CS344 Operating Systems Lab Assignment 3

Group Number: C40

Members:

Nayanika Ghosh - 200123036

Abhinav Kumar Choudhary - 200101005

Yash Garg - 200101113

Aarti Meena - 180101001

Part A : Lazy Memory Allocation

The **sbrk** system call is used by the current process to notify the xv6 OS whenever it requires more memory than it has been allotted. To meet this need, sbrk employs the growproc() function that is described in **proc.c**. An examination of the

```
case T_PGFLT:
  if(handlePageFault()<0){
    cprintf("Could not allocate page. Sorry.\n");
    panic("trap");
  }
break;
```

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

The implementation of **growproc()** reveals that growproc() calls **allocuvm()**, which is in charge of creating more pages and mapping virtual addresses to their corresponding physical locations inside page tables in order to allocate the needed additional memory.

Our goal in this assignment is to hold back memory when it is required. Instead, we provide the memory upon access. The term "**lazy memory allocation**" refers to this. To achieve this, we comment out the sbrk system function's call to **growproc()**.

We alter the current process's size variable to the desired value, giving the impression that memory has been allotted to the process. This process experiences a **PAGE FAULT** when it attempts to access the page (which it believes has already been brought into memory), resulting in a **T_PGFLT** trap being sent to the kernel. By invoking handlePageFault in **trap.c**, this is handled. **rcr2()** provides the virtual address where the page fault takes place in this. rounded _addr points to the page's beginning address where this virtual address is located. After that, we run **kalloc()**, which pulls a free page from the system's linked list of free pages (**freelist** inside kmem). We now have access to a physical page.

Now, using mappages, we must map it to the virtual address rounded **addr ()**. We remove the **static** keyword from front of mappages() in **vm.c** and declare its prototype in **trap.c** in order to use it there. The parameters for **mappages(**) are the current process's page table, the virtual address of the data's start, its size, the physical memory location of the physical page (we provide this parameter by using the V2P macro, which converts our virtual address to the physical address by subtracting **KERNBASE** from it), and the permissions associated with the page table entry. Check out mappages now (). The first page of the data that has to be loaded is indicated by "a," and the last page is indicated by "last."

```
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;
}
```

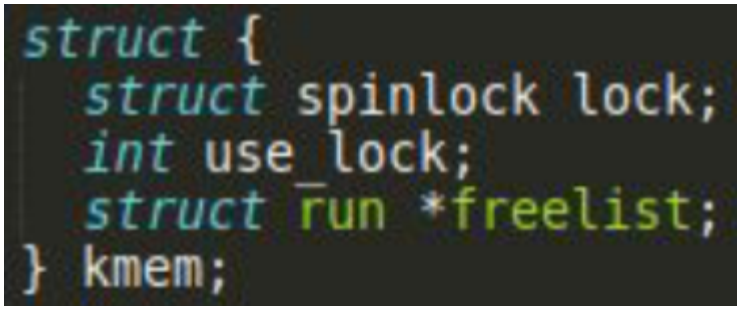
Since it is a two-level page table, it obtains the page directory entry that points to the page table using the first 10 bits of the virtual address (using the PDX macro). It then retrieves the matching entry from the page table using the following 10 bits (using the PTX macro) and returns it. We save the pointer to the first element of the page table corresponding to the page directory entry in pgtab if the page table for that entry is already existent in memory (we use the **PTE ADDR** macro to unset the last 12 bits to make the offset zero). We load the page table and set the permission bits in the page directory if it isn't already loaded in memory.

```
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)];
}
```

A pointer to the page table item corresponding to the virtual address is then returned. The **mappages()** function now understands which entry the current virtual address needs to be mapped to. The PRESENT bit of that item is checked to see if it has already been set, indicating that it has already been mapped to a virtual address. If the answer is true, an error message indicating a remap is generated. If there is no error, it maps the current page table entry to a virtual address, sets its PRESENT bit to indicate that, and correlates the page table entry with the virtual address.

Part B :-

Q1: How does the kernel know which physical pages are used and unused?



Ans :- In kalloc.c, xv6 keeps a linked list of available pages called kmem.

XV6 invokes kinit1 through main() to add 4 MB of free pages to the list as it is initially empty.

Q2: What data structures are used to answer this question?

