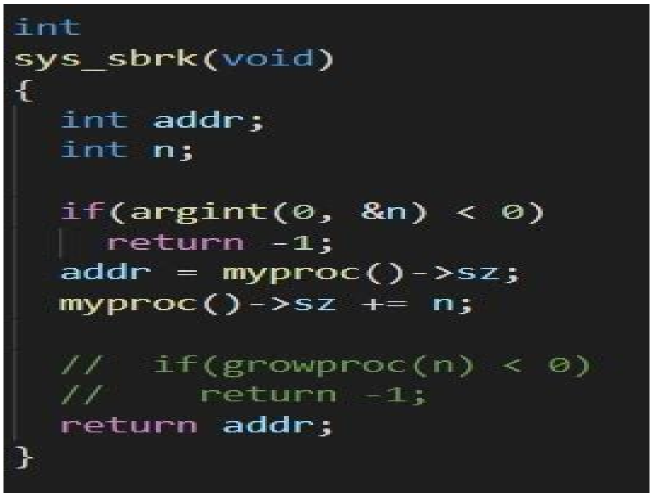
CS344 (OS LAB) ASSIGNMENT 3
GROUP G15

NAME	ANKET KOTKAR	RITIK MANDLOI	RAHUL MALA	ARYAN CHAUHAN
ROLL NO.	180101037	180101066	180101062	180101012

PART A)

LAZY MEMORY ALLOCATION

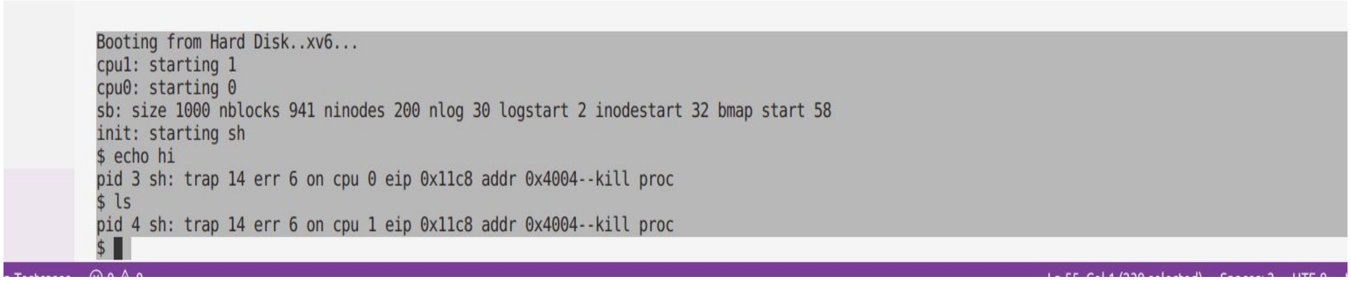
Lazy memory allocation means not allocating memory to a process until it is actually needed.



```
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;
}
```

sbrk() allocates physical memory and maps it into the process's virtual address space.

Error that we got due to changing the sbrk system call.(echo hi and ls giving exceptions due to page fault)



```
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 0 eip 0x11c8 addr 0x4004--kill proc
$ ls
pid 4 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$
```

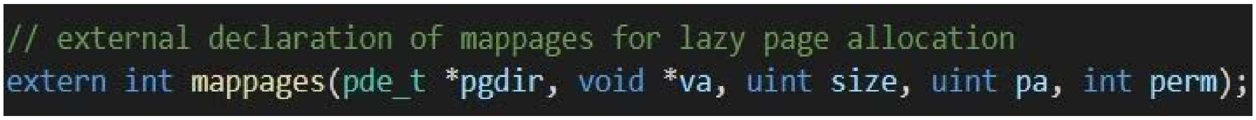
Effect of changes made in sbrk system call:

sbrk(n) system call helps to increase the memory by n bytes whenever a process requires extra memory for execution.Majorly it is handled by growproc() function which allocates the physical memory.As we have commented it we are not actually increasing the memory but the process thinks that we have increased it.(as we have increased the **myproc()->sz by n**)

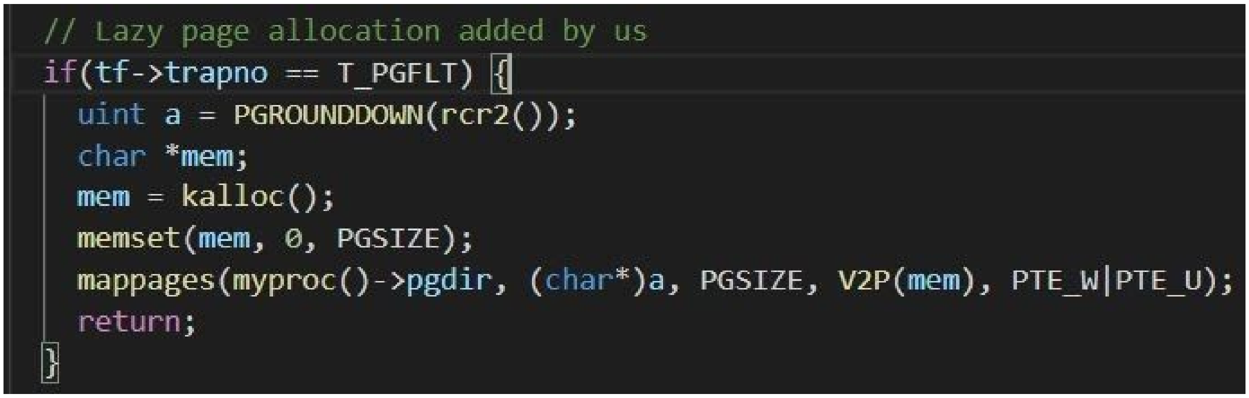
Now as physical memory is not actually allocated this results in page fault which the general xv6 cannot handle .(**T_PGFLT** exception is created as control is transferred to **trap.c**)

