CS344, OPERATING SYSTEMS LAB
ASSIGNMENT-03:

Aayush Suthar, 220123004
Aaditya Jalan, 220123001
Avanish Gadhikar, 220123075
Chaitanya Chabra, 220123012

Code Link Here: *OS_LAB_3*

PART A: Implementing Lazy Memory Allocation

The current running process indicates its requirement of additional memory requirement to the Operating System using the `sbrk()` system call. `sbrk()` further makes a system call to the `growproc()` function. `growproc()` is a function defined inside *proc.c*, and inside `growproc()` further calls are made to `allocuvm()`. `allocuvm()` is responsible for allocating the desired extra memory by allocating extra pages and mapping the virtual addresses to their corresponding physical addresses inside the page tables.

Lazy Memory Allocation tries to defer the allocation of the memory until it is needed, rather than giving memory as soon as it is requested.

xv6 currently provides memory as soon as it is requested. In order to change this method of allocation, we need to comment out the call made 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 memory has been allocated. When this process tries to utilise the memory, which it feels is already allocated to it, it encounters a **PAGE FAULT.**

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

The process tries to utilise the memory, thus raising a page fault.

This **PAGE FAULT** generates a `T_PGFLT` trap to the kernel. We add the functionality to handle this trap. We define a function called `pageFaultHandler()` inside *trap.c* for this purpose.

```
case T_PGFLT:
int x=pageFaultHandler();
  if(x<0){
    cprintf("page is not getting allocated\n");
    panic("trap");
  }
break;
```

Handling of the trap T_PGFLT inside the trap handler

The `pageFaultHandler()` function handles this **PAGE FAULT.** In this the `rcr2()` function gives the virtual address at which the page fault occurs. `rounded_addr` points to the starting address of the page where this virtual address resides. We then call the `kalloc()` function which returns a free page from a linked list of free pages (stores as `freelist` inside *kmem*). We now possess a physical page at our disposal. We then map this page 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.*

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

int pageFaultHandler(){
  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;
}
```

Implementation of page fault handler

`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 pages reside and the permissions corresponding to the page table as the arguments. (We give this parameter by using V2P macro which converts our virtual address to physical address by subtracting KERNBASE from it).

```
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;
}
```

Implementation of mappages inside vm.c

As we delve into `mappages()`, we have *'a'* denoting the first page and *'last'* denoting the last page of the data to be loaded. It runs a loop until all pages from the first to the last have been loaded successfully. Every page is loaded into the page table using the `walkpgdir()` function.

```
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)];
}
```

`walkpgdir()` takes as input a page table and returns the page table entry corresponding to that virtual address inside the page table. Since it uses a 2 level page table, it uses the first 10 bits (using *PDX* macro) of the virtual address to obtain the page directory entry which corresponds to the page table. Thereafter, it uses the further 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`, using `PTE_ADDR` macro to unset the last 12 bits, thereby making the offset 0.
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.

PART B: Questions

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

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

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

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

A linked list called `freelist` as shown in the image above is used for this purpose. 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 called `kmem`. Every node is of the type `struct run` which is also defined in `kalloc.c`.

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

Due to the limit on the size of the ptable (which is defined in a MACRO called `NPROC` set to 64 by default), the number of user processes are limited to 64 by default 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` running which in turn forks other processes, such as `sh` leading to further process creation. Also, since a process can have a virtual address space of *2GB* (**KERNBASE**) and the assumed maximum physical memory is *240MB*(**PHYSTOP**), one process can take up all of the physical memory. Hence, the **answer is 1.**

There cannot be **0** processes after boot since all user interactions need to be done using user processes, which are forked in `sh`/`initproc`.

PART B: Tasks

TASK 1: kernel processes

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 the entry point (Since the entry point is the address of a function). `allocproc()` assigns the process a spot in the `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**.

