6.033 - Design Project 1

Isolation and Sharing with Segments

Alexandros Kyriakides

Table of Contents

Abstract

This paper describes a general-purpose memory system. This system has two main advantages:

- A *Single address space* to allow sharing of memory between programs.
- *Enforced modularity* to allow data isolation.

The core of the design is the Memory Management Unit (MMU). The MMU is implemented in hardware and acts as a mediator between CPU and main memory. Programs in this system use Virtual Addresses which are translated by the MMU to Physical Addresses. Any memory access performed by this system has to pass through the MMU. In this way the MMU has the ability to enforce access policies. Modularity in achieved by dividing the memory into segments. This allows each segment to have its own permissions for each running process.

1 Introduction

The problem I will be trying to solve in this paper is that of enforcing modularity on a memory system with a single address space. The principal advantage of a single address space is that virtual addresses have a globally unique interpretation - a given piece of data appears at the same virtual address regardless of where it is stored or which programs access it. [1] The single address space enhances sharing and cooperation [2] between running processes. Because processes can share this single address space however, it is important to design a system that enforces modularity such that certain parts of the memory are isolated from certain processes. Therefore, using modularity and a single address space, the system will be able to provide both isolation and sharing of memory. This is important because in some cases it is advantageous for processes to share memory with other processes, while at the same time it is important for a process to make sure that certain parts of its data in memory cannot be accessed or modified by another process.

To make this clearer, consider the following scenario. A user executes the command `emacs` on a computer and starts the editor emacs running on a computer. While emacs is running, there will be a part of memory that is storing the emacs executable code. At the same time another part of memory will be storing the variable data that emacs is using (e.g. the text typed into one of the emacs buffers). We therefore have two memory areas, which I will call: a) the code area and b) the data area. If another user now starts another instance of the emacs program (which runs exactly the same code), we see a need for both sharing and isolation. The second instance of emacs will share the code area of the first instance of emacs, thus saving memory. It is also important however, to restrict access to the data area of the first emacs so that the second instance of emacs does not have access to this data. Similarly, the first instance of emacs should not have access to the data area of the second instance of emacs. This is isolation of the data areas.

2 Overview

I will now describe how I propose to solve the above problem. The main idea is to partition the memory into **segments**. There will be two types of memory addresses:

i) Virtual addresses

ii) Physical Addresses

Both addresses are 32 bits. The top 8 bits of the Virtual Address define a segment in memory. The 24 remaining bits define an offset in that segment. This is shown in **Figure 1.**

8-bit	24-bit
Segment ID	Offset

Figure 1: Bit description of a Virtual Memory address

The Physical Address is simply the address that the physical memory (RAM) accepts on its address port. Programs use only Virtual memory addresses. These Virtual addresses are translated to Physical addresses before being sent to the physical memory. I will provide a Memory Management Module (which I will just call MMU from now on) that will be responsible for the translation from Virtual to Physical addresses. The translation process will be described in detail later. **Figure 2** shows a simple schematic of how the CPU, MMU and RAM are connected such that the translation can take place.

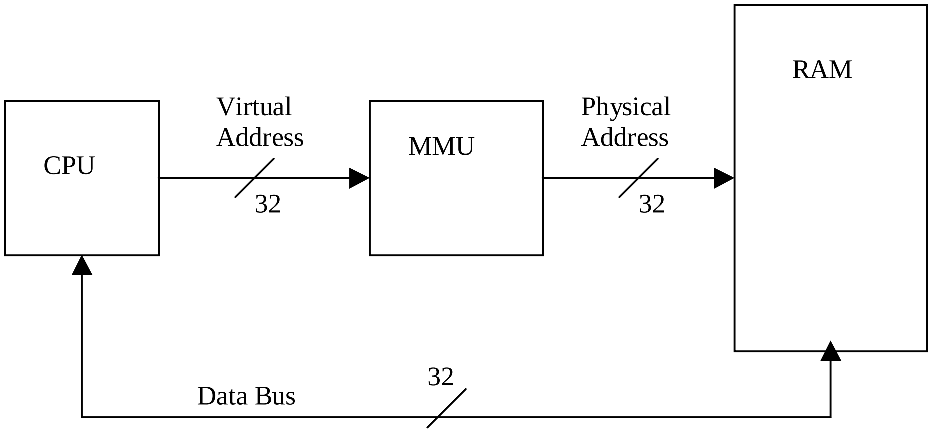

Figure 2: The CPU issues Virtual memory addresses in virtue of the running code. The MMU translates Virtual addresses to Physical addresses. The Physical address is passed to the RAM. During a read from memory, the RAM drives the Data Bus with the data located at the Physical Address. During a write to RAM, the CPU drives the Data Bus with the data to be written, and the RAM writes it to the Physical Address.

