INSTITUTE OF ENGINEERING AND TECHNOLOGY

Igniting Innovation

Semester 5-2015

Operating Systems

Virtual Memory Management System

Team Members:

Sr. Number	Name	Roll No.	Email Id
1.	Janki Shah	131017	jankishah1995@gmail.com
2.	Netra Pathak	131029	netra.20@gmail.com
3.	Prem Shah	131036	premparag@gmail.com
4.	Sharvil Shah	131050	shhharvil@gmail.com

TABLE OF CONTENTS

Contents

1. PROJECT DESCRIPTION

1. Project Description

Physical and virtual memory are the two forms of memory (internal storage of data). Physical memory exists on chips (RAM - Primary Memory) and on storage devices such as hard disks (Secondary Memory). For a process to be executed, it is required to first load into RAM physical memory (also termed main memory). The processor uses only that code which is lying into the main memory. But there may be case when the whole code is larger than the main memory. The other problem being that if multiple processes are lying on the main memory, all the processes cannot be stored onto main memory because of shortage of memory size.

Main memory and registers (built within the CPU itself) are the only storage the CPU can access directly. However, the processor normally needs to stall because of the frequency difference between the CPU clock and the main memory. Thus, the data required to complete the instructions is not available. One of the solutions for this problem is the use of fast memory (i.e. cash memory which contains the most frequently used functions or the most frequently run queries and transactions), to accommodate the speed differential between the CPU and the main memory. On the other side, virtual memory technique allows virtualizing the computer architecture (such as RAM, and disk storage) for processes to be executed in areas other than the main memory. These techniques facilitate the execution of programs in the computer systems with larger capacity than the physical memory.

In modern operating systems, data can be constantly exchanged between the hard disk and RAM memory via virtual memory. It is actually a specialized secondary type of data storage (hard drive space) acting as temporary storage for computer processes. It allows logical (virtual) addressing of actual physical addresses (via mapping). Hence, the use of virtual memory makes it show that a computer has a greater RAM capacity because virtual memory allows the emulation of the transfer of whole blocks of data, enabling programs to run efficiently. Instead of trying to put data into often-limited volatile RAM memory, data is actually written onto the non-volatile hard disk. It is basically an illusion provided by the operating system to all the applications so that there is no worry about actual available main memory. For example: on older 32 bit CPUs all the addresses are of 32 bits. Thus, the virtual memory can have (2^32) total addresses which is 4 GB. While 4 GB RAM at that time was rare, in this way virtual memory which is larger compared to the 2 GB RAM is used. Nowadays physical address extension is used for 32 bit architecture to have larger memory or 64 bits processors are available thus it can have up to (2^64) total addresses.

Modern operating systems provides functionality to select the size of virtual memory. Accordingly, the size of virtual memory is limited only by the size of the hard disk, or the space allocated to virtual memory on the hard disk. Virtual memory does not store the entire block of data or programming (e.g., an application process) which is being executed. It only stores a part of the code and it lies there until new function is needed. When new information (function) is needed in RAM, the virtual memory managing system tries to find it in the virtual memory and if it is not available in the virtual memory, the exchanges system rapidly swaps blocks of memory (also often termed pages of memory) between RAM and the hard disk. But, if it would have been found in the virtual memory than the management system would have swapped that part of the code with the currently lying code in the RAM. Accordingly, the use of virtual memory allows operating systems to run many programs and thus, increase the degree of multiprogramming within an operating system.

Virtual memory lies on the disk, thus operations including transfer of data to the main memory includes I/O operation, making it slower than the main memory. But, virtual memory also provides protection against memory holes. For example if a user has a runtime error like accessing a memory address which has no data or modifying memory of another code, than it can corrupt main memory, so the virtual memory gives protection against it.

Virtual Memory:

Virtual memory integration is accomplished through either demand-segmentation process or through another process termed demand-paging. Demand-paging is more common because it has a simple design. Demand-paging virtual-memory processes transfer data from disk to RAM only when the program calls for the page (fixed size block of data). Operating systems also utilize anticipatory paging processes that attempt to read ahead and execute the transfer of data before the data is actually required to be in RAM. After data is paged, data is constantly called back and forth by the paging processes, between RAM and the hard disk. Page states (valid/invalid, available/unavailable to the CPU) are registered in the virtual page table. When applications attempt to access invalid pages, a virtual-memory manager that initiates memory swapping intercepts the page fault message.

There are some reasons for having virtual memory and physical memory of the same size. In virtual memory, different processes can have their own address spaces. Hence, one process's data cannot be overwritten by another process. It also lets you give different permissions to different address spaces, so that some of the users of the system can have higher read/write privileges. But the storing benefits of Virtual Memory are eliminated by having the same amount of physical and virtual memory.

