Memory Management Unit (MMU)

Author: Gabriel Hofer

Date: May 2, 2021

The Purpose of Virtual Memory

Operating Systems support virtual memory for the following reasons:

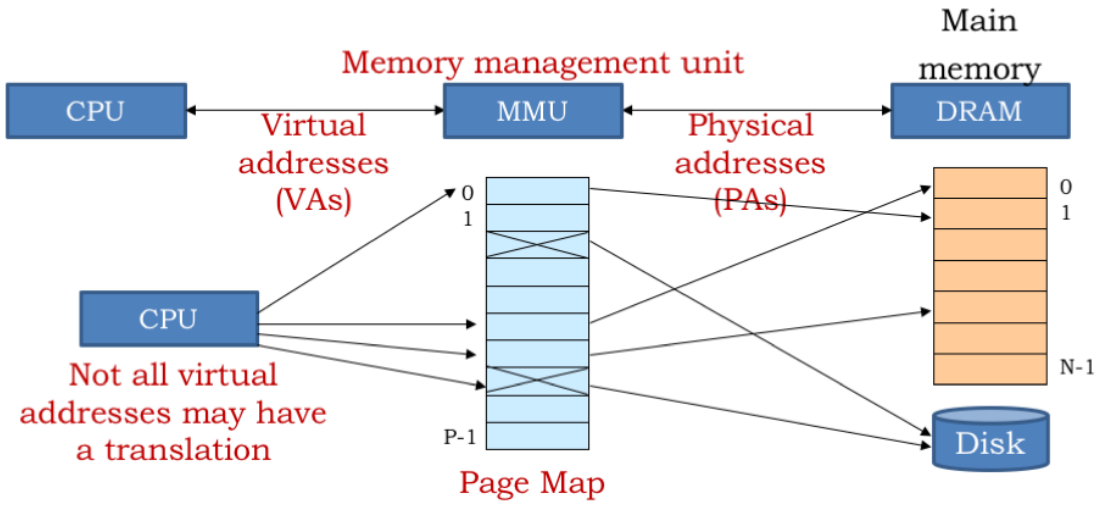
- Applications can have their own separate or isolated chunk of memory that can't be accessed by other applications.
- Memory isolation increases security.
- It is possible to create arbitrarily large memory addresses that may or may not exist in actual memory

What are Segmentation, Paging, and Page Tables?

In order for different processes to have their own memory, memory needs to be partitioned or divided into smaller pieces. This is called segmentation. The range of virtual addresses is partitioned/divided into smaller chunks called pages. Physical memory is divided into smaller pieces called frames. Moverover, an address in virtual memory belongs to a unique page, and this address also has an offset in that page that tells us where the address is relative to the beginning of the page. An address in a frame also has an offset relative to the beginning address of that frame. Page tables are used to map virtual addresses to physical addresses.

Virtual Memory

- Two kinds of addresses:
  - CPU uses virtual addresses
  - Main memory uses physical addresses
- Hardware translates virtual addresses to physical addresses via an operating system (OS)-managed table, the page map



