CS 344 : Operating Systems Laboratory

Lab 4

Members~

Harsh	*200123022*
Pragyan Banerjee	*200123080*
Sahil Kumar Gupta	*200123081*

PART A

What is lazy memory allocation?

Lazy allocation of memory means not allocating memory to a process until it is actually needed. By delaying allocation of memory until you actually need it, we can decrease startup time, and even eliminate the allocation entirely if we never actually use the process. Most modern operating systems perform lazy allocation of heap memory, though **xv6 does not.** Xv6 applications ask the kernel for heap memory using the sbrk() system call. In the original xv6 kernel, **sbrk() allocates physical memory and maps it into the process's virtual address space.**

Task: We have to add support for this lazy allocation feature in xv6 by delaying the memory requested by sbrk() until the process actually uses it.

Step 1: Eliminate allocation from sbrk():

To delete the page allocation from the sbrk(n) system call implementation, which is in the function sys sbrk() in sysproc.c, we have used the given patch file in which the call to the function growproc(n) is commented out. The new sbrk(n) will just increment the process's size by n and return the old size. It does not allocate memory. However, it still increases the size of the process by n bytes to trick the process into believing that it has the memory requested.

Typing echo hi to the shell gives us the following output:

```
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0201198 s, 254 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 6.5864e-05 s, 7.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
349+1 records in
349+1 records out
178728 bytes (179 kB, 175 KiB) copied, 0.000593334 s, 301 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -dr
ive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc
$
```

The “pid 3 sh: trap...” message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. The “addr 0x4004” indicates that the virtual address that caused the page fault is 0x4004.

Step 2: Adding Lazy Allocation

We have modified the code of trap() function in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. We have allocated a new memory page, added suitable page table entries, and returned from the trap, so that the process can avoid the page fault the next time it runs.

Commented out portion of sbrk in sysproc.c provided in patch

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

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

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

We have added the following piece of code in trap.c

Now we can begin to handle this page fault error starting from trap.c. In trap.c, there are various cases to handle trap errors using case-switch statements so we add one more for T_PGFLT since we have a page fault error and there is no pre-existing case for handling page-fault errors. To make mappages function available in trap.c we made a new header file named vm.h and included it in trap.c

vm.h

```
int             mappages(pde_t *, void *, uint, uint, int);
```

Mappages function

```
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;
}
```

Changes in trap.c trap function

```
void
trap(struct trapframe *tf)
{
  char *mem;
  uint a;
```

```
case T_PGFLT:
  mem = kalloc();
  a = rcr2();
  a = PGROUNDDOWN(a);
  memset(mem, 0, PGSIZE);
  mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U);
  break;
```

Important Points:-

1. tf->trapno = T_PGFLT check whether a fault is a page fault
2. PGROUNDDOWN(rcr2()) is used to round the faulting virtual address down to start of page
3. We have returned in order to avoid the cprintf() and the proc->killed = 1 statements
4. We have made a vm.h to access mappages in trap.c

Description of functions we changed and why we changed and what effect it had

PGROUNDUP(address) which rounds up the address to the starting of a page or we can say it gives us the page-aligned address since we can have errors at any address in-btw page.

kalloc() allocates one-page of 4096 bytes from physical memory & returns a pointer to that page.

memset() makes the whole page null.

mappages() maps the page to our process's page directory by converting the given virtual address (rounded) to physical address by the use of V2P(mem) which basically subtracts the Kernel base address from the virtual address. It creates a new (page table entry) PTE for that particular virtual address.

We use these functions in our lazy allocation implementation by giving the argument myproc() which contains the current process's directory and rcr2() which gives the address at which page fault occurs and the rest of the implementation is done in the lazy_allocation function. We need not use the old size and new size arguments since we only need to allocate a page so that our program can execute normally in userspace.

The page table flags used in mappages() function are PTE_W or PTE_U which declares that the page is user accessible and writeable as seen below.

Hence our new function takes the faulty address, rounds up and makes it page aligned, allocates a fresh page from the physical memory to the process using mappages() function and the process starts to execute normally if the allocation is done correctly.

