Lab 5 Design Document

Group 2
Aaron Meurer
Oran Wallace
Sheng Lundquist

Introduction

The memory manager will be an essential part of our operating system. The memory manager will be in charge of allocating and deallocating user and OS memory, generating page tables for processes, and the page fault process.

Page Replacement Algorithm

We will use the least recently used (LRU) replacement algorithm. This chooses the least recently used frame in memory to replace whenever there is a page fault. This algorithm does not suffer from Belady's anomaly (Silberschatz, Galvin, and Gagne 377).

This will be achieved by simulating a hardware counter. Whenever a page is accessed, the hardware will copy the value of the counter to a slot in a table representing that memory. When a page is victimized, the memory manager will search through this table for the page with the smallest counter value. In the rare case when the counter overflows, all counter values will be reset to 0.

All processes are lumped together for the choice of victim. Thus, processes that use memory be able to take over the physical memory used by processes that are not using their memory.

This algorithm does not require any special casing for the first start up of the machine. The counter table will begin with 0's. Thus, when it searches for a minimum value, it will take one of these empty pages with a 0 counter value.

The command `PAGE_HIT page_table_id page_num` will be added to active the hardware simulation. Accessing a page for the first time (i.e., a fault) will also count as a "reference" for the LRU algorithm. This command will give an error if the page is not in memory (we will require to use the page_fault command to handle that case).

Page/Frame Size

We decided to use a 4K page/frame size. This number is used to reduce the memory needed for the operating system. If a 1K page/frame were used, 25% of the physical memory space would have been used for the kernel, instead of the 7.8% required by a 4K page size. A 4K page size is a good estimate of how much memory will need to be brought in based on spatial locality.

System memory

We estimated how much memory the kernel will use using the following formulas. For the memory manager, we need to consider the (worst case) size of the page tables for all processes, the size of the table used for the LRU computation, and the size of a data type used to keep track of what memory is free in the backing store. For the page table, it can use up to:

$$max_{processes} * max_{pages/process} * sizeof(page\ table\ entry) =$$
$$16\ processes * (4\,MB / 4KB) * 4B$$

The 4 comes from using two byte (unsigned char) (one for the physical memory address and one for bitmasks) and one short int type (for the backing store address) in each page table entry.

For the LRU table, there will be:$2(frames\ in\ physical\ memory * sizeof(unsigned\ int)) = 2 * 256 * 4B$. Since there will also be a pointer to the page table that is using the current frame, it will be twice the size of the unsigned int.

For the backing store, we will use one bit to store the information of free or allocated for each frame, so it will require:

$$\frac{4096\,frames}{8\,bits/byte} * 1\,bit\,/\,frame = 512\,bytes.$$

Adding these up and dividing by $4096\,bytes\,/\,frame$, we get that the memory manager alone will require roughly 16.6 frames. The other parts of the kernel will not require much space in comparison (for example, we computed that all the queues in the process manager will use roughly 1.5 frames), so we estimate that allocating 20 frames for the system should be sufficient. It will be easy to modify this if it ends up not being enough, and it will also be straightforward to do further optimizations, such as changing data types in other parts of the system from ints, short ints, bytes, or even bits.

Page Table Layout

Our page tables will be created from the “ALLOC_PT” command, which is to be used by a process requesting address space. The size of each page will be 4 KB--the decision of this size was discussed earlier. Since each page is 4 KB, there can be up to 265 frames (pages) in the physical memory at any time. Our page table will contain a column to indicate the addresses that the pages map to in backing store. and will also contain the frame number that is mapped to the page. It will also contain a byte for bitmasks. Our bitmasks are a valid bit for physical memory, a valid bit for backing memory, a dirty bit, a page initialized . The size a specific page table will depend on the process it was created for, but it can range from 1 - 1024 entries. The maximum number of page tables will depend on the maximum number of processes allowed in our OS, right now this is 16, but this can be easily changed.