Handling the exception in trap.c:

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.



```
// external declaration of mappages for lazy page allocation
extern int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```



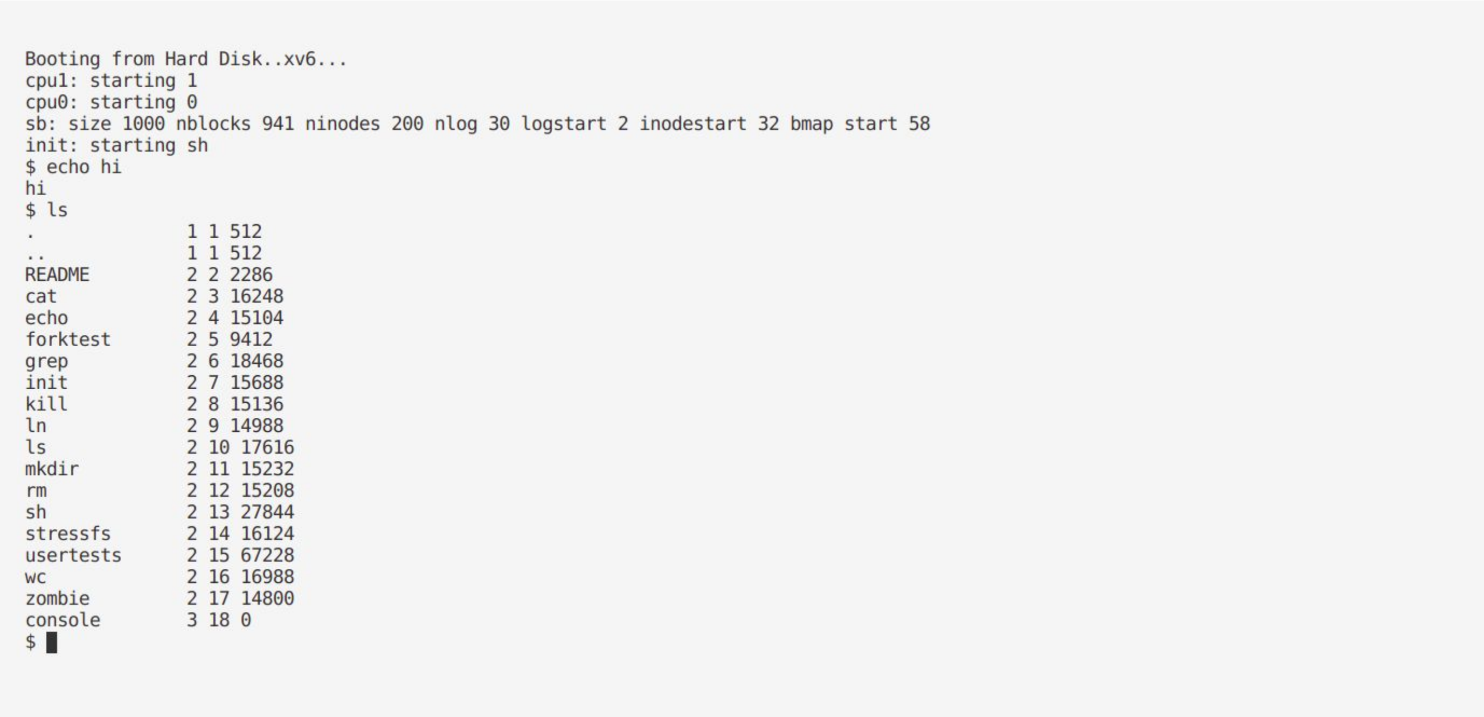
```
  // Lazy page allocation added by us
  if(tf->trapno == T_PGFLT) {
    uint a = PGROUNDDOWN(rcr2());
    char *mem;
    mem = kalloc();
    memset(mem, 0, PGSIZE);
    mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U);
    return;
  }
```

How it works:

We find the virtual address and round it down to find the corresponding page using **PGROUNDDOWN** where **rcr2** register stores the address of the program due to which page fault is created.Physical memory is then assigned using the call to **kalloc()** and page table entries are modified corresponding using mappages function.

The process is actually paused and then this exception is handled and then the process resumes(this is what is termed as **lazy allocation**).

After handling the exception: (echo hi and ls working correctly)