2. ARCHITECTURE/MODEL/DIAGRAMS

2. Architecture/model/diagrams

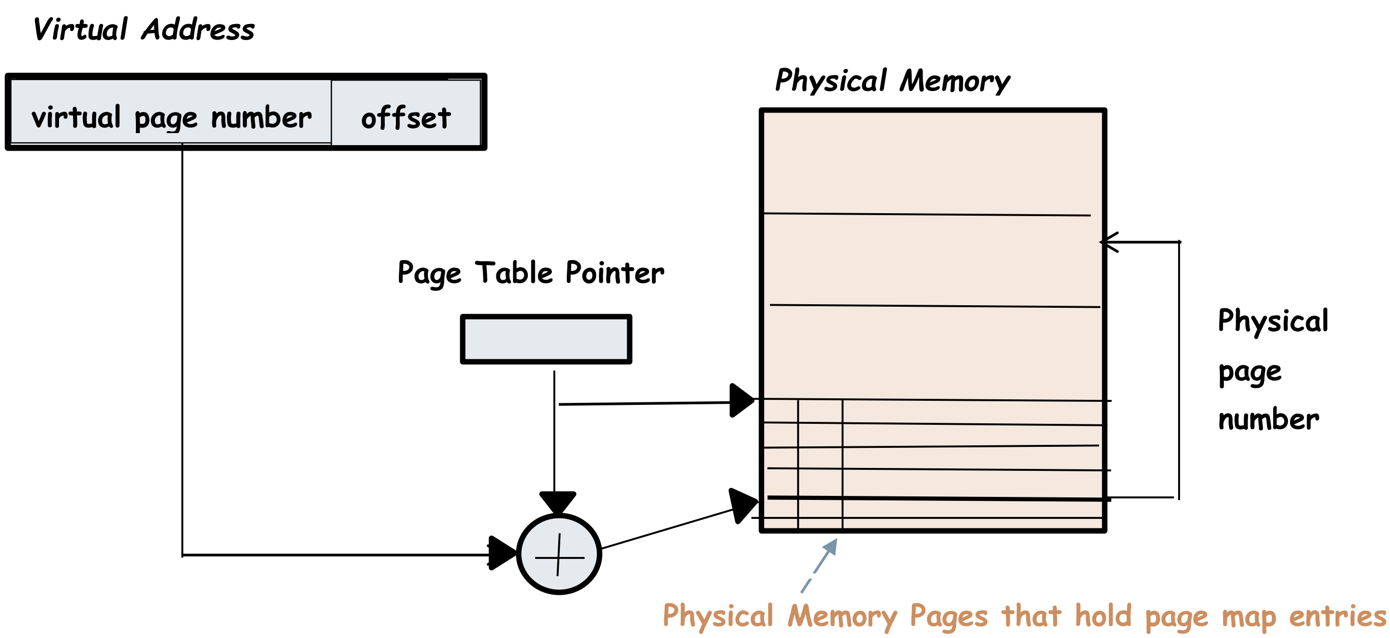


Figure 1: Virtual Memory: Address translation

Virtual Memory:

(3 bits)	(5 bits)
Page Number	Offset

Page Entry:

(2 Bits)	(2 Bits)
Flag Bits	Frame Number

Physical Memory:

(2 Bits)	(5 Bits)
Frame Number	Offset

- Number of physical pages $= 2^2 = 4$
- Number of Virtual Pages $= 2^3 = 8$
- Byte in a physical page $= 2^5 = 32$
- Page map entries $= 2^3 = 8$
- Bits in page map $= 4*2^3 = 32$