MIT OpenCourseWare slides

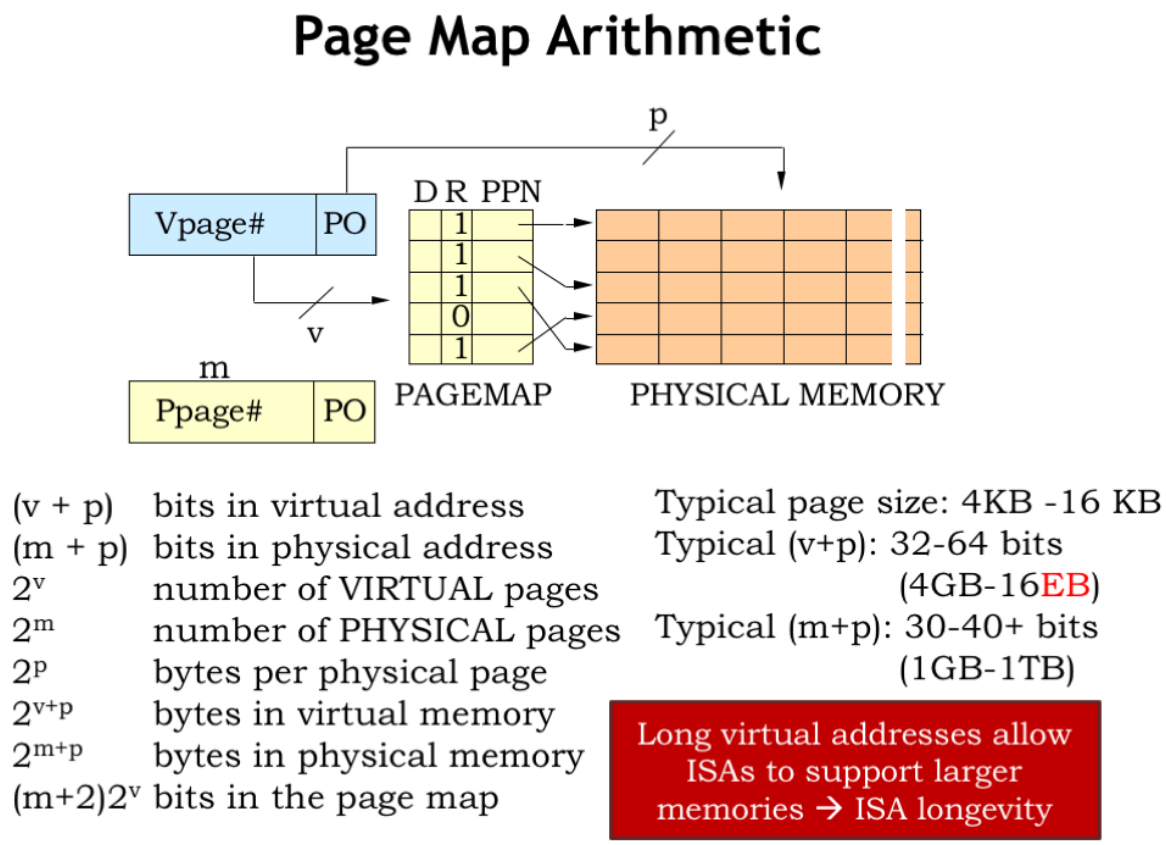
This project's simulation of virtual memory and paging was implemented based on MIT's OpenCourseWare slides on virtual memory.

There are three architectural parameters that characterize a virtual memory system and hence the architecture of the MMU.

P is the number of address bits used for the page offset in both virtual and physical addresses. V is the number of address bits used for the virtual page number. And M is the number of address bits used for the physical page number. All the other parameters, listed on the right, are derived from these three parameters.

```
/*****************************************//**
 *    8 bits in the page offset
 *    4 bits for the virtual page number
 *    3 bits for the physical page number
 ********************************************/
const int p = 8;
const int v = 4;
const int m = 3;
```

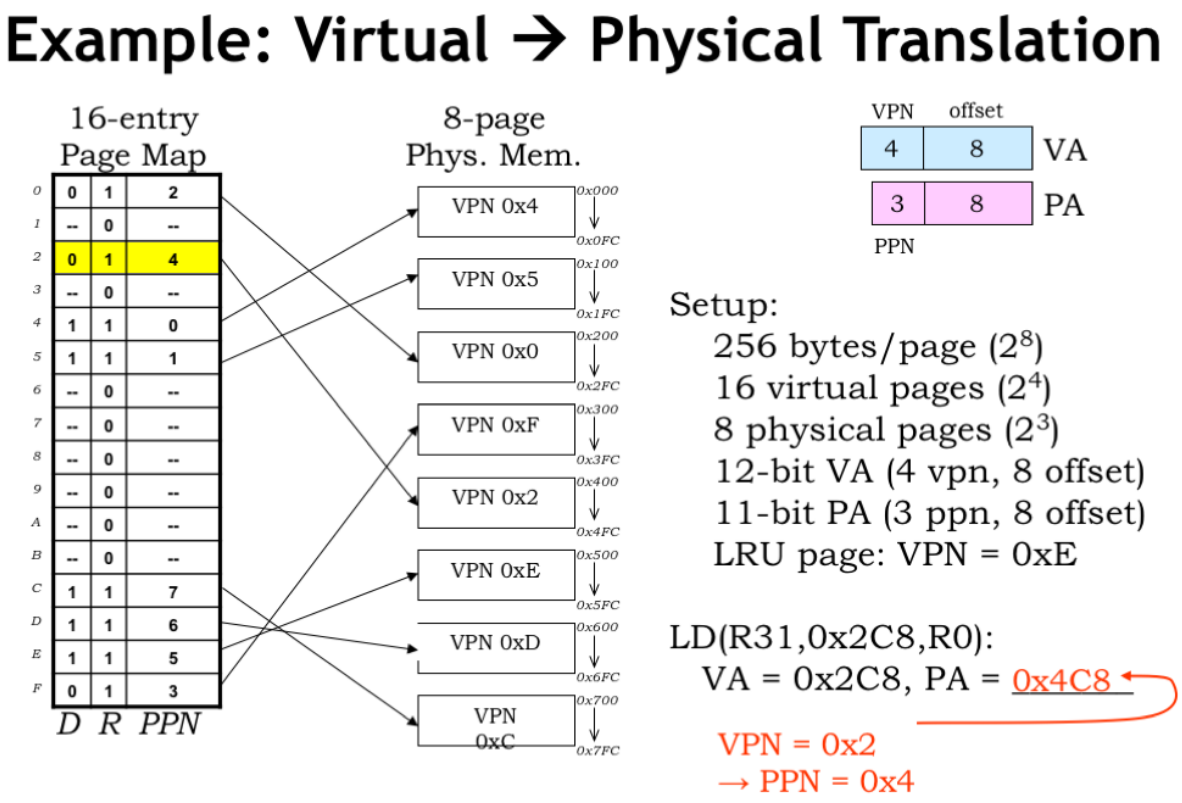
As we can see in the code, p, v and m are constant integers. So every time that the program runs, p has the vale of 8, v has the value of 4, and m has the value of 3. This can be changed in the future to let the user choose values for these variables, but I kept them constant to make explaining the implementation easier.



