Lab Report - Assignment 3

Group 9 MnC
Chandrashekhar Dharmarajan - 220123076
Srinjoy Som - 220123074
Shubham Kumar Jha - 220123081
Dhruva Kuthari - 220123015

For all the patch and zip files visit
https://drive.google.com/drive/folders/1CxvdZPrWMt1S5TQpP46P2QsKKw4HkIsj?usp=sharing

Part A : Lazy Memory Allocation

When a process in the xv6 operating system **requires additional memory** beyond its allocated limit, it signals this need using the **`sbrk` system call**. The `sbrk` function then invokes **`growproc()`**, which is implemented in **`proc.c`**. Upon inspection of **`growproc()`**, we find that this function relies on **`allocuvm()`** to fulfil the memory request. **`allocuvm()` handles the allocation of additional memory pages and updates the page tables by mapping the new virtual addresses to the corresponding physical addresses.**

Our goal is to implement Lazy Memory Allocation, where memory is allocated only when it is accessed, rather than immediately upon request. To achieve this, we **modify the `sbrk` system call** by **commenting out the call to `growproc()`**. Instead, we simply **update the `size` variable of the current process** to the requested value, creating the illusion that memory has been allocated. When the process later tries to access this memory (believing it has already been allocated), it **triggers a PAGE FAULT**, resulting in a **T_PGFLT trap to the kernel**. This is handled in trap.c by **calling Pfhandler().**

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

int Pfhandler(){
  uint addr=rcr2();
  uint rounded_addr = PGROUNDDOWN(addr);
  char *mem=kalloc();
  if(mem==0){
       return -1;
       }
  else{
    memset(mem, 0, PGSIZE);
    if(mappages(myproc()->pgdir, (char*)rounded_addr, PGSIZE, V2P(mem), PTE_W|PTE_U)<0)
      return -1;
    return 0;
  }
}

//PAGEBREAK: 41
```

`Description of Pfhandler()`

```c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpuid(), tf->cs, tf->eip);
    lapiceoi();
    break;
  case T_PGFLT:
    if(Pfhandler()<0){
      cprintf("PAGE ALLOCATION FAILED DURING AT %d\n",rcr2());
    }
  break;
  //PAGEBREAK: 13
  default:
    if(myproc() == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
```

Added case for T_PGFLT

In this scenario, the function **`rcr2()` provides the virtual address where the page fault occurred**. To find the beginning of the page where this virtual address lies, we compute **`rounded_addr`**, which **points to the starting address** of that page. After that, we call **`kalloc()`**, which **allocates a free page from the system's free memory pool**, managed as a **linked list called `freelist`** within the **kmem structure.** This free page is then used to resolve the page fault.

Once we have a physical page allocated, the **next step is to map it to the virtual address `rounded_addr`**. This is accomplished **using the `mappages()`** function. To call **`mappages()`** from within **`trap.c`**, we **first remove the `static` keyword** from its definition in **`vm.c`** and **declare its prototype in `trap.c`.**

`mappages()` takes several parameters: the page table of the current process, the virtual address where the mapping starts, the size of the data, the **physical address of the allocated page** (which we **obtain using the V2P macro** that converts a virtual address to a physical address **by subtracting KERNBASE**), and the permissions for the page table entry.

Now, let's explore the details of how **`mappages()`** works.