Ans :- A linked list called freelist, as seen in the image above. The linked list's nodes are all instances of the struct run, which is defined in kalloc.c (pages are typecast to (struct run *) when inserted into the freelist by kfree (char *v)).

Q3: Where do these reside?

Ans :- This linked list is declared inside kalloc.c inside a structure kmem.

Every node is of the type struct run which is also defined inside kalloc.c .

Q4: Does xv6 memory mechanism limit the number of user processes ?

Ans :- Due to a limit on the size of ptable (a maximum of NPROC elements which is set to 64 by default), the number of user processes are limited in xv6. NPROC is defined in param.h .

Q5: If so, what is the lowest number of processes xv6 can “have” at the same time (assuming the kernel requires no memory whatsoever) ?

Ans :- There is only one process running when the xv6 operating system boots up, and its name is initproc (this process forks the sh process which forks other user processes). A process can also use up all of the physical memory because it can have a virtual address space of 2 GB (KERNBASE) and 240 MB (PHYSTOP) of maximum physical memory (We added this since the question asks from a memory management perspective). So, the answer is 1.

Since all user interactions must be performed through user processes that are forked from initproc/sh, there cannot be 0 processes after boot.

Task 1 :-

In proc.c, the create kernel process() method was developed. The kernel process will continue to operate exclusively in kernel mode. Therefore, there is no need to initialise its trapframe, user space, or the user sector of its page table (trapframes hold userspace register values). The address of the following instruction is kept in the process' context's eip register. At the entry point, we want the process to begin executing (which is a function pointer). Consequently, we set the context's eip value to entry point (Since entry point is the address of a function). The process is given a place in the ptable by allocproc. The kernel portion of the process' page table, which converts

virtual addresses above KERNBASE to physical addresses between 0 and PHYSTOP, is configured by setupkvm.

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

  struct proc *p = allocproc();

  if(p == 0)
    panic("create_kernel_process failed");

  //Setting up kernel page table using setupkvm
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");

  //This is a kernel process. Trap frame stores user space registers. We don't need to initialise tf.
  //Also, since this doesn't need to have a userspace, we don't need to assign a size to this process.

  //eip stores address of next instruction to be executed
  p->context->eip = (uint)entrypoint;

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

  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);