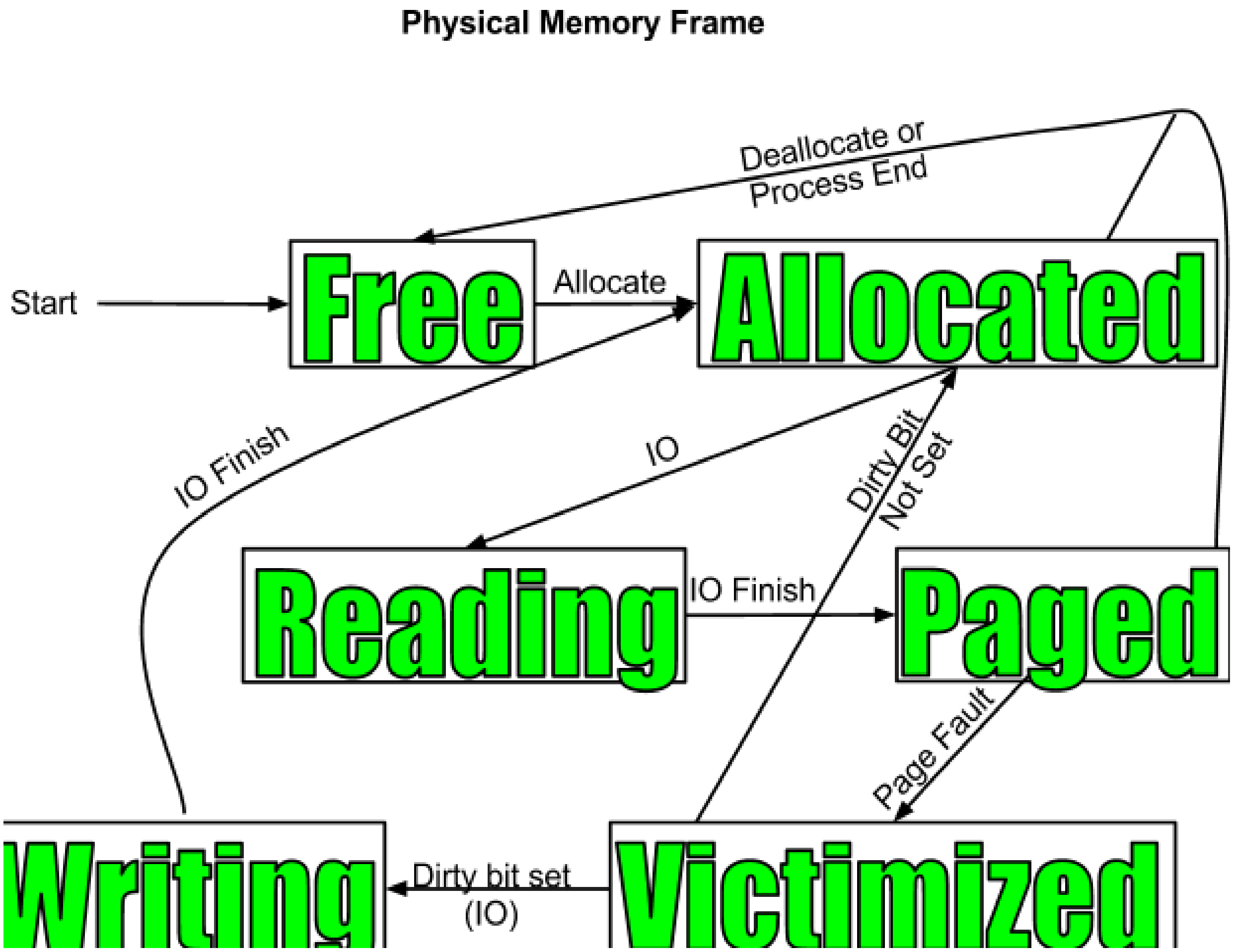
Physical Memory Layout

Our physical memory layout will be a 1MB array that is split into 256 4 KB frames. This is based on our choice in the page/frame size. The physical memory will be an array of 1024 entries (frames). We have decided to dedicate 20 4KB frames for the OS at all times, and the rest of the 236 will be used for the user processes.

Logical Memory Limit

Our logical memory limit that a process can have will be 4 MB. This number was chosen based on our process manager's process limit. The current limit of our process manager is 20. We will change this to the nearest power of 2, 16, for ease of calculation. Our backing store of 64 MB can be split for these 16 processes, which creates a 4 MB limit of memory per process. The memory manager will still check for full backing store memory and throw an error if it is full (for example, if we increase the max process number, this case will become possible).

Memory States



There are six possible states for a frame of physical memory. When the OS starts, all physical memory frames (except for those that are being used by the OS) will be in the “Free” state.

When a process is created, memory for that process goes into the “Allocated” state. Memory in this state is not being used, but is set aside by the OS for use (a slot in a page table exists for it).

When the process reads in data to the memory, that memory is in the “Reading” state. It must wait in this state until the memory has been read in.