```
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
hi
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16248
echo           2 4 15104
forktest       2 5 9412
grep           2 6 18468
init           2 7 15688
kill           2 8 15136
ln             2 9 14988
ls             2 10 17616
mkdir          2 11 15232
rm             2 12 15208
sh             2 13 27844
stressfs       2 14 16124
usertests      2 15 67228
wc             2 16 16988
zombie         2 17 14800
console        3 18 0
$ █
```

PART B)

CREATING A KERNEL PROCESS:

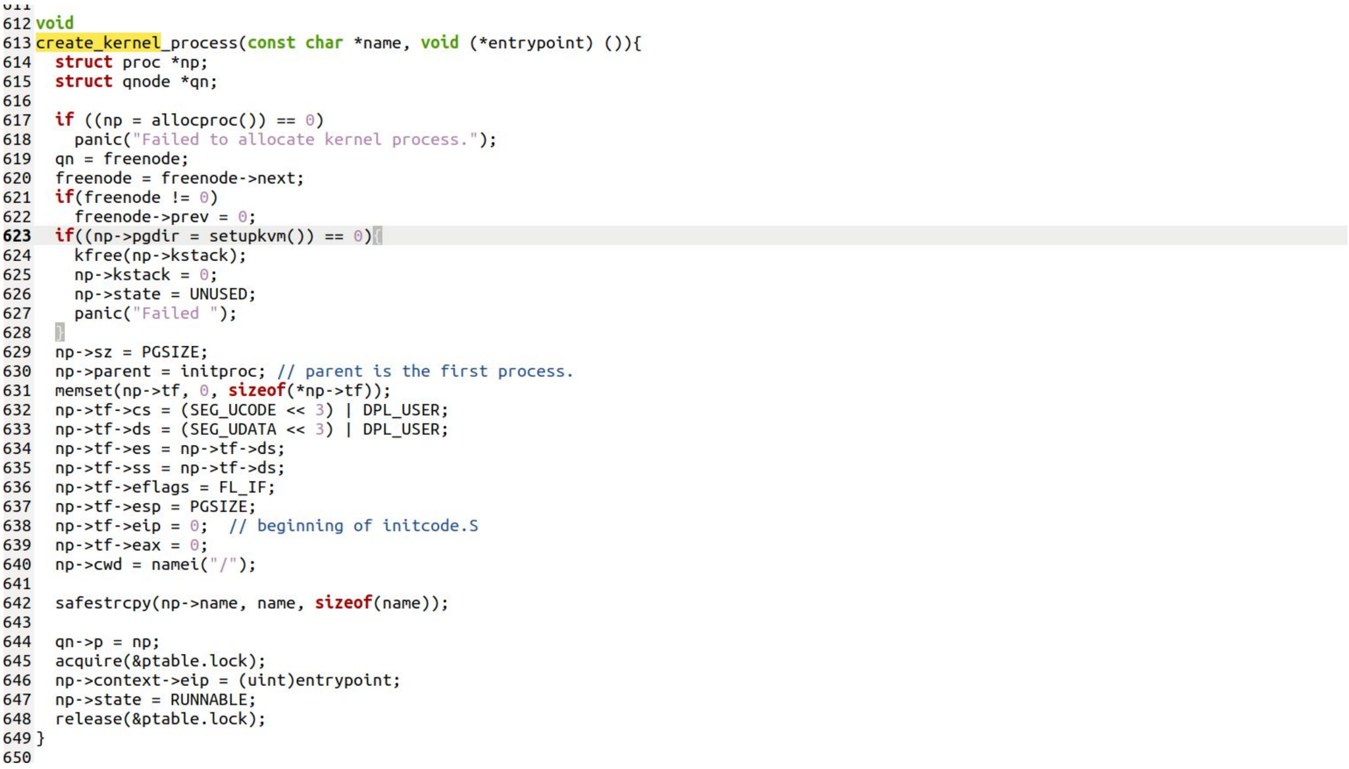
(these functions are created in proc.c)

In order to implement paging mechanism, we are implementing the function void create_kernel_process(const char *name, void (*entrypoint)());

This function creates a kernel process and adds it to the processes queue.

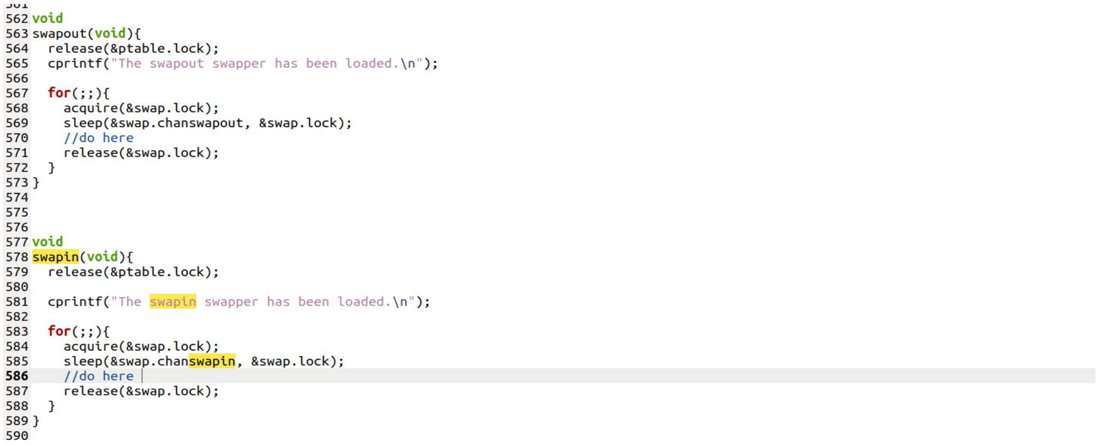
The below code has been implemented in proc.c

create_kernel_process function:



```c
void
create_kernel_process(const char *name, void (*entrypoint) ()){
  struct proc *np;
  struct qnode *qn;

  if ((np = allocproc()) == 0)
    panic("Failed to allocate kernel process.");
  qn = freenode;
  freenode = freenode->next;
  if(freenode != 0)
    freenode->prev = 0;
  if((np->pgdir = setupkvm()) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    panic("Failed ");
  }
  np->sz = PGSIZE;
  np->parent = initproc; // parent is the first process.
  memset(np->tf, 0, sizeof(*np->tf));
  np->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  np->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  np->tf->es = np->tf->ds;
  np->tf->ss = np->tf->ds;
  np->tf->eflags = FL_IF;
  np->tf->esp = PGSIZE;
  np->tf->eip = 0;  // beginning of initcode.S
  np->tf->eax = 0;
  np->cwd = namei("/");

