

# ASSIGNMENT 3

## CS344 - Operating Systems Lab

### G4 Group Members

B Venkatesh	180101013
Harsh Motwani	180101028
Nishchay Manwani	180101051
Vaibhav Kumar Singh	180101086

### Part A: Lazy Memory Allocation

Whenever the current process needs extra memory than its assigned value, it indicates this requirement to the xv6 OS using sbrk system call. sbrk uses growproc() defined inside proc.c to cater to this requirement. A closer look at the implementation of growproc() shows that growproc() calls allocvm() which is responsible for allocating the desired extra memory by allocating extra pages and mapping the virtual addresses to their corresponding physical addresses inside page tables.

In this assignment, our objective is to refrain giving memory as soon as it is requested. Rather, we give the memory when it is accessed. This is known as Lazy Memory Allocation. We do this by commenting out the call to growproc() inside the sbrk system call. We change the size variable associated with the current process to the desired value which gives the process a false feel that the memory has been allocated. When this process tries to access the page (which it thinks has been already brought inside memory), it encounters a PAGE FAULT, thus generating a T\_PGFLT trap to the kernel. This is handled in trap.c by calling

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

handlePageFault(). In this, rcr2() gives the virtual address at which the page fault occurs. rounded \_\_addr points to the starting

```
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;
}
```

address to the page where this virtual address resides. Then we call kalloc() which returns a free page from a linked list of

free pages (freelist inside kmem) in the system. Now we have a physical page at our disposal. Now we need to map it to the virtual address rounded\_\_addr which is done using mappages(). To use mappages() in trap.c, we remove the static keyword in front of it in vm.c and declare its prototype in trap.c. mappages() takes

the page table of the current process, virtual address of the start of the data, size of the data, physical memory at which the physical page resides (we give this parameter by using V2P macro which converts our virtual address to physical address by subtracting KERNBASE from it) and permissions corresponding to the page table entry as parameters. Now let's have a deeper look at `mappages()`.

```
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;
}
```

```
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)];
}
```

In this, 'a' denotes the first page and 'last' denotes the last page of the data that has to be loaded. It then runs a loop until all the pages from the first to last have been loaded successfully. For every page, it loads it into the page table using `walkpgdir()`. `walkpgdir()` takes a page table and a virtual address as input and returns the page table entry corresponding to that virtual address inside the page table. Since it is a two-level page table, it uses the first 10 bits (using PDX macro) of the virtual address to obtain the page directory entry which points to the page table. It then uses the next 10 bits (using PTX macro) to get the corresponding entry in the page table and returns it. If the page table

corresponding to the page directory entry is already present in memory, we store the pointer to its first entry in `pgtab` (We use `PTE_ADDR` macro to unset the last 12 bits thereby making the offset zero). If the page table isn't present in memory, we load it and set the permission bits in the page directory. After this, we return a pointer to the page table entry corresponding to the virtual address. Now, the `mappages()` knows the entry to which the current virtual address has to be mapped. It checks if the `PRESENT` bit of that entry is already set indicating that it is already mapped to some virtual address. If yes, it generates an error telling that remap has occurred. If no error takes place, it associates the page table entry to the virtual address, sets the permission bits and sets its `PRESENT` bit indicating that the current page table entry has been mapped to a virtual address.

# ANSWERS TO PART-B QUESTIONS

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

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

xv6 maintains a **linked list** of free pages in **kalloc.c** called **kmem**. Initially, the list is empty, so xv6 calls **kinit1** through **main()** which adds 4MB of free pages to the list.

<Q2> What data structures are used to answer this question?

➤ A **linked list** named **freelist** as shown in the above image. Every node of the linked list is a structure defined in **kalloc.c** namely **struct run** (pages are typecast to **(struct run \*)** when inserting into **freelist** in **kfree(char \*v)**).

<Q3> Where do these reside?

➤ 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?

➤ Due to a limit on the size of **ptable** (a max. 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)?

➤ When the xv6 operating system boots up, there is only one process named **initproc** (this process forks the **sh** process which forks other user processes). Also, since a process can have a virtual address space of 2GB (**KERNBASE**) and the assumed maximum physical memory is 240 MB (**PHYSTOP**), one process can take up all of the physical memory (We added this since the question asks from a memory management perspective). Hence, **the answer 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**.

## Part B (All tasks completed)

### Task 1:

The `create_kernel_process()` function was created in `proc.c`. The kernel process will remain in kernel mode the whole time. Thus, **we do not need to initialise its trapframe** (trapframes store userspace register values), user space and the user section of its page table. The `eip` register of the process' context stores the address of the next instruction. We want the process to start executing at the entry point (which is a function pointer). Thus, **we set the eip value of the context to entry point** (Since entry point is the address of a function). `allocproc` assigns the process a spot in `ptable`. `setupkvm` sets up the kernel part of the process' page table that maps virtual addresses above `KERNBASE` to physical addresses between 0 and `PHYSTOP`.