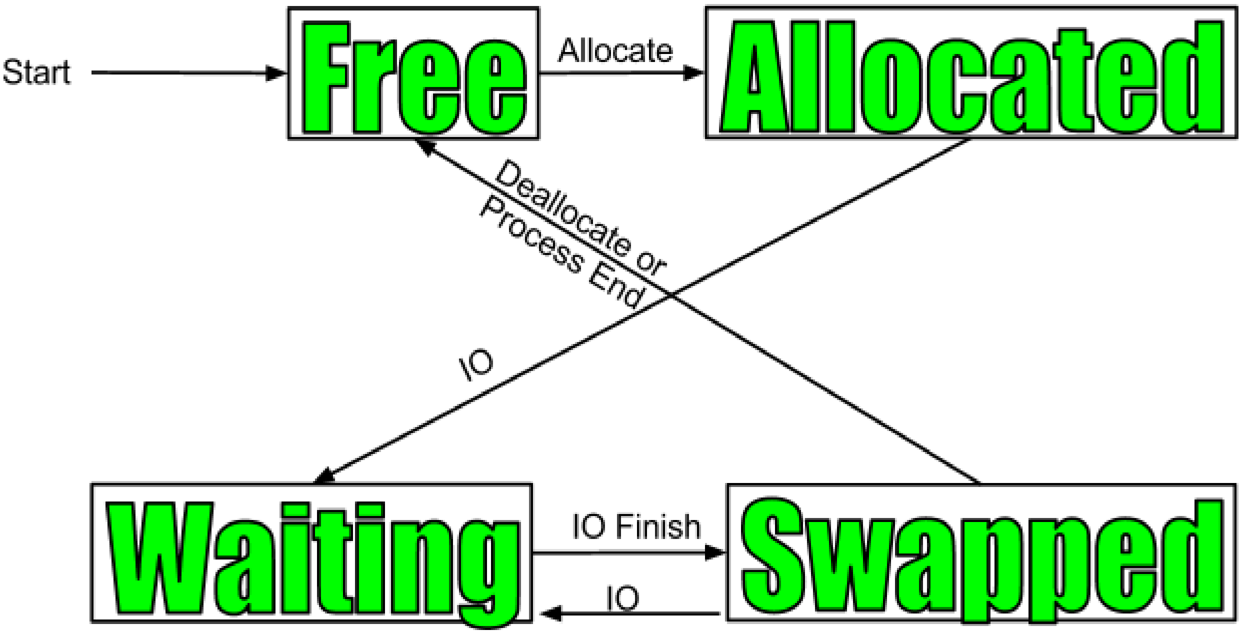
When the IO finishes, the memory is in the “Paged” state. Memory in this state is immediately usable.

When the page replacement algorithm decides to victimize a page, that frame moves into the “Victimized” state. If the dirty bit is not set, it moves from there immediately to the “Allocated” state. Otherwise, the data is written back.

While it is being written, it is in the “Writing” state. It must wait in this state until the IO finishes. When the IO finishes, it moves to the “Allocated” state.

Finally, when the process ends, or decides to voluntarily free the memory, it goes back to the “Free” state. This can happen either from the “Paged” or the “Allocated” states (“Reading” and “Writing” are waiting states, so it does not make sense to deallocate from them, and “Victimized” is a meta-state that memory will not actually stay in.