}
```

Task 2 :-

It consists of several components. We must first create a process queue to keep track of the processes that were denied more memory because there were no vacant pages. A circular queue structure named rq was developed. And rqueue is the particular queue that contains processes with swap out requests. Additionally, we developed the rq-related functions rpush() and rpop (). We need to use a lock that has been initialised in pinit to access the queue. Additionally, we set the starting values of s and e in userinit to zero. We also introduced prototypes in defs.h because the queue and the functions related to it are required in other files.

Now, kalloc returns zero every time it is unable to allot pages to a process. This informs allocuvm that no memory was allocated for the required amount (mem=0). In this case, we must first set the process status to sleeping.

(*Note: The process sleeps on a unique channel named sleeping channel, which is protected by a lock called sleeping channel lock. When the system boots, sleeping channel count is utilised for special circumstances. Afterward, we must add the running process to the rqueue queue for swap out requests.

If a page isn't already allocated for this process, create kernel process here generates a swapping out kernel process to do so. The swap out process exists variable, which was initialised to 0 in proc.c and declared as extern in defs.h, is set to 0 when the swap out process completes. It is initially set to 1 (as can be seen above). This is done to prevent the emergence of multiple swap out processes. Later on, swap out process is explained.

Next, we develop a system that wakes up all processes sleeping on the sleeping channel whenever free pages become available. In essence, all processes that were sent to sleep on the sleeping channel after being preempted because there weren't any pages available were. We wake all processes currently sleeping on sleeping_channel by calling the wakeup() system call.

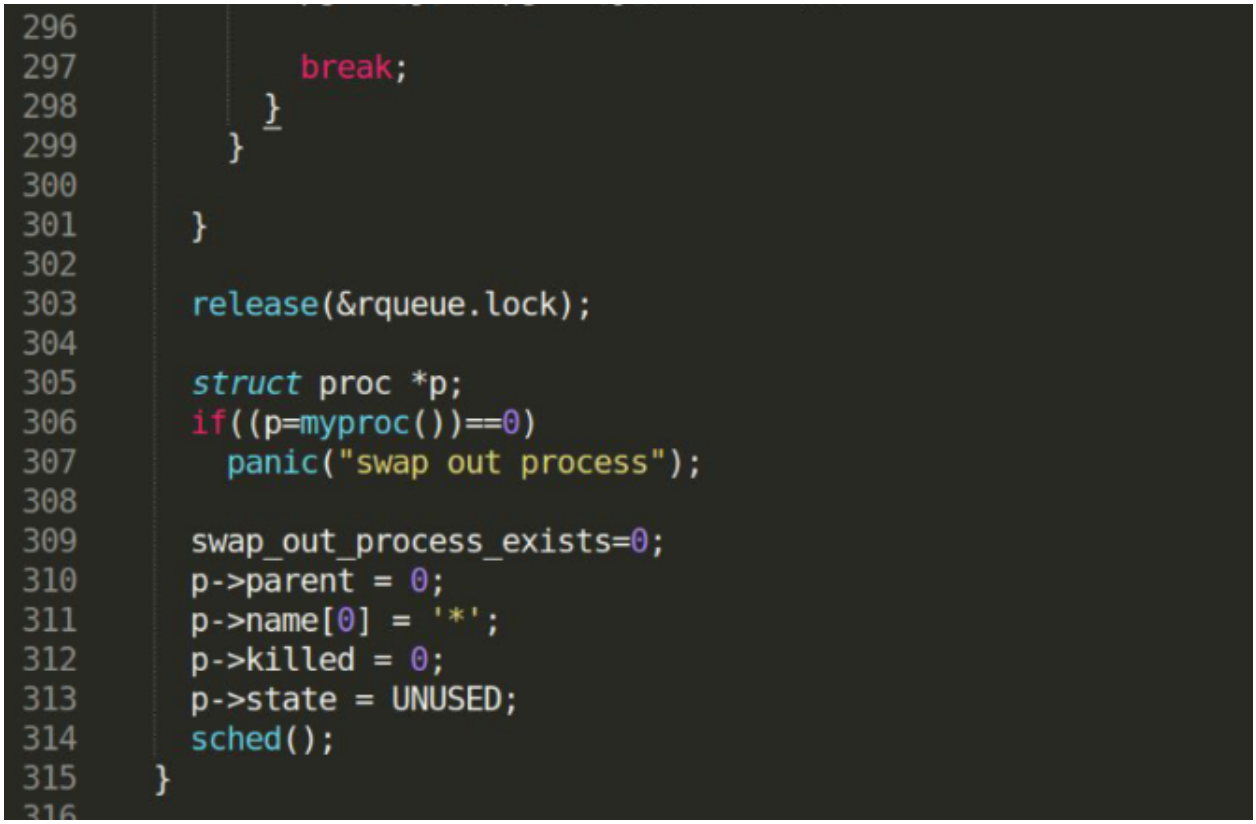
```
170  struct rq{
171    struct spinlock lock;
172    struct proc* queue[NPROC];
173    int s;
174    int e;
175  };
176
177  //circular request queue for swapping out requests.
178  struct rq rqueue;
```

```
502  // Set up first user process.
503  void
504  userinit(void)
505  {
506    acquire(&rqueue.lock);
507    rqueue.s=0;
508    rqueue.e=0;
509    release(&rqueue.lock);
```

```
381  void
382  pinit(void)
383  {
384    initlock(&ptable.lock, "ptable");
385    initlock(&rqueue.lock, "rqueue");
386    initlock(&sleeping_channel_lock, "sleeping_channel");
387    initlock(&rqueue2.lock, "rqueue2");
388  }
```

```
180  struct proc* rpop(){
181
182    acquire(&rqueue.lock);
183    if(rqueue.s==rqueue.e){
184      release(&rqueue.lock);
185      return 0;
186    }
187    struct proc *p=rqueue.queue[rqueue.s];
188    (rqueue.s)++;
189    (rqueue.s)%=NPROC;
190    release(&rqueue.lock);
191
192    return p;
193  }
```

```
void swap_out_process_function(){

  acquire(&rqueue.lock);
  while(rqueue.s!=rqueue.e){
    struct proc *p=rpop();

    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;
      //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;
        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;
      }
```