```
//creating our function here create kernel process
void create_kernel_process(const char *name, void (*entrypoint)()){

  struct proc *p = allocproc();

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

  //setupkvm sets up kernel virtual memory and page table
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");


  //eip stores the pointer to 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);

}
```

'create_kernel_process()' inside proc.c

TASK 2: swapping out mechanism

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.

Attached below are the screenshots of the changes made.

```
struct rq{
  struct spinlock lock;
  struct proc* queue[NPROC];
  int s;
  int e;
};

struct rq rqueue;
```

The circular queue struct `rq`.

```
struct proc* rpop(){

  acquire(&rqueue.lock);
  if(rqueue.s==rqueue.e){
    release(&rqueue.lock);
    return 0;
  }
  struct proc *p=rqueue.queue[rqueue.s];
  (rqueue.s)++;
  (rqueue.s)%=NPROC;
  release(&rqueue.lock);

  return p;
}
```

```
int rpush(struct proc *p){

  acquire(&rqueue.lock);
  if((rqueue.e+1)%NPROC==rqueue.s){
    release(&rqueue.lock);
    return 0;
  }
  rqueue.queue[rqueue.e]=p;
  rqueue.e++;
  (rqueue.e)%=NPROC;
  release(&rqueue.lock);

  return 1;
}
```

Implementation of `rpop()` and `rpush()`.

```
extern struct rq rqueue;
int rpush(struct proc *p);
struct proc* rpop();
```

```
extern          char * sleeping_channel;
extern struct spinlock sleeping_channel_lock;
extern          int sleeping_channel_count;
```

```
struct superblock;
struct rq;
```

Inside defs.h (above 3 images)

Now, whenever `kalloc()` is not able to allocate pages to a process, it returns zero. This notifies `allocuvm()` that the requested memory was not 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`:

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

```
void
userinit(void)
{
  acquire(&rqueue.lock);
  rqueue.s=0;
  rqueue.e=0;
  release(&rqueue.lock);
```

Inside vm.c (above 2 images)

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&rqueue.lock, "rqueue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&rqueue2.lock, "rqueue2");
}
```

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

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

   rpush(myproc());
  if(!swap_out_process_exists){
    swap_out_process_exists=1;
    create_kernel_process("swap_out_process", &swap_out_process_function);
  }

  return 0;
```

Inside `allocuvm`

Note: create_kernel_process here creates a swapping out process to allocate a page for this process if it already doesn't exist. When the swap out process ends, the `swap_out_process_exists` (declared as **extern** in *defs.h* and intitalised to 0 in *proc.c)* variable is set to 0. When it is created, it 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:

```
void
kfree(char *v)
{

  struct run *r;
  // struct proc *p=myproc();

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP){
    panic("kfree");
  }

  // Fill with junk to catch dangling refs.
  // memset(v, 1, PGSIZE);
  for(int i=0;i<PGSIZE;i++){
    v[i]=1;
  }

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);