After making the following changes, the lazy memory allocation is implemented correctly in xv6 and now we are not encountering any errors after typing echo hi **OR** ls **to the shell as shown in the screenshot attached below:**

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
hi
```

<u>PART B</u>

Answers to Questions

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

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

In kalloc.c, xv6 maintains a linked list of unoccupied pages under the name kmem. When the list is initially empty, xv6 runs kinit1 through main(), adding 4MB of free pages to the list.

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

freelist, a linked list 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 in kfree(char *v)).

3. Where do these reside?

Within the structure kmem of the file kalloc.c, this linked list is defined. Every node is of the struct run type, which is specified in kalloc.c as well.

4. Does xv6 memory mechanism limit the number of user processes?

The number of user processes is constrained in xv6 due to a restriction on the size of ptable (a maximum of NPROC elements which is set to 6$ by default). The param.h file defines NPROC.

5. If so, what is the lowest number of processes xv6 can ‘have’ at the same time (assuming the kernel requires no memory whatsoever)?

There is just 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). Additionally, since a process can use up to 2GB (KERNBASE) of virtual address space and 24O MB (PHYSTOP) of maximum physical memory, one process can use up the whole physical memory (We added this since the question asks from a memory management perspective). So, the response is 1. Additionally, since all user interactions must be performed through user processes that are forked from initproc/sh, there cannot be zero 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 next instruction is kept in the process' context's eip register. At the entrance 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.

proc.c:

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

  struct proc *p = allocproc();
  if(p == 0) panic("create_kernel_process failed");
  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

There are several elements to this task. 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 methods 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 since the queue and the functions related to it are required in other files.

proc.c:

```c
struct rq{
```

```c
C];
```

```c
struct rq rqueue, rqueue2;
struct proc* rpop(){
  acquire(&rqueue.lock);
  if(rqueue.s==rqueue.e){
        release(&rqueue.lock);
```

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

  acquire(&rqueue2.lock);
  rqueue2.s=0;
  rqueue2.e=0;
  release(&rqueue2.lock);
```

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

```c
  struct proc *p;
  (rqueue.e)%=NPROC;
  release(&rqueue.lock);
  return 1;
}
```

defs.h:

```c
void            yield(void);
void            create_kernel_process(const char *name, void
(*entrypoint)());
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;
extern struct rq rqueue2;
int rpush(struct proc *p);
int rpush2(struct proc* p);
struct proc* rpop();
struct proc* rpop2();
```

```c
struct rq;
```

Now, kalloc returns 0 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. The process then has to be added to the queue for swap out requests (*Note: The process sleeps on a unique sleeping channel named sleeping channel, which is protected by a lock called sleeping channel lock. sleeping channel count is used for exceptional instances when the system boots.)

vm.c:

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

```

```
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    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;
    }
    memset(mem, 0, PGSIZE);
```

allocuvm:

Now, kalloc always returns 0 if a process cannot receive a page allocation. Allocuvm is informed that the required memory wasn't allocated (mem=0) by this. We must first set the process status in this case to sleeping. (*Note: The process sleeps on a unique channel named sleeping channel, which is protected by a lock called sleeping channel lock; sleeping channel count is used for exceptional circumstances when the system boots.) Next, we must add the current process to the queue for swap out requests.

Next, we develop a system that wakes up all processes sleeping on the sleeping channel whenever free pages become available. In kalloc.c, we change kfree as follows:

```c
void
kfree(char *v)
{
  struct run *r;

  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 process 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);
}
```

In essence, all processes that were sent to sleep on the sleeping channel after being preempted because there weren't any pages available were. The wakeup() system function is used to wake up all sleeping processes on sleeping channel.

We'll now describe the changing out procedure. the function's swap out process entry point, which starts the switching out process. Due to the length of the function, We've included multiple screenshots:

screenshots:

```c
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++){
      if(pd[i]&PTE_A)continue;
      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]));

        int pid=p->pid;
        int virt = ((1<<22)*i)+((1<<12)*j);
        char c[50];


        int_to_string(pid,c);
        int x=strlen(c);
        c[x]='_';


        int_to_string(virt,c+x+1);
```

```c

        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]));
```

```c
        kfree((char*)pte);
        memset(&pgtab[j],0,sizeof(pgtab[j]));

        pgtab[j]=((pgtab[j])^(0x080));
        break;
      }
    }

  }

  release(&rqueue.lock);

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

  swap_out_process_exists=0;

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

Until the swap out requests queue (rqueue1) is not empty, the procedure iterates in a loop. A sequence of instructions are carried out to end the swap out process when the queue is empty (Image 2). The loop begins by removing the first process from the rqueue, then it searches its page table for a victim page using the LRU policy. We repeatedly go over each entry in the page table (pgdir) for the process and for every secondary page table, retrieves the physical address. The accessed bit (A), which is the sixth bit from the right on each entry in the page table for each secondary page table, is examined throughout each iteration of the page table. By examining the bitwise & of the entry, we may determine if it is set and PTE_A (which we defined as 32 in mmu.c)).

Important information on the Accessible flag: When the scheduler switches the context of a process, all accessed bits are unset. As a result of our actions, the accessed bit detected by the swap out process function will show whether the item was accessed during the previous iteration of the process:

```
    for(int i=0;i<NPDENTRIES;i++){
      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;

    swtch(&(c->scheduler), p->context);
```

This function, which is included in the scheduler, essentially resets all bits that have been accessed in the process' primary and secondary page tables.

Returning to the swap out process function now. The victim page is selected by the function (using the macros stated in section A report) as soon as it locates a secondary page table entry with the accessed bit unset. The following step is to replace and store this page on drive.

To name the file that holds this page, we utilise the process' process id (pid, line 267 in the picture), as well as the virtual address of the page that has to be deleted (virt, line 268 in the image). A new function we call "int to string" converts an integer into a specified string. We create the filename using the numbers pid and virt using this method. This is the declaration for the function in proc.c:

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

  while(x>0){
    c[i]=x%10+'0';
    i++;
    x/=10;
  }
  c[i]='\0';

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

}
```

The victim page's contents must be written to a file called _.swp. But this is where we get into trouble. The filename is kept in a string named c. Proc.c cannot make calls to file system functions.

The answer was to copy the open, write, read, close, etc. functions from sysfile.c to proc.c, modify them because sysfile.c methods took arguments in a different way, and then rename them to proc open, proc read, proc write, proc close, etc. so we could use them in proc.c.

Now, we write a page back to storage utilising these functions. We open a file with the permissions O CREATE and O RDWR (using proc open) (we have imported fcntl.h with these macros).

If this file doesn't already exist, O CREATE creates it, and O RDWR stands for read/write. The integer fd contains the file descriptor. We use proc write to write the page to this file using this file descriptor. Then, this page is put to the free page queue using kfree so that it is ready for usage (remember, when kfree adds a page to the free queue, we also wake up any processes sleeping on sleeping channel). The page table entry is then cleared using memset.

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


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

}
```

The next step is crucial since Task 3 requires us to determine whether or not the page that triggered a problem was replaced. We set the secondary page table entry's eighth rightmost bit (27) to indicate that this page has been switched out. To do this work, we use xor.

The loop is broken and the procedure is put on hold when the queue is empty. We can't wipe the kstack of the ongoing kernel processes while they are being terminated since they won't know which process to execute next as a result. Their kstack has to be cleared from outside the process. To do this, we preempt the process first and then wait for the scheduler to locate it. The scheduler clears the kstack and name of any kernel processes it discovers that are in the UNUSED state. By looking at the name of the kernel process, which had its initial character altered to the symbol * when the process stopped, the scheduler may determine if it is in an idle state. Thus the ending of kernel processes has two parts:

from within process

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

  swap_out_process_exists=0;

