OPERATING SYSTEM LAB 3

GROUP 35

Name	Roll Number
Mohan Kumar	180101042
Saurabh Baranwal	180101072
Rahul Krishna	180123035
Pankaj Kumar	180123031

PART A:

Two steps need to be done

1). Comment out the **growproc()** and

2). Add an extra line of code i.e. **myproc()->sz += n**;

Since we have removed growproc() it does not allocate memory. However, increasing the size of myproc() by n will make it believe that it has the memory requested and will results error on 'echo hi' as our process will unable to locate memory.

Lazy allocation:

- In the file trap.c -> in the function trap() we need to check the Page Fault error by tf->trapno == T_PGFLT, which is shown in the below image.
- rcr2() function gives the memory address of page fault.
- We need to declare lazy_page_allocate() function which takes memory address as an argument and allocate memory at that address.
- Since we need to use mappage() function in this lazy_page_allocate() we will define lazy_page_allocate() function in vm.c file where mappage() function is defined and also declare it in defs.h file and call it in trap.c file.

```
case T_PGFLT:
  //rcr2() function gives the memory address where the page fault occur
  if (lazy_page_allocate(rcr2()) < 0) {
    myproc()->killed = 1;
  }
  break;
```

```
//=======================================new function to allocate lazy page
int lazy_page_allocate(uint addr) {
  uint a = PGROUNDDOWN(addr);
  char *mem = kalloc();
  if (mem == 0) {
    return -1;
  }
  memset(mem, 0, PGSIZE);
  if (mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_U|PTE_W) < 0){
    return -1;
  }
  return 0;
}
```

PART B:

TASK-1 CREATING KERNEL PROCESS

For creating a Kernel process, we write the following function **createInternalprocess** in the file proc.c. Its respective declaration is also added in file defs.h

```
void createInternalProcess(const char *name, void (*entrypoint)()){
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
      cprintf("createInternalProcess error in allocproc\n");

   // Copy process state from p.
  if((np->pgdir = setupkvm(kalloc)) == 0)
   cprintf("createInternalProcess error in setupkvm\n");

//Set up the trap frame
  memset(np->tf, 0, sizeof(*np->tf));
  np->tf->cs = (SEG_UCODE << 3) | 0;
  np->tf->ds = (SEG_UDATA << 3) | 0;
  np->tf->es = np->tf->ds;
  np->tf->ss = np->tf->ds;
  np->tf->eflags = FL_IF;

  np->sz = initproc->sz;
  np->parent = initproc;
  *np->tf = *initproc->tf;
  // Clear %eax so that fork returns 0 in the child.
  //np->tf->eax = 0;
  // Set starting point of inswapper
  //np->cwd = idup(initproc->cwd);
  np->cwd = namei("/");
//Set the context eip to entrypoint given as parameter
  np->context->eip = (uint)entrypoint;
//add it to the processes queue by setting it runnable
 np->state = RUNNABLE;

//copy the name of kernel process (given as parameter) to the structure's name
 safestrcpy(np->name, name, (strlen(name) + 1));

}
```

Implementation of createInternalprocess() is similar to that of fork(), allocproc(), and userinit(). It has the same structure of fork() but there are some differences. fork() copies the address space, registers, etc from the parent process but createInternalprocess() does not do this. In places where fork() copies data from the parent process, createInternalprocess() sets up the data from scratch the same way allocproc() does.

Firstly, we set the entire trap frame in the function just like userinit() function does (by several lines of code as evident in above snippet). At the end of createInternalprocess(), WE set np->context->eip to the entrypoint function pointer that was provided in the parameter WHICH means that the process will start running at the function entrypoint when it starts.
Lastly, we also copy its name (given as argument to the function) to the process structure parameter. Also to add the process to the processes queue, we set the state of the process to be RUNNABLE.

TASK-2 & TASK-3 SWAP IN, SWAP OUT MECHANISM

In order to implement the swapper, we define two extra process states on our own: **SLEEPING_SUSPENDED** and **RUNNABLE_SUSPENDED.**

A process being blocked should be immediately swapped out by the kernel to the disk and have his state changed to sleeping _suspended. When a sleeping _suspended process is no longer blocked, his state should change to runnable_suspended. At this point, having a runnable state, the process can be selected by the scheduler to run. Nevertheless, it cannot run because its memory is swapped out (stored at the disk and not at the main memory). Thus, it first must be swapped back to the main memory and only then it can be scheduled to run.

Further, we also make changes to the proc structure in proc.h. We add the following things:

```
  struct file* swapped_file;    // Process swappped file
  int swapped_file_fd;         // Swapped file file descriptor
  int swapped;           // 1 => proc was swapped, else 0
```

For the swapper to function continuously, we define the following Kernel process: ***inswapper*** by creating it using the function call createInternalprocess() just after the userinit() function has ended.

```
void inSwapper(){
  release(&ptable.lock);
  for(;;){
      struct proc *p;
      acquire(&ptable.lock);
      //cprintf("inSwapper\n");
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

//Run through the process table and find any process that has been swapped out and
// has its state to be Runnable SUspended. In that case it has to be swapped in
      if(p->state == RUNNABLE_SUSPENDED && p->swapped){
    //cprintf("calling swapIn process %d\n",p->pid);
    swapIn(p);
    p->swapped = 0;
    p->state = RUNNABLE;
      }
      }
//Change current process state to SLEEPING now
```

```
      proc->state = SLEEPING;
      //cprintf("inSwapper finished proc->pid %d\n",proc->pid);
//Rerun the scheduler
      sched();
      release(&ptable.lock);
  }
}
```

Note here that for swapping purposes, we have defined two functions: **swapOut** and **swapIn** in file proc.c : The description of the code can be understood through the comments.