```c
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}

// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
```

Walkpgdir function

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

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

Mappages function

In this process, **'`a`' represents the first page** and **'`last`' represents the last page** of the data that needs to be mapped. **`mappages()` iterates through all pages** from the first to the last, **loading each one into the page table**. For every page, it uses **`walkpgdir()` to locate the corresponding page table entry (PTE).**

`walkpgdir()` takes two inputs: the page table and the virtual address. It returns the PTE associated with that virtual address by navigating through the **two-level page table structure**. First, it uses the **PDX macro to extract the first 10 bits** of the virtual address **to identify the page directory entry**, which points to the relevant page table. Then, it uses the **PTX macro to extract the next 10 bits to identify the specific entry within the page table**. The function then returns a pointer to the **PTE**.

If the page table already exists in memory, the function stores the pointer to its first entry in `pgtab`. It uses the **`PTE_ADDR` macro to clear the last 12 bits of the address, ensuring that the offset is set to zero**. If the page table is not present, it allocates it and sets the permission bits in the page directory. Once done, **`walkpgdir()`** returns the pointer to the **PTE** corresponding to the virtual address.

Back in **`mappages()`**, after retrieving the correct page table entry, the function checks if the **PRESENT** bit of that entry is set, which would indicate that the page is already mapped to a virtual address. If so, it raises an error indicating a remap has occurred. Otherwise, the page table entry is linked to the virtual address, permission bits are set, and the **PRESENT** bit is marked, signalling that the mapping has been successfully completed.

Here is a **stepwise implementation of Lazy Memory Allocation:**

Task 1 - Eliminate allocation from sbrk()

In this task we have made the following changes -

sysproc.c

```
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;
}
```

Output on calling **echo hi** cmd:

```
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
```

The “addr 0x4004” indicates that the virtual address that caused the page fault is 0x4004.

Task 2: Lazy Allocation

Changes made -

1) proc.h - Added uint oldsz; in struct proc
2) proc.c - p->oldsz = 0;

   Here, we initialised the above declared variable with zero value inside allocproc() function.

3) trap.c - extern int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

Here, we declared prototype of mappages in trap.c after removing static keyword for mappages function in vm.c

Output for working of **echo hi** cmd after Lazy Allocation:

```
init: starting sh
$ echo hi
hi
$ 
```

Output for working of **ls** cmd after Lazy Allocation:

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15464
echo           2 4 14348
forktest       2 5 8792
grep           2 6 18308
init           2 7 14968
kill           2 8 14432
ln             2 9 14328
ls             2 10 16896
mkdir          2 11 14456
rm             2 12 14436
sh             2 13 28492
stressfs       2 14 15364
usertests      2 15 62864
wc             2 16 15892
zombie         2 17 14012
console        3 18 0
$ 
```

Part - B

Q.1) How does the kernel know which physical pages are used and unused?

Ans)

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;
```

A linked list of free pages is maintained in **kalloc.c** called **kmem**. **kinit1** is called through **main()** which adds 4MB of free pages to the list.

Q.2) What data structures are used to answer this question?

Ans) A Linked List is used. A new structure named **struct run** is made in kalloc.c and used as a linked list node.

Q.3) Where do these reside?

Ans) They reside in **kalloc.c**, where **kmem** structure is instantiated. It contains a lock and linked list head.

Q.4 & Q.5) Does xv6 memory mechanism limit the number of user processes? If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

Ans) Yes, the maximum number of user processes that can be active simultaneously are 64, set by default. We can change it if we want.

At a time, lowest process running is 1, i.e. **sh.initproc** is initially made runnable, but later it sleeps continuously. After every command execution, the shell sleeps and again becomes runnable. So the lowest number is 1.

There cannot be zero processes after boot since all user interactions need to be done using user processes which are forked from **initproc/sh**.

Task-1: (kernel processes):

We have created the **create_kernel_process()** function in the **proc.c** file. We make a new process using **allocproc** after which we have not initialized the **trapframe** because we don't need to do it rather we set the instruction pointer of this process to be the entrypoint parameter taked which is basically a function pointer indicating the entrypoint of the function we intend to run. And then we set the state of the process to **runnable**.The **setupkvm()** function in the code initializes a page-table for the kernel process which will map virtual address (as it is a kernel process the virtual address space will be from **KERNBASE** to **KERNBASE+PHYSTOP**) to physical address (from **0** to **PHYSTOP**).

