Lab 5 Design Document

Group 2
Aaron Meurer
Oran Wallace
Sheng Lundquist

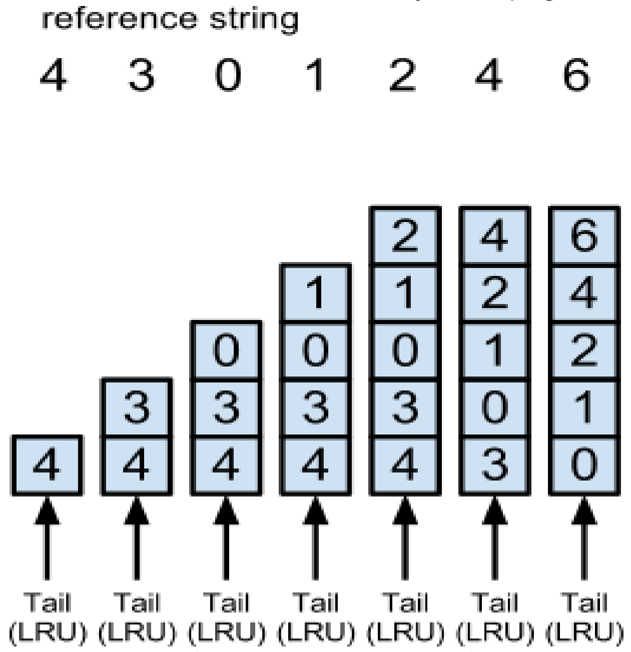
Introduction

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).

We will use the stack structure described on page 377 of the textbook to implement this algorithm. Whenever a frame is brought into memory, a reference to it is pushed onto the stack. When a page is accessed, its reference is moved to the top of the stack. The stack is implemented as a doubly linked list. This makes updates a little more expensive, but there is no need to search for the least recently used page to replace when there is a fault, because the tail pointer will always point to the reference to the least recently used page in the stack.



For example, suppose we had the following sequence of page accesses, with a page table size of five. The first five accesses fill up the stack. The second access of frame 3 moves it to the top of the stack. The access to 6 requires a swap. So we remove 3, which is at the bottom of the stack, because this corresponds to the least recently used frame, and then push 6 onto the stack.

We can implement this on top of our queue manager infrastructure by adding a function to "push" onto the queue, and then treat the queue as a stack ("pop" will just be delete). This will work because both data structures are built on top of a doubly linked list.

There will be a single stack that references all frames used in physical memory (except for those used by the system, because those should never be paged out). Thus, 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.

Because this algorithm requires knowledge of how often a frame is used, not just when it is faulted. The command will be `page_access page_table_id page_num`. 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 stack 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 $max_{processes} * max_{pages} / process * size(page\ table\ entry) = 16\ processes * 64\ MB / (16\ process$, where the 4 comes from using two short int types in each page table entry. For the stack, there will be $frames\ in\ physical\ memory * size(stack\ entry) = 256 * 4B$, where the 4 again comes from storing two short int types in each stack entry. For the backing store, we will use one bit to store the information of free or allocated, 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 16.3 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 using even smaller data-types than short ints, or changing data types in other parts of the system from ints to short ints.

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 4KB; the decision of this size was discussed earlier. Since each page is 4 KB, there will 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 page map to in backing store. and will also contain the frame number that is mapped to the page. 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, which can be easily changed.

Page Size: 4KB
Page Table Size: 1 - 1024 entries
Max # of Page Tables: 16 (max number of processes)

