Part 1

In part 1, we had to implement demand paging. The pgtPrint from the previous assignment was used to print the page table. To implement demand paging, the following changes were made:

1. In exec.c, while loading the program into memory, for every segment, we only allocate filesz amount of memory instead of memsz amount of memory.
2. We increase sz by memsz. This is to get the virtual addresses of segments in the program correct.
3. In trap.c, we add a handler for page faults. The trap number is 14, but there is also a variable defined T_PGFLT set to 14.
4. In the trap handler, we check if the address being accessed (given by rcr2()) is a valid address, i.e. its value should be less than process size.
5. If the address is valid, we allocate a new page using kalloc and we map this page into the page directory of the process using the mappages function. mappages function is present in the vm.c file. I made it not static and added it to the defs.h file. So I am able to use it in traps.c file.
6. I created mydemandPage.c file as required and added _mydemandPage to UPROGS in makefile.
7. After making these changes, I was able to obtain the desired result as described in the problem statement.

Learnings:

I learnt how a process is created from a file. I learnt about the ELF file which is an intermediate representation of code which can be loaded into memory. I learnt about how traps are handled and how memory is allocated to a process. I also learnt about allocating memory in kernel mode using kalloc function.

Part 2

In part 2, we had to implement copy-on-write. I undid changes from demand paging and started afresh. To implement copy-on-write, the following changes were made:

1. In the vm.c file, in the copyuvm function, a new physical page was being created for every page present in the parent process using the kalloc process. This new page was being mapped to the page table of the new process. Instead, I mapped the same physical page to the page table of the new process and marked the pages as not-writable in both parent and child processes.

2. In the kalloc.c file, we need to keep track of how many times we allocate a physical page. After a process exits, the kernel frees the physical memory as well. Because of this, if a child exits, the parent might not be able to access the physical memory. Thus we need to modify kernel functions such that we free physical memory only if no process is referring to that particular physical page.
3. In the kalloc.c file, a struct is maintained called kmem. This refers to free pages available for allocation. To this struct, I also integrated a count variable that maintains how many processes are referring to a particular physical page. I wrote two functions: increase and decrease that update the kmem.count variable. (Actually kmem.count is an array of length same as the number of available physical pages).
4. In kalloc, we set the count to 1. In copyuvm, we increase the count. kfree is used to free kernel memory. If count is 1, we free the page otherwise we only decrease the count. To make increase and decrease available outside kalloc.c, these functions were added to the defs.h file.
5. In trap.c file, we handle page faults to implement copy-on-write. When a page fault occurs, we check if the accessed address is less than process size. If so, we create a new physical page. We get the pointer to the page table entry and modify its value to the physical address of the new physical page. We decrease the kmem.count value by 1 for the old physical page.
6. In both the trap handler and copyuvm, we need to clear the translation lookaside buffer for the process for which we are changing page table entries. This is done using lcr3 function.
7. myCOW.c file is written to test this functionality. The output is as expected. _myCOW was added to UPROGS in Makefile.

Learnings:

I learnt more about memory management in the kernel. Kernel effectively maintains a stack of free physical pages. I learnt about the importance of clearing the translation lookaside buffer. I learnt about how fork creates new processes. While debugging my code, I also learnt about how the kernel boots up in the main.c file and how the first init process is created. From there, the scheduler process is created. I learnt about how wait and exit functions work. I learnt about how the cpu scheduler works. To debug the kernel code, I also came across a remote gdb debugger and used that for debugging. I was able to set up the remote gdb connection, set breakpoints and go through the boot process of the kernel. I enjoyed the assignment and learnt a lot.