SWAPOUT:

```
void swapOut(struct proc* p){
  //create swap flie
  char id_as_str[3]; // need to pre determine number of digits in p->pid
  itoa(p->pid,id_as_str);
  char path[strlen(id_as_str) + 5];
  strcat(path,0,id_as_str,".swap");
//Assign the created swap file to the corresponding proc structure
  p->swapped_file = kernel_open(path,O_CREATE | O_WRONLY);

  pte_t *pte;
  int i;
  uint pa;
  for(i = 0; i < p->sz; i += PGSIZE){
//Check if a page table entry exists
      if((pte = walkpgdir(p->pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");

//check if page is present
      if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
      pa = PTE_ADDR(*pte);
      //cprintf("p->swapped_file %d\n",p->swapped_file);
//Do file write
      if(filewrite(p->swapped_file,p2v(pa),PGSIZE) < 0)
      panic("filewrite: error in swapOut");
  }
//Close the swapped file now
  int fd;
  for(fd = 0; fd < NOFILE; fd++){
      if(p->ofile[fd] && p->ofile[fd] == p->swapped_file){
      fileclose(p->ofile[fd]);
      p->ofile[fd] = 0;
      break;
      }
```

```
  }
  p->swapped_file = 0;
//Since it has been swapped out, set swapped variable to be 1
  p->swapped = 1;

//Deallocate memory
  deallocuvm(p->pgdir,p->sz,0);

//Once it has been swapped out, set the state to be sleeping suspended
  p->state = SLEEPING_SUSPENDED;

}
```

SWAP IN

```
void swapIn(struct proc* p){
//   cprintf("swapIN\n");
  //create flie
  char id_as_str[3]; // need to pre determine number of digits in p->pid
  itoa(p->pid,id_as_str);
  char path[strlen(id_as_str) + 5];
  path[6] = '\0';
//   path[0] = '/';
  strcat(path,0,id_as_str,".swap");
  //cprintf("swapIn - passed strcat path: %s\n",path);
  release(&ptable.lock);
  int test;

  p->swapped_file = kernel_open(path,O_RDONLY);
//   p->swapped_file = p->ofile[p->swapped_file_fd];
//   cprintf("swapIn - passed open pid %d p->sz %d\n",p->pid,p->sz);
  p->pgdir = setupkvm();

//ALLOCATE SPACE FOR A PAGE IN THE PAGE DIRECTORY
  test = allocuvm(p->pgdir,0,p->sz); //changed from KERNBASE
//   cprintf("swapIn - passed allocuvm pid %d returned %d\n",p->pid,test);
//   cprintf("swapFile ip: %d\n",p->swapped_file->ip->size);
//LOAD THE CORRESPONDING SWAP FILE IN THE PAGE DIRECTORY USING THIS FUNCTION
  test = loaduvm(p->pgdir,0,p->swapped_file->ip,0,p->sz);
//   cprintf("swapIn - passed loaduvm pid %d returned %d\n",p->pid,test);
  test++;
  int fd;
//CLose the swap file corresponding to it first
  for(fd = 0; fd < NOFILE; fd++){
      if(p->ofile[fd] && p->ofile[fd] == p->swapped_file){
      fileclose(p->ofile[fd]);
      p->ofile[fd] = 0;
```

```
      break;
      }
  }

//Since it has now been swapped in, remove the swa
p file corresponding to it
  p->swapped_file = 0;
//   cprintf("swapIn - passed fileclose pid %d\n",p->pid);
  test = kernel_unlink(path);
  //test++;
//   cprintf("swapIn - passed kernel_unlink pid %d returned %d\n",p->pid,test);
  acquire(&ptable.lock);
}
```

The above two functions are used to **swapIn and swapOut** pages, which helps us in designing the page replacement technique. The process to be swapped out can be chosen using an LRU mechanism, to implement the LRU page replacement technique.

TASK-4

We write the following program memtest.c . In it, we have created 20 child processes, and in each process you have 10 iterations where in each iteration a memory of 4KB is allocated using the standard malloc() function. We introduce the **while(wait()>=0)** condition at last so that the swapper can work accordingly and swap in and swap out pages if required.
The code along with commented description is below:

```
#include "types.h"
#include "stat.h"
#include "user.h"

void
child_process(void)
{
  int *mem;
  // int i;
  //Run 10 iterations and in each iteration allocate 4 KB of memory using malloc
  for(int i=0; i<10; i++){
      mem = malloc(4096); //ALlocate memory of 4KB
      if (mem == 0){
      printf(1, "Out of memory bound. We must exit now\n");
      exit();
      }
  }
  mem[0] = 1;
  printf(1, "%d\n", mem[0]);
  exit();
```

```
}

int
main(int argc, char* argv[])
{
  int i, pid;
  int pids[20]; //Array to store process ids of 20 children that will be forked

  // Fork 20 children.
  for (i = 0; i < 20; i++) {
      pid = fork();
      if (pid == 0){
      child_process(); //Call the child process function
      }
      pids[i] = pid; //Set the pid value
  }

  printf(1, "first child pid: %d\n", pids[0]);

  while(wait() >= 0); //Run while loop till the wait() call returns value >=0
  exit();
}
```
