ASSIGNMENT 3

Group 2:-
1)Ritish Bansal 190101076
2)Suryansh Singh 190101089
3)Anant Shankhdhar 190101011
4)Mayank Chandak 190101052

Part A

Lazy Memory allocation

For this part of assignment mainly 3 files were modified:

1) sysproc.c - The **sbrk()** function was changed so that physical memory was not allocated before it was needed. When any process needs physical memory then only it is allocated. We commented out the function which allocates physical memory and also increased **myproc()->sz** by n.

```
int
sys_getpid(void)
{
  return myproc()->pid;
}

int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  myproc()->sz += n;

  //  if(growproc(n) < 0)
  //    return -1;
  return addr;
}
```

2) trap.c - Here we add condition to detect page fault in the switch section. As when a page fault occurs **tf->trapno** equals **T_PGFLT.** Here to handle this we added a function **PageFaulthandle()**. This function handles all the requests for page faults. **kalloc()** and **mappages( )** were used to allocate required space or to inform user that no more memory is available.

```
case T_PGFLT:
  if(PageFaulthandle()<0){
      cprintf("Memory Not Allocated!!!");
  }
break;
```

```c
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

int PageFaulthandle(){
  int addr1=rcr2();
  int addr2 = PGROUNDDOWN(addr1);
  char *mem=kalloc();
  if(mem!=0){
    memset(mem, 0, PGSIZE);
    if(mappages(myproc()->pgdir, (char*)addr2, PGSIZE, V2P(mem), PTE_W|PTE_U)<0)
      return -1;
    return 0;
  } else
    return -1;
}
```

3) vm.c - Return type of **mappages( )** was changed from static int to int so that it can be used in trap.c

```c
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;
```

Answers to some Question

Q1 How does the kernel know which physical pages are used and unused?
Ans - kernel maintains the list of free pages in **kalloc.c** called **kmem**.

Q2 What data structures are used to answer this question?
Ans - A linked list is used as data structure for storing free pages.

Q3 Where do these reside?
Ans - Linked List is declared inside **kalloc.c** inside structure **kmem**.

Q4 Does xv6 memory mechanism limit the number of user processes?
Ans - Number of user processes are limited to 64 as defined by **NPROC** in **param.h**

Q5 If so, what is the lowest number of processes xv6 can 'have' at the same time?
Ans - Minimum number of processes in xv6 can be 1 as while starting only one process named initproc which initiates all other user processes.

Part B

Task 1

The **create_kernel_process()** function was created in **proc.c.** It took the name of the process and entrypoint function as arguments. It always remains in kernel mode. The parent process was set to **initproc** and **p->context->eip** was set to entrypoint argument and all other values as default. **allocproc** allocates the process a spot in the ptable. **setupkvm** maps the virtual address to physical address (from 0 to PHYSTOP).

```
void create_kernel_process(const char *name, void (*entrypoint)())
{

  struct proc *kp  = allocproc();
  if(kp == 0){
    panic("Failed to allocate kernel process.");
  }
  kp->pgdir = setupkvm();
  if(kp->pgdir == 0)
  {
    kfree(kp->kstack);
    kp->kstack = 0;
    kp->state = UNUSED;
    panic("Failed to setup pgdir for kernel process.");
  }
  kp->sz = PGSIZE;
  kp->parent = initproc;
  memset(kp->tf, 0, sizeof(*kp->tf));
  kp->tf->cs = (SEG_KCODE << 3) | DPL_USER;
  kp->tf->ds = (SEG_KDATA << 3) | DPL_USER;
  kp->tf->es = kp->tf->ds;
  kp->tf->ss = kp->tf->ds;
  kp->tf->eflags = FL_IF;
  kp->tf->esp = PGSIZE;
  kp->tf->eip = 0;
  kp->tf->eax = 0;
  kp->cwd = namei("/");

  safestrcpy(kp->name, name, sizeof(name));


  acquire(&ptable.lock);

  kp->context->eip = (uint)entrypoint;
  kp->state = RUNNABLE;

  release(&ptable.lock);

  return;
}
```

