CIS657 Spring 2019

Assignment 2

Submitted by –

Name – Anish Nesarkar

SUID – 368122582

Subject – Operating Systems

Assignment 2

CIS657 Spring 2019

Assignment Disclosure Form

Assignment #: 2

Name: Anish Nesarkar

1. Did you consult with anyone other than instructor or TA/grader on parts of this assignment?

2. Did you consult an outside source such as an Internet forum or a book on parts of this assignment?

I assert that, to the best of my knowledge, the information on this sheet is true.

Signature:_____Anish Nesarkar_____ Date : 4/13/2019

Design and Implementation:

This Assignment consists of designing a Virtual Memory System in the Nachos. The operating system uses virtual memory during page faults. The main objective of this assignment is to minimize the page faults by doing some modifications in the nachos. The modifications have to be made in the address space, translation entries, paging etc. TLB will be used to speed up address translation. If there is a miss in the TLB, page fault occurs which causes a trap to the OS kernel. The address translation happens, the TLB is loaded with the mapping entry and the program starts running again. FIFO is used as page replacement algorithm during page fault.

The assignment consists of three tasks. The first task is to implement system call and exception handling for user programs. The second task is to implement multiprogramming in round-robin fashion. The third task is to implement inverted page table with software TLB i.e. virtual memory. The bonus task is also implemented in the assignment.

The task 1 requires implementing the following system calls:

1. Halt
2. Exit
3. Fork
4. Yield
5. Read
6. Write

The implementations for the above system calls were done as follows:

Exception.cc

```
void
TestThread(int which)
{
    int num;

        cout << ">-----------< Forked thread output >-----------<" << endl <<
endl;
        printf("*** thread %d looped %d times\n", which, num);
        cout << endl;
        cout << ">-----------------------------------------------<" << endl <<
endl;

}


void
ExceptionHandler(ExceptionType which)
```

```
{
  int len;
    int type = kernel->machine->ReadRegister(2);

    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    switch (which) {
    case SyscallException:
      switch(type) {
      case SC_Halt:
  DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
  cout << endl;
  cout << ">================================< Halt System Call
>====================================<" << endl << endl;
  SysHalt();
  cout <<
">================================================================================
============<" << endl << endl;
  ASSERTNOTREACHED();
  break;

      case SC_Add:

  DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " + " << kernel-
>machine->ReadRegister(5) << "\n");

  /* Process SysAdd Systemcall*/
  int result;
  result = SysAdd(/* int op1 */(int)kernel->machine->ReadRegister(4),
      /* int op2 */(int)kernel->machine->ReadRegister(5));

  DEBUG(dbgSys, "Add returning with " << result << "\n");
  /* Prepare Result */
  kernel->machine->WriteRegister(2, (int)result);

  /* Modify return point */
  {
    /* set previous programm counter (debugging only)*/
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));

    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);
```

```
    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);
  }

  return;

  ASSERTNOTREACHED();

  break;


case SC_Exit:

    SysExit((int)kernel->machine->ReadRegister(4));

{
    /* set previous programm counter (debugging only)*/
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));

    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);

    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);
  }

  return;


case SC_Read:

cout << endl;
cout << ">==================================< Read System Call
>====================================<" << endl << endl;
    cout << ">-------------< Reading from Console >-----------------< " << endl
<< endl;

    SysRead((int)kernel->machine->ReadRegister(4),(int)kernel->machine-
>ReadRegister(5),(int)kernel->machine->ReadRegister(6));
    cout << "The above string is read from Console!!" << endl << endl;
```

```
    cout << ">-------------< Done Reading >-----------------< " << endl << endl;
{
    /* set previous programm counter (debugging only)*/
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));

    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);

    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);
  }
cout <<
">=============================================================================
============<" << endl << endl;
  return;

case SC_Write:
{
    cout << endl;
    cout << ">==================================< Write System Call
>====================================<" << endl << endl;
    cout << ">-----------------< Writing the following on Console: >------------
-----< " << endl << endl;


      len = SysWrite((int)kernel->machine->ReadRegister(4),(int)kernel->machine-
>ReadRegister(5),(int)kernel->machine->ReadRegister(6));
      kernel->machine->WriteRegister(2, (int)kernel->machine->ReadRegister(5));
      cout << "Done Writing onto console!!" << endl << endl;
      cout << ">------------------< Done Writing >----------------< " << endl <<
endl;


    /* set previous programm counter (debugging only)*/
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));

    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);
```

```
    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);
  }
  cout <<
">=============================================================================
============<" << endl << endl;
  return;

case SC_UserYield:
{
  cout << endl;
  cout << ">==================================< Yield System Call
>====================================<" << endl << endl;
    SysUserYield();
    /* set previous programm counter (debugging only)*/
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));

    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);

    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);

}
    //ASSERTNOTREACHED();
  return;

case SC_UserFork:
{
cout << endl;
  cout << ">==================================< Fork System Call
>====================================<" << endl << endl;
    Thread *t = new Thread("forked thread");
    t->Fork((VoidFunctionPtr) TestThread, (void *) 1);

    kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));
```

```
    /* set programm counter to next instruction (all Instructions are 4 byte
wide)*/
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
4);

    /* set next programm counter for brach execution */
    kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg)+4);


cout <<
">=================================================================================
============<" << endl << endl;
}
  return;

      default:
  cerr << "Unexpected system call " << type << "\n";

  break;
      }
      break;

  //Page fault exception
   case PageFaultException:
   {
     cout << endl;
     kernel->stats->numPageFaults++;
    cout << ">==================================< Page Fault
>====================================<" << endl << endl;
    unsigned int vpn, offset;

      //bad virtual address will give vpn not there in main memory
      int badVaddress = kernel->machine->ReadRegister(39);
      vpn = (unsigned) badVaddress / PageSize;
      cout << "Virtual Page Number : " << vpn << endl;

      int getSwapCount = kernel->vpnToSwapMap[vpn];
      cout << "Getting Page from Swap Space.." << endl;
      //copy from swap space to main memory

      int ppn = kernel->physicalPagesBitmap->FindAndSet();

    char *buffer = new char[PageSize];
    kernel->swapExecutable->ReadAt(buffer, PageSize, getSwapCount * PageSize);
```

```
    //if main memory is empty copy directly from swap space to main memory and in
TLB
    if(ppn != -1)
    {
        cout << "Space available in Main Memory!!!"<< endl;
        cout << "Available Physical Page : " << ppn << endl;
        cout << "Adding Entry in TLB.." << endl;
        for(int i = 0; i < TLBSize ; i++)
      {
        if(kernel->machine->tlb[i].physicalPage = ppn)
        {
        kernel->machine->tlb[i].valid = TRUE;
        kernel->machine->tlb[i].virtualPage = vpn;
        kernel->machine->tlb[i].physicalPage = ppn;
        kernel->machine->tlb[i].id = kernel->currentThread->getID();
        break;
        }
      }

        cout << "Adding Entry in IPT.." << endl;
        kernel->machine->pageTable[ppn].valid = TRUE;
        kernel->machine->pageTable[ppn].virtualPage = vpn;
        kernel->machine->pageTable[ppn].physicalPage = ppn;
        kernel->machine->pageTable[ppn].id = kernel->currentThread->getID();

        //clear the existing space
        bzero(&(kernel->machine->mainMemory[ppn * PageSize]), PageSize);

        //copy from buffer to main memory
        bcopy(buffer, &(kernel->machine->mainMemory[ppn * PageSize]), PageSize);
        cout << "Loading to Main Memory" << endl;
        //add page table to list of pages used ( for FIFO to remove later )
        kernel->listOfPagesUsed->Append(ppn);

    }

    //if all pages in main memory are occupied
    //remove the first page according to FIFO.(also remove from the list)
    //get swap space count of the removed page and store the page back to the
swap executable
    //then clear the main memory and copy new page from swap executable to main
memory
    else
    {
```

```
      ppn = kernel->listOfPagesUsed->RemoveFront();
       cout << ">----< Main Memory Full >---< Page Replacement using FIFO >---<"
<< endl;
       //get the evicted swap count
       int swapCountEvicted = kernel->vpnToSwapMap[kernel->machine-
>pageTable[ppn].virtualPage];
      // cout << "evicted swap count " << swapCountEvicted << endl;
        char *tempbuffer = new char[PageSize];
       //copy from memory to buffer
       bcopy(&(kernel->machine->mainMemory[ppn * PageSize]), tempbuffer,
PageSize);
        //copy from buffer to swap file
       kernel->swapExecutable->WriteAt(tempbuffer,PageSize,swapCountEvicted *
PageSize);
      cout << "Evicted Physical Page : " << ppn << endl;

//////////////////////////////////////

    //load page in tlb

      for(int i = 0; i < TLBSize ; i++)
      {

        if(kernel->machine->tlb[i].physicalPage == ppn)
        {
          cout << "Adding Entry in TLB.." << endl;
        kernel->machine->tlb[i].valid = TRUE;
        kernel->machine->tlb[i].virtualPage = vpn;
        kernel->machine->tlb[i].physicalPage = ppn;
        kernel->machine->tlb[i].id = kernel->currentThread->getID();
        }
      }

      cout << "Adding Entry in IPT.." << endl;

    //change entry of new page - vpn and process ID
      kernel->machine->pageTable[ppn].valid = TRUE;
      kernel->machine->pageTable[ppn].virtualPage = vpn;
      kernel->machine->pageTable[ppn].physicalPage = ppn;
      kernel->machine->pageTable[ppn].id = kernel->currentThread->getID();
      //clear main memory
      bzero(&(kernel->machine->mainMemory[ppn * PageSize]), PageSize);

      //copy from buffer to main memory
        bcopy(buffer, &(kernel->machine->mainMemory[ppn * PageSize]), PageSize);
```

```
      cout << "Loading Page to Main Memory from swap space..." << endl;
      cout << ">------< Page Replacement Successful >-------< " << endl;

        //add page table to list of pages used ( for FIFO to remove later )
        kernel->listOfPagesUsed->Append(ppn);
     delete tempbuffer;

    }
     delete buffer;


     cout << endl;
     cout << ">================================< Page Fault Handled
>==================================<" << endl << endl;

   }
  return;
    default:
      cerr << "Unexpected user mode exception" << (int)which << "\n";
      break;
    }
    ASSERTNOTREACHED();

}
```