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

  qn->p = np;
  acquire(&ptable.lock);
  np->context->eip = (uint)entrypoint;
  np->state = RUNNABLE;
  release(&ptable.lock);
}
```

Swap in and Swap out kernel processes:



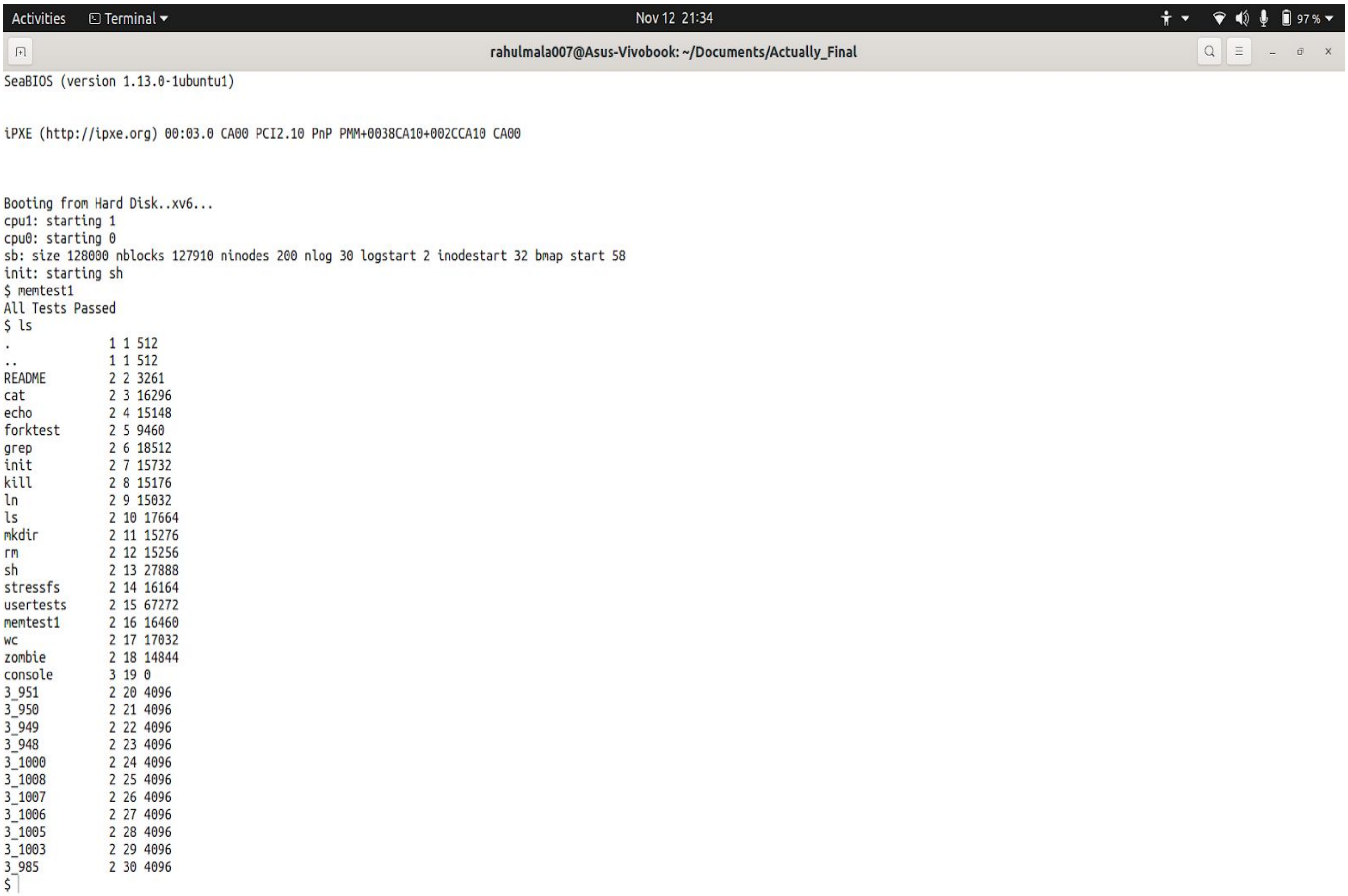
```c
void
swapout(void){
  release(&ptable.lock);
  cprintf("The swapout swapper has been loaded.\n");

  for(;;){
    acquire(&swap.lock);
    sleep(&swap.chanswapout, &swap.lock);
    //do here
    release(&swap.lock);
  }
}



void
swapin(void){
  release(&ptable.lock);

  cprintf("The swapin swapper has been loaded.\n");