2. ARCHITECTURE/MODEL/DIAGRAMS

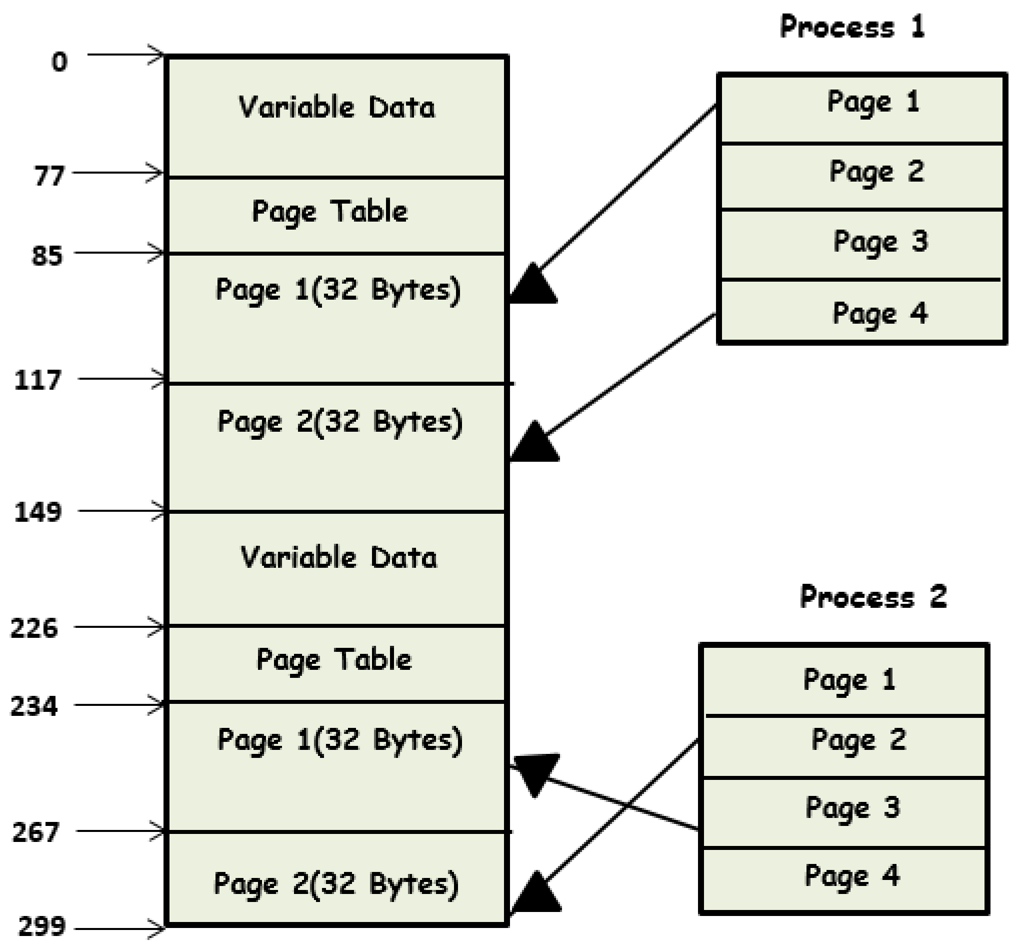


Figure 2: Virtual Memory: Allocation of processes (Memory Layout)

- ## Demand Paging

To use physical memory efficiently currently virtual pages are being used by the executing program. The technique of only loading virtual pages into memory as they are accessed is known as demand paging.

A Page Table is the data structure used by a virtual memory system in a computer operating system to store the mapping between *virtual address* and *physical addresses.*

When a currently running process accesses a memory page that is mapped into the virtual address space, but not actually loaded into main memory, hardware raises a page fault.

2. ARCHITECTURE/MODEL/DIAGRAMS

If the faulting virtual address is invalid i.e. the process has attempted to access non-existing a virtual address, then the operating system will terminate the process, protecting the other processes in the system from this rogue process.

The operating system brings the appropriate page into memory from the memory if the faulting virtual address was valid but the page that it refers to is not currently in memory. During that time other process performs the task. The targeted process is then restarted from where the memory fault was occurred. This time the virtual memory access is made and the process can proceed.

- **Swapping**

If a process is in need of bringing a page into physical memory from the virtual memory and there is no free page space available. Hence, page has to be removed. For the page that has to be removed there can be two conditions, one, the page needs to be saved first or the page does not need to be saved.

If the page does not need to be saved (i.e. the page has not been modified), it can be discarded and then it can be brought again into memory. If the page has been modified, then the page contents have to be saved in order to access it at a later time. This is known as a dirty page and when it is removed from the memory it is saved in a special file. Accessing that file takes a very long time relative to the processor speed and hence should be used the least. If the algorithm used to decide which pages to swap or discard is not efficient then it constantly accesses that file and pages are constantly written to the disk. This does not allow the operating system to do real work and it keeps on switching between reading and writing. This phenomenon is known as thrashing. If a page is being regularly accessed it is not a good candidate for swapping. The set of pages currently being used is known as the working set. An efficient swapping algorithm makes sure that all the processes have their working set in physical memory. Linux Operating system uses LRU technique to choose pages that need to be swapped. More number of times a page is accessed, younger it is. Hence it is favorable to remove old pages.

3. TECHNICAL SPECIFICATIONS/DETAILS

3. Technical specifications/details

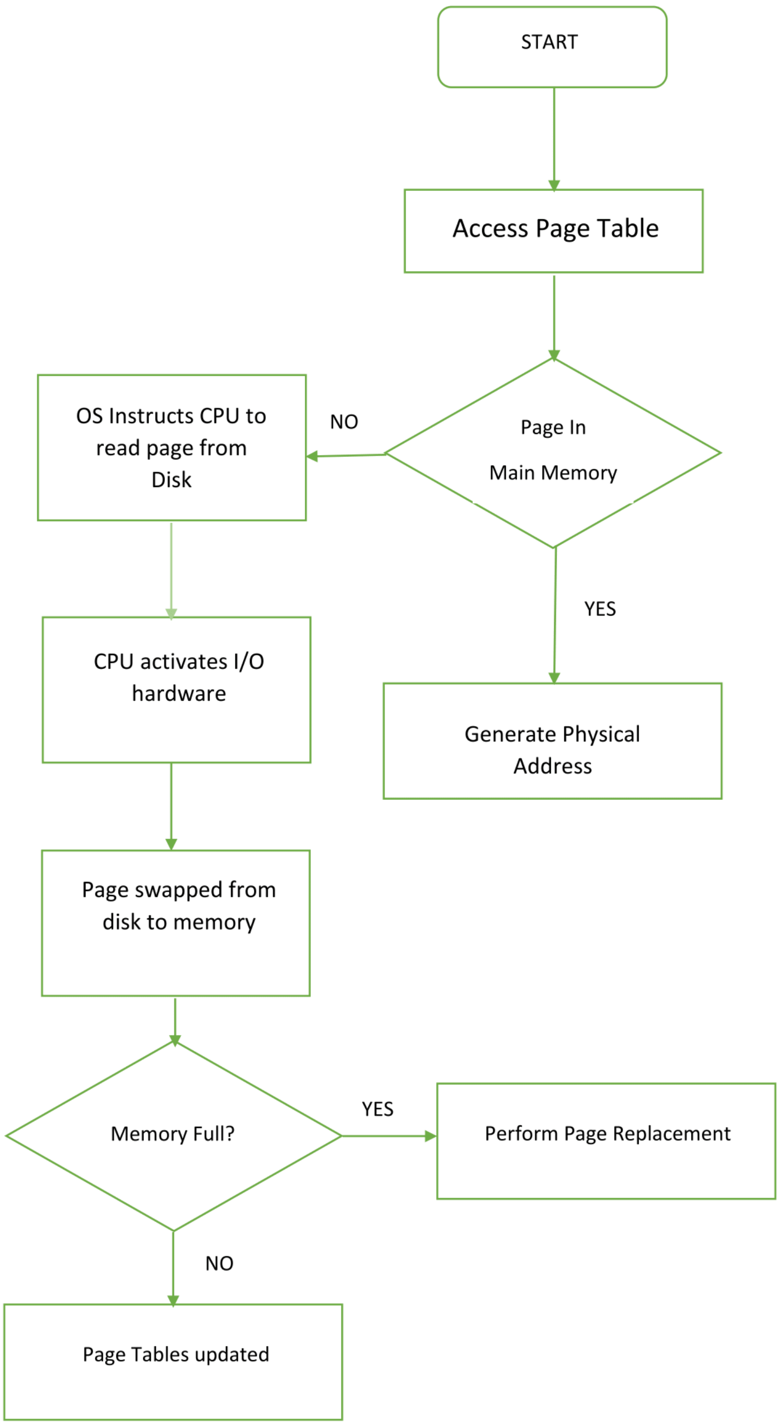
- Operating System: Windows
- Libraries: pthread
- Programming Language: C