- The system call Exceptions for task 1 are called in the switch - case statements.
- The systems calls are made for each exception and the program counter register is incremented for the next instruction
- There is also a case for page fault exception for task 3.
- If there is a page fault, following events happen in the handler
    - The bad virtual address is read from the register 39 and virtual page number is calculated
    - Corresponding swap count is returned from the map for that virtual page number.
    - A page number is found from the bitmap if it is available
    - If any page is empty in the main memory, the page is loaded from swap file and put into main memory
    - The page is loaded into tlb also.
    - If the Main memory is full, the page is removed according to FIFO Page replacement algorithm.
    - The evicted page is stored back to the swap file according to its swap count.

- After evicted page is stored back to swap file, the required page is loaded back to the same evicted page position.
- Thus, Page replacement is successful.

Syscall.h

```
#define SC_Halt     0
#define SC_Exit     1
#define SC_Exec     2
#define SC_Join     3
#define SC_Create   4
#define SC_Remove       5
#define SC_Open     6
#define SC_Read     7
#define SC_Write    8
#define SC_Seek         9
#define SC_Close    10
#define SC_UserFork 11
#define SC_UserYield    12
#define SC_ExecV    13
#define SC_ThreadExit   14
#define SC_ThreadJoin   15
```

- The Syscall.h defines the systems calls with a number as follows
- Halt – This system call halts the user program - 0
- Exit – This system call exits the current user program - 1
- Useryield – This system call yields the current user thread - 12
- Userfork – This system call forks the thread in the Kernel - 11
- read – This System call takes the user input from the Console - 7
- Write – This System call writes the output to the console - 8