  for(;;){
    acquire(&swap.lock);
    sleep(&swap.chanswapin, &swap.lock);
    //do here
    release(&swap.lock);
  }
}
```

SWAPPING IN AND SWAPPING OUT:

To swap out pages we have actually written the evicted out page to the disk and then we have created a file storing the evicted out page and we directly swap in the required page from the disk.

This shows the correct working of the test case that we have created and on ls we can see the swap file created following the file notation specified.)

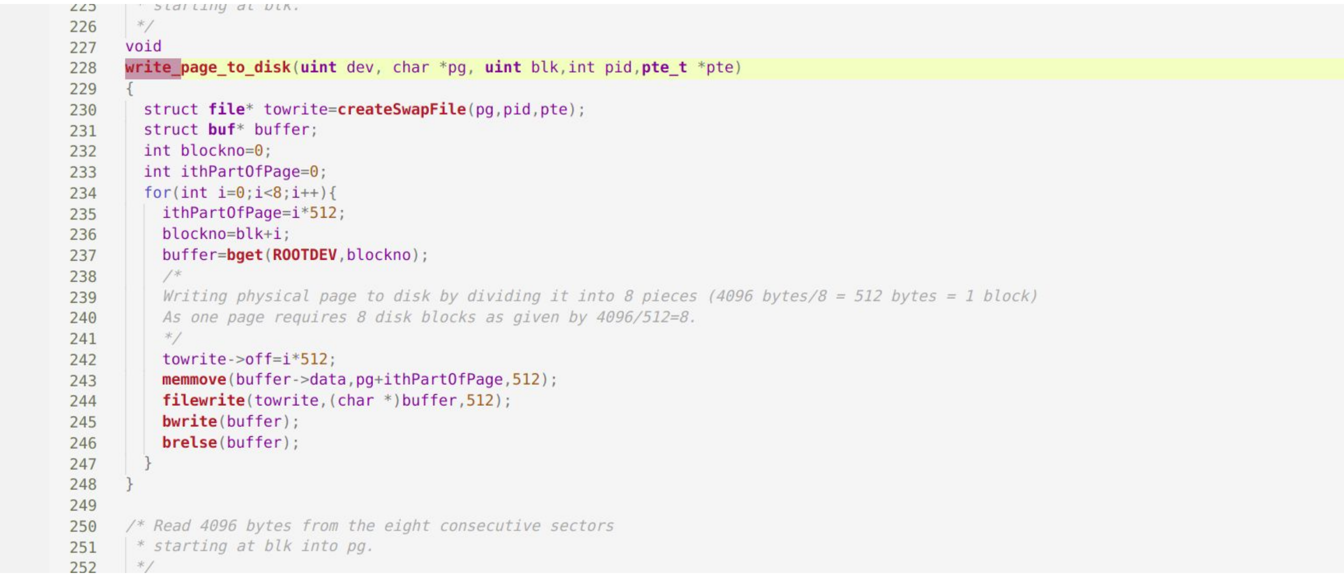
(See 3_951, 3_958.. These are the swap files created)



```
SeaBIOS (version 1.13.0-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+0038CA10+002CCA10 CA00