Data structures required:

- Process page tables
  - Array of Page Table Entries
    - Each entry contains a frame number, present bit and modified bit.
- Physical memory map: It describes the allocation of physical memory among the processes, which includes the information needed to identify free frames.

4. ALGORITHMS/FLOW CHART

4. Algorithms/Flow chart



4. ALGORITHMS/FLOW CHART

Above algorithm will be applied for each individual process.
For memory management algorithm: Demand Paging
In this, for the page replacement policy, we have used FIFO (First-In-First-Out).

Expected Output:

- Process ID
- Virtual Address that process is addressing to
- Frame Number
- Physical Address
- Page number corresponding to virtual address (index into process's page table)
- Offset into page for virtual address

We have taken two processes for the data set. Both of these processes pass together in the parser. Also, there is a non-preemptive algorithm, hence once a process starts, it will be allowed to complete. On occurrence of a page fault, swapping between physical and virtual memory occurs.

List of programs:

- MMU.h (header file for Memory Management Unit)
- Queue.h (FIFO Implementation)
- main.c (Main Program)

Test data sets:

- One.txt (Input Data Set)
- Two.txt (Input Data Set)

5. Implementation

MMU.h (header file for Memory Management Unit)

```
#ifndef MMU_H_INCLUDED
#define MMU_H_INCLUDED
#include "Queue.h"
/*This will translate Virtual address to physical address and will
retun if the page is in the memory*/

int Translate(char VA,int PTP,char *frame,char *offset,char
*P_no,char *MEM)
{
      *P_no=(VA&0xE0)>>5;
      char P_entry=MEM[*P_no+PTP];
      *frame=P_entry&0x03;
      *offset=(VA&0x1F);
      return ((P_entry>>2)&0x01);
}

void Page_Fault(int pid,char *page_file,struct queue *q,char *MEM,int
PTP,char page_no,char frame_cnt)
{
      if(q->elements<2)
      {
            enque(q,page_no);
            FILE *fp=fopen(page_file,"r");
            if(frame_cnt==0)
            {
                  int i=0;
                  int frame_start=(pid-1)*149+86;
                  int data=0;
                  while((data=fgetc(fp))!=EOF)
                  {
                        MEM[frame_start++]=(char)data;
                  }
                  char P_entry=MEM[page_no+PTP];
                  if(pid==1)
                        P_entry=P_entry|0x00;
                  else
                        P_entry=P_entry|0x02;
                  P_entry=P_entry|0x04;
                  MEM[page_no+PTP]=P_entry;
                  frame_cnt++;
```

```
		}
		else
		{
			int i=0;
			int frame_start=(pid-1)*149+118;
			int data=0;
			while((data=fgetc(fp))!=EOF)
			{
				MEM[frame_start++]=(char)data;
			}
			char P_entry=MEM[page_no+PTP];
			if(pid==1)
				P_entry=P_entry|0x01;
			else
				P_entry=P_entry|0x03;
			P_entry=P_entry|0x04;
			MEM[page_no+PTP]=P_entry;
			//frame_cnt++;

		}


	}
	else
	{
		int p=deque(q);
		enque(q,page_no);
		FILE *fp=fopen(page_file,"r");
		char P_entry1=MEM[p+PTP];
		P_entry1=P_entry1&0x03;
		char frame_n=P_entry1&0x03;
		MEM[p+PTP]=P_entry1;
		char P_entry2=MEM[page_no+PTP];
		P_entry2=P_entry2|0x04;
		P_entry2=P_entry2|frame_n;
		MEM[p+PTP]=P_entry1;
		int frame_start;
			if(frame_n==0)
			{
				frame_start=86;
			}
			else if(frame_n==1)
			{
				frame_start=118;
			}
			else if(frame_n==3)
```

```
                {
                      frame_start=235;
                }
                else if(frame_n==3)
                {
                      frame_start=268;
                }
                int data=0;
                while((data=fgetc(fp))!=EOF)
                {
                      MEM[frame_start++]=(char)data;
                }
      }
}
void Load(char *str,int Variable_counter,int Var_ptr,char
*MEMORY,char *P_n[],char *Physical_add,int PTP,int pid,struct queue
*q,char *length)
{
                char var_nm=str[0];
                int index=0;
                int variable_found=0;
                for(index=0;index<Variable_counter;index++)
                {
                      if(var_nm==MEMORY[index*3+Var_ptr])
                      {
                            variable_found=1;
                            break;
                      }
                }
                if(variable_found==0)
                {
                      printf("Could not find variable\n");
                      remove(P_n[0]);
                      remove(P_n[1]);
                      remove(P_n[2]);
                      remove(P_n[3]);
                      pthread_exit(0);
                }
                char var_vadd=MEMORY[(index*3)+2+Var_ptr];
                //printf("%d 0x%02x",index,var_vadd);
                char frame=0x00;
                char offset=0x00;
                char P_no=0x00;

      if(Translate(var_vadd,PTP,&frame,&offset,&P_no,MEMORY)==0)
                {
```

```
                        printf("Page Fault\n");
                        Page_Fault(pid,P_n[P_no],q,MEMORY,PTP,P_no,q-
>elements);

     Translate(var_vadd,PTP,&frame,&offset,&P_no,MEMORY);

                }
                printf("Pid=%d Page No:0x%02x\n",pid,P_no);
                printf("Pid=%d FrameNo:0x%02x\n",pid,frame);
                printf("Pid=%d Offset:0x%02x\n",pid,offset);
                *Physical_add=((frame<<5)|(offset))+(char)PTP+8;
                printf("Pid=%d Offset:0x%02x\n",pid,offset);
                //printf("Pid=%d Physical
Address:0x%02x\n",pid,*Physical_add);
                *length=MEMORY[(index*3)+1+Var_ptr];
                printf("Pid=%d Length:0x%02x\n",pid,*length);
}
#endif
```