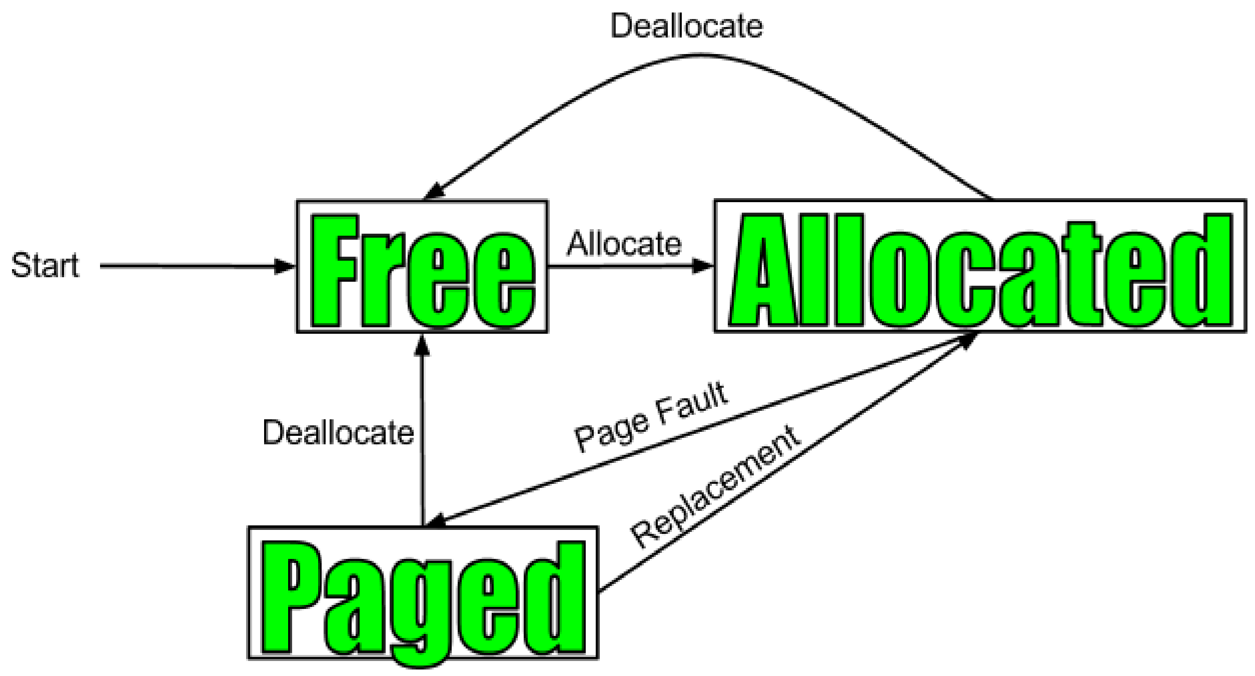
Physical Memory Layout

Our physical memory layout will be a 1MB array that is split into 256 (4KB) 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 4MB. This number was chosen based on our process manager’s process limit. The current limit of our process manager is 20. Rounded to the nearest power of 2, we can assume a process limit of 16. Our backing store of 64MB can be split for these 16 processes, which creates a 4MB limit of memory per process. The memory manager will still check for full backing store memory and throw an error if it is full.

