Group no C38
Vatsal Gupta 200101105
Sweeya Reddy 200101079
Pranshu Kandoi 200101086

Assignment-3 OS344

PART A

In this part of the assignment we need to implement lazy allocation. Normally in xv6 memory is allocated as soon as any process ask for it. But here we only need to allocate memory when it is actually required. Applications ask the kernel for heap memory using the **sbrk()** system call. Thus we need to delay the memory requested by **sbrk()** until the process actually uses it. Thus we have just increased the size of process by n and return old size thus tricking the process into believing that it has the memory requested and commented the growproc line thus memory is not allocated. We can also see this by running echo hi command in shell leading to following error.

```
Booting from Hard Disk..xv6...
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
pid 3 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$ 
```

Thus an error with trap number 14 occurs similar to that in the assignment at addr 0x4004. From the below code in trap.c we see that the rcr2 command gives the address at which page fault occurs. Also from traps.h we see that corresponding to number 14 `T_PGFLT` macro is defined.

```
//PAGEBREAK: 13
default:
  if(myproc() == 0 || (tf->cs&3) == 0){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
            tf->trapno, cpuid(), tf->eip, rcr2());
    panic("trap");
  }
  // In user space, assume process misbehaved.
  cprintf("pid %d %s: trap %d err %d on cpu %d "
          "eip 0x%x addr 0x%x--kill proc\n",
          myproc()->pid, myproc()->name, tf->trapno,
          tf->err, cpuid(), tf->eip, rcr2());
  myproc()->killed = 1;
}

// Force process exit if it has been killed and is in user space
```

Thus now to handle the page fault we have implemented the pageFaultHandler function. THis function is similar to **allocuvm** firstly rcr2 call is used to get the address at which the page fault occurs then **PGROUNDDOWN** is used to round the faulting virtual address down to the start of a page boundary. Then a page is allocated using **kalloc** which basically returns a free page from the list of free pages defined in kalloc.c. If an error occurs then return -1 else fill the page with 0 and then fill the page table entry using the mappages. Since we need to use the mappages we need to declare it in the trap.c and also the static keyword needs to be removed from the mappages function defined in the vm.c file for this purpose.

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

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

Note that the rcr2 function uses the cr2 register which holds the page fault address.

```c
}

static inline uint
rcr2(void)
{
  uint val;
  asm volatile("movl %%cr2,%0" : "=r" (val));
  return val;
}
```

Given below is the implementation of mappages

Mappages firstly using the walkpgdir function returns the page table entry corresponding to the virtual address and then at this location the physical address of the free page is mapped. Also note that the walk page directory function (walkpgdir) using the virtual address finds the page directory entry using the first 10 bits of the MSB and then finds the page table entry using the next 10 bits of the MSB in the page table and then returns the pointer to the page table entry.

```c
// be page-aligned.
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;
}
```

```c
// create any required page table pages.
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)];
}
```

PART B

Questions & Answers

- [ ] **How does the kernel know which pages are used and unused?**

  xv6 has a structured object called kmem which has as a part of it, a "freelist" which is is simply a linked list of free pages.

  Below is a screenshot of kmem creation code in kalloc.c. To fill up this list initially, or to initiate this list kinit1 is called through main which adds 4MB of free pages to the list.

```
struct run {
  struct run *next;
};

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

// Initialization happens in two phases.
// 1. main() calls kinit1() while still using entrypgdir to place just
// the pages mapped by entrypgdir on free list.
// 2. main() calls kinit2() with the rest of the physical pages
// after installing a full page table that maps them on all cores.
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}
```

- [ ] **What data structures are used to answer this question?**

  A linked list is used. The list is singly linked with the nodes being of struct run type. Page pointers are typecasted into this type before insertion into the list.

```
#define NPROC        64  // maximum number of processes
```

- [ ] **Where do these reside?**

  The data structures (linekd lists and corresponding nodes) themselves reside in kernel space and are declared in kalloc.c within kmem. Every node is of type struct run (also defined in kalloc.c)

- [ ] **Does xv6 memory mechanism limit the number of user processes?**

  The ptable size is constrained and so the number of user processes are limited in xv6.(Max number of elements is 64 constrained by NPROC as in param.h)

- [ ] **If so what is the lowest number of processes xv6 can have at the same time assuming the kernel requires no memory whatsoever**

  *There can not be 0 processes after boot since to take user input and interaction there needs to be at least one processes active.

  *The initproc processes which is the first processes on booting the OS, forks the sh processes which in turn forks other user processes. So you can have just 2 processes initially and then 3 when a user interaction takes place as is generally the case.