Ksyscall.h

```
//halt System Call
void SysHalt()
{
  kernel->interrupt->Halt();
```

```
}


int SysAdd(int op1, int op2)
{
  return op1 + op2;
}

//Exit System call
void SysExit(int s)
{

  if(s == 0)
    {
    for (int i = 0; i < TLBSize; i++)

    {
      if(kernel->machine->tlb[i].id == kernel->currentThread->getID())
    {

      kernel->physicalPagesBitmap->Clear(i);
            kernel->machine->tlb[i].valid = FALSE;
          kernel->machine->tlb[i].use = FALSE;
          kernel->machine->tlb[i].dirty = FALSE;
          kernel->machine->tlb[i].readOnly = FALSE;
            kernel->machine->tlb[i].id = -1;
            kernel->machine->tlb[i].virtualPage = -1;
            //clearing out the occupied main memory
            bzero(&(kernel->machine->mainMemory[i * PageSize]),PageSize);
    }
    }

    //clearing the page table entry associated with that process
    for(int i = 0; i < kernel->machine->pageTableSize; i++)
  {

    if(kernel->machine->pageTable[i].id == kernel->currentThread->getID())
    {

      kernel->physicalPagesBitmap->Clear(i);
            kernel->machine->pageTable[i].valid = FALSE;
          kernel->machine->pageTable[i].use = FALSE;
          kernel->machine->pageTable[i].dirty = FALSE;
          kernel->machine->pageTable[i].readOnly = FALSE;
            kernel->machine->pageTable[i].id = -1;
            kernel->machine->pageTable[i].virtualPage = -1;
```

```
            kernel->listOfPagesUsed->Remove(i);
            //clearing out the occupied main memory
            bzero(&(kernel->machine->mainMemory[i * PageSize]),PageSize);
    }
  }
  //displaying performance statistics
      cout << endl;
      cout << ">================================< User Program : " << kernel-
>currentThread->getName() << " : Exiting >============================<" << endl
<< endl;
      cout << ">------------------< Displaying statistics of the User program >--
------------------<" << endl << endl;
      cout << "Number of Page Hits : " << kernel->stats->numPageHits << endl;
      cout << "Number of Page Faults : " << kernel->stats->numPageFaults << endl
<< endl;
      cout << ">----------------------------------------------------------------
------------------<" << endl << endl;
      cout <<
">================================================================================
============================<" << endl << endl;

      kernel->currentThread->Finish();

    }

}

//Yield system call to yield user thread
void SysUserYield()
{
  cout << endl;
  cout << ">---------< Yield test >---------< " << endl << endl;
  cout << "User Thread Yielding!!" << endl << endl;
  cout << ">--------------------------------<" << endl << endl;
  kernel->currentThread->Yield();
}

//Write system call to write onto console
int SysWrite(int b, int s,int r)
{
  int buffer = b;
   int size = s;
  int outputBuffer;
  int count = 0;
  string writeoutput;
```

```
  while(count != size)
  {
    kernel->machine->ReadMem(buffer, 1, &outputBuffer);
    writeoutput = writeoutput + (char)outputBuffer;
    count++;
    buffer++;
  }
  cout << ">" << writeoutput << "<" << endl;
  return 0;
}

//Read System Call to read from the console
void SysRead(int b, int s,int r)
{
  char buf[s];
  cout << "Enter string to read from Console: " << endl;
  cin >> buf;
  cout << "The ouput read from Console is : " << buf << endl;
}
```

- The above file gives the implementation of the system calls
- The Halt system call halts the program
- The Exit system call exits the program if the argument for Exit is “0”.
    - All the entries of the program stored in inverted page table is removed and the memory is cleared out.
- The yield system call yields the user program.
- The fork system call forks the user program thread
- The read system call reads an input from the console and stores in the memory.
- The write system call writes the output to the console.

The following user programs are used to test the task1 system calls.

- forktest.c
- yieldtest.c
- readtest.c
- writetest.c
- exittest.c
- halt.c

The Virtual memory along with TLB (bonus) is designed and implemented as follows:

machine.cc

```
//create inverted page table

pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = -1;
    pageTable[i].physicalPage = i;
    pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    pageTable[i].id = -1;
    }

    pageTableSize = NumPhysPages;
//creating TLB of size 4
    tlb = new TranslationEntry[TLBSize];
    for (int i = 0; i < TLBSize; i++) {
    tlb[i].virtualPage = i;
    tlb[i].physicalPage = -1;
    tlb[i].valid = FALSE;
    tlb[i].use = FALSE;
    tlb[i].dirty = FALSE;
    tlb[i].readOnly = FALSE;
    tlb[i].id = -1;
    }
```

- Inverted page table of type Translational Entry is created and the pageTable size is set to number of physical pages.
- The inverted page table is indexed by the physical page number.
- TLB is created with TLB size of 4.

main.cc

```
else if (strcmp(argv[i], "-x") == 0) {
        ASSERT(i + 1 < argc);
```

```
        userProgName = argv[i + 1];
        userproglist->Append(userProgName);
        i++;
    }
```

```
// run an user program if requested to do so
    while (!userproglist->IsEmpty()) {

        char * userprog = userproglist->RemoveFront();
        Thread *t = new Thread(userprog);
        t->Fork((VoidFunctionPtr)RunUserProg, (void *) userprog);
        cout << "    " << userprog << endl;
    }
    cout << endl;
    cout << "========================== Number of Physical Pages = " <<
NumPhysPages << " ==========================" << endl << endl;
    delete userproglist;
```

- A list is used to load multiple user programs into the nachos.
- The arguments read from the console are stored in the list.
- Each item of the list is the user program which is loaded and executed in the RunUserProg function.

kernel.cc

```
physicalPagesBitmap = new Bitmap(NumPhysPages);
    listOfPagesUsed = new List<int>();
```

```

    kernel->fileSystem->Create("swap");
    swapExecutable = kernel->fileSystem->Open("swap");
```

- In the above file, a bitmap is used as data structure to get the available physical pages.
- Swap space is created
- A list is created to store the number of physical pages stored for FIFO later as page replacement algorithm if there is page fault.

