate number of *nanoseconds*
  - Disk access is measured in a small number of *milliseconds*
    - SSDs help this, but a lot less than we'd hope – too much bus overhead
  - Disk access is *literally thousands of times slower* than memory access
- **Page faults need to be *rare***

Oversubscription and Page Replacement (9.4)

- Up until now we've assumed we have enough frames of physical memory to arbitrarily swap pages in at will
- What happens when we need to swap pages back *out*?
- Eventually, we're going to look for a free frame and there's not going to be one
- That means we need to choose a page to remove from a frame and swap back out to disk
- That means we need to choose a *victim page* – and that means an algorithm

The Working Set and Thrashing (9.6)

This is presented in the text as a swapping strategy, but we need it as a general observation first and will examine it in more detail later.

- Without loss of generality, each process has a *working set* of pages that it needs to use in a given short time
- This working set will be somewhere between one page and the process’s total number of pages
- Consider the combined working sets of all active processes
- If there are not enough physical frames to hold them all, then the system will undergo *thrashing*: a very high proportion of memory accesses will begin to result in page faults
- Thrashing *immediately and invariably* results in unacceptable system performance

Virtual Memory Algorithms

- When we have oversubscribed memory and need to choose a page to swap out, we need to figure out how to select victim pages that will not be needed again soon
  - This is done by a *page replacement algorithm*
- The other side of this is fairly and effectively allocating frames to processes
  - This is done by a *frame allocation algorithm*
- Swapping entire processes still has its place: if the choice is between swapping and thrashing, we may be able to at least keep the system running by swapping out a process
  - This is done by a *thrashing prevention algorithm*

NEXT TIME: MEMORY ADDRESSING