```
296
297             break;
298           }
299         }
300
301       }
302
303       release(&rqueue.lock);
304
305       struct proc *p;
306       if((p=myproc())==0)
307         panic("swap out process");
308
309       swap_out_process_exists=0;
310       p->parent = 0;
311       p->name[0] = '*';
312       p->killed = 0;
313       p->state = UNUSED;
314       sched();
315     }
```

Image 1: The process runs a loop until the swap out requests queue (rqueue1) is non empty. When the queue is empty, a set of instructions are executed for the termination of swap_out_process (Image 2). The loop starts by popping the first process from rqueue and uses the LRU policy to determine a victim page in its page table. We iterate through each entry in the process page table (pgdir) and extracts the physical address for each secondary page table. For each secondary page table, we iterate through the page table and look at the accessed bit (A) on each of the entries (The accessed bit is the sixth bit from the right. We check if it is set by checking the bitwise & of the entry and PTE_A (which we defined as 32 in mmu.c)). Important note regarding the Accessed flag: Whenever the process is being context switched by the scheduler, all accessed bits are unset. Since we are doing this, the accessed bit seen by swap_out_process_function till indicate whether the entry was accessed in the last iteration of the process.

```
751
752     for(int i=0;i<NPDENTRIES;i++){
753       //If PDE was accessed
754
755       if(((p->pgdir)[i])&PTE_A){
756
757         pte_t* pgtab = (pte_t*)P2V(PTE_ADDR((p->pgdir)[i]));
758
759         for(int j=0;j<NPTENTRIES;j++){
760           if(pgtab[j]&PTE_A){
761             pgtab[j]^=PTE_A;
762           }
763         }
764
765         ((p->pgdir)[i])^=PTE_A;
766       }
767     }
768
769     // Switch to chosen process.  It is the process's job
770     // to release ptable.lock and then reacquire it
771     // before jumping back to us.
772     c->proc = p;
773     switchuvm(p);
```

```
149  void int_to_string(int x, char *c){
150    if(x==0)
151    {
152      c[0]='0';
153      c[1]='\0';
154      return;
155    }
156    int i=0;
157    while(x>0){
158      c[i]=x%10+'0';
159      i++;
160      x/=10;
161    }
162    c[i]='\0';
163
164    for(int j=0;j<i/2;j++){
165      char a=c[j];
166      c[j]=c[i-j-1];
167      c[i-j-1]=a;
168    }
169
170  }
```

```
33  int
34  proc_write(int fd, char *p, int n)
35  {
36    struct file *f;
37    if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
38      return -1;
39    return filewrite(f, p, n);
40  }
```

```
20    int
21    proc_close(int fd)
22    {
23      struct file *f;
24
25      if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
26        return -1;
27
28      myproc()->ofile[fd] = 0;
29      fileclose(f);
30      return 0;
31    }
```

This code resides in the scheduler and it basically unsets every accessed bit in the process page table and its secondary page tables. 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 (pid) and virtual address of the page to be eliminated (virt) 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): 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. We store the filename in a string called c. File system calls cannot be called from proc.c. The solution was that 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 .

```
195  int rpush(struct proc *p){
196
197    acquire(&rqueue.lock);
198    if((rqueue.e+1)%NPROC==rqueue.s){
199      release(&rqueue.lock);
200      return 0;
201    }
202    rqueue.queue[rqueue.e]=p;
203    rqueue.e++;
204    (rqueue.e)%=NPROC;
205    release(&rqueue.lock);
206
207    return 1;
208  }
```

There are many more functions (proc open, proc_fdalloc etc.) in proc.c. 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.

```
240     if(mem == 0){
241       // cprintf("allocuvm out of memory\n");
242       deallocuvm(pgdir, newsz, oldsz);
243 
244       //SLEEP
245       myproc()->state=SLEEPING;
246       acquire(&sleeping_channel_lock);
247       myproc()->chan=sleeping_channel;
248       sleeping_channel_count++;
249       release(&sleeping_channel_lock);
250 
251        rpush(myproc());
252       if(!swap_out_process_exists){
253         swap_out_process_exists=1;
254         create_kernel_process("swap_out_process", &swap_out_process_function);
255       }
256 
257       return 0;
```

```
60  void
61  kfree(char *v)
62  {
63
64    struct run *r;
65    // struct proc *p=myproc();
66
67    if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
68      panic("kfree");
69    }
70
71    // Fill with junk to catch dangling refs.
72    // memset(v, 1, PGSIZE);
73    for(int i=0;i<PGSIZE;i++){
74      v[i]=1;
75    }
76
77    if(kmem.use_lock)
78      acquire(&kmem.lock);
79    r = (struct run*)v;
80    r->next = kmem.freelist;
81    kmem.freelist = r;
82    if(kmem.use_lock)
83      release(&kmem.lock);
84
85    //Wake up processes sleeping on sleeping channel.
86    if(kmem.use_lock)
87      acquire(&sleeping_channel_lock);
88    if(sleeping_channel_count){
89      wakeup(sleeping_channel);
90      sleeping_channel_count=0;
91    }
92    if(kmem.use_lock)
93      release(&sleeping_channel_lock);
94
95  }
```

Thus the ending of kernel processes has two parts:

1. from within process:

2. From scheduler.

All check marks in assignment accomplished.