Address space

```
    cout << ">============================< Loading
>==============================<" << endl << endl;
    cout << "Initializing address space for thread : " << kernel->currentThread-
>getName() << endl;
    cout << "Number of Pages of the Thread : "<< numPages << endl;
    cout << "Size of the Thread : " << size << endl << endl;
    cout <<
">=============================================================================<" <<
endl << endl;

int ppn;

//page table

    char *buffer;

for(int i = 0; i < numPages; i++)
{
 //make an entry in map for swap location
    kernel->vpnToSwapMap.insert({i,getNextSwapLoation()});
    //temporary buffer
   buffer = new char[PageSize];
   //read from executable into the buffer
    executable->ReadAt(buffer,PageSize,noffH.code.inFileAddr + i * PageSize);
    //get the available page
    ppn = kernel->physicalPagesBitmap->FindAndSet();
    //if page is availabe load, into main memory
    if(ppn != -1)
    {
        //load into tlb
        if(i < TLBSize)
        {

            kernel->machine->tlb[i].valid = TRUE;
        kernel->machine->tlb[i].virtualPage = i;
        kernel->machine->tlb[i].physicalPage = ppn;
        kernel->machine->tlb[i].id = kernel->currentThread->getID();
        }

        kernel->machine->pageTable[ppn].valid = TRUE;
        kernel->machine->pageTable[ppn].virtualPage = i;
        kernel->machine->pageTable[ppn].physicalPage = ppn;
        kernel->machine->pageTable[ppn].id = kernel->currentThread->getID();
```

```
        //clear the existing space
        bzero(&(kernel->machine->mainMemory[ppn * PageSize]), PageSize);

        //copy from buffer to main memory
        bcopy(buffer, &(kernel->machine->mainMemory[ppn * PageSize]), PageSize);

        //add page table to list of pages used ( for FIFO to remove later )
        kernel->listOfPagesUsed->Append(ppn);

    }
    //write into the swap executable
    kernel->swapExecutable->WriteAt(buffer,PageSize,kernel->vpnToSwapMap[i] *
PageSize);

}
delete [] buffer;

    delete executable;          // close file

    return TRUE;            // success
}
```

In Address space, the load method is modified for virtual memory implementation as follows:

- The main memory and swap file are loaded with the number of pages of the user program.
- If the page is loaded to main memory, the valid bit is set “true” and other information about the page are stored in page table like virtual page number, physical page number and thread ID.
- Similarly set all the parameters for the TLB.
- If the main memory is full, the pages are stored to swap file.

translate.cc

```
// calculate the virtual page number, and offset within the page,
// from the virtual address
    vpn = (unsigned) virtAddr / PageSize;
    offset = (unsigned) virtAddr % PageSize;

    //look for page into tlb first
    if(tlb != NULL)
```

```
    {
        for(int i = 0;i < TLBSize; i++)
        {
            if(tlb[i].id == kernel->currentThread->getID() && tlb[i].virtualPage
== (int)vpn)
        {
            kernel->stats->tlbHits++;
            entry = &tlb[i];
            break;
        }

        }

    }
            // => page table => vpn is index into table
    //check if there is entry for current process in IPT
    if(entry == NULL || !entry->valid)
    {
    for(int i = 0; i < pageTableSize; i++)
    {
        if(pageTable[i].id == kernel->currentThread->getID() &&
pageTable[i].virtualPage == (int)vpn)
        {
        kernel->stats->numPageHits++;
            entry = &pageTable[i];
            break;
        }

    }
    }

    //if there is no entry or entry not valid => page fault exception
    if(entry == NULL || !entry->valid)
    {

        return PageFaultException;
    }
```

- The translate function is modified for virtual memory implementation as follows:
- The tlb is scanned and if the thread ID of currently running thread is same as id in the tlb and if there exists virtual page number in the tlb, the entry is found and execution continues.

- The Inverted page table is scanned and if the thread ID of currently running thread is same as id in the inverted page table and if there exists virtual page number in the inverted page table, the entry is found and execution continues.
- If there is no entry in the page table or if the entry is invalid, there is page fault exception and Exception handler is called to handle the page fault and bring in the page from the swap space.

Deleting Garbage Collections

1. Exception.cc and Addrspace.cc

```
    delete [] buffer;
    delete [] tempbuffer;
```

2. In kernel.cc

```
   delete physicalPagesBitmap;
    delete listOfPagesUsed;
    delete swapExecutable;
```

3. In main.cc

```
delete userproglist;
```

4. In Machine.cc

```
delete [] mainMemory;
    delete[] pageTable;
    if (tlb != NULL)
        delete [] tlb;
```

Testing

How to run the Test

1. Copy the nachos folder from your local machine to the server.
2. Copy the modified Machine.cc, Main.cc, Kernel.cc, addressspace.cc, translate.cc, exception.cc, syscall.h, ksyscall.h from the local machine to the server
3. Navigate to the directory nachos/code/build.linux
4. Execute the below commands
    a. make clean
    b. make depend

c. make nachos

5. To run the user programs, execute the commands as below
   a. ./nachos -x ../test/writetest (for write user program test)
   b. ./nachos -x ../test/readtest (for read user program test)
   c. ./nachos -x ../test/yieldtest (for yield user program test)
   d. ./nachos -x ../test/forktest (for fork user program test)
   e. ./nachos -x ../test/exittest (for exit user program test)
   f. ./nachos -x ../test/halt (for halt user program test)
6. To run multi-programs, execute as follow :

   Ex. ./nachos -x ../test/writetest -x ../test/matmult

Added Files:

writetest.c , readtest.c, yieldtest.c, forktest.c, exittest.c to the location /nachos/code/test/

Modified Files:

translate.cc in the location /nachos/code/machine/

machine.cc in the location /nachos/code/machine/

thread.cc in the location /nachos/code/threads/

kernel.cc in the location /nachos/code/threads/

main.cc in the location /nachos/code/threads/

syscall.h in the location /nachos/code/userprog/

ksyscall.h in the location /nachos/code/userprog/

exception.cc in the location /nachos/code/userprog/

addrspace.cc in the location /nachos/code/userprog/

Makefile (In test folder)

