ICSI412 – 5 - Paging

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.

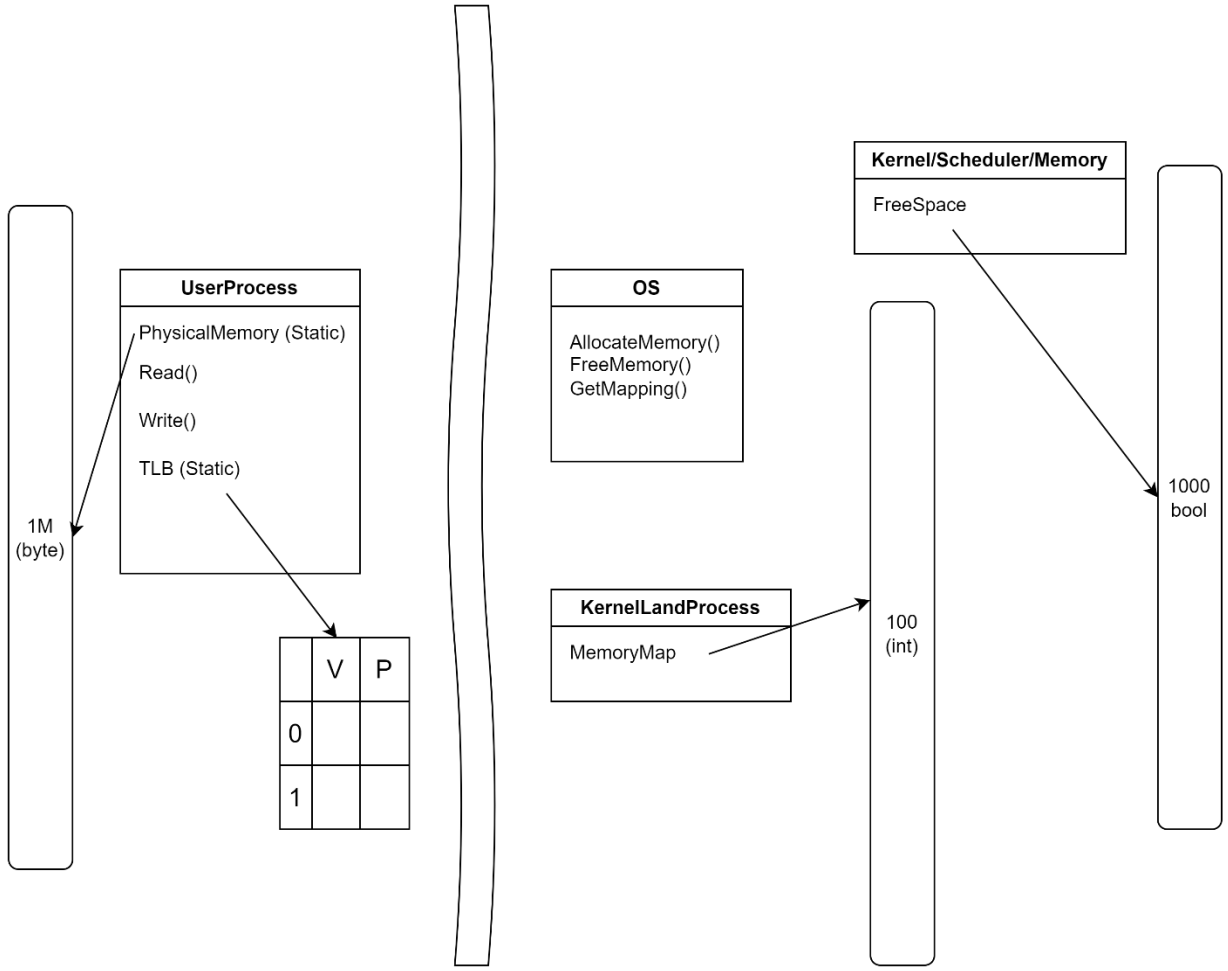
Overview

In operating systems, we manage three key sets of resources – processor, devices, and memory. In paging, we start addressing the memory. Normally, a significant amount of the work is done in hardware. Our simulation is a little … unusual because we don't have hardware. Let's review the main ideas.

Memory is arranged as a contiguous block of memory cells from 0 → 16 billion (for 16GB); we call this physical address. Because programs come and go, we end up with "holes" – small areas of memory that are not usable because they are so small. To fix this, we break memory up into pages – fixed size blocks of memory, typically around 4KB. Most programs want more than 4KB at a time and they want it contiguous. To provide this interface, the hardware manufacturers provide the idea of a virtual address – a translation from virtual→physical address, and they do that in hardware. With this mapping, I can ask for 16KB, get pages 904, 10, 30 and 47 but they look to me like addresses 0-16383. Note that this mapping is on the **page level**. That means that to map 4 pages, we need 4 virtual→physical mappings, but all of the addresses are remapped.