  *However, purely from a memory management standpoint, a processes can have a virtual space of 2GB and a physical maximum of 240MB. Since one processes can theoretically take up all the space, **the answer is 1 in that sense.**

Tasks

1. **Task 1: kernel processes**

```
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");

  //kernel proccesses doesn;t need userspace related memory allocations like trapframes
  //In the absence of a userspace, size is also not necessary

  //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);

}
```

create_kernel_process() function is made in proc.c. We first allocate a new process for this kernel processes using allocproc(which will allocate a new process and put an entry in the ptable). We then setup it's kernel virtual memory using setupkvm(which will setup the kennel page table accordingly). If either of these fail we panic and send error with the appropriate message. Notice as in comments that we do not need to initalise a trapframe- since the the processes remains in kernel mode, it circumvents the need of a userspace register and hence it doesn't need a trapframe or user section of page table/userspace. We also don't need to define a size.
We then setup the eip or the (extended) instruction pointer register to point to the entrypointer which is the address of the function that we want to allocate the process to i.e. where we want the processes to jump and begin execution.

2. **Task 2: swapping out mechanism**
We create a process queue to track processes which require additional memory but have been refused it due to absence of free pages.
A queue which loops back on itself (circular queue) is created- loop_queue. To

enqueue and dequeue we have mnade the functions cqpush() and cqpop().

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

//circular request queue for swapping out requests.
struct circq loop_queue;

struct proc* cqpop(){

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

  return p;
}

int cqpush(struct proc *p){

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

  return 1;
}
```

The queue works after acquiring a lock to avoid conflicts, this lock is initlaised in pinit. s and e (start and end values) are also initalised to 0 in userint.

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&loop_queue.lock, "loop_queue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&loop_queue2.lock, "loop_queue2");
}
```

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

To make the queue and its functions accessible to other modules, we have added their declarations in defs.h.

```
struct circq;
```

```
extern struct circq loop_queue;
extern struct circq loop_queue2;
int cqpush(struct proc *p);
struct proc* cqpop();
struct proc* cqpop2();
int cqpush2(struct proc* p);
```

allocuvm calls kalloc which returns 0 if it is not able to allocate new pages. This error is then handled by putting this process into sleep by changing its state and adding to a special channel called the sleeping_channel which is secured by a sleeping_channel_lock. sleeping_channel_count counts how many processes are sleeping. The current process is queued to the loop_queue we made earlier. declarations are defined as below:

(vm.c)

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

(defs.h)

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

allocuvm:

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    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);

       cqpush(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);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

create_kernel process here calls a swapping out process to allocate a page for the process. swap_out_process_exists is set to 0 when the swap out process completes(swap_out_process_exists is declared extern in defs.h and intialised in proc.c to 0 and 1 when created). This prevents multiple creations. The swap_out_process_function is as below:

```c
void swap_out_process_function(){

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

    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){

      //skip page table if accessed. chances are high, not every page table was accessed.
      if(pd[i]&PTE_A)
        continue;
      //else
      pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
      for(int j=0;j<NPTENTRIES;j++){

        //Skip if found
        if((pgtab[j]&PTE_A) || !(pgtab[j]&PTE_P))
          continue;
        pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));

        //for file name
        int pid=p->pid;
        int virt = ((1<<22)*i)+((1<<12)*j);

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

        // file management
        int fd=proc_open(c, O_CREATE | O_RDWR);
        if(fd<0){
          cprintf("error creating or opening file: %s\n", c);
          panic("swap_out_process");
        }

        if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
          cprintf("error writing to file: %s\n", c);
          panic("swap_out_process");
        }
        proc_close(fd);

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

        //mark this page as being swapped out.
        pgtab[j]=((pgtab[j])^(0x080));

        break;
      }
    }

  }

  release(&loop_queue.lock);

  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();
}
```

The process initially loops through all the process in the queue(loop_queue). We implement the LRU policy to determine victim page in it's page table. We iterate through the second level index (pgdir) and extract the address of each page table. We henceforth iterate through the page table and see the accessed bit A. (by performing bitwise & with the entry and PTE_A(32 in mmu.h)).

If the function notices that a particular page was never accessed, the corresponding physical page is chosen for eviction i.e. to be unassigned to the virtual page to free up memory. The page is then swapped out and its content is stored in the memory (a new file is created with the .swp extension <pid>_<virt>.swp). We use process id (pid) and virtual address (virt) to name this swp file.