Task 2

We create a container to contain the processes who have asked for additional memory but do not have any free pages. Therefore, we implement a circular queue and functions to give entry (cq_push) and exit (cq_pop) from the circular queue in proc.c .

```c
struct circular_queue{
  struct spinlock lock;
  struct proc* queue[NPROC];
  int head;
  int tail;
};

// circular process queue for swapping out requests
struct circular_queue cq;
```

```c
struct proc* cq_pop(){
  acquire(&cq.lock);
  if(cq.head == cq.tail){
    release(&cq.lock);
    return 0;
  }
  struct proc *p = cq.queue[cq.head];
  cq.head = (cq.head + 1) % NPROC;
  release(&cq.lock);
  return p;
}
```

```c
int cq_push(struct proc *p){
  acquire(&cq.lock);
  if ((cq.tail + 1) % NPROC == cq.head){
    release(&cq.lock);
    return 0;
  }
  cq.queue[cq.tail] = p;
  cq.tail = (cq.tail + 1) % NPROC;
  release(&cq.lock);

  return 1;
}
```

We initialise the queue in userinit (user initialisation) function and lock for the queue in pinit function.

```c
void
userinit(void)
{
  acquire(&cq.lock);
  cq.head = 0;
  cq.tail = 0;
  release(&cq.lock);

  struct proc *p;
```

```c
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&cq.lock, "cq");
}
```

We want to use the circular queue globally therefore we give its definition in defs.h

```c
struct circular_queue;
```

```c
extern struct circular_queue cq;
int cq_push(struct proc *p);
int cp_pop();
```

Whenever a process needs to access some data it calls the **walkpgdir** function: If the data is not present in main memory, then **growproc** function is called which calls the **allocuvm** function which ultimately calls the **kalloc** function. If any free page is available then kalloc assigns it to the process else we need to swap out a page according to LRU policy to get a free page.

```c
    if(mem == 0){
      // cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);

      // SLEEP
      myproc()->state = SLEEPING;
      acquire(&sleeping_channel_lock);
      myproc()->chan=sleeping_channel;
      sleeping_channel_count++;
      release(&sleeping_channel_lock);

      cq_push(myproc());
      if(!swap_out_process_exists){
        swap_out_process_exists = 1;
        create_kernel_process("swap_out_process", &swap_out_process_function);
      }
      return 0;
    }
```

To swap out the page, we first need to move the process in the sleeping state on a special channel called **sleeping_channel.** We create this special channel in vm.c

```
struct spinlock sleeping_channel_lock;
int sleeping_channel_count = 0;
char *sleeping_channel;

// Set up CPU's kernel segment descriptors.
```

Then we declare it in defs.h to use it globally.

```
void                clearpteu(pde_t *pgdir, char *uva);
extern struct spinlock sleeping_channel_lock;
extern int          sleeping_channel_count;
extern char*        sleeping_channel;
```

When we have a free page (either already had or after swap out) we need to assign it to the process, for this processes sleeping on sleeping_channel need to be woken up by **wakeup()** system call.

```
    release(&kmem.lock);

  // wake up processes sleeping on a sleeping channel
  if(kmem.use_lock)
    acquire(&sleeping_channel_lock);
  if(sleeping_channel_count){
    wakeup(sleeping_channel);
    sleeping_channel_count = 0;
  }
  if(kmem.use_lock)
    release(&sleeping_channel_lock);
}
```