Based on the slide above we calculate the following for our simulation:

- (v+p) = 12 bits in virtual address
- (m+p) = 11 bits in physical address
- 2^v = 16 virtual pages
- 2^m = 8 physical pages
- 2^p = 256 bytes per physical page
- 2^(v+p) = 4096 bytes in virtual memory
- 2^(v+m) = 2048 bytes in physical memory

This slide illustrates the size of our page map, the size of virtual memory, and the size of physical memory for our program.



Looking at the Code

main() and REPL()

The main function immediately calls REPL. The REPL function is an infinite loop which reads in a line from the user. If the user enters the command "simulation", then the virtual memory simulation is started. If the user enters the "exit" command, the program terminates.

```
/*******************************************//**
 *    Two commands exist in the shell:
 *
 *            dash> simulation
 *            dash> exit
 ***********************************************/
void REPL(){
  char cwd[PATH_MAX];
  char * buf=NULL;
  size_t leng=64;
  char str[] = "simulation\n\0exit\n\0";
  for(;;){
    getcwd(cwd, sizeof(cwd));
    printf("%s> ", cwd);
    getline(&buf, &leng, stdin);
    if(!strcmp(buf, str)) simulation();
    else if(!strcmp(buf, &str[12])) return;
  }
}

/*******************************************//**
 *    Start the diagnostic shell
 ***********************************************/
void main(){
  REPL();
}
```

simulation()

Let's assume that the user enters the virtual memory simulation. Next, the user enters the number of processes they want to create. Then a menu prompts the user to decide which type of page table to simulate: multiple separate page tables, one for each process, or an inverted page table that all processes share.

```
/*********************************************//**
 *    Simulation Menu
 ***********************************************/
void simulation(){
  printf("--------------  Paging Simulation  --------------\n");

  /* Ask user for number of processes and number of pages */
  int q, p, pid;
  printf("\nNumber of processes: ");
  scanf("%d", &NUM_THREADS);

  /* Ask user which type of page table to simulate */
  int option;
  printf("Choose page table simulation type: \n");
  printf("\n\t1. Separate Page Tables\n");
  printf("\t2. Inverted Page Table\n\n");
  printf("Option: ");
  scanf("%d", &option);

  switch(option){
    case 1:
      separatePageTables();
      break;
    case 2:
      invertedPageTables();
      break;
    default:
      printf("Invalid Option\n");
      break;
  }

}
```

separatePageTables()

If the user chooses separate page tables, each process will be assigned it's own page table.

In this function, multithreading is used to create multiple lightweight processes. Threads were used in this program because threads share the same memory.

We want the threads to share the same memory because they need to be able to access the same physical memory.

As shown in the code, the number of threads/"processes" created is equal to the NUM_THREADS variable which was entered by the user.

When `pthread_create()` is called, a new thread starts execution in the `makeTableAndRequests()` routine.

```
/*****************************************//**
 *    Each process gets its own page table
 *****************************************/
void separatePageTables(){
  printf("------------  Separate Page Tables  -------------\n");

  pthread_barrier_init (&barrier, NULL, NUM_THREADS+1 );

  pthread_t threads[NUM_THREADS];
  int rc;
  for(int i=0; i<NUM_THREADS; i++){
    rc = pthread_create( &threads[i], NULL, makeTableAndRequests, (void *)i);
    if(rc){
      fprintf( stderr, "Error: unable to create thread, %d\n", rc);
      exit(-1);
    }
  }

  pthread_barrier_wait( &barrier );
  for(int i=0; i<NUM_THREADS; i++)
    pthread_join(threads[i], NULL);
  pthread_exit(NULL);
}
```

makeTableAndRequests

The function allocates a page table for one process. The page table is actually implemented as three separate arrays. A page table consists of

- R - a character array where element i represents the resident bit for page i
- D - a character array where element i tells us if the contents in physical memory match the contents in secondary storage (on the disk).
- PPN - an integer array where element i store the physical page number PPN for page i in the page table.