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

}
```

Inside `kalloc.c` we edit `kfree()`.

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

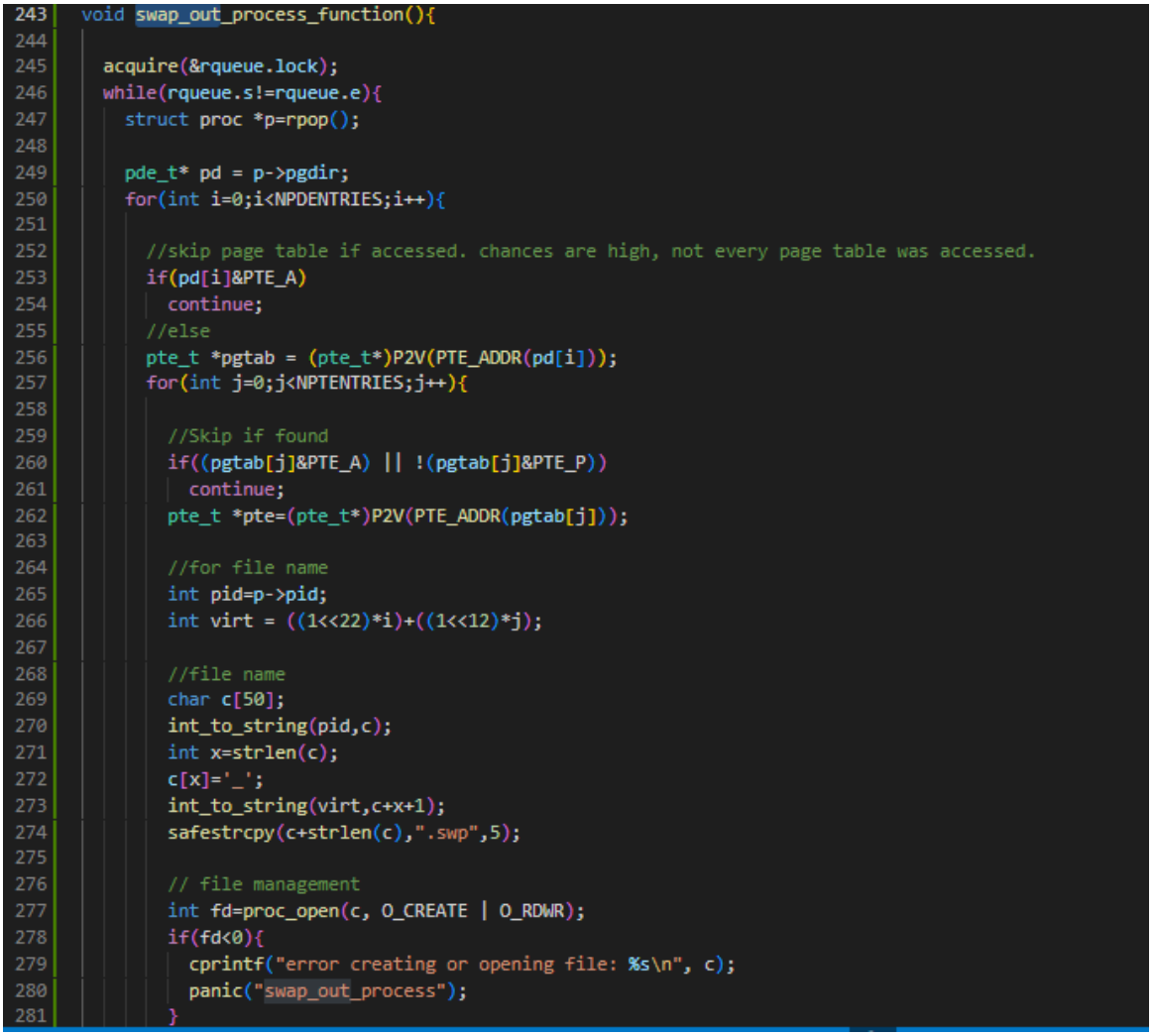
The entry point for the swap out process in `swap_out_process_function`. The screenshots for the same are attached below:

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. The loop starts by popping out the first process from queue and uses the LRU policy to determine a victim page in its 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 bit ise & 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:



```
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");
        }
```

```
282
283             if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
284               cprintf("error writing to file: %s\n", c);
285               panic("swap_out_process");
286             }
287             proc_close(fd);
288
289             kfree((char*)pte);
290             memset(&pgtab[j],0,sizeof(pgtab[j]));
291
292             //mark this page as being swapped out.
293             pgtab[j]=((pgtab[j])^(0x080));
294
295             break;
296           }
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     }
```

Implementation of swap_out_function inside `proc.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;
  }
}

// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchuvm(p);
p->state = RUNNING;
```

Changes in `scheduler()` function in `proc.c`

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. 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*):

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:

```
148  //made a conversion function
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  }
```

Added a new conversion function `int_to_string()` in proc.c

There are many more functions (**proc_open**, **proc_fdalloc,** etc.) and you can check them out 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 (LINE 295 in image).