We now implement the swap_out_process to really swap out the page.
To determine the victim page using LRU policy, we iterate through each entry in the process page table and look at the accessed bit which is obtained by bitwise & of the entry and PTE_A. The access bit indicates whether the page was accessed in the last iteration or not.

```
#define PTE_PS          0x080   // Page Size
#define PTE_A           0x020   // Accessed
```

```c
void swap_out_process_function(){
  acquire(&cq.lock);
  while (cq.head != cq.tail){
    struct proc *p = cq_pop();

    pde_t *pd = p->pgdir;
    for(int i = 0; i < NPDENTRIES; i++){

      // skip page table if accessed
      if(pd[i] & PTE_A)
        continue;
      pte_t *pt = (pte_t *) P2V(PTE_ADDR(pd[i]));
      for(int j = 0; j < NPTENTRIES; j++){
        // skip if found
        if((pt[j] & PTE_A) || !(pt[j] & PTE_P))
          continue;
        pte_t *pte = (pte_t *) P2V(PTE_ADDR(pt[j]));

        // for file name
        int pid = p->pid;
        int virt = ((1 << 22) * i) + ((1 << 12) * j);

        // file name
        char c[50];
        itoa(pid, c);
        int x = strlen(c);
        c[x] = '_';
        itoa(virt, c + x + 1);
        safestrcpy(c + strlen(c), ".swp", 5);

        // file management
        int fd = proc_open(c, O_CREATE | O_RDWR);
        if (fd < 0){
          cprintf("Error creating or opening file: %s\n", c);
          panic("swap out process");
        }

        if(proc_write(fd, (char *) pte, PGSIZE) != PGSIZE){
          cprintf("Error writing to file: %s\n", c);
          panic("swap out process");
        }
        proc_close(fd);

        kfree((char *) pte);
        memset(&pt[j], 0, sizeof(pt[j]));

        // mark this page as swapped out
        pt[j] = pt[j] ^ 0x008;

        break;
      }
    }
  }
  release(&cq.lock);
```

```c
    release(&cq.lock);

    struct proc *p;
    if ((p = myproc()) == 0)
      panic("swap out process");

    swap_out_process_exists = 0;
    p->parent = 0;
    p->name[0] = '*';
    p->killed = 0;
    p->state = UNUSED;
    sched();
}
```

In the scheduler, we unset the accessed bit.

```c
      for(int i = 0; i < NPDENTRIES; i++){
        // if PDE were accessed

        if(((p->pgdir)[i]) & PTE_P && ((p->pgdir)[i]) & PTE_A){

          pte_t *pt = (pte_t *) P2V(PTE_ADDR((p->pgdir)[i]));

          for(int j = 0; j < NPTENTRIES; j++){
            if(pt[j] & PTE_A){
              pt[j] ^= PTE_A;
            }
          }
          ((p->pgdir)[i]) ^= PTE_A;
        }
      }
```

Now the swapped out page needs to be stored(written) in secondary storage for that we have copied open, write, close, functions from proc.c to sysfile.c and named as proc_open, proc_write, proc_close

```c
#include "spinlock.h"
#include "fcntl.h"
#include "stat.h"
#include "sleeplock.h"
#include "fs.h"
#include "file.h"
```

, etc.

```c
int
proc_close(int fd)
{
  struct file *f;

  if(fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
    return -1;
  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}

int
proc_write(int fd, char *p, int n)
{
  struct file *f;

  if(fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
    return -1;
  return filewrite(f, p, n);
}
```
.

We clear the stack of the kernel process while exiting from it.

```c
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      // if the swapped out process has stopped running, free its stack
      if(p->state == UNUSED && p->name[0] == '*'){
        kfree(p->kstack);
        p->kstack = 0;
        p->name[0] = 0;
        p->pid = 0;
      }
```

Task-3

When a process had requested a data which is not in main memory, i.e, in case of a page fault, we need to swap the page containing data into the main memory.
First we need to create a container to satisfy the swap-in requests, just like we did in task-2. We create a circular queue and functions to enter and exit the container in proc.c.

```c
struct circular_queue cq1;

int cq_push1(struct proc *p){
  acquire(&cq1.lock);
  if ((cq1.tail + 1) % NPROC == cq1.head){
    release(&cq1.lock);
    return 0;
  }
  cq1.queue[cq1.tail] = p;
  cq1.tail = (cq1.tail + 1) % NPROC;
  release(&cq1.lock);

  return 1;
}

struct proc* cq_pop1(){
  acquire(&cq1.lock);
  if(cq1.head == cq1.tail){
    release(&cq1.lock);
    return 0;
  }
  struct proc *p = cq1.queue[cq1.head];
  cq1.head = (cq1.head + 1) % NPROC;
  release(&cq1.lock);
  return p;
}
```

We initialise the queue in userinit (user initialisation) function and lock for the queue in pinit function.

```c
void
userinit(void)
{
  acquire(&cq.lock);
  cq.head = 0;
  cq.tail = 0;
  release(&cq.lock);

  acquire(&cq1.lock);
  cq1.head = 0;
  cq1.tail = 0;
  release(&cq1.lock);
```

```c
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&cq.lock, "cq");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&cq1.lock, "cq1");
}
```