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 128000 nblocks 127910 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ memtest1
All Tests Passed
$ ls
.              1 1 512
..             1 1 512
README         2 2 3261
cat            2 3 16296
echo           2 4 15148
forktest       2 5 9460
grep           2 6 18512
init           2 7 15732
kill           2 8 15176
ln             2 9 15032
ls             2 10 17664
mkdir          2 11 15276
rm             2 12 15256
sh             2 13 27888
stressfs       2 14 16164
usertests      2 15 67272
memtest1       2 16 16460
wc             2 17 17032
zombie         2 18 14844
console        3 19 0
3_951          2 20 4096
3_950          2 21 4096
3_949          2 22 4096
3_948          2 23 4096
3_1000         2 24 4096
3_1008         2 25 4096
3_1007         2 26 4096
3_1006         2 27 4096
3_1005         2 28 4096
3_1003         2 29 4096
3_985          2 30 4096
$
```

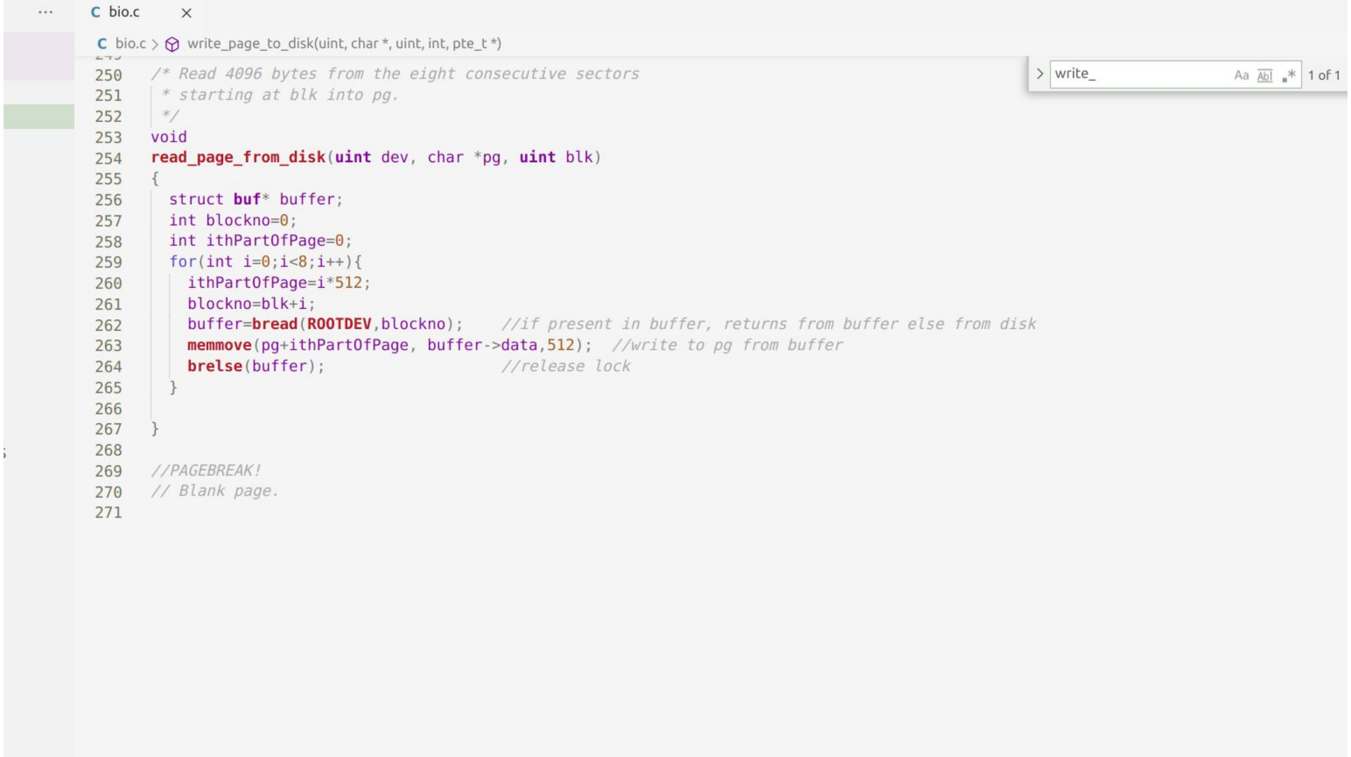
To write to the disk and create the swap file the following functions are used:

write_page_to_disk : this function is used to write the page to disk.



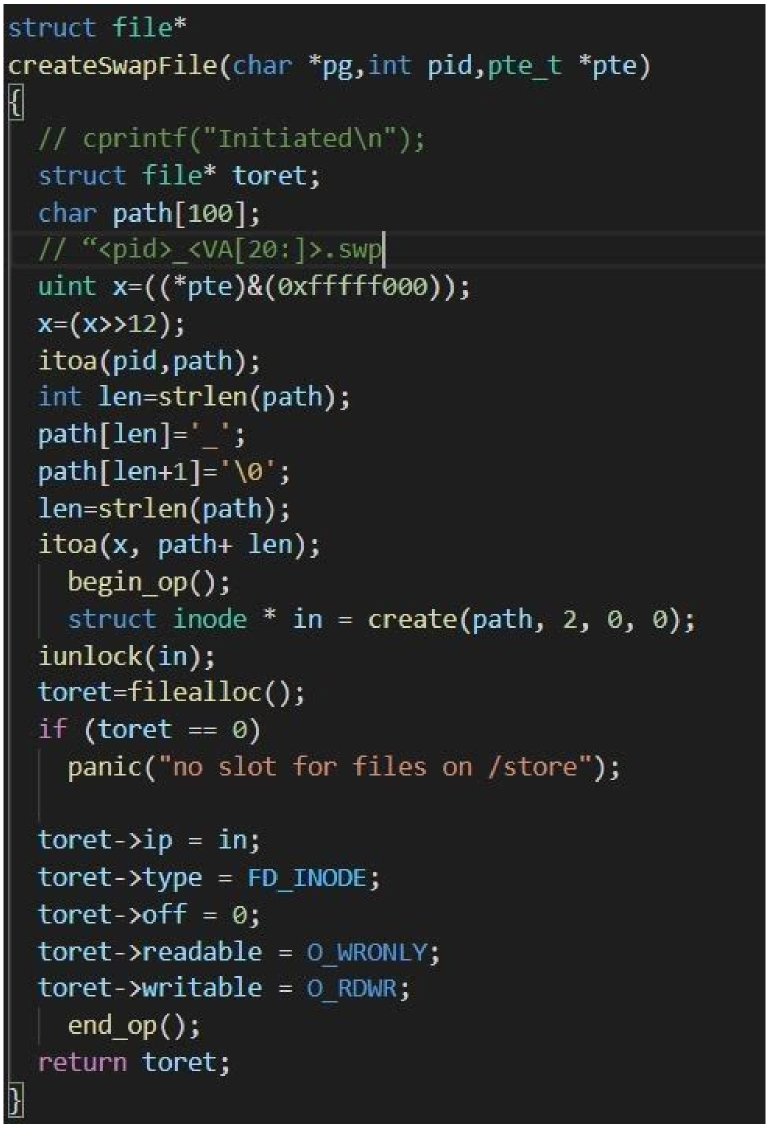
```c
 * starting at blk.
 */
void
write_page_to_disk(uint dev, char *pg, uint blk,int pid,pte_t *pte)
{
  struct file* towrite=createSwapFile(pg,pid,pte);
  struct buf* buffer;
  int blockno=0;
  int ithPartOfPage=0;
  for(int i=0;i<8;i++){
    ithPartOfPage=i*512;
    blockno=blk+i;
    buffer=bget(ROOTDEV,blockno);
    /*
    Writing physical page to disk by dividing it into 8 pieces (4096 bytes/8 = 512 bytes = 1 block)
    As one page requires 8 disk blocks as given by 4096/512=8.
    */
    towrite->off=i*512;
    memmove(buffer->data,pg+ithPartOfPage,512);
    filewrite(towrite,(char *)buffer,512);
    bwrite(buffer);
    brelse(buffer);
  }
}

/* Read 4096 bytes from the eight consecutive sectors
 * starting at blk into pg.
 */
```

read_page_from_disk: this function is used to read page from disk.


bio.c
bio.c > write_page_to_disk(uint, char *, uint, int, pte_t *)
write_
Aa
1 of 1

```c
/* Read 4096 bytes from the eight consecutive sectors
 * starting at blk into pg.
 */
void
read_page_from_disk(uint dev, char *pg, uint blk)
{
  struct buf* buffer;
  int blockno=0;
  int ithPartOfPage=0;
  for(int i=0;i<8;i++){
    ithPartOfPage=i*512;
    blockno=blk+i;
    buffer=bread(ROOTDEV,blockno);    //if present in buffer, returns from buffer else from disk
    memmove(pg+ithPartOfPage, buffer->data,512);  //write to pg from buffer
    brelse(buffer);                   //release lock
  }