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

from scheduler

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

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

      if(p->state != RUNNABLE)
        continue;
```

Task 3

First, a swap in request queue has to be created. To make a swap in the request queue known as rqueue2 in proc.c, we utilised the same struct (rq) as in Task 2. In defs.h, we additionally specify 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.

Since all of the variables and functions are similar to those in Task 2, I won't attach screenshots of them here.

Next, we add a new entry named addr to the struct proc in proc.h (int). The swapping in function will be informed by this item of the virtual address where the page fault occurred.

proc.h

```c
  char name[16];
  int addr;
};
```

The page fault (T PGFLT) traps generated by trap.c must then be handled. In a function named handlePageFault(), we carry it out.

trap.c

```c
void handlePageFault(){
  struct proc *p=myproc();
  int addr=rcr2();
  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){
    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 {
```

```c
    lapiceoi();
    break;
  case T_PGFLT:
    handlePageFault();
  break;
  //PAGEBREAK: 13
  default:
```

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). Beginning on the fifth page of this report, this is discussed.

So, using bitwise & and 0x080, we examine the page's 7th order bit to determine whether or not the page was switched out. Initiating swap in process (assuming it doesn't already exist - verify using swap in process exists) is done if it is set. Otherwise, as instructed by the assignment, we may safely halt the operation using exit().

We now proceed with the switching in procedure. As you can see in handlePageFault, the entry point for the swapping out process is swap in process function (defined in proc.c).

Due to its length, swap in process function is displayed on the next page. For the real function, see the page after this one.

In the Task 2 section of the report, We have previously discussed how We introduced file management features in proc.c. Here, We will only briefly discuss the functions We used and how We used them. The loop in the function continues until rqueue2 is not empty. In the loop, a process is selected from the queue, and its pid and addr values are extracted to get the file name. The filename is then created as a string named "c" using the int to string function. Then, it used proc open to open this file using file descriptor fd in read-only mode (O RDONLY). Then, using kalloc, we provide this process access to a free frame (mem). Using proc read, we read into this free frame data from the file with the fd file descriptor. Then, after removing the static keyword from it in vm.c, we make mappages accessible to proc.c by declaring a prototype there. The physical page that was obtained using kalloc and read into is then mapped using mappages to the page corresponding to addr (mem).

Then, using wakeup to correct the page fault, we wake up the process for which a new page was allocated. We execute the kernel process termination instructions when the loop has finished.

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

```c
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");
                }
```

```c

                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->killed = 0;


        p->state = UNUSED;
        p->parent = 0;


        p->name[0] = '*';
        sched();
```

Task 4

In this section, we'll build a testing framework to evaluate the functions that we developed in the other sections. We'll utilise a user-space application called memtest to carry out this task on our behalf. Below is a description of how memtest is implemented.

By examining the implementation, we may draw the following conclusions:

- Using the fork() system function, the primary process generates 2O child processes.
- Every child process runs a 1O iterations long loop.
- Using malloc, 4O96B (4KB) of memory is allocated per loop ()
- The mathematical equation that is calculated using math func provides the value stored at array index We ().
- The number of bytes that have the correct values is kept in a counter called matched. This is accomplished by comparing the value provided by the function for each index with the value saved at each position.

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


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");
                        int j = 0;
                        while(j<10){
                                int *arr = malloc(4096);
                                for(int k=0;k<1024;k++)arr[k] = k*k- 4*k +1;
                                int matched=0;
                                for(int k=0;k<1024;k++){
                                        if(arr[k] == k*k- 4*k +1)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);

                        j++;

                        }


                        printf(1, "\n");

                        exit();
                }
        }





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

}
```

We must add memtest to the Makefile's UPROGS and EXTRA sections so that the xv6 user can access it in order to execute it.

We get the following output from memtest.

```
$ 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
```

The result shows that our implementation passes the sanity test since every index stores the right value.

We now run the tests on various PHYSTOP values to better evaluate our implementation (defined in memlayout.h). OxEOOOOOO is the default value for

PHYSTOP (224MB). Its value was modified to 0x0400000 (4MB). We choose 4MB since xv6 requires this amount of RAM in order to run kinit1. The output from memtest was similar to the prior output, proving that the implementation is sound.