We were unable to get swapping working. However, all the code for managing the swap file and most of the code for marking page table entries as swapped out has been written.

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?

Our virtual address space structure contains 3 important parts:

1) A pagetable structure, which is a two-level page table for the virtual address space.
2) A vnode, corresponding to the ELF file that contains the currently running program.
3) An array of region structures. A region corresponds to a region defined in the ELF file. The region is used to determine if a virtual address is supposed to be loaded from the ELF file.

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

In our page table, each page entry takes up 32-bits. This entry needs to contain the page frame number, which takes up only 20-bits. There are 12 bits left in each entry to play around with, so we used these bits to flag details about the page. The three relevant flags are PAGE_FREE, PAGE_IN_MEM and PAGE_IN_SWP.

PAGE_FREE: The page has not been touched yet (and thus is not loaded into memory)

PAGE_IN_MEM: The page is loaded into memory and the upper 20-bits correspond to its page frame number.

PAGE_IN_SWP: The page has been loaded into memory before, but is out of memory and in the swap file right now.

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?

There are 2 cases here:

1) The required page has not been touched yet and must be loaded from the ELF file. To detect this case, the address space keeps track of all the regions defined when load_elf is originally called. Whenever a TLB miss occurs and the desired page is flagged as PAGE_FREE (see Question 6), we check all the defined regions to see if the fault address overlaps with any of them. If it does, the corresponding region structure contains the details required by load_segment to load the data from the ELF file. It then calculates the offsets into the file and the size to load (which is upper bounded by PAGE_SIZE) and calls load_segment.
2) The required page is in the swap file. Before a page is swapped out, a function gets called which updates its page table entry. The upper 20-bits (which previously corresponded to the page frame number) are used to store the offset into the swap file that the page's data is being stored at. When an address within this page is used, we read its flags (in the lower 12-bits of the entry) and realize the PAGE_IN_SWP flag is set. To get the page back, we ask coremap for a page and then load the contents into the page from the defined offset in the swap file.

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

Only 3 of the 12-bits in the page table entry have been used up. We used 3 more bits as flags for read, write and executable. When a TLB miss occurs, we check to see if the writeable flag is set. If it is then when we add it to the TLB we set the TLBLO_DIRTY flag to false. This causes an EX_MOD exception when someone tries to change the contents of the page. The entry is loaded from the page table and the bits are checked to make sure that it isn't writeable. If it isn't then we return an error immediately.

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.