We want to use the circular queue globally therefore we give its definition in defs.h

```c
extern struct circular_queue cq1;
int cq_push1(struct proc *p);
struct proc* cq_pop1();
```

We create an integer variable to store the virtual address where the page fault has occurred in proc.h .

```c
char name[16];               // Process name (debugging)
int va;                      //Virtual Address of the process
```

Whenever a page fault occurs the process traps the os, therefore to handle the page fault we add the following in trap.c

```c
  break;
case T_PGFLT:
  pfhandling();
  break;
```

In pfhandling function, we set the process in sleeping state and obtain the virtual address where the page fault has occurred. Then we check whether the page was swapped out or not. If not then allow the default way of handling page fault else call **swap_in()** function.

```c
void pfhandling(){
    struct proc *p=myproc();
    int va = rcr2();
    acquire(&swaplock);
    sleep(p,&swaplock);
    pde_t *pde = &(p->pgdir)[PDX(va)];
    pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

    if((pgtab[PTX(va)])&0x008){
        p->va = va;
        cq_push1(p);
        if(swap_in_process_exists==0)
        {
            swap_in_process_exists=1;
            create_kernel_process("swap_in_process", &swap_in);
        }
    } else {
        exit();
    }
}
```

Then we declare swap_in() in defs.h to use it globally.

```c
void            swap_out_process_function();
void            swap_in();

extern int      swap_out_process_exists;
extern int      swap_in_process_exists;
```

We have already implemented other file management functions in task-2. We here implement read_process which is basically copy function.

```c
int read_process(int fd, int n, char *p)
{
  struct file *fl;
  if(fd < 0 || fd >= NOFILE)return -1;
  fl =myproc()->ofile[fd];
  if(fl==0)return -1;
  return fileread(fl, p, n);


}
```

We allocate a free page to the process in main memory and read the page from secondary storage to the allocated free page in the main memory.

```
void swap_in(){
  acquire(&cq1.lock);
  while(cq1.head!=cq1.tail){
    struct proc *p=cq_pop1();
    int process_id=p->pid;
    int va=PTE_ADDR(p->va);
    char pagename[50];
      int_to_string(process_id,pagename);
      int length =strlen(pagename);
      pagename[length]='_';
      int_to_string(va,pagename+length+1);
      safestrcpy(pagename+strlen(pagename),".swp",5);

      int fd=proc_open(pagename,O_RDONLY);
      if(fd<0){
        release(&cq1.lock);
        cprintf("Page could not be found in memory: %s\n", pagename);
        panic("swap in failed");
      }
      char *mem=kalloc();
      read_process(fd,PGSIZE,mem);
      int x = PTE_W|PTE_U;
      int mp = mappages(p->pgdir, (void *)va, PGSIZE, V2P(mem),x );
      if(mp<0){
        release(&cq1.lock);
        panic("page mapping");
      }
      wakeup(p);
  }
  release(&cq1.lock);
  struct proc *p = myproc();
  if(p==0)
    panic("swap in failed");
  swap_in_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

Task-4:Sanity Test

We will create a user-space program to test our swapping mechanism.

20 child processes are created using fork().
10 4Kb blocks of memory is allocated for each process.
For a given process_id <pid>, block number <j> and offset <k>, the memory location stored in address field is pid*100000+j*10000+k.