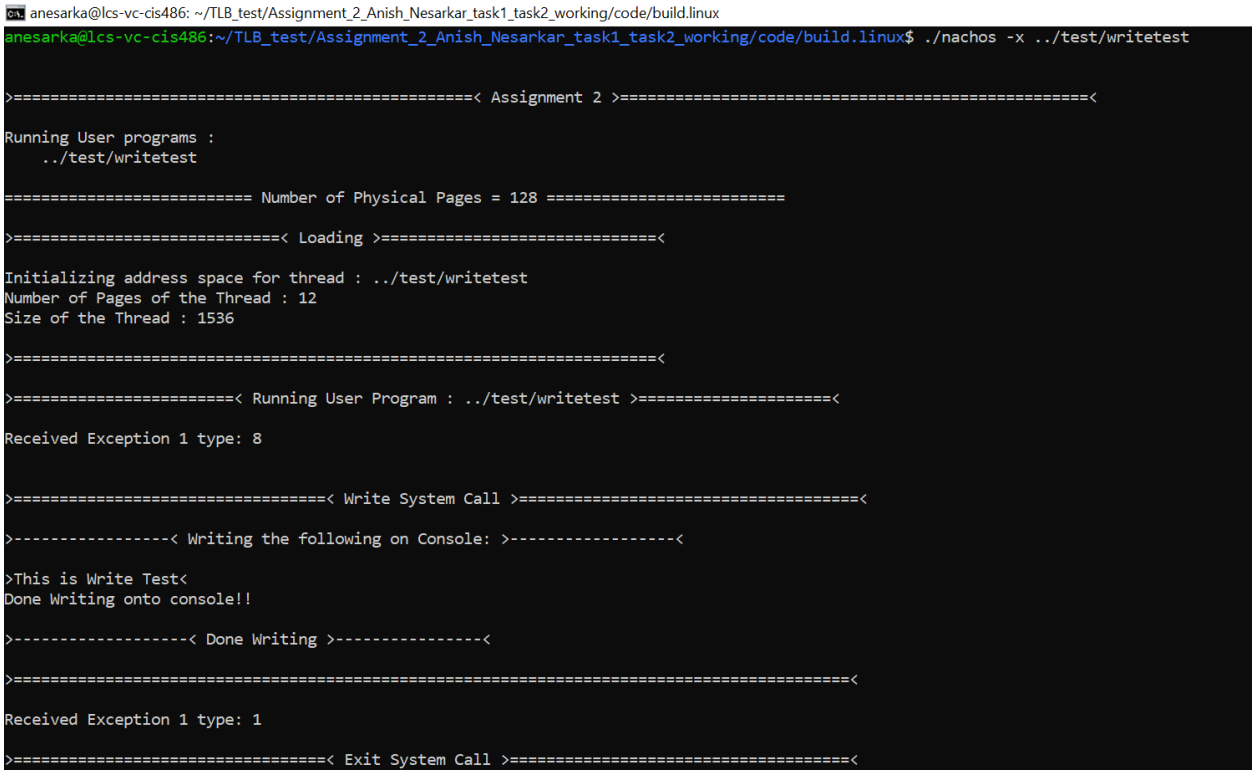
```
# change this if you create a new test program!
PROGRAMS = add halt shell matmult sort segments yieldtest writetest forktest
readtest exittest
endif
```

```
all: $(PROGRAMS)

start.o: start.S ../userprog/syscall.h
	$(CC) $(CFLAGS) $(ASFLAGS) -c start.S

halt.o: halt.c
	$(CC) $(CFLAGS) -c halt.c
halt: halt.o start.o
	$(LD) $(LDFLAGS) start.o halt.o -o halt.coff
	$(COFF2NOFF) halt.coff halt

yieldtest.o: yieldtest.c
	$(CC) $(CFLAGS) -c yieldtest.c
yieldtest: yieldtest.o start.o
	$(LD) $(LDFLAGS) start.o yieldtest.o -o yieldtest.coff
	$(COFF2NOFF) yieldtest.coff yieldtest

writetest.o: writetest.c
	$(CC) $(CFLAGS) -c writetest.c
writetest: writetest.o start.o
	$(LD) $(LDFLAGS) start.o writetest.o -o writetest.coff
	$(COFF2NOFF) writetest.coff writetest

forktest.o: forktest.c
	$(CC) $(CFLAGS) -c forktest.c
forktest: forktest.o start.o
	$(LD) $(LDFLAGS) start.o forktest.o -o forktest.coff
	$(COFF2NOFF) forktest.coff forktest

readtest.o: readtest.c
	$(CC) $(CFLAGS) -c readtest.c
readtest: readtest.o start.o
	$(LD) $(LDFLAGS) start.o readtest.o -o readtest.coff
	$(COFF2NOFF) readtest.coff readtest

exittest.o: exittest.c
	$(CC) $(CFLAGS) -c exittest.c
exittest: exittest.o start.o
	$(LD) $(LDFLAGS) start.o exittest.o -o exittest.coff
	$(COFF2NOFF) exittest.coff exittest
```