}

//PAGEBREAK!
// Blank page.
```

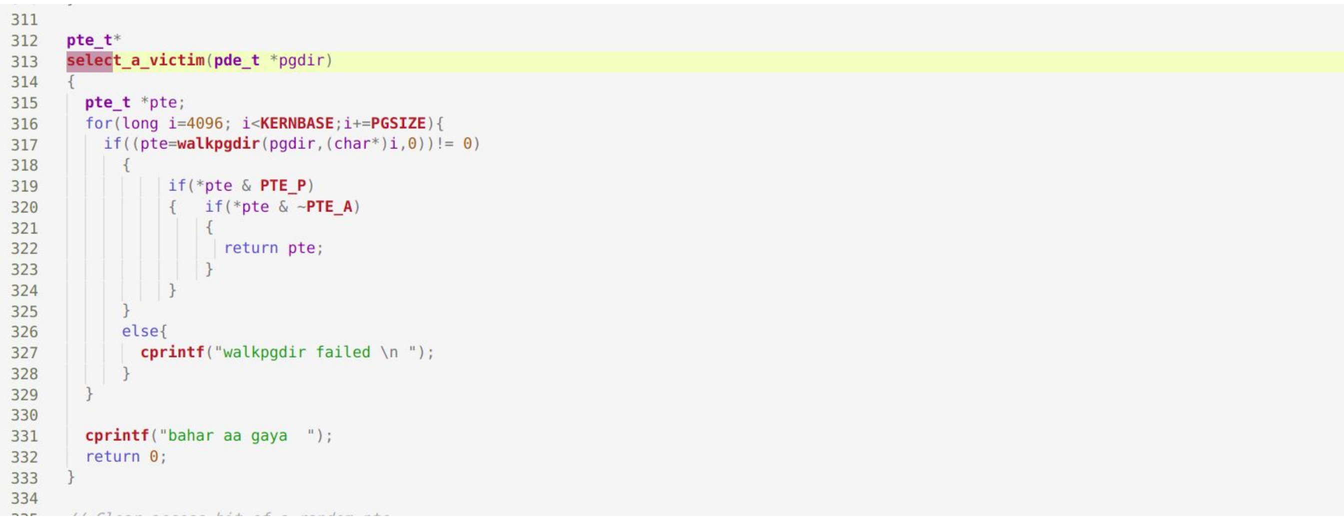
Creation of swap file :



```
struct file*
createSwapFile(char *pg,int pid,pte_t *pte)
{
  // cprintf("Initiated\n");
  struct file* toret;
  char path[100];
  // “<pid>_<VA[20:]>.swp
  uint x=((*pte)&(0xfffff000));
  x=(x>>12);
  itoa(pid,path);
  int len=strlen(path);
  path[len]='_';
  path[len+1]='\0';
  len=strlen(path);
  itoa(x, path+ len);
    begin_op();
    struct inode * in = create(path, 2, 0, 0);
  iunlock(in);
  toret=filealloc();
  if (toret == 0)
    panic("no slot for files on /store");

  toret->ip = in;
  toret->type = FD_INODE;
  toret->off = 0;
  toret->readable = O_WRONLY;
  toret->writable = O_RDWR;
    end_op();
  return toret;
}
```

IMPLEMENTATION OF PAGE REPLACEMENT POLICY:

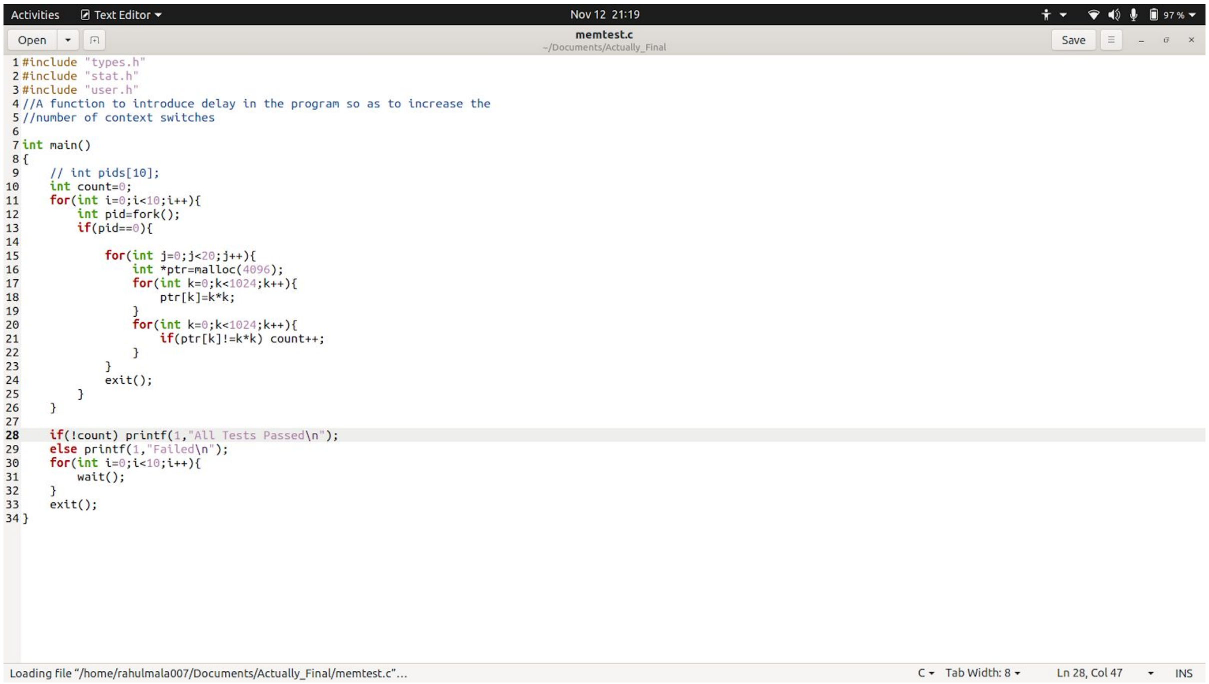
We have implemented an approximate LRU scheme for choosing the page to be evicted the function which is used to select the victim page is shown below:



```
pte_t*
select_a_victim(pde_t *pgdir)
{
  pte_t *pte;
  for(long i=4096; i<KERNBASE;i+=PGSIZE){
    if((pte=walkpgdir(pgdir,(char*)i,0))!= 0)
    {
        if(*pte & PTE_P)
        {   if(*pte & ~PTE_A)
            {
              return pte;
            }
        }
    }
    else{
      cprintf("walkpgdir failed \n ");
    }
  }

  cprintf("bahar aa gaya  ");
  return 0;
}
```

TESTING:

(Code is found in memtest.c)

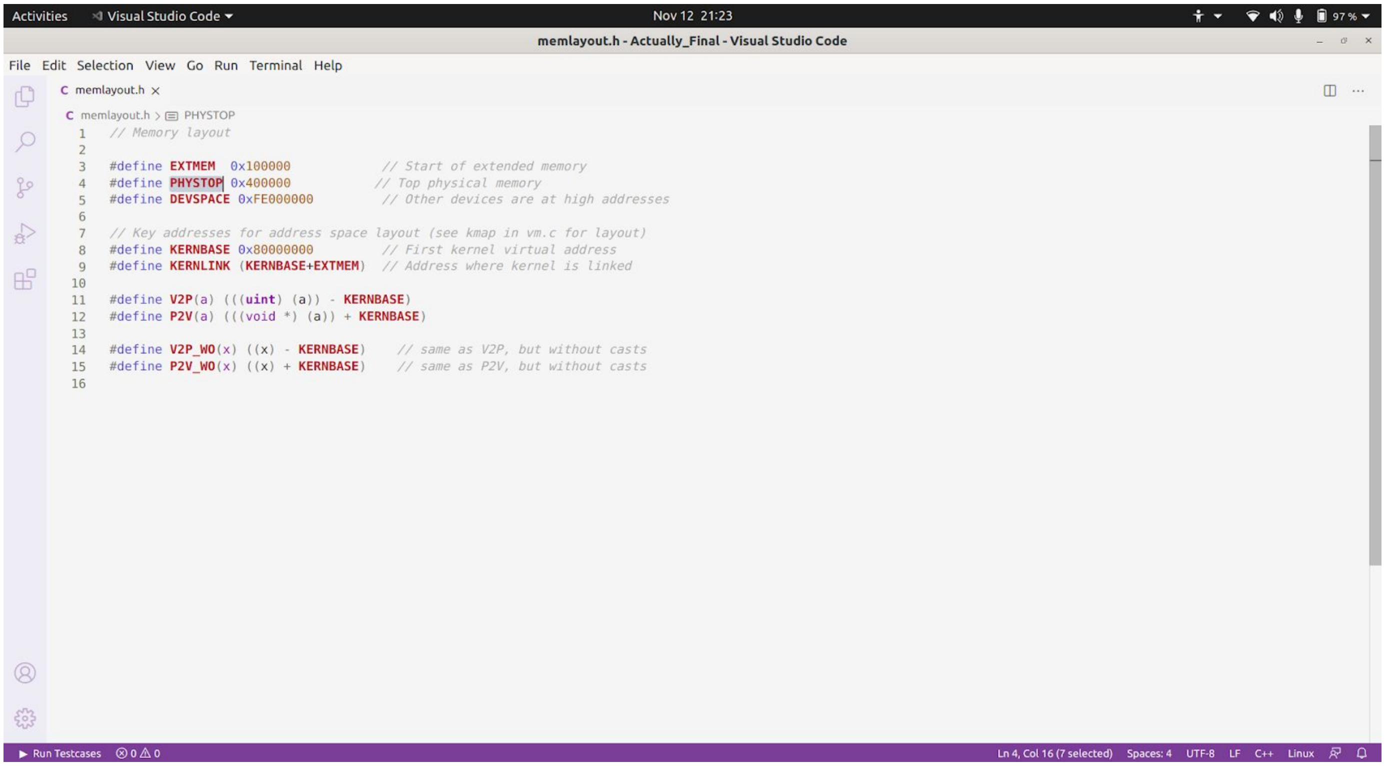


```
#include "types.h"
#include "stat.h"
#include "user.h"
//A function to introduce delay in the program so as to increase the
//number of context switches

int main()
{
    // int pids[10];
    int count=0;
    for(int i=0;i<10;i++){
        int pid=fork();
        if(pid==0){

            for(int j=0;j<20;j++){
                int *ptr=malloc(4096);
                for(int k=0;k<1024;k++){
                    ptr[k]=k*k;
                }
                for(int k=0;k<1024;k++){
                    if(ptr[k]!=k*k) count++;
                }
            }
            exit();
        }
    }

    if(!count) printf(1,"All Tests Passed\n");
    else printf(1,"Failed\n");
    for(int i=0;i<10;i++){
        wait();
    }
    exit();
}
```

Loading file "/home/rahulmala007/Documents/Actually_Final/memtest.c"...

Parent process forks 10 child processes and each has a loop iterating for 20 times.A function f(k)=k*k is used to assign at ptr[k] and final value at ptr[k] is compared to the function .Any changes in the value represents a faulty paging mechanism.(**This is checked using the count variable in memtest.c**).

Note:
PHYSTOP is changed to 0x40000 from 0xE0000 so that the physical memory gets exhausted faster resulting in more page faults.


Activities
Visual Studio Code
Nov 12 21:23
97 %
memlayout.h - Actually_Final - Visual Studio Code
File Edit Selection View Go Run Terminal Help
memlayout.h
memlayout.h > PHYSTOP
Run Testcases
0 0
Ln 4, Col 16 (7 selected) Spaces: 4 UTF-8 LF C++ Linux

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0x400000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```