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?
Our VM system has X main data structure components: core map (to manage physical memory), address spaces (to manage virtual memory), segments (to store information about each address space segment), and page tables (to manage page information for a particular segment).

The core map keeps track of information for our page replacement algorithm (a clock pointer, since we use the clock algorithm), how much memory is used for kernel memory that cannot be freed (such as memory core map uses itself) and an entry for every physical page of RAM. For each page of ram, we keep track of the virtual page that owns the current memory, whether the kernel has marked the memory as not pageable, and pointers to the next and previous free page of memory (if the memory is free). We initialize core map very early in the bootstrap process since it is used by many other systems. To initialize it we calculate how much memory will be unfreeable by using ram_getsize(), and set the other pages to: have no virtual page owning it, not in use by the kernel, and part of the free list.

The address space structure keeps an array of segment information, and a vnode for the elf file so demand paging may take place. This structure is initialized when a thread is created, by reading from the elf file for segment information, and then storing the vnode for later access. A separate stack segment for each address space is also created by the kernel.

The segment structure mainly stores the base virtual address, the size of the segment, whether the segment is write-able, and a page table to store information about each specific page. There are also fields for offset information to find the location of the segments data in the elf-file on a demand load. Segment data is initialized by reading from the elf file when a program is loaded. For the stack segment, special zeroed elf file information is stored, so that when trying to load the page from elf, it will always zero fill.

The page table stores for each page: the base virtual address, the physical frame number the page is loaded in (or not loaded), the swap file location the page is stored in (or not stored), a valid bit, a dirty bit, and a use bit (used for page replacement). The page table is initialized during segment initialization, and all pages are initially set to be not loaded, not used, and not stored in physical memory, causing them to be demand paged on a fault.

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?
When a single physical frame is needed, the kernel will first check the core maps free frame list, and immediately return the first frame it finds (setting kernel use to one so that it will not be altered), and removes it from the free list. If no free frames are found, our kernel will run the clock algorithm on the frames to select a victim. This clock algorithm only uses the use bit, and not a modified to select its victim. If a kernel page is needed, we just look for the first non-kernel page, and put it there, swapping out that page if it isn't free. This is done to avoid fragmenting kernel pages preventing large kmalloced that require more than 1 page of continuous memory to work. To avoid constantly paging out user pages to make way for kernel pages, user pages go to the beginning of the free list when freed, and kernel pages go to the end of the free list. Also, the free list starts at the last page and works backwards, while kernel pages start looking at the first page and go forwards. This prevents unnecessary “bullying” by the kernel.

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?

Yes. The physical memory system have to handle requests to allocate/free multiple (physically) contiguous frames. Under the circumstances when kmalloc is called, and the size requested is greater than LARGEST_SUBPAGE_SIZE. Then we will try to allocate (size + PAGE_SIZE - 1)/PAGE_SIZE contiguous frames. Our physical manager supports this through the data structure provided by core map. Up on the request of k number of pages, the core map will check and see if there are k free contiguous pageable frames inside the core map. If such k frames exist, all k frames will be cleared out by either pushing the contents to swap or taking it away from the list of unused frames and marking them as not pagable kernel memory and finally returns the physical memory location.

Question 4: Are there any synchronization issues that arise when the above data structures are used? Why or why not?
During the design and development of our kernel, we decided that we will use any necessary synchronization mechanisms to ensure critical atomic operations. Because the low levelness of virtual and physical memory, and to accommodate multiple threads, throughout the virtual memory handling codes we've used several forms synchronization from locals to disabling interrupts to handle situations such as updating the vmstats, making sure the writing to the swap space is atomic, and allocating and deallocating core map details. We also mark pages as kernel when allocating them, even if they are user pages, then setting kern back to 0 (if it's a use program) through the function cm_finish_paging. This prevents a page from being swapped out while it's being setup since kernel pages are never swapped out.

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?
The kernel uses the address space structure to keep track of the virtual address space, which contains the segments array (the main structure keeping track of the space). These structures are described in question 1. Each address space has information about the segments from the elf file, and the stack segment. These segments then contain the page tables, so the address space keeps information about the current status of each virtual page. The address space also contains a vnode for the elf file, allowing for demand paging from the elf file. The number of segments is also stored.

Question 6: When your kernel handles a TLB miss, how does it determine whether the required page is already loaded into memory?
First the kernel checks the address space to find the segment that the BADVADDR belongs to. After finding the segment, it checks the segments page table for the physical frame the page occupies. If this is set to be a non-physical frame number, we know that the frame is not in memory, otherwise we have the frame that the page is loaded in.

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?
First trap.c calls vm_fault, which determines if the address of the fault is a valid address, after the segment is determined from the address space with the given fault address. Finally, pt_page_in(faultaddress, s) is called with the fault address, and the segment that the fault address belongs in. Inside the pt_page_in function we first check to see if the sfn in page_details is not -1 if so, swap in from swap file. Otherwise, we check to see if the page detail is valid, and the physical frame number is not -1, then we just add the page into the TLB from the given physical frame number, and finally, we will load the page from the elf file.

Question 8: How does your kernel ensure that read-only pages are not modified?
Inside the segments, and page tables, the writable/dirty property is defined, and whenever adding tlb entries for the page, we always use this bit to decide if we set the dirty flag. as such if the page is modified, a read-only fault will be generated and we can kill the process.

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?

We don't have an explicit swap file structure. However, we keep that information in the page_table structure with the "sfn" swap frame number field. We are only recording the sfn because only this information is required to swap back to memory, as well as keeping the proper swap index when information is swapped out to disk. There is a global free list in swapfile.c that keeps track of free pages. Interrupts are disabled when searching for a free page to avoid synchronization issues. Also, when swapping out a core, the "sfn" for that page is set to -2 until we're done writing to the swap, which means that if an application requests that page while it's being swapped, it will yield until it's done swapping. This prevents memory from being loaded twice and freed once.

THE SWAP SIZE CAN BE CHANGED BY CHANGING THE #DEFINE SWAP_SIZE LINE AT THE BEGINNNING OF SWAPFILE.C (not swapfile.h, but swapfile.c)

Question 10: What page replacement algorithm did you implement? Why did you choose this algorithm? What were some of the issues you encountered when trying to design and implement this algorithm?

We used the clock algorithm for the page replacement algorithm, we choose this algorithm because it was unique, conceptually simple, mildly efficient, and at the same time interesting and easy to implement. The main issue we came across during design and implementation, was trying to use a modified bit to improve our clock algorithm performance. We found this would have required strategically setting tlb entries to read only, even when they were write-able, on fault checking the page table for readability, and then setting the modified bit before re-adding the tlb entry as writeable. This proved overly complex and time consuming, so we decided to go with the simpler clock algorithm.