1: i am using a “coremap” data structure to manage all the physical memory. Specifically, the “coremap” data structure contains a mapping to the “logical page”, a status bit(either free or used), a occupation bit(kernel,user or unknown) and a “last” bit used to handle contiguous frames. When the vm system is initialing, first i use ram_getsize() to get the available memory range, and use this info to calculate total physical frames. Then i don’t use kmalloc() to build coremap, instead, i manually allocate memory for it. Because after ram_getsize() is being called, ram_stealmem() won’t work. So coremap must be manually initialized. Also, we must ensure that the memory which allocated by coremap structure cannot be touched by any program. Last we set every coremap entry to a initial value.

2:when a single physical frame needs to be allocated, coremap_alloc_one_page() will be called, it searches the entire coremap entries from top to down, trying to find a entry whose status is free. If we find that entry, we mark it as “used” and return the corresponding physical address. When the physical frame needs to be freed, coremap_free() will be called, it finds the beginning entry and will fresh everything in that coremap entry, repeat this process, until detecting the “last” bit.

3: If the system requests multiple physical frames, coremap_alloc_multi_page() will be called. Since i implement on-demand loading, i am pretty confident that it is kernel who requested contiguous physical frames. To be more specific, user program can request at most one page at a time. To allocate contiguous frames, kernel searches the entire coremap entry from down to top to try to find a requested contiguous frames. if fail to find, kernel will perform eviction: the frame used by kernel cannot be evicted; the frame used by user program can be evited. If there is such a eviction, kernel will also do a TLB invalidation.

4: there are some synchronization issues. For example, when the kernel has found a free coremap entry, and is trying to occupy, there could be some other program also requesting memory, and coincidently they choose the same entry. At this time the kernel is not consistent and could be crash. So we must ensure that only one program can access coremap_alloc_* functions at a time. In my implementation, it is protected by coremap_lock.

5: i am using segmentation to describe the virtual address space of each process. Currently my kernel only supports three segments(code,data,stack). In each segment, it contains the permission info(read,write,execute) ,segment virtual address range (vbase and vtop) and a logical page table. For every page table, it contains a valid bit, a mapping to physical address, the location of the swap file and the virtual address(it is used when swap out the page. kernel needs this info to invalidate TLB)

(6)(7): on a TLB miss, the vm_fault() will do the following things:
1 : determine which segment this fault address resident in.
2: computer the index of the page table in that segment
3: check whether this entry has been initialized
If yes, go to 4
If no, allocate a new page entry, set it as valid ,go to 4
4: check whether the entry is valid
If yes, we just update TLB, finish
If no, we understand this page is currently resident in swap file. kernel allocates a fresh page and swap the page in it. Then we set it as valid, and update TLB, finish

the valid bit in every logical page is responsible to determine whether the required memory is already loaded. (1 means in memory, 0 means in swap file).

Every page data structure contains the info(off_t swap_loc) that if the page is not resident in memory, the position we can swap it out in the swap file.

8: as i described in question 5, each segment contains the permission info. Generally, when as_prepare_load() is running, we must ensure that every segment is writeable, because this is the first time we load program data. When as_complete_load() is running, we then reinforce the permission(by setting the code segment as read-only) so that if we get a TLB miss again, we will not issue a dirty bit to that TLB entry. After that, any behaviors trying to modify code segment data is not permitted. If there is such a behavior happens, a “vm_fault_readonly” will arise and that process will be killed.

9:i am using a “bitmap” data structure to manage swap file. As i described in question 5, the location of the swap file will be recorded in page data structure. We need this info so that when we trying to read from swap file, we can know the exact position so that we get the right page. Also, the bitmap is used to monitor the usage of swap file, If bitmap finds that there is no enough space to swap out, it will panic because we have limited swap space. There are also synchronization issues. For example, if two programs are trying to request a swap location in bitmap, they could coincidently find the same position, then at least one page info will be permanently lost. So we must ensure that only one program can request a swap location at a time. Swap_alloc() is protected by swap_lock.

10: i am using a likely-FIFO algorithm. It is not strictly FIFO because we can not swap out kernel page. If the function detects that the next victim is a kernel page, it will skip that page and find the next victim. It is not a very good choice because its performance is not good. There are some issues. Form example, we must ensure that only one program can request to get a victim and evict it.