To guarantee a single address space there is a unique one-to-one mapping between Virtual addresses and Physical addresses. This solves the problem of sharing. To ensure modularity, the MMU does not act as a simple translator. The MMU has the ability to decide, during any memory access, if that memory access is valid or not. If a running program does not have read access for example, to a certain memory location, the MMU will stop the read operation. The segment ID's allow the system to form a simple logical structure for permissions. For each segment, the system stores a set of permissions that dictate the type of access that each process has on that specific segment. Therefore, when the CPU performs a memory operation (instruction fetch, LOAD, STORE) it sends a Virtual address to the MMU. The MMU can identify the memory segment that the CPU is trying to access, because the Segment ID is simply the top 8 bits of the Virtual address. The MMU can find out what permissions the currently running process has on this memory segment. Depending on the type of operation being performed, and depending on the permissions of the current process on that segment, the MMU acts accordingly. This solves the problem of isolation.

3 Design

The goal of my design is to make a simple system that can enforce modularity on a single address space. The design does not try to make optimizations. I will discuss possible alternatives and optimizations, but these are not implemented in my design. I have tried to make the design as simple as possible.

3.1 The Memory Management Unit (MMU)

The MMU is the main part of the memory system that I will describe in this paper. The MMU acts as a mediator between the CPU and the memory, and it has two main functions:

i) Translate Virtual memory addresses to Physical memory addresses

ii) Decide if memory operations are valid

In order for the MMU to perform its function properly, it needs to communicate with the CPU and with the Operating System's kernel. The MMU has an internal register called the StateRegister. The MMU has internal memory of at least 4096 bytes (4Kb).

3.1.1 MMU - kernel interface

An operating system compatible with the MMU has to provide the MMU with information about memory segments. The kernel alone manages the memory segments. The MMU simply enforces the decisions of the kernel. This follows the principle of *separation of mechanism and policy*. The MMU is the mechanism and the kernel provides the policy. Because the MMU is hardware and the kernel is software, it will be easy to change the policy in the future by changing the kernel.

Communication is achieved by using a **Memory Management Segment**. The Memory Management Segment is a segment in memory which always has the Segment ID of 00h and always begins at the Physical Address location of 0000 0000h. This segment always has a length of 2^{12} = 4096 bytes. The first byte of this segment is called the Control Byte. The format of this byte is shown in **Figure 3**.

1-bit	3 bits	4-bit
Valid Bit	Reserved	Counter

Figure 3: The format of the Control Byte. This byte is always located at Virtual Address 0000 0000h which is always mapped to Physical Address 0000 0000h.

The 3 bytes after the Control Byte are reserved. The next 3072 bytes of the Memory Management Segment make up the **Segment Table.** The remaining 1020 bytes are reserved. **Figure 4** shows the format of the Memory Management Segment**.**

<table>
<tr><td><i>0000 0000h</i>
Control Byte</td><td><i>0000 0001h</i>
Reserved</td><td><i>0000 0002h</i>
Reserved</td><td><i>0000 0003h</i>
Reserved</td></tr>
<tr><td colspan="4">Segment Table
(contains exactly 256 Segment Descriptors)
<i>0000 0004h - 0000 0C03h</i>
(3072 bytes)</td></tr>
<tr><td colspan="4">Reserved
<i>0000 0C04h - 0000 1000h</i>
(1020 bytes)</td></tr>
</table>

Figure 4: The Memory Management Segment. This segment has ID 0, and always starts at Physical Address location 0000 0000h. The numbers in italics indicate memory addresses. They are both Virtual and Physical memory addresses.

The **Segment Table** is an array of 256 **Segment Descriptors**. Each Segment Descriptor is indexed in the array by its Segment ID. Therefore, to find the Segment Descriptor for Segment 34, for example, we retrieve the array element 34. Segment ID's range from 0 to 255, inclusive. Each Segment Descriptor has a length of 12 bytes. **Figure 5** shows the format of each Segment Descriptor.

<table>
<tr><td colspan="2">PhysAddr (4 bytes)</td></tr>
<tr><td colspan="2">Length (4 bytes)</td></tr>
<tr><td>Mode (1 byte)</td><td>Reserved (3 bytes)</td></tr>
</table>

Figure 5: The format of the **Segment Descriptor**. PhysAddr is the start of the segment in physical memory. Length is the length of the segment in bytes. Mode is a special byte that describes permissions.