OUTPUT

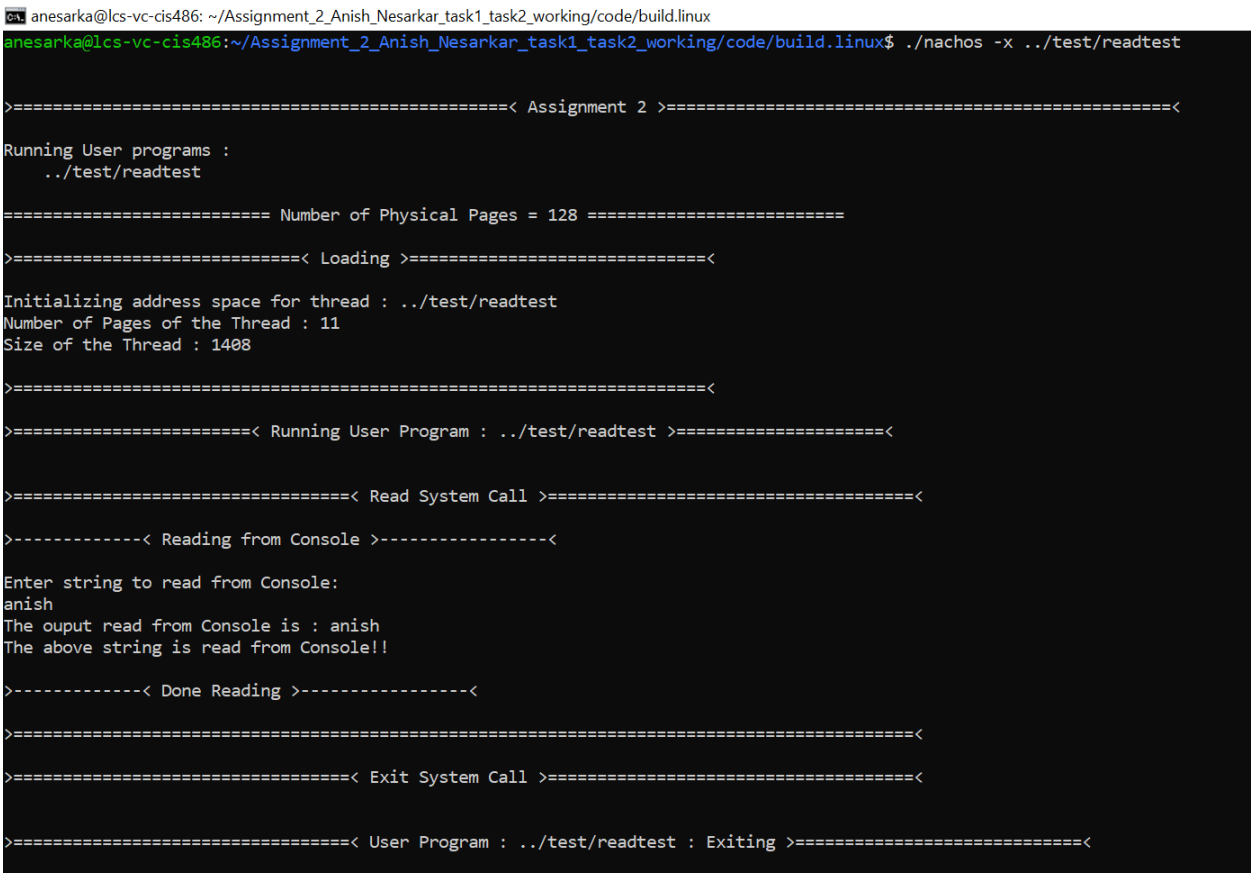
Task 1 – System Calls

1. Write System Call



- The write system call is called when the user program calls write.
- The above output shows the system call exception when something needs to written onto the console.

2. Read System Call



- The user is prompted with an input from the console to be written into the memory.

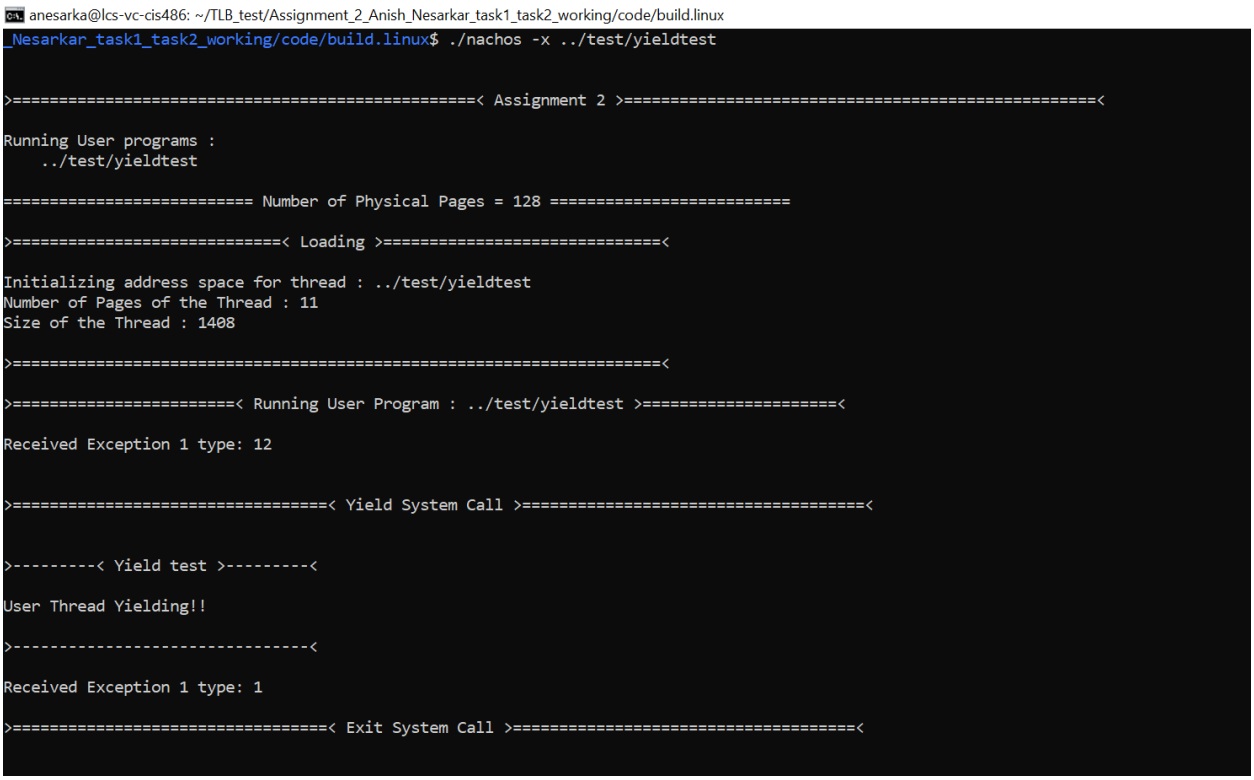
3. Fork System Call

anesarka@lcs-vc-cis486: ~/TLB_test/Assignment_2_Anish_Nesarkar_task1_task2_working/code/build.linux

```
Virtual Page Number : 0
Getting Page from Swap Space..
Space available in Main Memory!!!
Copying to Main Memory

>================================< Page Fault Handled >=================================<

Received Exception 1 type: 1

>================================< Exit System Call >===================================<


>================================< User Program : ../test/forktest : Exiting >============================<

>-----------------< Displaying statistics of the User program >--------------------<

Number of TLB Hits : 19
Number of Page Hits : 2
Number of Page Faults : 3

>-------------------------------------------------------------------------------------<

>=====================================================================================================<

>-----------< Forked thread output >-----------<

*** thread 1 looped 0 times

>---------------------------------------------<
```