```
304       struct proc *p;
305       if((p=myproc())==0)
306         panic("swap out process");
307
308       swap_out_process_exists=0;
309       p->parent = 0;
310       p->name[0] = '*';
311       p->killed = 0;
312       p->state = UNUSED;
313       sched();
314     }
315
```

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

    //If the swap out process has stopped running, free its stack and name.
    if(p->state==UNUSED && p->name[0]=='*'){

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

Note: the swapping out process must support a request queue for the swapping requests.

Note: whenever there are no pending requests for the swapping out process, this process must be suspended from execution.

Note: whenever there exists at least one free physical page, all processes that were suspended due to lack of physical memory must be woken up.

Note: only user-space memory can be swapped out (this does not include the second level page table) (since we are iterating all top tables from to bottom and all user space entries come first (until KERNBASE), we will swap out the first user space page that was not accessed in the last iteration.

Task 3:-

First, a swap in request queue needs to be created. To make a swap in the request queue known as rqueue2 in proc.c, we used the same struct (rq) as in Task 2. In defs.h, we additionally declare an extern prototype for rqueue2. We also developed the corresponding functions for rqueue2 (rpop2() and rpush2()) and stated their prototype in defs.h along with declaring the queue. Additionally, we initialised pinit's lock. Additionally, we set its s and e variables to zero in userinit.

Next, we add a new entry called addr to the struct proc in proc.h (int).

This entry will tell the swapping in function at which virtual address the page fault occurred.

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

```
19  void handlePageFault(){
20    int addr=rcr2();
21    struct proc *p=myproc();
22    acquire(&swap_in_lock);
23    sleep(p,&swap_in_lock);
24    pde_t *pde = &(p->pgdir)[PDX(addr)];
25    pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
26
27    if((pgtab[PTX(addr)])&0x080){
28      //This means that the page was swapped out.
29      //virtual address for page
30      p->addr = addr;
31      rpush2(p);
32      if(!swap_in_process_exists){
33        swap_in_process_exists=1;
34        create_kernel_process("swap_in_process", &swap_in_process_function);
35      }
36    } else {
37      exit();
38    }
39  }
```

Similar to Part A, handlePageFault uses rcr2 to identify the virtual address at which the page fault occurred (). We then use a brand-new lock called the swap in lock to put the active process to sleep (initialised in trap.c and with extern in defs.h). The page table entry corresponding to this address is then obtained (the logic is identical to walkpgdir). We must now determine if this page was switched out. When switching out a page in Task 2, we set the page table entry's bit of the seventh order (27). So, using bitwise & and 0x080, we examine the page's 7th order bit to determine whether or not the page was switched out.

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 handlePageFault.

```
void swap_in_process_function(){

    acquire(&rqueue2.lock);
    while(rqueue2.s!=rqueue2.e){
        struct proc *p=rpop2();

        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(&rqueue2.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(&rqueue2.lock);
            panic("mappages");
        }
        wakeup(p);
    }

    release(&rqueue2.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();

