

# **OPERATING SYSTEMS FOUNDATIONS**

## **MSIT 5170**

University of Massachusetts Lowell  
Department of Computer Science  
Spring Semester 2020

### **Week #7 Lesson**

1. The material in this lesson follows the reading assignment for ch 3.5 through the end of chapter 3 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 3 will focus on issues of memory management and address space.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some of the dimensions of memory management.
5. During this week you must complete the “First Half Exam”, as described in the weekly agenda. The exam will be available to take as of Wednesday morning, March 4, but I advise you to wait at least until the Wednesday chat session is over before you consider taking the exam.

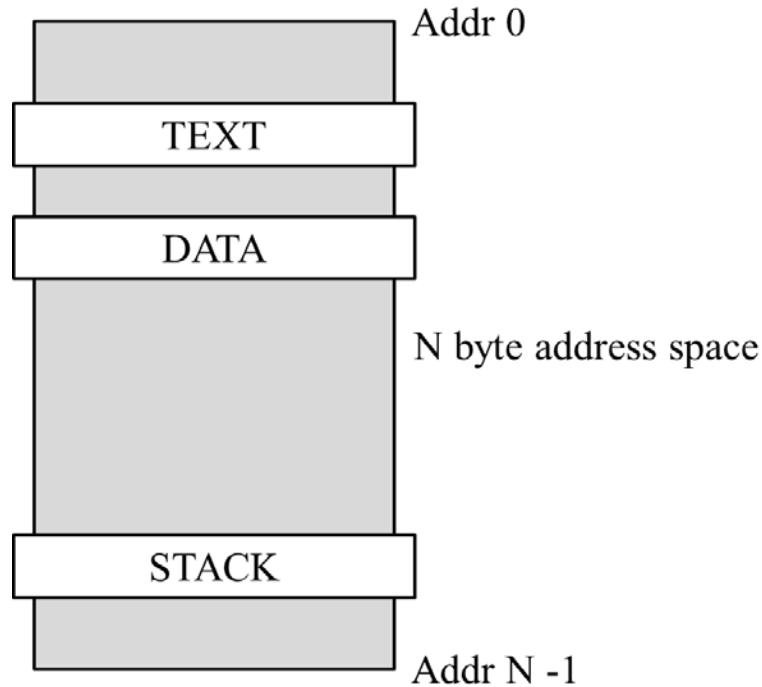
#### **Design Issues for Paging Systems:**

Thus far, the material in chapter 3 has considered the need for memory management, discussed some methods for achieving this with a physically contiguous address space, and considered a more practical approach using a contiguous virtual address space with an underlying physical paging mechanism. Some sort of paging mechanism is supported by most CPU implementations, and all but the simplest embedded operating systems typically exploit these hardware resources. Most well known operating systems such as Linux, OSX, Windows, VMS, etc., deploy a virtual, paged memory management implementation.

Page replacement algorithms play a central role in any virtual memory implementation, as we've seen, but there's more to consider on the backup side of replacement. If a page is selected for replacement, and that page has been modified since it was last paged in (the VM hardware support provides a hardware update to a bit in the mapping PTE called the dirty or modified bit, that is set anytime a store operation is performed on the PTE's target physical page), then the page-purger must first write out the modified version of the page to a safe disk location. This is the most expensive part of page replacement, because it requires an IO operation. The page-purger would much rather select a page for replacement that has not been modified since it was last paged in, since such a page can simply be added to the free list without need to write its content anywhere. There are times, however, when most of the likely replacement victims are all dirty, and there is little choice but to page-out modified pages.

To understand the mechanisms for paging out modified pages, we have to revisit our address space model. We know that a process address space must be populated by a minimum of 3 memory object, called the text, data and stack objects. Like any memory object, each of these

objects must occupy a contiguous range of virtual addresses, and are never allowed to overlap in the virtual address space. Recall our earlier address space diagram:



Each memory object in an address space (and there can be many more than 3) has 2 principal characteristics; whether the object is a shared or private object, and whether the object is backed up by the file system or the swap disk. When we designate an object as shared, we mean that we allow such an object to appear in several separate address spaces concurrently. The text object is in this category, meaning that if we have 5 processes in the system all running the bash shell, they will all share the same text object in their private address spaces. For a text object, this is sensible, since the object is usually mapped as read only, and the code within cannot be modified, so it's easy to share. A stack object, on the other hand, is a private object, and we expect that any given stack object can only be mapped into a single address space and used exclusively in just one process. When an object is backed up by the file system, we say it is a file based object, and when it is backed up by the swap disk we say it is an anonymous object. Here are the categorizations of the common memory objects found in both Windows and Linux:

- Text object    usually shared, file based object
  - Can be linked private if we want to include any self-modifying code, but such pages can then only be backed up to the swap space, since we are not allowed to modify the original executable file.
- Data object    private, anonymous
  - Although initialized global data is initially loaded in from an executable file just like text is, any modified global data must be written to swap if a data page of data is selected for replacement.
- Stack object    private, anonymous
  - The entire content of the stack object is dynamically constructed from zeroed out physical pages when a process address space is created at process create time.
- Shared memory object    shared, anonymous

- A shared memory object can be mapped into the address space of many processes, but like a stack object, its initial content is constructed from zeroed out physical pages when the object is first mapped.
- Memory mapped file object      usually shared, file based object
  - An entire file or some range of a file can be directly mapped into an address space to allow file access using simple memory reference instructions. When the object is mapped shared, modified pages of the object are written directly back to the file itself, and multiple processes can share the object concurrently. If the object is mapped privately, then any modified pages of the file that are selected for replacement must be written to swap, since a private mapping cannot modify the original state of the mapped file.

The page-purger has a propensity for selecting private object pages for replacement, since they can only appear in one address space, and if a page from such an object is reclaimed for the free list by the page-purger, only one page table has to be adjusted to reflect the replacement. Private pages are also added to the free list whenever a process terminates, since they cannot be used in any other address space than the one that was just torn down when the owning process terminated. Notice that pages from a shared object cannot be automatically added to the free list when a process that has that object mapped terminates, since the pages may be in use by other processes that also have that shared object mapped into their address spaces.

## Thrashing

Even the best page replacement algorithms will not be able to keep a system from thrashing under certain load conditions. Thrashing is a term used to describe a degenerate system that spends all of its CPU and IO cycles processing page-purger operations, with little to no resources applied to actually getting any application work done. Such a system looks very busy, with constant disk activity and context switching, but no thread ever gets to run for more than an instruction or two before it page faults and starts running fault handler code and the page-purger kernel thread. Thrashing occurs because none of the runnable threads on a given system can get enough physical page frames to hold what is known as their current “working set”. The current working set of a given process is the set of physical pages that its threads need during some phase of its operation to avoid page faulting. Consider a thread that is running through an iterative loop of code that spans 5 pages for one million iterations. If all 5 pages are memory resident, then each iteration can run very fast by avoiding any page faults. If the system is so short of pages however, that it can only come up with 4 of the needed 5 pages, then each iteration will lead to a page fault, with one of the 5 necessary pages being swapped out so the fifth page can be brought in to complete an iteration. This would lead to one million extra page faults than would have occurred if we could have just found one more physical frame to give to the thread. There are times, however, when just such a scenario can form in a system, leading to thrashing.

## Load Control

Load control mechanisms are programmed into most memory management systems, to both detect and react to thrashing. The operating system collects various statistics about general system behavior, but one of the most important statistics is the rate of execution of the page-purger. The system wide page fault rate is also important, but we really don't worry about a high fault rate if we have enough free pages to cover the faults. If we see the page-purger running frequently in an effort to reclaim in-use pages to build up the free list to handle page

faults, then we know that we're in trouble. Now we are "robbing Peter to pay Paul", and this leads us into thrashing. We need to either add more physical memory to our system (an expensive and disruptive endeavor), or we need to reduce the number of threads that are competing for page frames. Load control is all about limiting the number of threads that can reach the run state (and thereby fault pages) until the system reduces its page-purger run rate. There are various approaches to this, but they broadly fall into two camps. The swap camp uses an approach of selecting a process to "swap out" and then collects all of the currently paged in physical pages of the process and writes them out to disk. This collection of physical pages can now be distributed among the remaining processes to attempt to resolve the current thrashing situation. The swapped out process can be recovered some time later when there are enough free pages available in the system to page the swapped process back into physical memory and resume their execution without thrashing. Selecting a process to swap out is based on different factors, not the least of which is the number of physical pages that we reclaim by giving a specific process the boot. The second approach (the one used by Linux) is to locate a process that has a large number of physical pages currently paged in and send that process a termination signal. When one of the threads in the process sees the termination signal, it is forced to leave the run state by the exit path and bring down the entire process with it. The pages of the now deceased process can then be put on the free list in the hopes of eliminating the thrashing.

There are several different topics discussed in the remainder of chapter 3 that you will visit as you read your way through, but the material included above and in the previous lesson is what we're primarily interested in and what I will test over in the first exam. Virtually every question on the first half exam will come from either some part of the notes that you've been looking at across these first 7 lessons, or some part of the assignment work you've been doing for the first 5 assignments, so these weekly notes should be your first resource in studying for the exam, followed by a review of the weekly assignments.