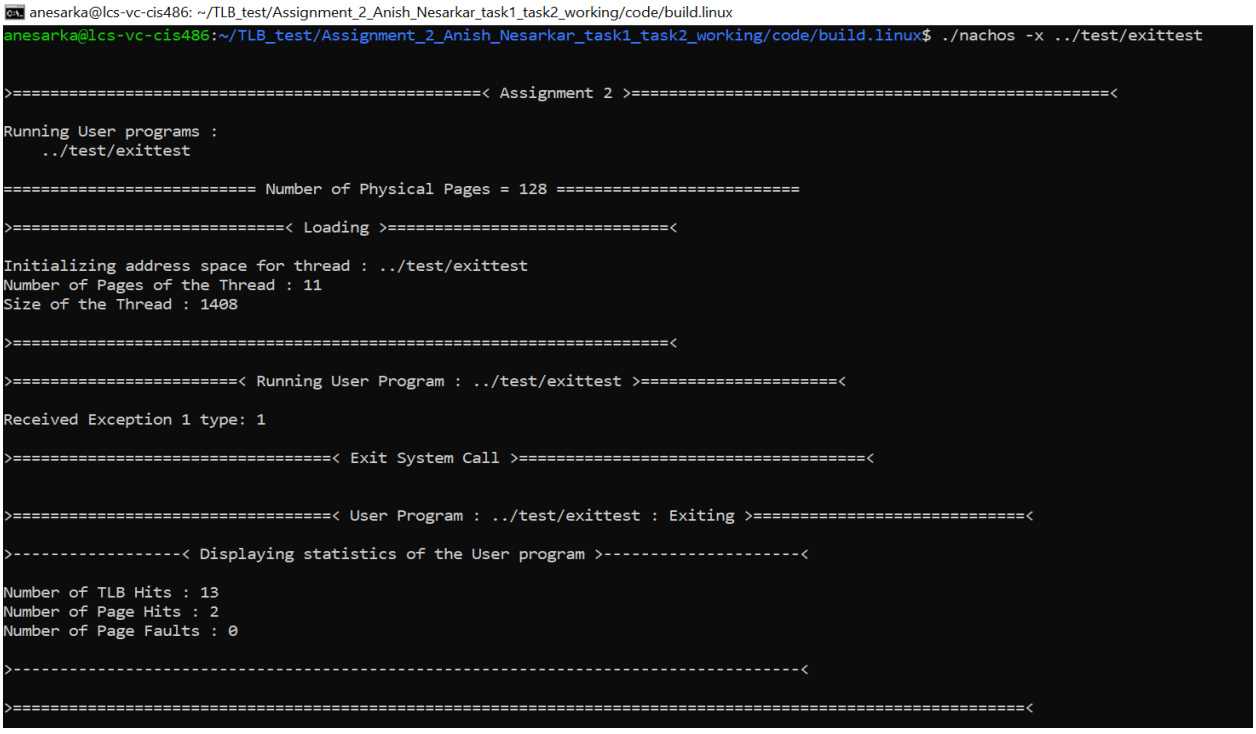
- The fork system call forks the thread in the kernel as shown above.

4. Yield System Call



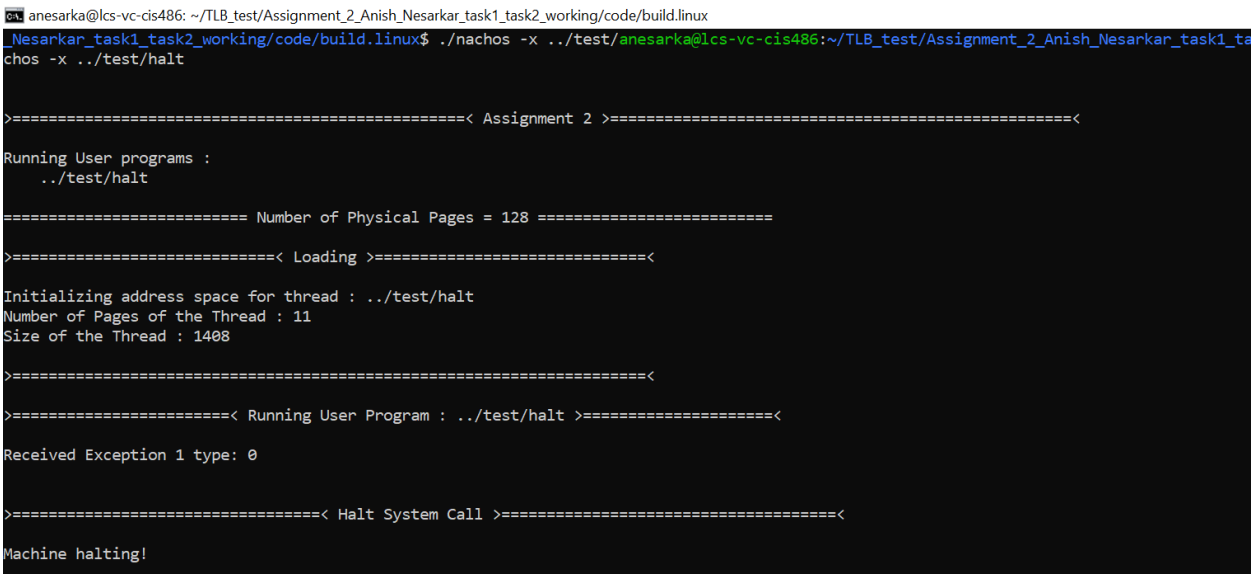
- The yield system call yields the current running user thread as shown above.

5. **Exit System Call**



- The exit system call exits the currently running user program as shown above.

