# JUDICIAL CASE MANAGER USING VIRTUAL MEMORY


GROUP MEMBERS:

ANIRUDH KUMAR (19BCI0246)

AGASTYA ZARABI(19BCE2421)

ABHAY KUMAR JAIN(19BCE2431)

SIDDHANT SINGH PATEL(19BCE2567)

# Introduction

- The basic concept behind this project is TLB and Page Tables.
- The basic idea is segregating judicial cases on the basis of their importance (priority) and giving a storage location for the cases which are not yet active in the court.
- This project consists of writing a program that translates logical to physical addresses for a virtual address space of size 2^16 = 65,536 bytes.
- A backing store which is a file that is big enough to store all values is used to check for TLB faults
- For each value that is encountered in the file, based on the logical address, a priority [critical , important , normal or low] is assigned to the value and is displayed.
- Also, the status, and the physical address of the file is reported.

# Abstract

- Segregating Judicial Cases on the basis of their importance and gives a storage location for those cases which are not yet active in the court.
- Writing a program that translates file number(logical) to storage location(physical addresses) for a virtual address space of size 2^16 = 65,536 bytes.
- The files will be read from file addresses.txt, which contains 1000 logical addresses ranging from 0 to 65535. Apart from translating each logical address to a physical address, it determine the contents of the signed byte stored at the correct physical address.
- The program is to output the following values:
    1) The logical address being translated (the integer value being read from addresses.txt).
    2) The corresponding physical address (what your program translates the logical address to).
    3) The signed byte value stored at the translated physical address.

# Motivation

- Every one is very well aware of the kind of mess that India's judicial system has become. Cases are pending from tens of years, and many files/records are missing as well and are not anymore.
- Management of such files seems to be very unorganized.
- Ranging from police stations to courts at all levels, the types of primitive methods used to store files is very traditional and therefore, outdated.
- The program aims to curb this technological gap by introducing the concept of virtual memory to generate storage locations for files, based on their activity status in the courts.

# Literature Survey

- Virtual memory was developed to automate the movement of program code and data between main memory and secondary storage to give the appearance of a single large store.
- This technique greatly simplified the programmer's job, particularly when program code and data exceeded the main memory's size.
- Virtual memory has now become widely used, and most modern processors have hardware to support it.
- Unfortunately, there has not been much agreement on the form that this support should take.
- The result of this lack of agreement is that hardware mechanisms are often completely incompatible.
- Thus, designers and porters of system-level software have two somewhat unattractive choices: they can write software to fit many different architectures or they can insert layers of software to emulate a particular hardware interface.

# Steps that are followed:

- First, a file containing all logical addresses is read.
- If the file contains any value that is more than 2^16 -1(65535), an error is thrown.
- Then, while the file exists, the following things occur:
- First, the page number is extracted from the logical address, and the TLB is consulted.
- In the case of a TLB-hit, the frame number is obtained from the TLB.
- In the case of a TLB-miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table or a page fault occurs.
- Then, the page table checks for the values.
- If the value is not found, a Page Miss is returned.
- If found, a page hit is reported.

# A basic idea of the code

- Before performing the first step : reading a file, a few things need to be done:
- A structure to store the values present in the frame [the content of the frame and the dirty bit] needs to be created.
- Another structure to store the frame number along with the dirty bit and the page number.
- Another structure to basically hold an array of tlb records and a tail.

# A basic idea of the code [Contd.]

- A few functions to read the data, update the TLB and the Page Table are required.
- Functions to print and search the TLB and Page Table are also required.

# A basic idea of the code [Contd.]

- Once all the functions and structures are created, the file is taken as input.
- The file is iterated through and all the values are checked for a TLB hit or miss.
- If the value is missed by the TLB, the page table is checked for the values.
- Then, a page hit or miss is reported.

# Modules of the code

- Creation of the structures that have been specified.
- Creation of the functions that have been specified.
- Declaration of the functions that have been specified.
- Creation of the page tables, TLB.
- Reading from the file.
- Invoking TLB search function to check for a TLB hit or a TLB miss and updating the TLB.
- Invoking page search function to check for a page hit or a page miss and updating the page table.

# Work division

- Creating the structures and the file reading function which checks for digits and providing the data for the file: Siddhant
- Creating the frame ,TLB printing function along with some part of the GUI and providing the data for the file: Abhay
- Creating the address translation function [logical to physical address] and assigning a priority to the address and calling the other functions: Anirudh
- Create the page table update function and page table printing function and reading from the files along with the GUI: Agastya

# The Output

- The final output of the project is:
  - The logical address that is being read, the status and file number of the case [in case of a TLB miss]
  - The net TLB hit rate
  - The net Page miss rate
  - If the user chooses to search for a particular value, returns whether the value exists or not in the given file.