Backing Store Frame



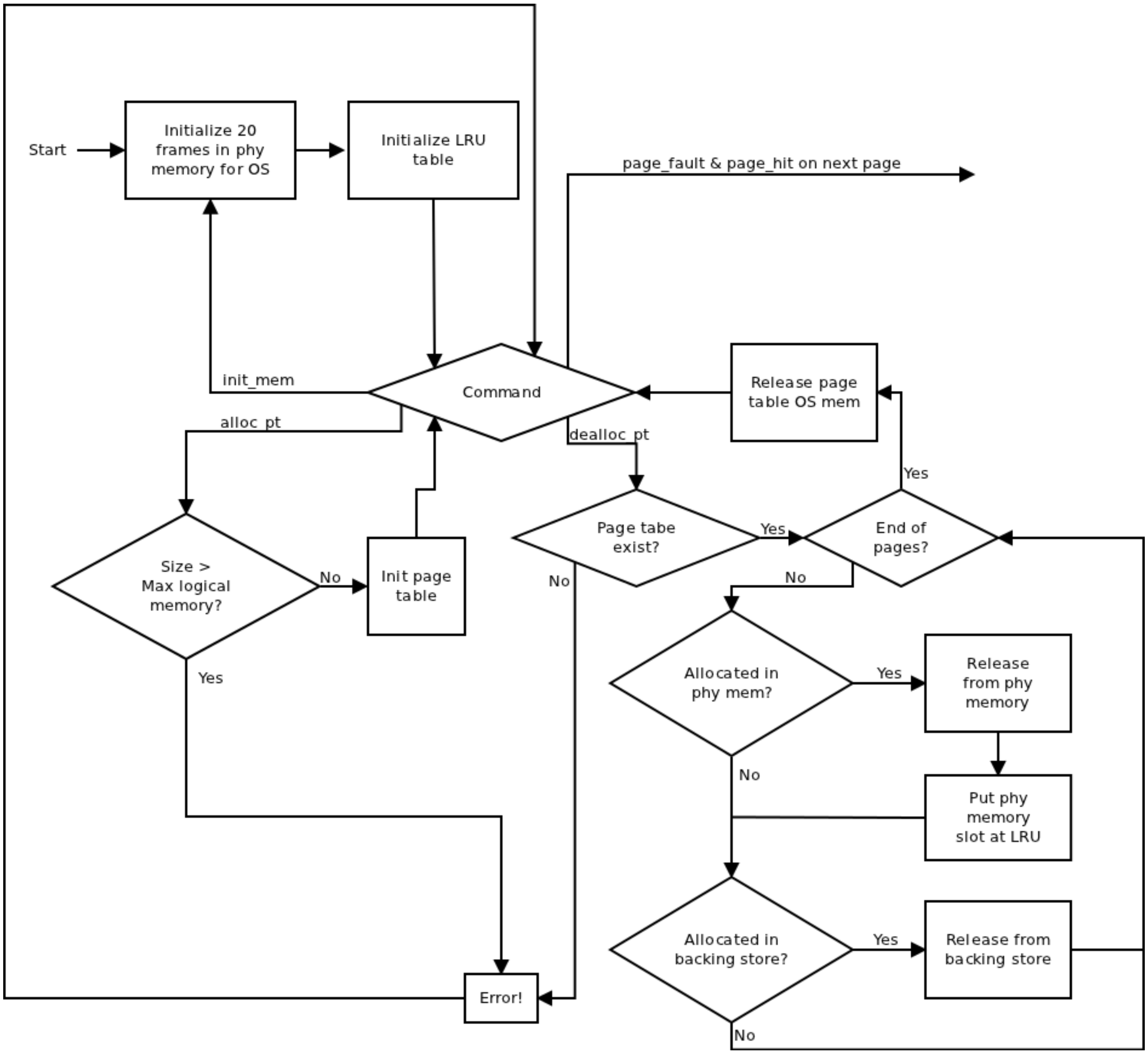
Memory in the backing store can be in four states. Again, when the OS starts, all backing store memory will be in the “Free” state.

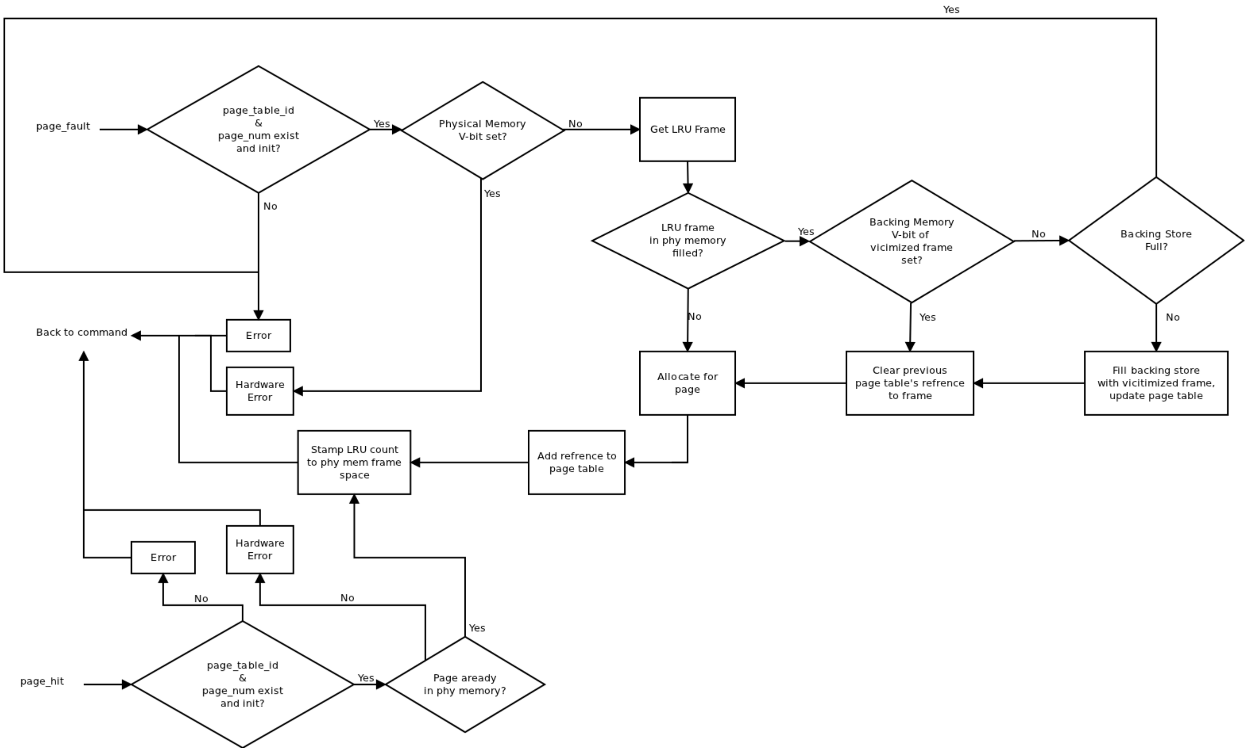
When a page fault occurs, the memory moves to the “Allocated” state, which just means that the memory is no longer marked as free.