Queue.h (FIFO Implementation)

```
#ifndef QUEUE_H_INCLUDED
#define QUEUE_H_INCLUDED
struct node
{
    int page_no;
    struct node *next;
};


struct queue
{
    struct node *first_list;
    int elements;
};
void enque(struct queue *lst,int element)
{
     lst->elements++;
    struct node *temp,*temp1=NULL;


        if (lst->first_list == NULL)
```

```
        {
            temp=(struct node *)malloc(sizeof(struct node));
            temp->page_no=element;
                temp->next=NULL;
            lst->first_list=(struct node *)malloc(sizeof(struct
node));
            lst->first_list=temp;
        }
        else
        {
            temp=(struct node *)malloc(sizeof(struct node));
            temp1=(struct node *)malloc(sizeof(struct node));
            temp1=lst->first_list;
            if (!(temp == NULL))
            {
                temp->page_no=element;
                temp->next=NULL;
                while(temp1->next!=NULL)
                    temp1=temp1->next;
                temp1->next=temp;
            }
            else
            {
                printf("\nQueue is Full...");
            }
        }

}
int deque(struct queue *q)
{
     q->elements--;
    if(q->first_list==NULL)
        printf("Queue is UnderFlow");
    else if(q->first_list->next==NULL)
    {
     int ret=q->first_list->page_no;
        q->first_list=NULL;
        return ret;
    }
    else
    {
        struct node *temp=NULL;
        temp=(struct node *)malloc(sizeof(struct node));
        temp=q->first_list;
     int ret=q->first_list->page_no;
           q->first_list=temp->next;
```

```
        free(temp);
        temp=NULL;
        return ret;
    }

}
#endif // QUEUE1_H_INCLUDED
```

main.c (Main Program)

```
#include<stdio.h>
#include<pthread.h>
#include<math.h>
#include<malloc.h>
#include<semaphore.h>
#include<stdlib.h>
#include<string.h>
#include<stdint.h>

#include "MMU.h"
#include "QUEUE.h"
char *MEMORY;//Main Memory of 300 Bytes 0-299.
float BtoF(const void *buf)
{
    const unsigned char *b = (const unsigned char *)buf;
    uint32_t temp = 0;
    temp = ((b[0] << 24) |
            (b[1] << 16) |
            (b[2] <<  8) |
             b[3]);
    return temp;
}
int FtoB(void *buf, float x)
{
    unsigned char *b = (unsigned char *)buf;
    unsigned char *p = (unsigned char *) &x;
      #if defined (_M_IX86) || (defined (CPU_FAMILY) && (CPU_FAMILY
== I80X86))
    b[0] = p[3];
    b[1] = p[2];
    b[2] = p[1];
    b[3] = p[0];
```

```
      #else
    b[0] = p[0];
    b[1] = p[1];
    b[2] = p[2];
    b[3] = p[3];
      #endif
    return 4;
}
double BtoD(const void *buf)
{
    const unsigned char *b = (const unsigned char *)buf;
    uint32_t temp = 0;
    temp =  (((long long unsigned)b[0] << 56) |
                ((long long unsigned)b[1] << 48) |
            ((long long unsigned)b[2] << 40) |
            ((long long unsigned)b[3] << 32) |
                (b[4] << 24) |
            (b[5] << 16) |
            (b[6] <<  8) |
             b[7]);
    return temp;
}
int DtoB(void *buf, double x)
{
    unsigned char *b = (unsigned char *)buf;
    unsigned char *p = (unsigned char *) &x;
      #if defined (_M_IX86) || (defined (CPU_FAMILY) && (CPU_FAMILY
== I80X86))

      b[0] = p[7];
    b[1] = p[6];
    b[2] = p[5];
    b[3] = p[4];
    b[4] = p[3];
    b[5] = p[2];
    b[6] = p[1];
    b[7] = p[0];
      #else
    b[0] = p[0];
    b[1] = p[1];
    b[2] = p[2];
    b[3] = p[3];
    b[4] = p[4];
    b[5] = p[5];
    b[6] = p[6];
    b[7] = p[7];
```