# Code for the structures

```
#define FRAME_SIZE 1024
#define PAGE_SIZE 1024
#define PHY_SIZE 1024
#define PAGE_TABLE_SIZE 1024
#define TLB_SIZE 50
//record of page table
typedef struct page_record
{
  int dirty;
int page;
  int frame;
} Record;

// record of page table
typedef struct v_frame
{
  int dirty;
  char content[FRAME_SIZE];
} Frame;
```

# Code for the structures

```
typedef struct v_tlb_record
{
  int dirty;
  int page;
  int frame;
} TLB_Record;

// circular array structure
typedef struct v_tlb
{
  TLB_Record records[TLB_SIZE];
  int tail;
} TLB;
```

# Code for functions

```
inline char *get_line(FILE* fp)
{
int len = 0, got = 0, c;
char *buf = 0;
while ((c = fgetc(fp)) != EOF) {
if (got + 1 >= len) {
len *= 2;
if (len < 4) len = 4;
buf = (char*)realloc(buf, len);
}
if (c == '\n') break;
   if (c - '0' < 0 || c - '0' > 9) {
     printf("Only numbers are allowed!\n");
     return 0;
   }
buf[got++] = c;
}
if (c == EOF && !got) return 0;
                        buf[got++] = '\0';
                        return buf;}
```

# Code for functions

```
inline int search_in_page_table(Record *page_table, int size, int page) {
  int i;
  for (i=0; i<size; i++) {
    if (page_table[i].dirty == 1 && page_table[i].page == page) {
      return page_table[i].frame;
    }
  }
  return -1;
}
```

# Code for functions

```
inline int get_available_frame(Frame *physical_memory, int size)
{
  int i;
  for (i=0; i<size; i++) {
    if (physical_memory[i].dirty == 0) {
      return i;
    }
  }
  return -1;
}
inline int copy_to_physical_memory(Frame *physical_memory, int size, char *buffer) {
  int i = get_available_frame(physical_memory, size);
  if (i >= 0) {
    memcpy(physical_memory[i].content, buffer, FRAME_SIZE);
    physical_memory[i].dirty = 1;
    return i;
  }
  return -1;
 }
```

# Code for functions

```
inline int get_available_record(Record *page_table, int size) {
  int i;
  for (i=0; i<size; i++) {
    if (page_table[i].dirty == 0) {
      return i;
    }
  }
  return -1;
}

inline void update_page_table(Record *page_table, int size, int page_number, int frame) {
  int a = get_available_record(page_table, size);
  if (a >= 0) {
    printf("\tUpdate in page table #%d, page %d, frame %d\n", a, page_number, frame);
    page_table[a].page = page_number;
    page_table[a].frame = frame;
    page_table[a].dirty = 1;
  } else {
    printf("No available record in page table found!\n");
  }
}
```

# Code for functions

```
inline void print_page_table(Record *page_table, int size) {
  int i;
  for (i=0; i<size; i++) {
    if (page_table[i].dirty == 0) {
      return;
    }
    printf("Page table #%-3d: page %-15d frame %-15d dirty %d\n", i, page_table[i].page, page_table[i].frame, page_table[i].dirty);
  }
}
inline void print_frame(Frame frame) {
  int i;
  for (i=0; i<512; i++) {
    printf("%c", frame.content[i]);
  }
}
inline void print_tlb(TLB tlb) {
  int i;
  printf("TLB tail: %d\n", tlb.tail);
  for (i=0; i<TLB_SIZE; i++) {
    printf("Page: %-10dFrame: %-10dDirty: %d\n", tlb.records[i].page, tlb.records[i].frame, tlb.records[i].dirty);
  }
}
```

# Code for functions

```
inline void update_tlb(TLB *tlb, int page, int frame) {
  tlb->records[tlb->tail].page = page;
  tlb->records[tlb->tail].frame = frame;
  tlb->records[tlb->tail].dirty = 1;
  //printf("\tUpdate in TLB #%d: page %d, frame %d\n", tlb->tail, tlb->records[tlb->tail].page, tlb->records[tlb->tail].frame);
  tlb->tail = (tlb->tail + 1) % TLB_SIZE;
}

inline int search_in_tlb(TLB tlb, int page) {
  int i;
  for (i=0; i<TLB_SIZE; i++) {
    if (tlb.records[i].dirty == 1) {
      if (tlb.records[i].page == page) {
        return tlb.records[i].frame;
      }
    }
  }
  return -1;
}
```

# Code for functions

```
inline int translate_to_physical_address(int frame, int offset) {
  int address = frame;
  // printf("%d\n", address);
  address = address << 8;
  // printf("%d\n", address);
  address = address | offset;
  // printf("%d\n", address);
  return address;
}
```