ICSI412–6 – Virtual Memory

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

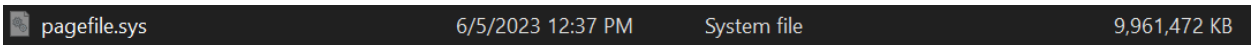
You must submit every file for every assignment.

You must submit buildable .java files for credit.

Overview

By implementing paging, we decouple the program's view of memory from the computer's view. The programmer sees memory as a contiguous block of storage. The computer sees blocks of memory and translates the program's perspective into blocks. This decoupling allows us to use "holes" in the physical memory address space effectively, but it also enables a few other possibilities.

When our computer doesn't have enough memory to fulfill all of the requests from userland programs, it uses disk space instead. While this is slower than running from memory, it does allow users to perform their tasks. The operating system can try to do a number of things to minimize the time-cost of this. We will not be modeling those. The space on disk has to belong to a file – this file is called the swap file. Windows calls this pagefile.sys. You can see this in C:\ if you enable viewing system files:



Developers often want to work with large files. We can do this by making a buffer (just allocated memory), loading a little bit of the file at a time, using that little bit, then loading some more. But that is a significant amount of code that we have to write. We have to manage the buffer, make sure it is big enough, ensure that we aren't accidentally looking at old data (we missed a "load from disk"), etc. This common pattern could be replaced with something simpler if we just loaded the entire file into memory. But you might say, that would take a long time (even with modern storage) and takes up a lot of memory, maybe more than we even have!

Believe it or not, both of these problems have the same solution! With paging, we had a simple number in the KernelandProcess that indicated the physical page number that fulfils a virtual page request. What if we replaced that array with an array of data structures? The data structure holds the physical page (like before), but it also stores a reference to a file and an offset into that file. If the physical page is present in memory, then paging works just like before. But if the page **isn't** in memory, we can read it from storage. But where do we put it? The answer is that it doesn't matter! Any page will do. We load the block from storage into physical memory and then update the data structure's physical page number. That covers the "large file" case completely.

But there is still the “not enough” memory case. In this case, some process asks for memory and the operating system has to borrow space from some other process. It does that by writing that process’ block out to disk and populating that process’ memory structure with the location of the disk block. It then reassigns that memory to the process that needed it. Of course, at some point, the process whose memory was borrowed will try to access that memory. At that point, the operating system acquires a block of memory from somewhere, loads the data from storage and updates the lender with the new physical address.

One interesting side effect of all of this is that when someone allocates memory, we now have a mechanism to grant that request without giving them any physical space. We can mark the memory data structure as valid (that is, we promised it) but unfulfilled. When the process tries to access a block, it gets fulfilled. This is used extensively in modern operating systems. Every process created is granted a very large amount of memory (stack + heap). What they actually use is paged in as it is used.

Details

Let’s start by creating our swap file. We can use fake file system and a simple call to open. Note that you might have to make some accessors depending on how you organized your code. Open the swap file on startup. You will need an integer to track the page number (remember, a page = 1024 bytes) that is the “next page to write out”. We won’t be trying to reuse pages on the disk file.

Initially, we used an array of int in our KernelandProcess to map virtual->physical address. Now we need to track more information. Create a new class (VirtualToPhysicalMapping) with public members for physical page number and on disk page number. Create a constructor (no parameters) that sets physical page number and disk page number to -1. Change the int[] in the KernelandProcess to an array of 100 of these classes. Fix the related code issues.

Previously, when the user called AllocateMemory, we did all of the virtual->physical mapping. We are going to remove that, since we now will have the ability to deal with physical pages that don’t exist. Now we can just create instances of the VirtualToPhysicalMapping (one for every page that the user is allocating) and populate the array. If an element of the array is not null, we know that the user has allocated space. The method for detecting block runs is no longer based on physical page being -1, but on the presence of mapping entries.

FreeMemory now needs to check to see if the physical page is not -1 before updating the physical memory in use. It also needs to set the array entry for each block back to null (freeing the VirtualToPhysicalMapping).

The last thing to do is to fix GetMapping. Previously, GetMapping assumed that there would be physical memory backing our virtual memory. Now, it cannot. Find the memory map entry as before. If the physical page is -1, find a physical page in the “in use” array and assign it. If there isn’t one available, that’s when we have to do a page swap to free one up.

> Start the page swap process by adding a new method to the scheduler:
>
> `public KernelLandProcess getRandomProcess()`
>
> Get a random process and find a page in the process that has physical memory. If there are none, pick a different process and repeat until you find a physical page. Write the victim page to

disk, assigning a new block of the swap file if they didn't have one already. Set the victim's physical page to -1 and ours to the victim's old value.

If we got a new physical page (remember – we could get here simply because the TLB didn't have the data we needed), we have to do one of two things – if data was previously written to disk (the on disk page number is not -1) then we have to load the old data in and populate the physical page. If no data was ever written to disk, we have to populate the memory with 0's.

Test your code!

Test reading and writing (to make sure that you get the same value). Test using more than the allocated memory (I made a process that used 100 * 1024 bytes called piggy. I then instantiated piggy 20 times.)

Rubric	Poor	OK	Good	Great
Swap file	None (0)			Is created using FFS (5)
Page object	None (0)	Is now a class (3)		Is a class with data items described (5)
AllocateMemory - Lazy Physical Page Allocation	Still allocating up front (0)			Allocated on demand (10)
FreeMemory – correctly frees memory	No changes (0)		Marks physical space free OR clears VirtualTo PhysicalMapping (5)	Marks physical space free, clears VirtualTo PhysicalMapping (10)
Physical pages are used	None – all memory access fails (0)			Free physical pages are mapped (10)
Swapping uses a random process	No swapping (0)		Swapping picks a fixed process (5)	Random process used (10)
Swapping writes out pages	No(0)			Writes out pages (10)
Physical pages are cleared when not loading	No (0)			Yes (10)
Physical pages are loaded when they were previously written out	No(0)		Yes, but wrong (5)	Read correctly (10)
Pages are preserved when the physical mapping already existed	No (0)			Yes – the existing mapping is copied into the TLB (10)
Testing	None (0)		Partially tested (6)	Test processes that show all functionality working - multiple processes reading and writing to memory and proving that they are not overwriting each other. Memory is filled and shows that paging works. (10)