```c
void create_kernel_process(const char* name, void (*entrypoint)()) {
    struct proc* ker_proc = allocproc();

    // Check if process allocation succeeded.
    if (ker_proc == 0) {
        panic("could not create a kernel process");
    }

    // Try to set up the kernel page table.
    if ((ker_proc->pgdir = setupkvm()) == 0) {
        // Clean up if page table setup fails.
        kfree(ker_proc->kstack);
        ker_proc->kstack = 0;
        ker_proc->state = UNUSED;
        panic("setting up kernel's page table failed");
    }

    // Set the entry point for the process.
    ker_proc->context->eip = (uint)entrypoint;

    // Copy the process name.
    safestrcpy(ker_proc->name, name, sizeof(ker_proc->name));

    // Mark the process as runnable.
    acquire(&ptable.lock);
    ker_proc->state = RUNNABLE;
    release(&ptable.lock);
}
```

TASK 2 : (swapping out mechanism):

To do this task first we have implemented a queue using a **linked list**.We created structures called **swap_queue** and **swap_manager** to make the linked list and maintain head and tail of the queue respectively. Which is used as the queue which stores the processes that are requesting for memory allocation in physical memory.

```c
struct swap_queue{
        struct proc * pro ;
        struct swap_queue * next ;
} ;

struct swap_manager{
    struct spinlock lock ;
    struct swap_queue * head ;
    struct swap_queue * tail ;
} ;
```

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

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

```c
struct proc* smpop(){
    acquire(&sm.lock) ;
    if(sm.head == 0){release(&sm.lock) ; return 0  ; }
    if(sm.head == sm.tail){
        struct proc * p  = sm.head->pro  ;
        sm.head = 0 ;
        sm.tail = 0  ;
        release(&sm.lock) ; return  p ;
    }
    struct proc *p = sm.head->pro ;
    sm.head = sm.head->next ;
    release(&sm.lock) ;
    return  p ;
}
```

```c
void smpush(struct proc * p ){
        struct swap_queue sq ;
        sq.pro = p ;
        sq.next = 0 ;
        acquire(&sm.lock) ;
        if(sm.head == 0){
          sm.head = &sq ; sm.tail = &sq ;
          release(&sm.lock) ;
        }
        else{
            sm.tail->next = &sq ;
            sm.tail = &sq ;
            release(&sm.lock) ;
        }
        return  ;

}
```

We have used the **userinit** function to initialise the start and end of the queue whereas **pinit** initialises the locks. Also the **smpop** function is used to pop from the head of the queue and **smpush** to push at the tail .We need these structures in other files so we declare them in **defs.h** and include this file where needed. We also added a special sleeping channel so that the processes that are put to sleep because of the occurrence of a page fault can sleep in that channel and we have also maintained their count. In the **allocuvm()** function when **mem** returns 0 we run **deallocuvm(pgdir , newsz , oldsz)** and then put the process to sleep and increase the count of such sleeping processes. We have a bool variable **swap_out_exists** , which allows us to control the concurrency of the **swap_out_function() .**

```
if(mem == 0){
  cprintf("allocuvm out of memory\n");
  deallocuvm(pgdir, newsz, oldsz);
  myproc()->state=SLEEPING;
  acquire(&swap_lock);
  myproc()->chan=swap_ch;

  release(&swap_lock);
  smpush(myproc());
  cprintf("%d\n" , swap_cnt) ;
  swap_cnt ++ ;
  cprintf("%d\n" , swap_cnt) ;
  if(!swap_out_process_exists){
    swap_out_process_exists=1;
    swap_out_process_function() ;
    //create_kernel_process("swap_out_process", &swap_out_process_function);
  }
  return 0;
}
```

In the **kfree** function in **kalloc.c** we have edited it so that whenever there is a free page available we wake up all the sleeping processes. So basically we have a sleeping channel for all processes . Whenever the process is not able to allocate memory(i.e mem == 0 in **allocuvm**) , the process gets pushed in swap_queue and it gets assigned to a sleeping channel . Now whenever some kind of memory is freed using **kfree** , we wake up the processes associated with that channel so that they can be scheduled via swap_out_function (Discussed Below) .

```
struct spinlock swap_lock ;
int swap_cnt = 0 ;
char * swap_ch ;
```

Now the code in kalloc.c for kfree to wake up the sleeping channel .