Then, the memory moves to the “Waiting” state, which means that it is waiting for the memory from the physical memory to be written back.

Once the IO is done, it is in the “Swapped” state. Memory in this state has actual data from the program. This data may not be up-to-date with the physical memory, though if the dirty-bit on the corresponding physical memory frame is not set, it will be. If a page fault happens again on the same frame and the dirty bit is set, then it will go again to the “Waiting” frame while the data is being written back, and will remain there until the IO is finished, after which it goes again to the “Swapped” state.

Manager Flow Diagram


Start
Initialize 20 frames in phy memory for OS
Initialize LRU table
page_fault & page_hit on next page
init_mem
Command
Release page table OS mem
alloc_pt
dealloc_pt
Yes
Page tabe exist?
Yes
End of pages?
Size > Max logical memory?
No
Init page table
No
No
Allocated in phy mem?
Yes
Release from phy memory
Yes
No
Put phy memory slot at LRU
Allocated in backing store?
Yes
Release from backing store
Error!
No



Test Cases

Testing will be performed on both the physical memory, backing store and page tables.
We will test that the addressing scheme is working correctly for both the physical memory and the backing store. We tested that memory can be in both physical and backing store. We will test that our LRU implementation works correctly, using the commands page_hit and fill_phy_mem. Once the backing store is full we test that a page fault does not work.

The following test cases are for both functions called by the test interface.

- INIT_MEM
  - Check if init_mem command works properly
  - Test if memory is fully cleared after its been allocated
- ALLOC_PT
  - Check is alloc_pt works correctly
  - Test for max pages for a process
  - Test for max page table limit
- DELLOC_PT
  - Check dealloc_pt works correctly
  - Test table not initialized
- FILL_PHY_MEM
  - Check to see if command fills physical memory
  - Test page table has enough pages to fill physical memory
  - Test table not initialized
  - Test page table argument from 0 to 15

- PAGE_FAULT

  Check if page_fault command works properly
  Test table not initialized
  Test page not initialized
  Test page table from 0 to 15
  Test hardware error of already in physical memory
- PAGE_HIT

  Check if page_fault command works properly
  Test table not initialized
  Test page not initialized
  Test page table argument from 0 to 15
  Test hardware error of not in physical memory
- LIST

  Check if list works correctly for all arguments
  Test list pagetable argument from 0 to 15
  Test table not initialized
- LRU_OVERFLOW

  Check next page hit resets LRU
  Check next page fault reset LRU

How to Run Memory Manager

After running the “make” command to compile, use the memory_test.o file for grading purposes. If ./memory/memory_test.o is given the argument memory/mem_tests the mem_tests.txt file will be read and executed by the test interface. If no argument is given when ./memory/memory_test.o is entered a CLI interface can be used. Use the HELP command to see function names.

References

Silberschatz, A., P. B. Galvin, and G. Gagne. *Operating system concepts*. John Wiley & Sons Inc, 2009. 377. Print.