```c
int main(int argc, char *argv[]){
    int *add_list[10];
    int id_list[100];
    int counter = 0;
    for (int i = 0; i < 10; i++)
    {
        if (fork() == 0){
            counter = counter + 1;
            for (int j = 0; j < 10; j++){
                int *addr = (int *)malloc(4096);
                int p_id;
                if ((add_list[j] = addr) == NULL){
                    p_id = getpid();
                    printf(1, "the process ID is: %d\n", p_id);
                    break;
                }
                p_id = getpid();
                id_list[counter] = p_id;
                for (int k = 0; k < 1024; k++){
                    *(addr + k)  = p_id * 100000 + j * 10000 + k;
                }
                if (j == 0)
                    printf(1, "block 1 index : %d, Beginning Address : %p , process ID : %d\n", counter, add_list[j], p_id);
                if (j == 4)
                    printf(1, "block 5 index : %d, Beginning Address : %p , process ID : %d\n", counter, add_list[j], p_id);
                if (j == 9)
                    printf(1, "block 10 index : %d, Beginning Address : %p , process ID : %d\n", counter, add_list[j], p_id);
            }
        }
        else break;
    }
    while (wait() != -1); // Execute all the child process
    if (counter == 0)
        exit();
    for (int i = 0; i < 10; i++){
        if (i == 0)
            printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 1st block: %d \n",  add_list[i], id_list[counter],*(add_lis
        if (i == 4)
            printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 5th block: %d \n",  add_list[i], id_list[counter],*(add_lis
        if (i == 9)
            printf(1, "Beginning Address : %p ,Process ID: %d, 100th value of the 10th block: %d \n",  add_list[i], id_list[counter],*(add_li
    }
    counter--;
    exit();
}
```

Output:

1) Details of block 1, 5, 10 are displayed on the console.
2) After all child processes stop executing the contents of memory locations are checked.

```
$ sanity
block 1 index : 1, Beginning Address : A000 , process ID : 4
block 5 index : 1, Beginning Address : 5FE0 , process ID : 4
block 10 index : 1, Beginning Address : FFF0 , process ID : 4
block 1 index : 2, Beginning Address : EFE8 , process ID : 5
block 5 index : 2, Beginning Address : 1A000 , process ID : 5
block 10 index : 2, Beginning Address : 14FD8 , process ID : 5
block 1 index : 3, Beginning Address : 13FD0 , process ID : 6
block 5 index : 3, Beginning Address : 1EFE8 , process ID : 6
block 10 index : 3, Beginning Address : 28FF8 , process ID : 6
block 1 index : 4, Beginning Address : 27FF0 , process ID : 7
block 5 index : 4, Beginning Address : 23FD0 , process ID : 7
block 10 index : 4, Beginning Address : 2DFE0 , process ID : 7
block 1 index : 5, Beginning Address : 2CFD8 , process ID : 8
block 5 index : 5, Beginning Address : 37FF0 , process ID : 8
block 10 index : 5, Beginning Address : 42000 , process ID : 8
Beginning Address : 2CFD8 ,Process ID: 8, 100th value of the 1st block: 800100
Beginning Address : 37FF0 ,Process ID: 8, 100th value of the 5th block: 840100
Beginning Address : 42000 ,Process ID: 8, 100th value of the 10th block: 890100
Beginning Address : 27FF0 ,Process ID: 7, 100th value of the 1st block: 700100
Beginning Address : 23FD0 ,Process ID: 7, 100th value of the 5th block: 740100
Beginning Address : 2DFE0 ,Process ID: 7, 100th value of the 10th block: 790100
Beginning Address : 13FD0 ,Process ID: 6, 100th value of the 1st block: 600100
Beginning Address : 1EFE8 ,Process ID: 6, 100th value of the 5th block: 640100
Beginning Address : 28FF8 ,Process ID: 6, 100th value of the 10th block: 690100
Beginning Address : EFE8 ,Process ID: 5, 100th value of the 1st block: 500100
Beginning Address : 1A000 ,Process ID: 5, 100th value of the 5th block: 540100
Beginning Address : 14FD8 ,Process ID: 5, 100th value of the 10th block: 590100
Beginning Address : A000 ,Process ID: 4, 100th value of the 1st block: 400100
Beginning Address : 5FE0 ,Process ID: 4, 100th value of the 5th block: 440100
Beginning Address : FFF0 ,Process ID: 4, 100th value of the 10th block: 490100
$
```

0xE000000

Results:-

1) When phystop=0xE000000, then we have all 20 processes in the output.
2) When phystop=0x0400000, then we have only 9 out of 20 processes in the output.

Explanation:-When we reduce the phystop then, we don’t have enough capacity to hold all the processes in the main memory.

NOTE:-

1) Part-A and Part-B are separately implemented on two different xv-6 directories.
2) Use <patch -ruN -strip -d xv6-public < patch_A.txt> to apply patch for A part.
3) Use <patch -ruN -strip -d xv6-public < patch_B.txt> to apply patch for B part.