}
```

The function runs a loop until rqueue2 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.

Task 4 :- Sanity Test

We created a testing mechanism to test the functionalities created in the previous parts. We will implement a user-space program named memtest for this.

Observations : -

i) The main process creates 20 child processes using fork().

ii) Each child process executes a loop with 10 iterations.

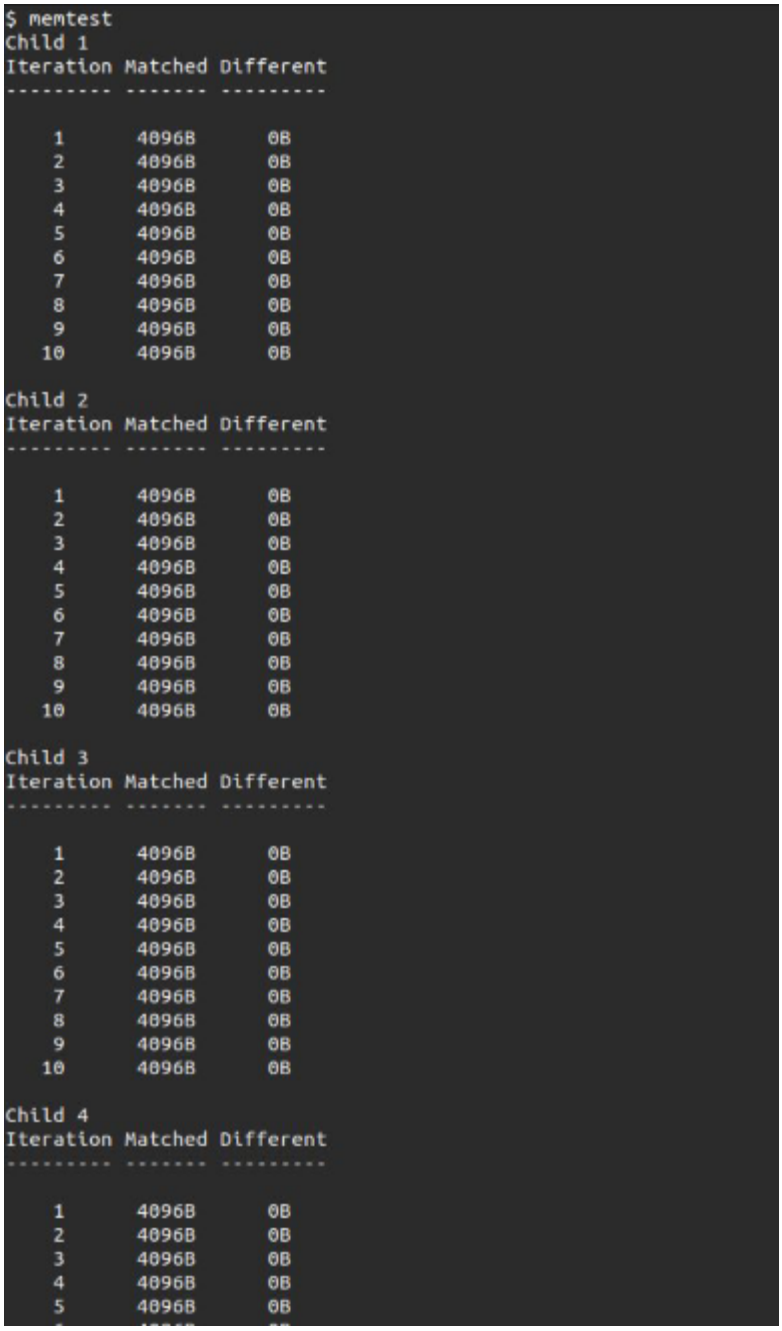
iii) At each iteration, 4 KB of memory is allocated using malloc().

iv) $i^2 - 4i + 1$ gives the value stored at index i of the array.

v) A counter named matched stores the no. of bytes with the right values.

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

int math_func(int num){
    return num*num - 4*num + 1;
}

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

    for(int i=0;i<20;i++){
        if(!fork()){
            printf(1, "Child %d\n", i+1);
            printf(1, "Iteration Matched Different\n");
            printf(1, "--------- ------- ---------\n\n");

            for(int j=0;j<10;j++){
                int *arr = malloc(4096);
                for(int k=0;k<1024;k++){
                    arr[k] = math_func(k);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    if(arr[k] == math_func(k))
                        matched+=4;
                }

                if(j<9)
                    printf(1, "    %d      %dB      %dB\n", j+1, matched, 4096-matched);
                else
                    printf(1, "   %d      %dB      %dB\n", j+1, matched, 4096-matched);

            }
            printf(1, "\n");

            exit();
        }
    }

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

}
```

To run memtest, include it in Makefile under UPROG and EXTRA to make it accessible to xv6 user.



The output shows that out implementation passes the sanity test. Also, we run the tests on different values of PHYSTOP(of memlayout.h). The default value of PHYSTOP is 0xE000000 (224 MB). We change its value but the obtained output is same as the previous output on running the memtest.

Thus, the implementation os proved correct.