Spending 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 as changed to '*' when the process ended.

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

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

  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}
```

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

defined functions `proc_write()` and `proc_close()` in `proc.c`

Ending of kernel process has **two** parts:

```
  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();
}
```

inside `swap_out_process()` function in `proc.c`

```
  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;
    }
```

Inside `scheduler()`

```
//creating our function here create kernel process
void create_kernel_process(const char *name, void (*entrypoint)()){

  struct proc *p = allocproc();

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

  //setupkvm sets up kernel virtual memory and page table
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");


  //eip stores the pointer to 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 3:

Defined structs for **task3:**

```
void            swap_out_process_function();
void            swap_in_process_function();
extern int swap_out_process_exists;
extern int swap_in_process_exists;
extern struct rq rqueue;
int rpush(struct proc *p);
struct proc* rpop();
extern struct rq rqueue2;
struct proc* rpop2();
int rpush2(struct proc* p);
```

Inside defs.h

We first need to create a swap in request queue. We use 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 created the corresponding functions for `rqueue2` (`rpop2()` and `rpush2()`) in *proc.c* and declared their prototypes in *defs.h*. Additionally, we initialised its lock in `pinit` and 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:

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

Inside struct proc in proc.h

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

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

Added this case inside function trap in trap.c

In `handlePageFault()` , similar to Part A, we use `rcr2()` to find the virtual address where the page fault happened. Next, we put the current process to sleep using a new lock called **swap_in_lock** (which is initialised in *trap.c* and with extern in *defs.h*). After that, we get the page table entry for this address (following the same logic as in `walkpgdir`). At this point, we need to check if

the page was swapped out. In Task 2, whenever we swapped out a page, we set the 7th bit (2^7) in its page table entry.

```
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)])&0x080){
    //This means that the page was swapped out.
    //virtual address for page
    p->addr = addr;
    rpush2(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

Implementation of handlePageFault

To check if the page was swapped out, we use a bitwise `&` operation with `0x080` to check its 7th bit. If the bit is set, we start the **swap_in_process** (if it doesn't already exist—this can be checked using **swap_in_process_exists**). If the bit is not set, we safely terminate the process using `exit()` , as required by the assignment.

Now, let's go through the swap-in process. The entry point for this process is the `swap_in_process_function ()` (declared in *proc.c*). You can see this function referenced in `handlePageFault()` . The actual function is shown on the next page since it's quite long. I've already explained how we implemented file management functions in *proc.c* in Task 2 of this report, so here, I will only mention which functions were used and how.

The function runs a loop until `rqueue2` is empty. In the loop, it pops a process from the queue and extracts its pid and `addr` value to get the file name. It then creates the filename as a string called "c" using `int_to_string()` (which is detailed in Task 2). After that, it opens the file in read-only mode (**O_RDONLY**) using **proc_open**, with a file descriptor **fd**.

Next, we allocate a free frame (mem) for this process using `kalloc`. We read data from the file into this free frame using **proc_read**. We 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 was using `kalloc` and read into (mem).

Finally, we wake up the process for which the new page was allocated to fix the page fault using `wakeup()`. Once the loop finishes, we execute the kernel process termination instructions.

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

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

Removed static keyword from mappages

```
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;
```

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

  }
```

TASK 4: Memtest.c

Using the above stated file we will test the functionalities that we have created so far. The implementation of the `memtest.c` is given below:

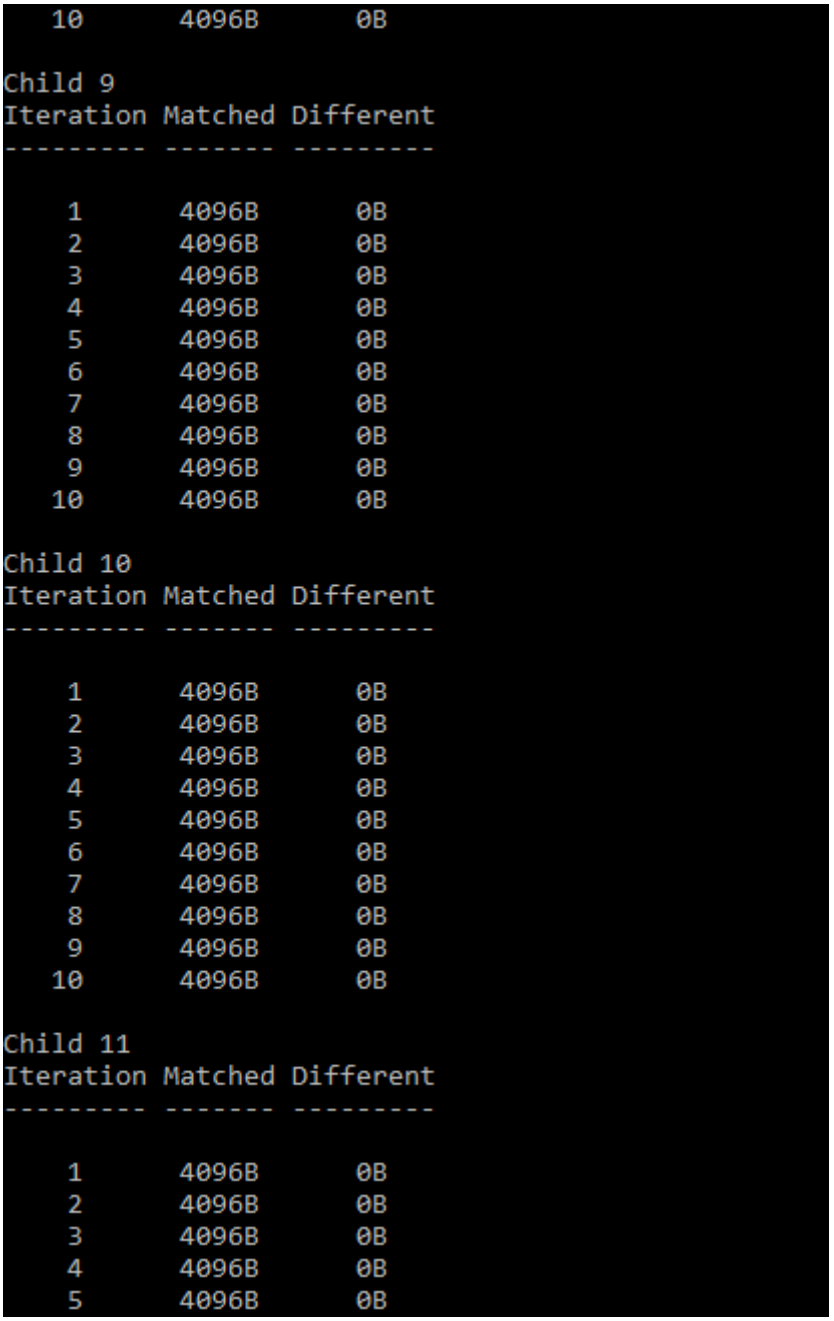
```c
#include "types.h"
#include "stat.h"
#include "user.h"

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();

}
```

We observe the following observations by looking at the the implementation:

- The main function forks and creates 20 child processes.
- Each child runs a loop with 10 iterations.
- At each iteration, *4KB* of memory is allocated using `malloc()`.
- The value stored at index i of the array is given by mathematical expression i*i -4i+1.
- Calculated by `math_func()`.
- A "matched" counter tracks bytes with correct values by comparing array values to expected results.

To run `memtest.c` we need to include it in the makefile under **UPROGS** and **EXTRAS**

We get following output on running `memtest` :



As we can see in the output, all sanity checks are passed as the value stored are correct.

Now we can test the implementation further by changing the value of **PHYSTOP**. We changed the value of **PHYSTOP** from *224 MB* to *16 MB*.
On running we got the same output indicating the implementation is correct.