**proc.c:**

```
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:

This task has various parts. First, we need a **process queue** that keeps track of the processes that were refused additional memory since there were no free pages available. We created a **circular queue struct** called `rq`. And the specific queue that holds processes with **swap out requests** is `rqueue`. We have also created the functions corresponding to `rq`, namely `rpush()` and `rpop()`. The queue needs to be accessed with a lock that we have initialised in `pinit`. We have also initialised the initial values of `s` and `e` to zero in `userinit`. Since the queue and the functions relating to it are needed in other files too, we added prototypes in `defs.h` too.

**proc.c:**

```
169
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;
179
```

```
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);
510 }
```

```
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 }
194
```



```

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 }

```

defs.h:

```

12 struct rq;
13
127 extern struct rq rqueue;
128 extern struct rq rqueue2;
129 int rpush(struct proc *p);
130 struct proc* rpop();
131 struct proc* rpop2();
132 int rpush2(struct proc* p);
133

```

**\*Note:** **rqueue2** (and correspondingly **rpupsh2** and **rpup2**) is used in **Task 3**.

Now, whenever **kalloc** is not able to allocate pages to a process, it returns zero. This notifies **allocvm** that the requested memory wasn't allocated (mem=0). Here, we first need to change the process state to sleeping. (**\*Note:** The process sleeps on a special sleeping channel called **sleeping\_channel** that is secured by a lock called **sleeping\_channel\_lock**. **sleeping\_channel\_count** is used for corner cases when the system boots) Then, we need to add the current process to the swap out request queue, **rqueue**:

vm.c:

**Global declarations (Note: These are also declared in defs.h as extern variables. I am not adding the defs.h screenshots):**

```

14 struct spinlock sleeping_channel_lock;
15 int sleeping_channel_count=0;
16 char * sleeping_channel;

```

allocvm:

```

240 if(mem == 0){
241     // cprintf("allocvm out of memory\n");
242     deallocvm(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;

```

**\*Note:** `create_kernel_process` here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. When the swap out process ends, the **`swap_out_process_exists`** (declared as **extern** in **`defs.h`** and initialised in **`proc.c`** to 0) variable is set to 1 (as seen above). This is done so multiple swap out processes are not created. **`swap_out_process`** is explained later.

Next, we create a mechanism by which whenever free pages are available, all the processes sleeping on

**`sleeping_channel`** are woken up. We edit **`kfree`** in **`kalloc.c`** in the following way:

Basically, all processes that were preempted due to lack of availability of pages were sent sleeping on the sleeping channel. We wake all processes currently sleeping on **`sleeping_channel`** by calling the **`wakeup()`** system call.

```
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 }
```

Now, I will explain the **swapping out process**. The entry point for the swapping out process in **`swap_out_process_function`**. Since the function is very long, I have attached two screenshots:

```
244 void swap_out_process_function(){
245
246     acquire(&rqueue.lock);
247     while(rqueue.s!=rqueue.e){
248         struct proc *p=rpop();
249
250         pde_t* pd = p->pgdir;
251         for(int i=0; i<NPENTRIES; i++){
252
253             //skip page table if accessed. chances are high, not every page table was accessed.
254             if(pd[i]&PTE_A)
255                 continue;
256             //else
257             pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
258             for(int j=0; j<NPENTRIES; j++){
259
260                 //Skip if found
261                 if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_H))
262                     continue;
263                 pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));
264
265                 //for file name
266                 int pid=p->pid;
267                 int virt = ((1<<22)*i)+((1<<12)*j);
268
269                 //file name
270                 char c[50];
271                 int_to_string(pid,c);
272                 int x=strlen(c);
273                 c[x]='_';
274                 int_to_string(virt,c+x+1);
275                 safestrcpy(c+strlen(c),".swp",5);
276
277                 // file management
278                 int fd=proc_open(c, 0_CREATE | 0_RDWR);
279                 if(fd<0){
280                     cprintf("error creating or opening file: %s\n", c);
281                     panic("swap_out_process");
282                 }
283
284                 if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
285                     cprintf("error writing to file: %s\n", c);
286                     panic("swap_out_process");
287                 }
288                 proc_close(fd);
289
290                 kfree((char*)pte);
291                 memset(&pgtab[j],0,sizeof(pgtab[j]));
292
293                 //mark this page as being swapped out.
294                 pgtab[j]=((pgtab[j])^(0x080));
295
296                 break;
297             }
298         }
299     }
300
301     release(&rqueue.lock);
302
303     struct proc *p;
304     if((p=myproc())==0)
305         panic("swap out process");
306
307     swap_out_process_exists=0;
308     p->parent = 0;
309     p->name[0] = '*';
310     p->killed = 0;
311     p->state = UNUSED;
312     sched();
313
314 }
```

