---- PART - 1 ---- USER PROCESSES ----

1) fork():
   The working of fork is as follows:
   a) Program calls fork() system call
   b) Kernel fork system call duplicates the process running the program.
   c) The kernel sets the return value for the system call for the original program and for the duplicate (PID of the duplicate and 0, respectively)
   d) The kernel puts both processes in the scheduler queue
   e) As each process is scheduled, the kernel 'returns' to each of the two programs.

   FUNCTIONS AND DATA STRUCTURES (made for fork()):
   a) fork_execute(): (userprog/process.c)
      similar to process_execute(), but instead of allocating the page to arguments, it uses the name of the parent process.
   b) fork_process(): (userprog/process.c)
      following data structure is passed into fork_process:

```
struct aux_fork{
   char *file;
   struct intr_frame *f;
   char *stack_ptr;
   char *stack;
   size_t size;
};
```

      - file contains the file to be loaded.
      - f contains the interrupt frame for parent process, which is passed to intr_exit.
      - stack pointer has the stack pointer of parent thread in its address space.
      - stack contains the entire stack of the parent.
      - size contains the size of parent stack.
   c) fork_create(): (thread/thread.c)
      similar to thread_create(), and also sets up the above struct aux_fork to be pass to fork_process, and adds fork_process to execute in ready_queue.

2) exec(char *filename):
   The working of exec is as follows:
   a) calls process_execute to run the new process.
   b) exits the current process.

3) Extra features implemented for proper testing:
   a) write system call:
      Implemented the write system call to print to console in user mode.
   b) Argument passing:
      Implemented passing arguments to user programs by setting up the process stack. (Hints taken from stanford assignments.)
      Since this was not a necessary feature to implement, only arguments
      of length upto 15(including filename) is implemented.

---- TEST CASES FOR USER PROCESSES ----
All test cases are in examples directory
1) echo.c: Tests the most basic working of fork() and exec() where in parent process creates only 1 child process.
2) calc_f.c: Solves the question in midsem on fork which takes 1 argument n and 'execs' compute_f from 0-n
3) compute_f.c: Prints a simple function value on argument (called by calc_f)
4) triple_fork_sh.c: Forks 3 times and calls compute_sh to completely test fork()! It also tests shared memory (explained later).

==

---- PART - 2 ---- VIRTUAL MEMORY USING PURE DEMAND PAGING ----
The basic working of virtual memory via pure demand paging is as follows:
1) The process gets loaded into the swap store instead of being loaded in the user pool directly.
2) When page fault occurs the faulting address is checked in the supplementary page table for the process. If it is there then a frame is allocated from the user pool and the page from the swap store corresponding to the faulting page is loaded into it.

Following data structures, files and functions were made/modified to implement the above.

Data Structures made:
1) Following entry is added to struct thread: struct list sup_list for storing the supplementary page table(list) for each process.
   - it is a list of following structure (in vm/sup_table.h):
2) struct sup_entry{

```
     uint8_t *page_no;
     uint8_t *kpool_no;
     bool writable;
     bool stack_page;
     bool shared_mem;
     struct list_elem elem;
   };
```

   - page_no stores the virtual address in the user space of page.
   - kpool_no stores the virtual aggress of above page in swap store/ backing store.
   - stack_page stores if the above page is for storing the stack
   - shared_mem stores if this page was added for shared memory

Files and functions made/modified for Virtual memory using pure demand paging:
1) vm/sup_table.c:
   Following functions are defined in it:
   a) swapspace_init(): - called in init.c during initialization.
      - initialises the swap store by allocating 64 pages from kernel pool.
      - initializes the swap pool (similar ro kernel and user pool in structure)
   b) swap_get_page(): - get a page from swap store initalized above
   c) swap_free_page(): - free the page from the swap store
   d) init_pool(): - to initialize the swap pool in swapspace_init

2) load_segment2(): (userprog/process.c)
   Following changes are made in load_segment2 to implement pure demand

```
    paging (this function is used instead of load_segment() in load()):
    a) Remove the line that allocates page from user pool and replace it
     to allocate page from swap pool.
    b) Hence the process is loaded in swap store.
    c) Make an entry of struct sup_entry (defined above) and add it to the
     supplementary page table (sup_list) of the thread.
    d) Remove the line that installs the page to the page directory (page
     is installed to page directory when page fault occurs)

 3) setup_stack(): (userprog/process.c)
    Does same tasks as load_segment but also sets the stack_page entry of
    structure above to be true.

 4) page_fault(): (userprog/exception.c)
    - It checks if there exists an entry in the supplementary list for the
      page for which fault occured. If there exists an entry then it loads
      the page from the backing store into a page allocated from the user
      pool over here and modifies the frame table accordingly (frame tables
      are not used as page replacement is not to be done.
    - page is installed in the page directory for the process.


 5) process_exit(): (userprog/process.c)
    - It removes the page from the frame table.
    - It frees the pages from the swap slot used in the swap store by this
      user process.

 ---- TEST CASES FOR VIRTUAL MEMORY VIA PURE DEMAND PAGING----
 - echo.c, compute_f.c, compute_sh.c, calc_f.c, and triple_fork_sh.c defined
   above in examples directory use virtual memry via pure demand paging to
   get executed.
 - This can be checked by replacing the code in userprog/exception.c of
   function pagefault() to its original version which page faults as soon as
   a user memory address is referered.

==============================================================================

---- PART - 3 ---- SHARED MEMORY ----
 Following data structures are defined for implementing shared memory:
 1) Following entry is added to struct thread: bool shared_mem to check if
    it has opened a shared memory.
 2) global varriable void *shared_mem in vm/shared_memory.c to store the
    location of globally allocated shared memory.

 Following File is made for implementing shared memory: vm/shared memory.c
 It implements the following functions:
 1) shared_memory_open_sys(): (vm/shared_memory.c)
    - called by syscall_handler when a system call for it is executed.
    - gets a free page from the user address space for the process using
      get_user_page() (defined below)
    - allocates a page from user pool to shared_mem (global variable) if
      it was not assigned any page before.
    - adds an entry for this page in the supplementary table(list) for the
      process (sup_list defined above).
    - install the page (returned by get_user_page) to shared_mem in the
      pagetable.

 2) shared_memory_close_sys(): (vm/shared_memory.c)
```

- called by syscall_handler when a system call for it is executed or it thread is to be exited and user has forgotton to close the shared memory.
- searches for the page in supplementary table which has shared_mem element to be true, removes that element from the supplementary table and removes its corresponding frame from the page table of the process.

3) get_user_page(): (vm/shared_memory.c)
   - gets a page from the user address space which is not present in supplementary table and returns it.

---- TEST CASES FOR SHARED MEMORY----

1) triple_fork_sh.c: Forks 3 times and 'execs' compute_sh, opens shared memory and puts a string over there without using any system call.
2) compute_sh.c: Opens the shared memory and prints the string in it.

===