5. IMPLEMENTATION

```
      #endif
     return 4;
}
void *parse(void *arg)
{
      int ii=*((int*) arg);
      char *name;
      if(ii==1)
      {
            name="One.txt";
      }
      else
      {
            name="Two.txt";
      }

      /*----------------Parser Variables------------------------------
*/
      FILE *fp;
      fp=fopen(name,"r");
      int ch;
      int i=1;
      int flag_space=0;//Space detected
      int flag_eq=0;//Assignment using equals to sign
      int semcol=0;//Assignment using equals to sign
      int line_no=0;//line no
      int var_name_pos=0;//Variable name index in the instruction
      int space_pos=0;
      int pid=ii;//Which process numberis this
      /*Beginning of Variables' data in Memory 1 Byte for Name 1 Byte
for Length and 1 Byte for Virtual Address
      So total 3 Bytes for each entry for a variable.As there is only
one character varible name allowed there will be 26 Variables
      with maximum size=8 Bytes(for Double) So jump of 3 to search
for a name of var and
      Max entries needed=26 and total bytes for that=26*3=78 Bytes
per process.
      After that comes the Page Table .
      There are 3 bits for a page# so total entries per process will
be 8 and physical address has 2 bits for frame number and
      5 bits for offsets.
      So each table entries will have 2 bits for flags 2 bits for
frame number.
      Each page has 2^5 Bytes memory so each page is of size 32 Bytes
and each process will have only 2 frame pages.
      */
```