```
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
316 }
```

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 into by the scheduler, all accessed bits are unset. Since we are doing this, the accessed bit seen by swap\_out\_process\_function will indicate whether the entry was accessed in the last iteration of the process:**

```

751
752     for(int i=0;i<NPENTRIES;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     switchvm(p);

```

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, line 267 in image**) and virtual address of the page to be eliminated (**virt, line 268 in image**) 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**):

```

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

```

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**. Some examples:

```

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 }
41

```

There are many more functions (**proc open, proc\_fmalloc etc.**) and you can check them out in **proc.c**. I can't paste all of them here.

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

```

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 }

```

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 (LINE 295 in image).**

### 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.**

Thus the ending of kernel processes has two parts:

1. from within process:

```

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

```

2. From scheduler

```

//If the swap out process has stopped running, free its stack and name.
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state==UNUSED && p->name[0]=='*'){
        kfree(p->kstack);
        p->kstack=0;
        p->name[0]=0;
        p->pid=0;
    }
}

```

All check marks in assignment accomplished:

- Note: the swapping out process must support a request queue for the swapping requests. (**page 1 and 2 of this report**).
- Note: whenever there are no pending requests for the swapping out process, this process must be suspended from execution. (**page 5 of this report, just above this**)
- 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. (**page 3 of this report: kfree and sleeping\_channel**)
- 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 top 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:

We first need to create a **swap in request queue**. We used the same struct (**rq**) as in Task 2 to create a swap in request queue called **rqueue2** in **proc.c**. We also declare an extern prototype for **rqueue2** in **defs.h**. Along with declaring the queue, we also created the corresponding functions for **rqueue2** (**rpop2()** and **rpush2()**) in **proc.c** and declared their prototype in **defs.h**. We also initialised its lock in **pinit**. We also initialised its **s** and **e** variables in **userinit**.

**Since all the functions/variables are similar to the ones shown in Task 2, I am not attaching their screenshots here.**

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:

**proc.h (in struct proc):**

```
char name[16];           // Process name (debugging)
int addr;                // ADDED: Virtual address of pagefault
};
```

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

**trap.c:**

```
96         break;
97     case T_PGFLT:
98         handlePageFault();
99         break;
100 //PAGEBREAK: 13
```

In **handlePageFault**, 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

```
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 }
```

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 ( $2^7$ )**. This is mentioned at the beginning of the 5th page of this report.

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 **handlePageFault**.

**Note:** **swap\_in\_process\_function** is shown on the next page since it is long. Refer to the next page for the actual function.

I have already mentioned how we have implemented file management functions in **proc.c** in the **Task 2** part of the report. I will just mention which functions I used and how I used them here. 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, page 4 of this report). 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 that were described on **page 5 of this report**.

In Task 3 too, all the check marks were accomplished.

```
17
18  int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
19
```

```
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,0_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();
}
```

## Task 4:Sanity Test

In this part, our aim is to create a testing mechanism in order to test the functionalities created by us in the previous parts. We will implement a user-space program named memtest that will do this job for us. The implementation of memtest is given below.

```
#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();
}
```

We can make the following observations by looking at the implementation:

- ❑ The main process creates 20 child processes using fork() system call.
- ❑ Each child process executes a loop with 10 iterations
- ❑ At each iteration, 4096B (4KB) of memory is being allocated using malloc()
- ❑ The value stored at index  $i$  of the array is given by the mathematical expression  $i^2 - 4i + 1$  which is computed using math\_func().
- ❑ A counter named matched is maintained which stores the number of bytes that contain the right values. This is done by checking the value stored at every index with the value returned by the function for that index.

In order to run memtest, we need to include it in the Makefile under UPROGS and EXTRA to make it accessible to the xv6 user.

On running memtest, we obtain the following output.

```
$ memtest
Child 1
Iteration Matched Different
-----
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

Child 2
Iteration Matched Different
-----
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

Child 3
Iteration Matched Different
-----
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

Child 4
Iteration Matched Different
-----
1      4096B      0B
2      4096B      0B
3      4096B      0B
4      4096B      0B
5      4096B      0B
6      4096B      0B
```

As can be seen in the output, our implementation **passes** the sanity test as all the indices store the correct value.

Now, to test our implementation even further, we run the tests on different values of **PHYSTOP** (defined in memlayout.h). The default value of PHYSTOP is 0xE000000 (224MB). We changed its value to 0x0400000 (4MB). We chose 4MB because this is the minimum memory needed by xv6 to execute **kinit1**. On running memtest, the obtained output is identical to the previous output indicating that the implementation is correct.