The kernel communicates with the MMU by writing to the Memory Management Segment (MMS), and then having the MMU read it. Only the kernel can read and write to the MMS. The MMU only reads from the MMS. The MMS provides all the information the MMU needs at any one time. It provides translation information as well as permission information. The kernel has the job of updating this table whenever needed. Before updating the table, the kernel clears the Valid Bit in the Control Byte. Once the kernel has finished updating the table, the Valid Bit is set. Then the Counter in the Control Byte is incremented. I use the Valid Bit so that if the MMU tried to read the Segment Table and the Valid Bit is cleared, then it knows that the Segment Table is not valid. This should never happen. If it does happen, then the MMU should signal a fatal error to the CPU, which can then cause an interrupt which forces a kernel error-handling routine.

The MMU has a register called the StateRegister. The StateRegister stores the last value of the Counter read from the Control Byte. If the Counter is different than the value in the StateRegister then the MMU reads the Segment Table from the Memory Management Segment in RAM and stores it in its internal memory. It is better that the MMU has this internal memory to store the Table, so that it does not have to read the whole table from RAM during every machine cycle. During the end of every machine cycle, the MMU simply has to read the Counter of the Control Byte to make sure if it needs to update its internal Table. **Figure 6** describes the MMU's algorithm in detail.

```
1      X = read_byte(0000 0000h);       //reads the Control Byte from the MMS
2      V = (X & 80h) >> 7;        //stores the Valid Bit in V
3      X = X & 0Fh;      //masks out the top 4-bits and leaves the Counter in X
4      if (StateRegister != X){
5             if (V != 1) {
6                           SignalError;
                            }
                    else{
7                           ReadTable; //Reads Segment Table into Internal Memory
8                           StateRegister = X;
                            }
       }
```

Figure 6: The MMU Table-update algorithm. X and V refer to 8-bit registers. They can be part of a larger 32-bit register.

In order to perform its function, the MMU requires a simple ALU and five 32-bit registers. The timing of the above algorithm is important. The MMU has to be synchronized with the CPU. The check for the update (line 4) is performed at the end of each machine cycle. The update (line 7) is performed at the end of the next machine cycle. The reason for this will be explained later.

3.1.2 CPU - MMU interface

The CPU and MMU are synchronized so that the MMU knows when the CPU is doing an instruction fetch (at the beginning of a machine cycle) and when the CPU is doing a memory access (at the end of a machine cycle). It is important for the MMU to differentiate between an instruction fetch, a memory read and a memory write. An instruction fetch is equivalent to execution of code. The read/write pin of the CPU has to go to the MMU so that it can differentiate between reads and writes. The MMU also has to signal the CPU if an error occurred. Connecting an interrupt signal line from the MMU to the CPU allows this to happen. If this interrupt occurs, the CPU jumps to a kernel service routine that handles the error. This routine is in the **Kernel Entry Segment**.

The Kernel Entry Segment

When a switch is made from user code to kernel code, the MMU needs to know that kernel code is running. To ensure this, we define a Kernel Entry Segment that has the segment ID of 1. Code in this segment has special privileges. When code from this segment is running, it can write to the MMS, irrespective of the Segment Table stored in the MMU. Also, all processes are given execute permission to this segment. Whenever we are not executing kernel code, the only way we can switch to kernel mode, is by executing code in the Kernel Entry Segment. The MMU has a register to keep track of the segment from which code is being executed. It updates this register every time an instruction fetch is made by the CPU. This register is called PC (Program Counter) because it follows the CPU's PC.

Figure 7 shows an updated schematic of the system.