To facilitate this, opening, reading,writing etc. of files needed to be implemented. These functions were taken from sysfile.c and modified slightly and copied down to proc.c for our use.(screenshot only for some attached below refer to code for all)

```
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);
}


static struct inode*
proc_create(char *path, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);

  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(type == T_DIR){  // Create . and .. entries.
    dp->nlink++;  // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");

  iunlockput(dp);

  return ip;
}
```

We use these functions and macros(for permissions like O_RDWR) which we have defined in fcntl.h to write back a page to storage.

```c
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
```

For opening the file, we use proc_open with O_CREATE(create if doesnt exist) and O_RDWR(read/write) permissions. The fd file descriptor handle links to this file and facilitates writing through proc_write.

Lastly, this page is added to the free page queue using kfree so it can be used later and is made available to processes which were earlier refused page allocations (implementation explained next)and we truly clear the page table entry using memset(deallocating space).

Implementation of kfree: Kfree is edited in kalloc.c to wakeup all the processes which were refused page allocation (and premepted) and put to sleep (by putting them on the sleeping channel). This is done by calling wakeup as a system call on the system_channel.

```
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
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);
```

NOTE:

*swapping out process supports a request queue for the swapping requests.
*when there are no pending requests, the process suspends correctly.
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:

```
  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();
```

2. from scheduler:

```
//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;
}
```

*whenever there exists at least one free physical page, all processes that were suspended due to lack of physical memory are woken up.
*only user-space memory can be swapped out (this does not include the second level page table) (since we are iterating all top tables from to bottom and all user space entries come first (until KERNBASE), we will swap out the first user space page that was not accessed in the last iteration.)
*Whenever the process is being context switched into the scheduler, all accessed bits are unset. Since we are doing this, the accessed bit seen by the swap_out_prcess_function will tell whether the entry was accessed in the last iteration of the process.

*int_to_string functionality is implemented in proc.c to use integers as strings in naming files(pid,virt are integer types)

```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;
  }

}
```

*for Task 3 we need to know if the page causing the page fault was swapped out. To mark this page correctly, we set the 8th bit from right on the secondary page table entry.

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

//mark this page as being swapped out.
pgtab[j]=((pgtab[j])^(0x080));
```

3. **Task 3: swapping in mechanism**
   We have to create a swap in request queue. We used the circq structure as before to create loop_queue2 in proc.c and declare in defs.h as before along with cpush2 and cpop2. Like before, the lock is initialised in pinot and s and e are in userinit(screenshots skipped as same as loop_queue, kindly refer code.)

   The proc struct now needs to know where the page fault occurred. To do this, addr (int) variable is added to the structure, which will indicate the virtual address of the page fault.

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int addr;                    // ADDED: Virtual address of pagefault

};
```

The page fault is handled through the handlePageFault() function when T_PGFLT (14) trap/error is raised.

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

```c
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;
    cqpush2(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

Like in part A, the virtual address where the page fault occurred is found by rcr2(). The current process is then put to sleep with the swap_in_lock (initialised in trap.c and made available through defs.h globally). The page table entry is then obtained as in walkpgdir.

Now, to check whether this page was swapped out, we check the 7th order bit (using bitwise & with 0x080) as we had set it in the previous task. If it set, we use swap_in_process to swap it back in(if it doesn't already exist - check using swap_in_process_exists (same functionality as before)). otherwise we suspend the process using exit() as instructed.

```
void swap_in_process_function(){

  acquire(&loop_queue2.lock);
  while(loop_queue2.s!=loop_queue2.e){
    struct proc *p=cqpop2();

    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(&loop_queue2.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(&loop_queue2.lock);
        panic("mappages");
      }
      wakeup(p);
  }

    release(&loop_queue2.lock);
    struct proc *p;
  if((p=myproc())==0)
    panic("swap_in_process");

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

}
```

The entry point for the swapping out process is swap_in_process_function (declared in proc.c). The function implements a while loop as long as loop_queue2 is not empty. It then pops a process and extracts its process id and virtual address into pid and addr to to get the file name. Then the filename is created in a string (named c) using int_to_string. Then proc_open opens this file with the file descript fd. Then a free frame mem is allocated to this process using kalloc. The file is read from this free frame using proc_read.

mappages is made available to proc.c by removing static from vm.c and declaring it 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 as before.

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

4. **Task 4: Sanity test**

We have to create a test for all the things we have implemented until now. We make a user program called memtest which adhocly links and subsequently tests each and every functionality.

```
#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();

}
```

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

- Using the fork() system call, the primary process generates 20 child processes.
- Every child process runs a 10 iterations long loop.
- Using malloc, 4096B (4KB) of memory is allocated per loop ()
- The mathematical expression that math func uses to calculate the value contained at index I of the array ().
- 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.

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 OxEOOOOOO (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.

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ 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
```

NOTE:

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