```
if(kmem.use_lock)
  acquire(&swap_lock);
if(swap_cnt){
  wakeup(swap_ch);
  swap_cnt = 0 ;
}
```

Let's go over the swapping-out process. The starting point for this process is the `swap_out_process_function`. This function runs a loop as long as the swap-out requests queue is not empty. When the queue becomes empty, a set of instructions is executed to terminate the `swap_out_process`.

Within the loop, the function retrieves the first process from the queue and applies the LRU (Least Recently Used) policy to identify a page to be swapped out from its page table. It iterates through each entry in the process's page table (`pgdir`), extracting the physical address for each corresponding secondary page table. For each secondary page table, it then checks the accessed bit (A) for each entry. The accessed bit is the sixth bit from the right, and we determine if it is set by performing a bitwise AND operation between the entry and `PTE_A` (which is defined as 32 in `mmu.c`).

```
void swap_out_process_function(){
  //acquire(&sm.lock);
  while(sm.head != 0){
    struct proc *p=smpop();
    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){
      //skip page table if accessed. chances are high, not every page table was accessed.
      if(pd[i]&PTE_A){
        continue;
      }

      int e = 1 ;
      //else
      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
      for(int j=0;j<NPTENTRIES;j++){
        //Skip if found
        if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P)){
          continue;
        }
        e = 1  ;
        pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));
        //for file name
        int pid=p->pid;
        int virt = ((1<<22)*i)+((1<<12)*j);
        //file name
        char c[50];
        int_to_string(pid,c);
        int x=strlen(c);
        c[x]='_';
        int_to_string(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(&pgtab[j],0,sizeof(pgtab[j]));
        //mark this page as being swapped out.
          pgtab[j]=((pgtab[j])^(0x080));
          break;
        }
        e ++  ;e -- ;
        if(e == 1){break ; }
      }

    }
    cprintf("Bye\n") ;

    //release(&sm.lock);
    cprintf("Bye1\n") ;

    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();
    return  ;
  }
```

Important note regarding the Accessed flag: Whenever a process is context-switched by the scheduler, all of its accessed bits are cleared. As a result, the accessed bit observed by the `swap_out_process_function` will reflect whether the page entry was accessed during the process's last iteration.

This logic is implemented in the scheduler, which clears the accessed bits for every entry in both the main and secondary page tables of the process. Once a victim page is identified, the contents of this page need to be written to a file named `<pid>_<virt>`.

```c
if(p->state==UNUSED && p->name[0]=='*'){

  kfree(p->kstack);
  p->kstack=0;
  p->name[0]=0;
  p->pid=0;
}
```

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

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

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

    for(int j=0;j<NPTENTRIES;j++){
      if(pgtab[j]&PTE_A){
        pgtab[j]^=PTE_A;
      }
    }

    ((p->pgdir)[i])^=PTE_A;
  }
}
```

Now, back to **swap_out_process_function**. As soon as the function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number (using macros mentioned in part A report) as the victim page. This page is then swapped out and stored to drive. We use the process pid and virtual address of the page to be eliminated to name the file that stores this page. We have created a new function called **'int_to_string'** that copies an integer into a given string. We use this function to make the filename using integers **pid** and **virt**. Here is that function (declared in proc.c):

```c
void int_to_string(int x, char *c){
  if(x==0)
  {
    c[0]='0';
    c[1]='\0';
    return;
  }
  int i=0;
  while(x>0){
    c[i]=x%10+'0';
    i++;
    x/=10;
  }
  c[i]='\0';

  for(int j=0;j<i/2;j++){
    char a=c[j];
    c[j]=c[i-j-1];
    c[i-j-1]=a;
  }

}
```

We need to write the contents of the victim page to the file with the name **<pid>_<virt>.swp**. But we encounter a problem here.

File system calls cannot be made from **proc.c**. So, we copied the **open**, **write**, **read**, **close** etc. functions from **sysfile.c** to **proc.c**, modified them since the **sysfile.c** functions used a different way to take arguments and then renamed them to **proc_open**, **proc_read**, **proc_write**, **proc_close** etc. so we can use them in **proc.c**.

```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;
}
```

```c
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);
}
```