This mapping is also stored in memory. In fact, this mapping is different for every process. The hardware has a single register (on Intel/AMD, this is called register CR3) that holds the address of the mapping. When we switch between processes, we update this one register and the hardware has the new process' memory mapping in place.

The "problem" is that this means that every memory access now costs 2 (or more) memory accesses – one to look up the virtual address in the mapping and get the physical and one access to do the "real" work that was asked for. Hardware designers recognized that most programs don't jump all over the place in memory – they look at a very small number of addresses at any point in time. So they added some very fast memory called the TLB – the Translation Lookaside Buffer – that holds a few translations. This fast memory has to be purged in between processes (which slows that down a little) but brings our average memory accesses from 2 accesses/request to very close to 1 access / request.



Details

A modern CPU has variable page sizes, 4KB or bigger. To keep our model reasonable for Java, we will use 1KB pages and 1MB (1024 pages) of memory. We will be adding new methods to our UserlandProcess for the first time:

```
byte Read(int address)
void Write(int address, byte value)
```

These two methods simulate accessing memory. The first thing that either of these methods need to do is find the page number. That is easy – address/page size. Next, they need to look at the TLB to see if this virtual page → physical page mapping is in there. The TLB (which, of course, normally is in hardware) should be a **static** array of integer holding 2 virtual addresses and 2 physical addresses. I would use a [2][2] array, but there are other schemes which work. If the mapping is found, then we need to calculate the physical address. Remember that the virtual page is virtual address/1024. Where we are within the page is virtual address % 1024. Once we know the physical page number, we multiply it by the page size and add the page offset to get the physical address.

Example:

Read(3100)
Virtual address = 3100, Virtual Page = 3100 / 1024 = 3, Page Offset = 3100%1024 = 28
We look in the TLB and find that this is in physical page 7.
Page offset (still) = 28, Physical address = 7 * 1024 + 28 = 7196

Make a static array of 1,048,576 (1024*1024) bytes in UserlandProcess – this will be our memory. If the virtual→physical mapping is in the TLB, we could now go get the byte and return it. If not, what do we do? In real hardware, this would cause an interrupt. Instead, we will perform an OS call:

```
void GetMapping(int virtualPageNumber)
```

To implement GetMapping(), we will add a new data element to the KernelandProcess – an array of integers. The virtual page number will be the index into the array, the value in the array will be the physical page number. Upon creation of a KernelandProcess, this array should be set to all -1 values (no mapping). Let's make the array 100 elements – 100 1k pages is 1/10 of our machine size. GetMapping should update (randomly) one of the two TLB entries. Back in user space, we should then try again to find a match.

This leaves only two pieces – allocating and freeing memory.

```
int AllocateMemory(int size) – returns the start virtual address
```

```
boolean FreeMemory(int pointer, int size) – takes the virtual address and the amount to free
```

In user space (OS), we should ensure that size and pointer are multiples of 1024; return failure if not. Inside the kernel, we need an efficient mechanism to track if pages are in use or not. An array of boolean will work. All blocks in memory start out as free, then as memory is allocated, mark the pages as in use and assign them to the process' array. This gets a bit tricky, though, with FreeMemory's existence – it can make holes in your virtual memory space. You will have to look for a space big enough to map into.

Finally, on termination of a process, we need to free all of its memory. You also must clear the TLB on task switch.

Test your code!

Test reading and writing (to make sure that you get the same value). Test extending your memory. Test trying to access memory that you shouldn't be able to and make sure that your process is killed.

Rubric	Poor	OK	Good	Great
Page table in KernelandProcess	None (0)			Exists and is right sized (5)
UserlandProcess - Memory array	None (0)			Exists and is right sized (5)
UserlandProcess - tlb	None (0)			Exists and is right (5)
Kernel – free list	None (0)			Exists and is right sized (5)

UserlandProcess – ReadMemory	None (0)	Gets virtual page, gets physical page (3)	Gets virtual page, gets physical page, checks TLB (6)	Gets virtual page, gets physical page, checks TLB and returns data correctly (10)
UserlandProcess – WriteMemory	None (0)	Gets virtual page, gets physical page (3)	Gets virtual page, gets physical page, checks TLB (6)	Gets virtual page, gets physical page, checks TLB and writes data correctly (10)
Kernel – AllocateMemory	None (0)	Finds number of pages to add, adds to the first correctly sized hole in virtual space (7)	Finds number of pages to add, adds mapping to the first correctly sized hole in virtual space, marks physical pages as in use (13)	Finds number of pages to add, adds mapping to the first correctly sized hole in virtual space, marks physical pages as in use, returns correct value (20)
Kernel - FreeMemory				Marks physical pages as not in use and removes mappings (10)
TLB Cleared	None (0)			TLB cleared when process switched (5)
Memory freed on end of process	None (0)			memory is freed when a process ends (10)
Testing	None (0)		Partially tested (9)	Test processes that show all functionality working - multiple processes reading and writing to memory and proving that they are not overwriting each other. (15)