Memory States

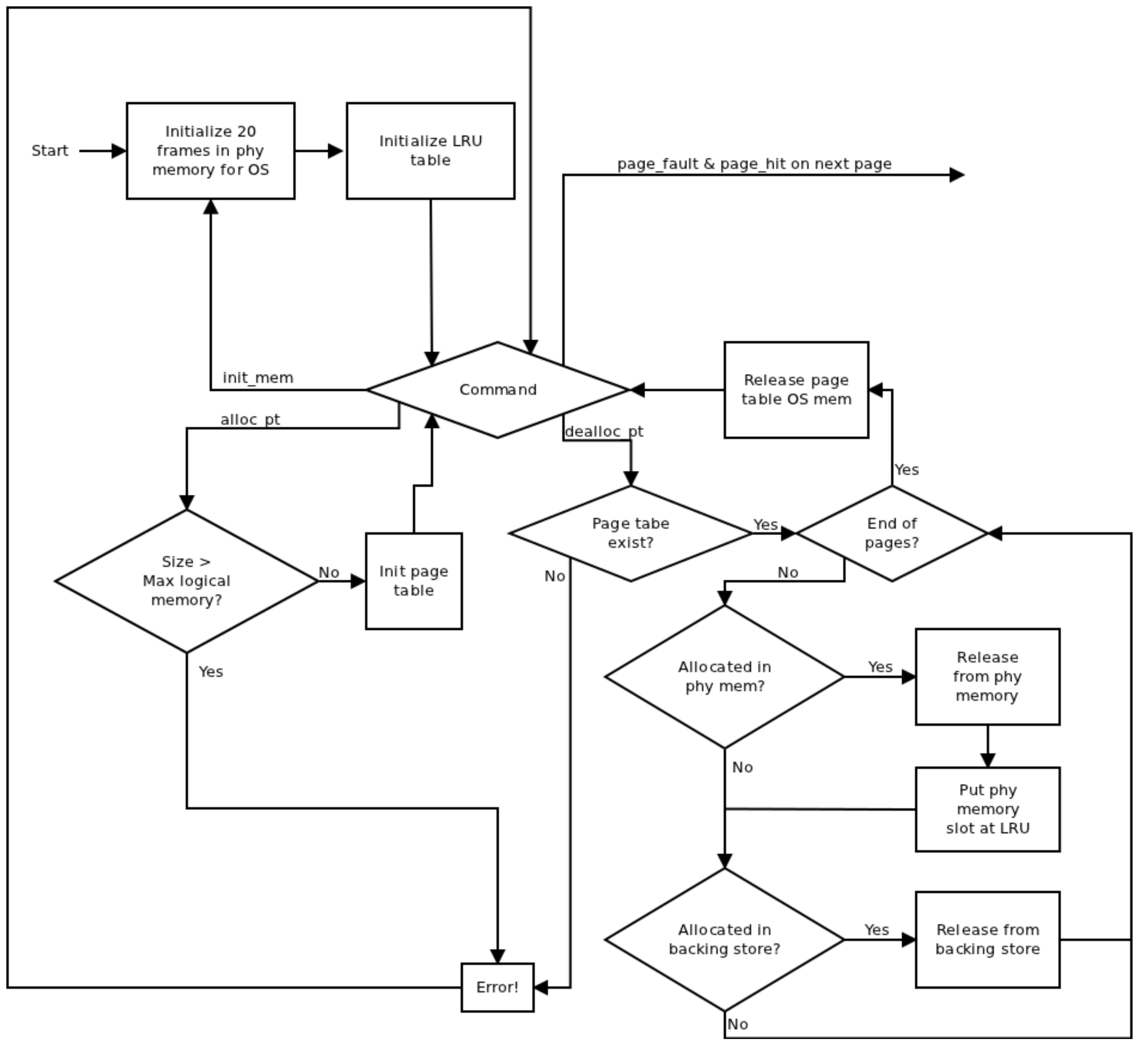


Memory blocks for our operating system will be in one of three different states at all times. Initially all memory blocks will start out in the “Free” state. In this state the block is simply sitting in the backing store waiting to be allocated for some process. When the function to allocate a page table is called the block of memory has a page table created for it and transitions into the “Allocated” state.