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

}
```

Now, using these functions, we write back a page to storage. We open a file (**using proc_open**) with **O_CREATE** and **O_RDWR** permissions (we have imported **fcntl.h** with these macros). **O_CREATE** creates this file if it doesn't exist and **O_RDWR** refers to read/write. The file descriptor is stored in an integer called **fd**. Using this file descriptor, we write the page to this file using **proc_write**. Then, this page is added to the **free page queue** using **kfree** so it is available for use (remember we also wake up all processes sleeping on **sleeping_channel** when **kfree** adds a page to the free queue). We then clear the page table entry too using memset.

After this, we do something important: **for Task 3, we need to know if the page that caused a fault was swapped out or not. In order to mark this page as swapped out, we set the 8th bit from the right (2^7) in the secondary page table entry. We use xor to accomplish this task.**

Suspending kernel process when no requests are left:
When the queue is empty, the loop breaks and suspension of the process is initiated. While exiting the kernel processes that are running, we can't clear their **kstack** from within the process because after this, they will not know which process to execute next. We need to clear their **kstack** from outside the process. For this, we first preempt the process and wait for the scheduler to find this process. **When the scheduler finds a kernel process in the UNUSED state, it clears this process kstack and name. The scheduler identifies the kernel process in unused state by checking its name in which the first character was changed to '*' when the process ended.**

TASK 3: (swapping in mechanism):

Similar to Task2 we maintain a data structure called swap_manager_in for the the linked list similar to the one made for the swapping out function.

```
struct swap_manager_in{
    struct spinlock lock ;
    struct swap_queue * head ;
    struct swap_queue * tail ;
} ;
```

```
struct proc* smpopin(){
    acquire(&smi.lock) ;
    if(smi.head == 0){release(&smi.lock) ; return 0  ; }
    if(smi.head == smi.tail){
        struct proc * p  = smi.head->pro  ;
        smi.head = 0 ;
        smi.tail = 0  ;
        release(&smi.lock) ; return  p ;
    }
    struct proc *p = smi.head->pro ;
    smi.head = smi.head->next ;
    release(&smi.lock) ;
    return  p ;
}
```

```
void smpushin(struct proc * p ){
    struct swap_queue sq ;
    sq.pro = p ;
    sq.next = 0 ;
    acquire(&smi.lock) ;
    if(smi.head == 0){
        smi.head = &sq ; smi.tail = &sq ;
        release(&smi.lock) ;
    }
    else{
        smi.tail->next = &sq ;
        smi.tail = &sq ;
        release(&smi.lock) ;
    }
    return  ;

}
```

Next, we add an additional entry to the **struct proc** in **proc.h** called **addr (int)**. This entry will tell the swapping in function at which virtual address the page fault occurred:

```
    int addr ;                    // Gives virtual address of fault
```

Next, we need to handle page fault (**T_PGFLT**) traps raised in trap.c. We do it in a function called **Haldling_page_fault()**:

trap.c:

```c
struct spinlock swap_in_lock;

void handlePageFault(){
  int addr=rcr2();
  struct proc *p=myproc();
  acquire(&swap_in_lock);
  sleep(p,&swap_in_lock);
  pde_t *pde = &(p->pgdir)[PDX(addr)];
  pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

  if((pgtab[PTX(addr)])&PTE_A){
    //This means that the page was swapped out.
    //virtual address for page
    p->addr = addr;
    smpushin(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

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

In **Haldling_page_fault()**, just like **Part A**, we find the virtual address at which the page fault occurred by using **rcr2()**. We then put the current process to sleep with a new lock called **swap_in_lock** (**initialised in trap.c and with extern in defs.h**). We then obtain the page table entry corresponding to this address (the logic is identical to **walkpgdir**). Now, we need to check whether this page was swapped out. In Task 2, **whenever we swapped out a page, we set its page table entry's bit of 7th order (2^7).** Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using **bitwise & with 0x080.** If it is set, we initiate **swap_in_process (if it doesn't already exist - check using swap_in_process_exists).** Otherwise, we safely suspend the process using exit() as the assignment asked us to do.