5. IMPLEMENTATION

```
int Var_ptr;
int PTP;//Location of Page Table Pointer in memory
int JUMP=3;//3 Bytes of jump
int Variable_counter=0;//How many variables have been stored.
char VA_counter=0x00;//Which virtual address is this
char *P_n[4];
if(pid==1)
{
      Var_ptr=0;
      PTP=78;
      P_n[0]="One_P1";
      P_n[1]="One_P2";
      P_n[2]="One_P3";
      P_n[3]="One_P4";
}
else if(pid==2)
{
      Var_ptr=150;
      PTP=228;
      P_n[0]="Two_P1";
      P_n[1]="Two_P2";
      P_n[2]="Two_P3";
      P_n[3]="Two_P4";
}
struct queue *q=calloc(1,sizeof(struct queue));
q->elements=0;
while((ch=fgetc(fp))!= EOF)
{
      char *str=calloc(1,sizeof(char));
      str[0]=ch;
      ch=fgetc(fp);
      while((ch!='\n')&&(ch!=EOF))
      {
            str=(char *)realloc(str,(++i)+1);
            str[i-1]=(char)ch;
            str[i]='\0';
            if(ch==' ')
            {
                  flag_space=1;
                  var_name_pos=i;
                  space_pos=i-1;
            }

            else if(ch=='=')
            {
                  flag_eq=1;
```

```
                }
                else if(ch==';')
                {
                        semcol=1;
                }
                ch=fgetc(fp);
        }
        line_no++;
        printf("%s\n",str);
        if(semcol==0)
        {
                printf("No semicolon found\n");
                break;
        }
        if(flag_space==1)//Definition of var
        {
                char var_name=str[var_name_pos];
                int index=0;
                int variable_found=0;
                if(Variable_counter==26)
                {
                        printf("Cannot define more than 26
variables\n");
                        pthread_exit(0);
                }
                for(index=0;index<Variable_counter;index++)
                {
                        if(var_name==MEMORY[index*3+Var_ptr])
                        {
                                printf("Variable already defined\n");
                                pthread_exit(0);
                        }
                }
                /*Variable is not defined yet so insert an entry for
this variable into MEMORY*/
                MEMORY[Variable_counter*3+Var_ptr]=var_name;
                /*Check which data Type it is and assign memory
accordingly*/
                char *data_type=calloc(1,sizeof(char));
                for(index=0;index<space_pos;index++)
                {
                        data_type=(char *)realloc(data_type,index+2);
                        data_type[index]=str[index];
                        data_type[index+1]='\0';
                }
```

```
                char incr=0x00;
                if(strcmp(data_type,"int")==0)//Variable is defined
as integer
                {
                     //Allocate 2 blocks of memory for an
integer(32bits)
                     MEMORY[((Variable_counter)*3)+1+Var_ptr]=0x02;

     MEMORY[((Variable_counter)*3)+2+Var_ptr]=VA_counter;
                     incr=0x02;
                     //VA_counter=VA_counter+0x04;
                     VA_counter=VA_counter&0x000000ff;
                     printf("Pid=%d Datatype=integer
0x%02x\n",pid,VA_counter);
                }
                else if(strcmp(data_type,"float")==0)//Variable is
defined as integer
                {
                     //Allocate 4 blocks of memory for a
float(32bits)
                     MEMORY[((Variable_counter)*3)+1+Var_ptr]=0x04;

     MEMORY[((Variable_counter)*3)+2+Var_ptr]=VA_counter;
                     incr=0x04;
                     //VA_counter=VA_counter+0x04;
                     VA_counter=VA_counter&0x000000ff;
                     printf("Pid=%d Datatype=float
0x%02x\n",pid,VA_counter);
                }
                else if(strcmp(data_type,"double")==0)//Variable is
defined as integer
                {
                     //Allocate 8 blocks of memory for a
double(64bits)
                     MEMORY[((Variable_counter)*3)+1+Var_ptr]=0x08;

     MEMORY[((Variable_counter)*3)+2+Var_ptr]=VA_counter;
                     incr=0x08;
                     //VA_counter=VA_counter+0x08;
                     VA_counter=VA_counter&0x000000ff;
                     printf("Pid=%d Datatype=double
0x%02x\n",pid,VA_counter);
                }
                else
                {
                     printf("This data type is not valid\n");
```

```
                    }
                    char page_no=(VA_counter&0xE0)>>5;
                    FILE *page_file=fopen(P_n[page_no],"ab+");
                    int sk=0;
                    for(sk=0;sk<incr;sk++)
                    {
                          fputc(0x00,page_file);
                    }
                    fclose(page_file);
                    /*DataTyoe Checked*/
                    Variable_counter++;
                    free(data_type);
                    VA_counter=VA_counter+incr;
                    /*Entry Inserted*/

              }
              else if(flag_eq==1)//assignement
              {
                    //printf("Eq\n");
                    char *Phy_add1=calloc(1,1);
                    char *Phy_add2=calloc(1,1);
                    char *length1=calloc(1,1);

      Load(str,Variable_counter,Var_ptr,MEMORY,P_n,Phy_add1,PTP,pid,q
,length1);
                    int l=0;
                    if(*length1==0x02)
                    {
                          short int load=0;
                          for(l=0;l<*length1;l++)
                          {
                                load=load|(MEMORY[*Phy_add1+l]<<8*l);
                          }
                          //printf("%s %d\n",str,strlen(str));
                          int str_i=2;
                          if((str[2]>96)&(str[2]<123))
                          {
                                char *inp_str=calloc(2,1);
                                inp_str[0]=str[2];
                                inp_str[1]='\0';
                                char *length2=calloc(1,1);

      Load(inp_str,Variable_counter,Var_ptr,MEMORY,P_n,Phy_add2,PTP,p
id,q,length2);
                                if(*length2==0x02)
                                {
```

```
                                        short int load2=0;
                                        for(l=0;l<*length2;l++)
                                        {
        load2=load2|(MEMORY[*Phy_add2+l]<<8*l);
                                        }
                                        int str_ind=0;
                                        char number[strlen(str)-4];
                                        for(str_ind=4;str_ind<(strlen(str)-
1);str_ind++)
                                        {
                                                number[str_ind-
4]=str[str_ind];
                                        }
                                        int num=atoi(number);
                                        //printf("num=%d\n",num);
                                        if(str[3]=='+')
                                        {
                                                load=load2+num;
                                        }
                                        else if(str[3]=='-')
                                        {
                                                load=load2-num;
                                        }
                                        else if(str[3]=='*')
                                        {
                                                load=load2*num;
                                        }
                                        else
                                        {
                                                load=load2/num;
                                        }
                                        //printf("num=%d\n",load);
                                }
                                else if(*length2==0x04)
                                {
                                        float load2=0;
                                        char *buf=calloc(4,1);
                                        for(l=0;l<*length2;l++)
                                        {
                                                buf[l]=MEMORY[*Phy_add2+l];
                                        }
                                        load2=BtoF((void*)buf);
                                        int str_ind=0;
                                        char number[strlen(str)-4];
```

```
                    for(str_ind=4;str_ind<(strlen(str)-
1);str_ind++)
                    {
                          number[str_ind-
4]=str[str_ind];
                    }
                    float num=atof(number);
                    //printf("num=%d\n",num);
                    if(str[3]=='+')
                    {
                          load=load2+num;
                    }
                    else if(str[3]=='-')
                    {
                          load=load2-num;
                    }
                    else if(str[3]=='*')
                    {
                          load=load2*num;
                    }
                    else
                    {
                          load=load2/num;
                    }
                    //for
              }
              else if(*length2==0x08)
              {
                    double load2=0;
                    char *buf=calloc(8,1);
                    for(l=0;l<*length2;l++)
                    {
                          buf[l]=MEMORY[*Phy_add2+l];
                    }
                    load2=BtoD((void*)buf);
                    int str_ind=0;
                    char number[strlen(str)-4];
                    for(str_ind=4;str_ind<(strlen(str)-
1);str_ind++)
                    {
                          number[str_ind-
4]=str[str_ind];
                    }
                    double num=atof(number);
                    //printf("num=%d\n",num);
                    if(str[3]=='+')
```

```
                    {
                         load=load2+num;
                    }
                    else if(str[3]=='-')
                    {
                         load=load2-num;
                    }
                    else if(str[3]=='*')
                    {
                         load=load2*num;
                    }
                    else
                    {
                         load=load2/num;
                    }
               }
               //printf("\na\n");
          /*char *RHS=calloc(1,sizeof(char));
          RHS[0]='\0';
          int var_pres=0;
          while(str[str_i]!='\0')
          {
               RHS=(char *)realloc(RHS,(str_i));
               RHS[str_i-1]='\0';
               RHS[str_i-2]=str[str_i];
               if(str[str_i]>'')
               {

               }
               str_i++;
          }*/
          }
          else
          {

          }

     }
}
else
{
     printf("Error at Line %d :Statement is not valid and
File:%s\n",line_no,name);
     break;
}
//printf("\naa\n");
```

```
            if(ch==EOF)
            {
                  break;
            }
            flag_space=0;
            flag_eq=0;
            semcol=0;
            free(str);
            i=1;
      }
      fclose(fp);
      remove(P_n[0]);
      remove(P_n[1]);
      remove(P_n[2]);
      remove(P_n[3]);
      /*----------------------------------------------------------*/
      pthread_exit(0);

}
int main()
{
      int i=1;
      pthread_t thread[2];
      MEMORY=calloc(300,1);
      //char name[7] ="One.txt";
      for(i=1;i<3;i++)
      {

            int *arg = malloc(sizeof(*arg));
            *arg = i;
            if(pthread_create(&(thread[i-1]), NULL,parse,arg)!=0)
            {
                  printf("Error creating thread\n");
            }
      }

      for(i=0;i<2;i++)
      {
            if(pthread_join(thread[i], NULL)!=0)
            {
                  printf("Error destroying thread\n");
            }
      }
}
```

