**Question 1: Brieﬂy describe the data structure(s) that your kernel uses to manage the allocation of physical memory. What information is recorded in this data structure? When your VM system is initialized, how is the information in this data structure initialized?**

We used a coremap consisting of an array of coremap\_page structs. Each coremap\_page contains the state of the page (free, allocated, or fixed), the address space/virtual address to which the page is allocated, and the number of pages allocated with it. The virtual address and address space is used when swapping out to notify the page table referring to the page that the page has been swapped out and the number of pages is used to free multiple up a block of pages that are allocated together.

When the VM system is initialized, we use ram\_getsize to get the amount of ram already ‘stolen’ during the initialization and the total amount of ram. We then create a coremap array to store all of the pages in memory and put it right after the memory that has already been stolen. Then finally, we mark every page up to the end of that array as fixed (unswappable).

**Question 2: When a single physical frame needs to be allocated, how does your kernel use the above data structure to choose a frame to allocate? When a physical frame is freed, how does your kernel update the above data structure to support this?**

The OS looks through the coremap to find an entry marked as free. It searches linearly through the coremap to find a free frame, but each time it goes to find a free frame, it starts off where it last stopped, circling back around to the start of the array if necessary. If it is unable to find a free frame, then it panics (or attempts to swap out a page if we get that working by the time this is submitted).

When a frame is freed, we simply find the frame’s entry in the coremap and mark it as free.

**Question 3: Does your physical-memory system have to handle requests to allocate/free multiple (physically) contiguous frames? Under what circumstances? How does your physical-memory manager support this?**

Our system only allocates/frees multiple contiguous pages if more than one page is requested by kmalloc. It searches through the coremap as stated above, but it looks for the given number of free pages in a row. The first page in the coremap then stores the number of pages allocated so that we can free all of them when kfree attempts to free the block of pages.

**Question 4: Are there any synchronization issues that arise when the above data structures are used? Why or why not?**

We didn’t encounter any synchronization issues, but to be safe, we used a lock in any functions that altered the coremap. Since the coremap is declared static, the only way to access it is through these functions so we’ll never have synchronization issues.

**Question 5: Brieﬂy describe the data structure(s) that your kernel uses to describe the virtual address space of each process. What information is recorded about each address space?**

**Question 6: When your kernel handles a TLB miss, how does it determine whether the required page is already loaded into memory?**

**Question 7: If, on a TLB miss, your kernel determines that the required page is not in memory, how does it determine where to ﬁnd the page?**

**Question 8: How does your kernel ensure that read-only pages are not modiﬁed?**

**Question 9: Brieﬂy describe the data structure(s) that your kernel uses to manage the swap ﬁle. What information is recorded and why? Are there any synchronization issues that need to be handled? If so what are they and how were they handled?**

Our swap file is managed using a *vnode* (which points to the actual swapfile) and an array of integer flags that indicate whether or not a swapfile entry is in use. There are functions to request a page in the swapfile (*swapfile\_prepareswap*), to store to that page (*swapfile\_performswap*), and finally to retrieve and free up the page (*swapfile\_getpage*). *swapfile\_prepareswap* returns an integer which is the index in the swapfile of the page. This integer is then passed to *swapfile\_performswap* and *swapfile\_getpage* to perform the other operations on that page. A page is marked free when *swapfile\_getpage* is called on it.

Like the coremap, the objects used by the swapfile are static and accessed only through function calls. These calls use locks to ensure that the swapfile is only being accessed by one thread at a time.

**Question 10: What page replacement algorithm did you implement? Why did you choose this algorithm? Is this a good choice, why or why not? What were some of the issues you encountered when trying to design and implement this algorithm?**

We chose to do random page replacement as it was the simplest to implement and it doesn’t require any additional overhead. We wanted to stick with a simple page replacement algorithm until we actually got swapping fully implemented so that we didn’t waste time on something that might not end up getting finished.