Additionally, we also allocate an array called pageFrequency which helps us keep track of how often pages are accessed by the CPU. pageFrequency is used to implement the page replacement algorithm (LFU).

After the page is allocated, the program starts simulating memory accesses by making several calls to the VtoP() function.

Inside the for loop, a random virtual address is generated.

Then VtoP() is called to lookup the physical address.

We don't do anything with the physical address, because we are mostly interested in implementing the "lookup".

```
/*****************************************//**
 *    1. Allocate new Page Table for thread
 *    2. Access virtual memory by calling VtoP()
 *****************************************/
void * makeTableAndRequests(void *threadid){
  long tid = (long)threadid;
  printf("Thread ID, %ld\n", tid);

  /* Thread allocates its own page table */
  /*  Allocate 2^v pages in page table */
  char * R = malloc( (1<<v) * sizeof(char));
  char * D = malloc( (1<<v) * sizeof(char));
  int * PPN = malloc( (1<<v) * sizeof(int));
  int * pageFrequency = malloc( (1<<v) * sizeof(int));

  /* Access virtual memory multiple times */
  for(int i=0;i<16;i++){

    /*  produce random address in virtual memory */
    int r = rand() % (1<<(v+p)+1);
    printf("Thread %ld requesting address: %d\n", tid, r);

    /*  request access to virtual memory location */
    VtoP(r, R, D, PPN, pageFrequency);
  }
```

```
  pthread_barrier_wait(&barrier);
  pthread_exit(NULL);
}
```

VtoP

This function accepts a virtual address and calculates the virtual page number (VPageNo) and the page offset (PO).

pageFrequency is updated to reflect accessing a virtual page in the page table.

Then, if the resident bit is set, the physical address is calculated and returned. Otherwise, a PageFault occurs.

```
/*****************************************//**
 *    Virtual Address --> Physical Address
 *    @return physical address
 *****************************************/
int VtoP(int Vaddr, char * R, char * D, int * PPN, int * pageFrequency){
  int VPageNo = Vaddr >> p;
  int PO = Vaddr & ((1 << p) - 1);

  pageFrequency[VPageNo] += 1;
  if(R[VPageNo] == 0)
    PageFault(VPageNo, R, D, PPN, pageFrequency);

  return (PPN[VPageNo] << p) | PO;
}
```

PageFault

First, we call the SelectLRUPage() to find the page that is accessed the least often. Then we check to see if the dirty bit for that page is set. If it is, we need to write the current page to secondary memory (the disk).

Then the page table is updated to map the virtual page to a physical memory frame.

```
/*****************************************//**
 *    Handle a missing page
 *****************************************/
void PageFault(int VPageNo, char * R, char * D, int * PPN, int *
pageFrequency){
  printf("\tPage Fault occurred\n");
  int i;
```

```
  i = SelectLRUPage(pageFrequency);
  if(D[i] == 1)
    WritePage(DiskAdr[i],PPN[i]);
  R[i] = 0;

  PPN[VPageNo] = PPN[i];
  ReadPage(DiskAdr[VPageNo], PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 1;
}
```

SelectLRUPage

This function select the virtual page that is accessed the least often. This is done by updating a frequency table called pageFrequency every time a memory address is accessed.

The function simply loops through the frequency array the returns the index of the smallest item in the array.

```
/*****************************************//**
 *    Page Replacement Algorithm:
 *    Least Frequently Used (LFU)
 ******************************************/
int SelectLRUPage(int * pageFrequency){
  // change to SelectLFUPage()
  int idx=0;
  int mn=100000;
  printf("\tpageFrequency[]: ");
  for(int i=0;i<16;i++){
    printf("%d ",pageFrequency[i]);
    if(pageFrequency[i]<mn){
      mn = pageFrequency[i];
      idx = i;
    }
  }
  printf("\n");
  pageFrequency[idx] = 1;
  printf("\tLeast Frequency Page: %d\n", idx);
  return idx;
}
```

Usage

```
$ tar -xzf final.tgz
$ cd final
$ make
$ ./dash
```

To keep things simple, there are two commands in this dash shell. We can run the simulation by entering the command "simulation" into the shell:

```
/home/gabriel/dash> simulation
```

- Enter the number of processes you want to run.
- Next, when the program asks for the user to choose a page table simulation type, enter ‘1’ because the inverted page table isn’t implemented.
- Then the program creates NUM_THREADS processes and accesses the virtual memory multiple times. The output prints which process accessed which address, when a page fault occurs, the pageFrequency table, and which page is swapped.

The second command is "exit" which exits the shell:

```
/home/gabriel/dash> exit
```

Conclusion

The inverted page table isn’t implemented.