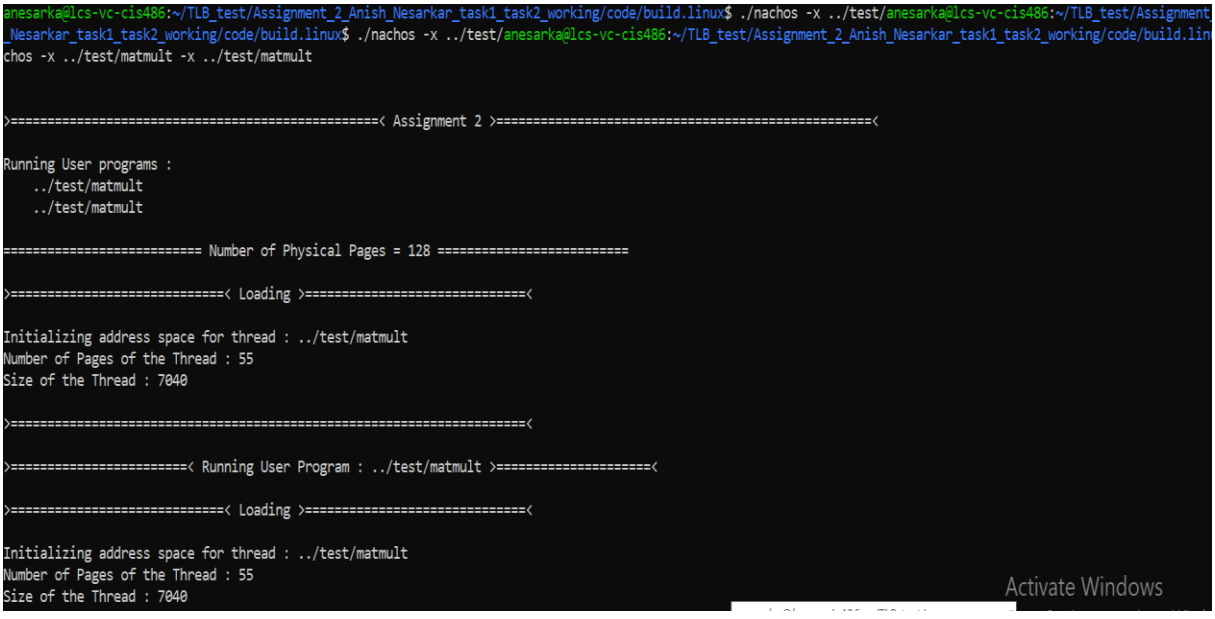
6. Halt System Call



- The halt system call halts the current process as shown above.

Virtual Memory Task

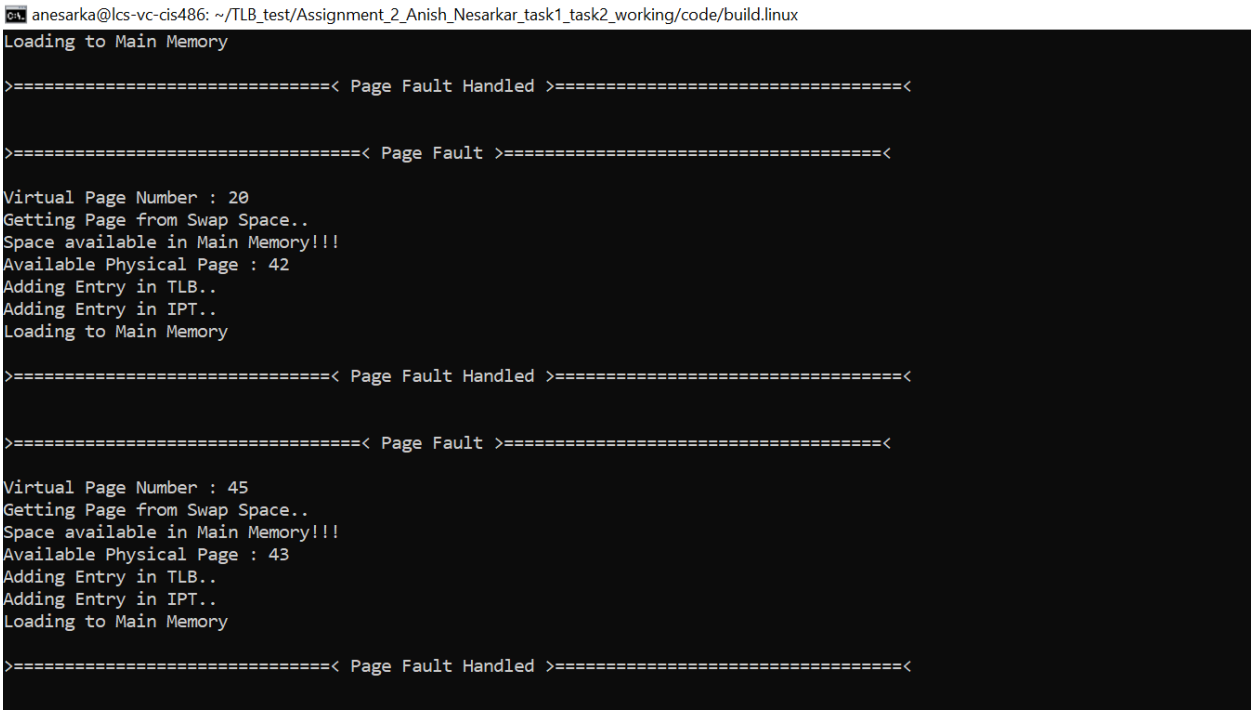
Running and Loading User program



- The above output shows multiple user programs loaded into the ready list
- It also shows the number of physical pages in the Main memory

- It also shows loading a user program in the address space and executing the program.

Handling Page Faults - When physical page is available in main memory



- The above output shows how the page faults are handled when page fault exception occurs

Handling Page Faults - When physical page is available in main memory

anesarka@lcs-vc-cis486: ~/TLB_test/Assignment_2_Anish_Nesarkar_task1_task2_working/code/build.linux

```
anesarka@lcs-vc-cis486:~/TLB_test/Assignment_2_Anish_Nesarkar_task1_task2_working/code/build.linux$ ./nachos -x ../test/matmult


>=================================================< Assignment 2 >=================================================<

Running User programs :
    ../test/matmult

========================= Number of Physical Pages = 50 =========================

>============================< Loading >============================<

Initializing address space for thread : ../test/matmult
Number of Pages of the Thread : 55
Size of the Thread : 7040

>=====================================================================<

>=======================< Running User Program : ../test/matmult >=======================<


>================================< Page Fault >================================<

Virtual Page Number : 54
Getting Page from Swap Space..
>----< Main Memory Full >---< Page Replacement using FIFO >---<
Evicted Physical Page : 0
Adding Entry in TLB..
Adding Entry in IPT..
Loading Page to Main Memory from swap space...
>------< Page Replacement Successful >-------<

>==============================< Page Fault Handled >==============================<
```

Displaying Performance Statistics

anesarka@lcs-vc-cis486: ~/TLB_test/Assignment_2_Anish_Nesarkar_task1_task2_working/code/build.linux

```
Initializing address space for thread : ../test/writetest
Number of Pages of the Thread : 12
Size of the Thread : 1536

>=====================================================================<

>=======================< Running User Program : ../test/writetest >=======================<

Received Exception 1 type: 8


>================================< Write System Call >================================<

>-----------------< Writing the following on Console: >-----------------<

>This is Write Test<
Done Writing onto console!!

>-------------------< Done Writing >----------------<

>=============================================================================================<

Received Exception 1 type: 1

>================================< Exit System Call >================================<


>================================< User Program : ../test/writetest : Exiting >============================<

>-----------------< Displaying statistics of the User program >--------------------<

Number of TLB Hits : 48
Number of Page Hits : 9
Number of Page Faults : 0

>-------------------------------------------------------------------------------<
```