5. IMPLEMENTATION

One.txt (Input Data Set)

```
int c;
double a;
double b;
double d;
double e;
double f;
double g;
c=c+1;
```

Two.txt (Input Data Set)

```
double a;
double b;
double c;
double d;
```

6. TEST RESULTS

6. Test Results

```
int c;
double a;
double b;
double d;
double e;
double f;
double g;
c=c+1;
```

```
Pid=2 Datatype=double 0x10
double d;
Pid=1 Datatype=double 0x12
double d;
Pid=2 Datatype=double 0x18
double e;
Pid=1 Datatype=double 0x1a
double f;
Pid=1 Datatype=double 0x22
double g;
Pid=1 Datatype=double 0x2a
c=c+1;
Page Fault
Pid=1 Page No:0x00
Pid=1 FrameNo:0x00
Pid=1 Offset:0x00
Pid=1 Offset:0x00
Pid=1 Length:0x02
Pid=1 Page No:0x00
Pid=1 FrameNo:0x00
Pid=1 Offset:0x00
Pid=1 Offset:0x00
Pid=1 Length:0x02
```

6. TEST RESULTS

```
double a;
double b;
double d;
double e;
double f;
double g;
double h;
double i;
double j;
double k;
double l;
double m;
double n;
double o;
double p;
double q;
```

```
Pid=2 Datatype=double 0x00
Pid=1 Datatype=double 0x00
double b;
double b;
Pid=1 Datatype=double 0x08
Pid=2 Datatype=double 0x08
double d;
Pid=1 Datatype=double 0x10
double c;
Pid=2 Datatype=double 0x10
double e;
Pid=1 Datatype=double 0x18
double d;
Pid=2 Datatype=double 0x18
double f;
Pid=1 Datatype=double 0x20
double g;
Pid=1 Datatype=double 0x28
double h;
Pid=1 Datatype=double 0x30
double i;
Pid=1 Datatype=double 0x38
double j;
Pid=1 Datatype=double 0x40
double k;
Pid=1 Datatype=double 0x48
double l;
Pid=1 Datatype=double 0x50
double m;
Pid=1 Datatype=double 0x58
double n;
Pid=1 Datatype=double 0x60
double o;
Pid=1 Datatype=double 0x68
double p;
Pid=1 Datatype=double 0x70
double q;
Pid=1 Datatype=double 0x78
```

6. TEST RESULTS

```
double a
double b;
double d;
double e;
double f;
double g;
double h;
double i;
double j;
double k;
double l;
double m;
double n;
double o;
double p;
double q;
double r;
double s;
double t;
double u;
double v;
double w;
double x;
double y;
```

```
double a
No semicolon found
double a;
Pid=2 Datatype=double 0x00
double b;
Pid=2 Datatype=double 0x08
double c;
Pid=2 Datatype=double 0x10
double d;
Pid=2 Datatype=double 0x18
```

7. REFERENCES

7. References

- Physical Computer Memory and Virtual Memory - Data, Ram, Hard, and Page - JRank Articles http://science.jrank.org/pages/1698/Computer-Memory-Physical-Virtual-Memory.html#ixzz3mIIJTD3B
- http://www.tutorialspoint.com/operating_system/os_virtual_memory.htm
- https://msdn.microsoft.com/en-us/library/windows/desktop/aa366916(v=vs.85).aspx
- http://www.ehow.com/facts_6857434_difference-virtual-memory-main-memory.html
- http://web.stanford.edu/class/ee282h/handouts/Handout31.pdf
- http://www.cs.princeton.edu/courses/archive/spr02/cs217/lectures/paging.pdf
- http://cis.poly.edu/cs220/cs3224OS/projects/mem%20mang/2.Memory%20Management/MemoryManagement.html#ProcessScheduler
- http://www.tldp.org/LDP/tlk/mm/memory.html
- http://homepage.cs.uiowa.edu/~ghosh/4-29-10.pdf
- https://bryansoliman.wordpress.com/2011/07/08/virtual-memory-vs-physical-memory/