Now, we go through the **swapping-in process**. The entry point for the swapping out process is **swap_in_process_function (declared in proc.c)** as you can see in **Haldling_page_fault.**

swap_in_process_function function runs a loop until the queue is not empty. In the loop, it pops a process from the queue and extracts its pid and addr value to get the file name. Then, it creates the filename in a string called "**c**" using **int_to_string (described in Task 2**). Then, it used **proc_open** to open this file in read only mode (**O_RDONLY**) with file descriptor **fd**. We then allocate a free frame (**mem**) to this process using **kalloc**. We read from the file with the **fd** file descriptor into this free frame using **proc_read**. We then make **mappages** available to **proc.c** by removing the **static** keyword from it in **vm.c** and then declaring a prototype in **proc.c**. We then use **mappages** to map the page corresponding to **addr** with the physical page that got using kalloc and read into (**mem**). Then we wake up, the process for which we allocated a new page to fix the page fault using **wakeup**. Once the loop is completed, we run the kernel process termination instructions.

```c
void swap_in_process_function(){

  acquire(&smi.lock);
  while(smi.head != 0 ){
    struct proc *p=smpopin();

    int pid=p->pid;
    int virt=PTE_ADDR(p->addr);

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

      int fd=proc_open(c,O_RDONLY);
      if(fd<0){
        release(&smi.lock);
        cprintf("could not find page file in memory: %s\n", c);
        panic("swap_in_process");
      }
      char *mem=kalloc();
      proc_read(fd,PGSIZE,mem);

      if(mappages(p->pgdir, (void *)virt, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
        release(&smi.lock);
        panic("mappages");
      }
      wakeup(p);
  }

  release(&smi.lock);
  struct proc *p;
  if((p=myproc())==0)
    panic("swap_in_process");

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

}
```

TASK 4: Sanity test:

In this task, our goal is to create a testing mechanism to validate the functionalities developed in the earlier tasks. To achieve this, we implement a user-space program called **sanitytest**. Below is the implementation of **sanitytest.c.**

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

int runfunc(int i , int j ){
    return (i*i*j+j*j*i);
}

int
main(int argc, char* argv[]){

    for(int i=0;i<20;i++){
        wait() ;
        if(!fork()){
            printf(1, "This is %dth  child\n", i+1);
            printf(1, "iter | correct_value | wrong_val\n");
            printf(1, "\n\n");
            int * arr[10] ;
            for(int j=0;j<10;j++){
                arr[j] = malloc(4096) ;
                for(int k=0;k<1024;k++){
                    arr[j][k] = runfunc(j, k);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    if(arr[j][k] == runfunc(j, k))
                        matched+=4;
                }


                printf(1, "    %d      %dB      %dB\n", j+1, matched, 4096-matched);


            }

            printf(1, "\n");

            exit();
        }
    }

    while(wait()!=-1);
    exit();

}
```

From the implementation, we can make the following observations:

- The main process spawns 20 child processes using the fork() system call.
- Each child process runs a loop that iterates 10 times.
- During each iteration, 4KB (4096 bytes) of memory is allocated using malloc().
- A demonstration function named runfunc() is created, which computes the expression $i^2 * j + j^2 * i$.
- A counter named matched keeps track of the number of bytes that hold the expected values. This is achieved by verifying that the stored value at each index matches the value returned by runfunc() for that specific index.

To execute **sanitytest,** it needs to be added to the Makefile under UPROGS and EXTRA, allowing it to be accessed by the xv6 user. When **sanitytest** is run, we see the following output:

```
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
    10       4096B       0B

his is 19th  child
ter | correct_value | wrong_val


    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
    10       4096B       0B

his is 20th  child
ter | correct_value | wrong_val


    1       4096B       0B
    2       4096B       0B
    3       4096B       0B
    4       4096B       0B
    5       4096B       0B
    6       4096B       0B
    7       4096B       0B
    8       4096B       0B
    9       4096B       0B
    10       4096B       0B

```

As shown in the output, our implementation passes the sanity check, as all indices store the correct values.

For further testing, we modified the value of PHYSTOP (defined in memlayout.h). The default PHYSTOP value is 0xE000000 (224MB). We adjusted it to 0x0400000 (4MB) because this is the minimum memory required by xv6 to run **kinit1**. After running **sanitytest** with the new setting, the output remained identical to the previous results, confirming that the implementation is correct.