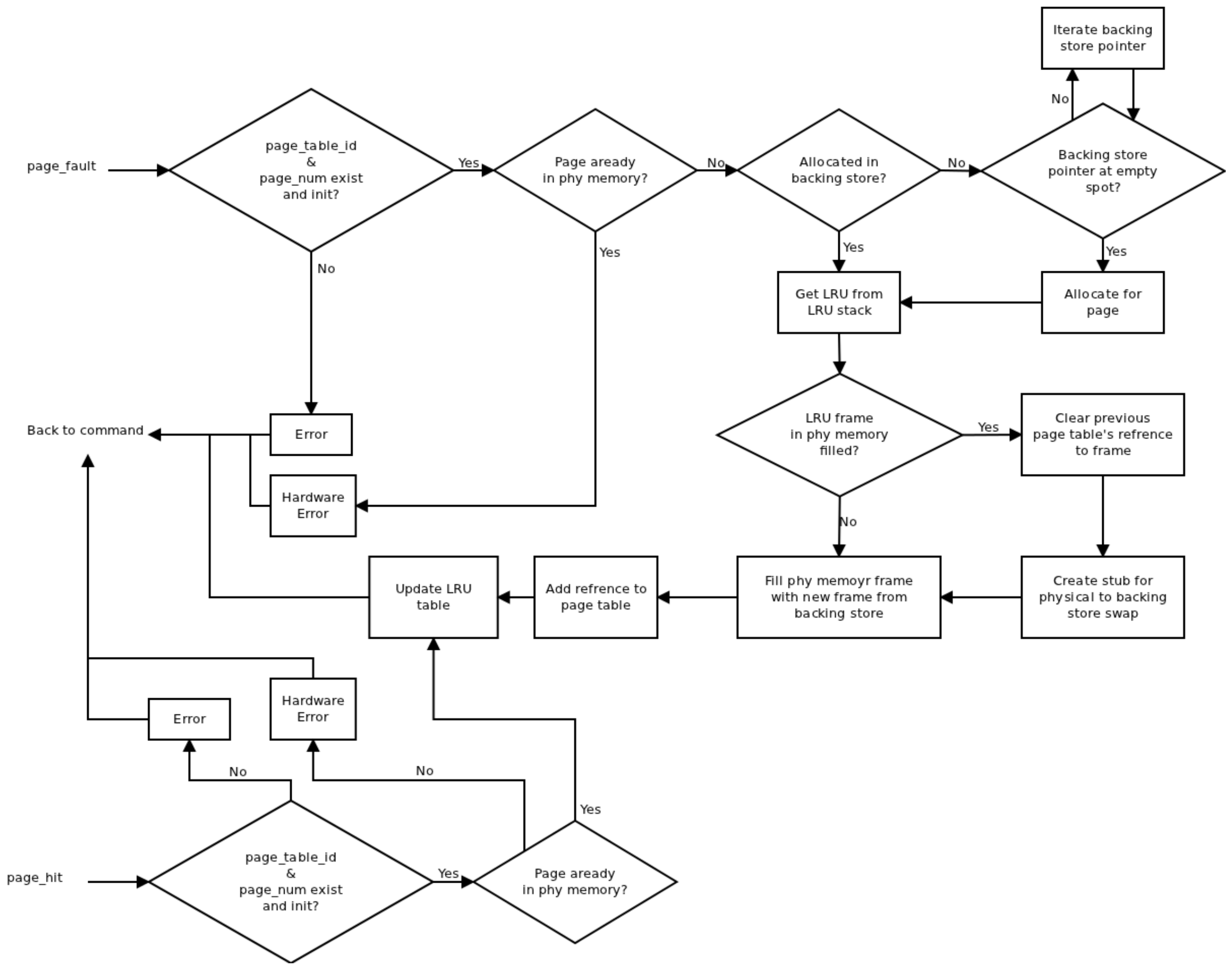
In this state the block of memory is referenced through the newly created page table. When a page fault occurs and the memory block will transition into the “Paged” state, when the page table is deallocated the block of memory will transition back to the “Free” state.

In the “Paged” state the memory block has been brought into the physical memory and is really to be used by some process. From this state a block of memory can transition to the “Free” state if a deallocation function is called for the block or it can transition back to the “Allocated” state if our page-replacement algorithm decides the page needs to be swapped back to the backing store.

Manager Flow Diagram



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


Iterate backing store pointer
page_fault
page_table_id & page_num exist and init?
Yes
Page aready in phy memory?
No
Allocated in backing store?
No
Backing store pointer at empty spot?
No
Yes
No
Yes
Yes
Get LRU from LRU stack
Allocate for page
Back to command
Error
LRU frame in phy memory filled?
Yes
Clear previous page table's refrence to frame
Hardware Error
No
Update LRU table
Add refrence to page table
Fill phy memoyr frame with new frame from backing store
Create stub for physical to backing store swap
Error
Hardware Error
No
No
Yes
page_hit
page_table_id & page_num exist and init?
Yes
Page aready in phy memory?

References

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