---- 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) struct semaphore fork\_sema is defined in struct thread to block the

forking thread untill its address space is copied by forked thread.

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:

(userprog/process.h)

struct aux\_fork{

struct intr\_frame \*f;

struct thread \*t;

};

- f contains the interrupt frame for parent

process, which is passed to intr\_exit.

- t is the thread which called fork whose user

address space is to be copied.

c) load\_addr\_space(): (userprog/process.c)

The forking thread is passed to it.

All the user address space mappings from supplementary

table of forking thread are copied to it. It does the

same work as load does to start the process.

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.