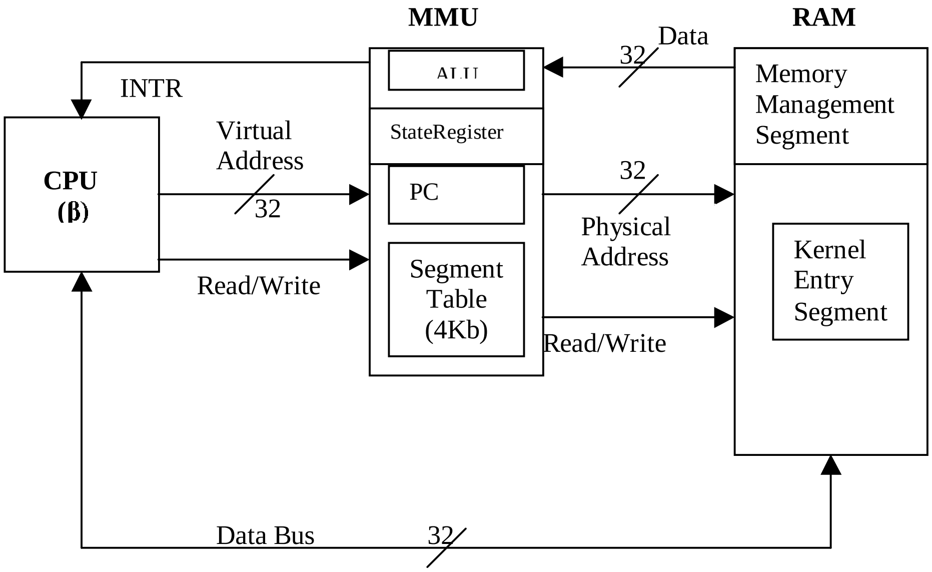

Figure 7: A schematic of the complete system

3.1.3 Address Translation

The Segment Table stored internally in the MMU provides all the information that the MMU requires in order to perform the address translation. The MMU uses the top 8 bits of the Virtual Address it receives from the CPU in order to index its internal Segment Table. It then retrieves the Segment Descriptor. The Segment Descriptor provides the start of the segment's physical address as well as its length and its mode. The MMU first checks the mode in order to decide what to do. **Figure 8** shows the format of the Mode byte.

1-bit	3 bits	1-bit	1-bit	1-bit
Valid	Reserved	Read	Write	Execute

Figure 8: The format of the Mode byte found in a Segment Descriptor

If the Valid bit is not set, then it means that the segment does not exist. The currently running process has permission to Read/Write/Execute if and only if the corresponding bit is set.

Figure 9 describes the algorithm that the MMU uses in the translation process. Due to its interface with the CPU, the MMU knows if the current memory operation is an instruction fetch (INSTR), a memory read (READ), or a memory write (WRT).

```
Translate (Virtual_Address) {


      E = PC >> 24;             //store the executing segment ID
      X = Virtual_Address;
      S = X >> 24;             //store segment ID into S
     M = Segment_Table[S].Mode; //store the Mode byte into M

      If (M.7 == 0) {                         //check Valid bit
                  Signal_Invalid_Error;      //Segment does not exist
                  }

      //Segment exists

      //give write permission to segment 1 code when writing to MMS
      If ((WRT) AND (E == 1) AND (S == 0)) goto perms_OK;

      If ((INSTR) AND (M.0 == 0)) {    //check for execute permission
      Signal_NoExecutePermission_Error;
      }

      If ((READ) AND (M.2 == 0)) {            //check for read permission
      Signal_NoReadPermission_Error;
      }

      If ((WRT) AND (M.1 == 0)) //check for write permission
      Signal_NoWritePermission_Error;
      }
      }

      :perms_OK

      //permissions OK - return physical address
      P = SegmentTable[S].PhysAddr + (X & 00FF FFFFh);

      return P;  //return the physical address
}
```

Figure 9: Address Translation Algorithm for the MMU

If an error is signaled, then the MMU sends an interrupt to the CPU, which then forces a kernel error-handling routine to execute. The code for this routine resides in segment 1.

3.2 The Kernel

The kernel is responsible for managing the memory. It updates the Segment Table in the MMS whenever needed. This allows the kernel to have total control of the permissions for each segment. Whenever the kernel wants to change the permissions of the current process, it updates the Segment Table in the MMS, and by then incrementing the Counter in the Control Byte of the MMS, it signals the MMU to update its internal Segment Table. This permissions change is usually performed during context-switching (from one process to another). The kernel needs to maintain a data structure that stores the permissions of each process, for each segment.

Entering the kernel has been described above. The only way to enter the kernel code is by jumping to code in the Kernel Entry Segment. This segment gives execute permission to all processes. Only the kernel has read and write permissions on this segment. System calls therefore, begin with a jump to the appropriate place in the Kernel Entry Segment. The code residing in this segment has the job of updating the Segment Table in the MMU so that the permissions for kernel mode are set. Control is then transferred to the appropriate segment where the kernel code is stored.

Leaving the kernel is done by updating the Segment Table in the MMU. The last instruction in the Table update has to be the incrementing of the Counter in the Control Byte in the MMS. The instruction following the increment, should then be a jump or a return to user code. As described above, the MMU updates its table at the end of the machine instruction following the Counter increment. This allows an extra instruction to take place (the jump or return instruction) with the current permissions, before the permissions are changed. If permission change was immediate, then the jump instruction following the permission change would be executing with user permissions (not kernel permissions). Because this jump or return instruction is in a segment executable only by the kernel, it would not have permissions to execute.

3.2.1 Kernel System Calls

Apart from the usual system calls that the kernel provides, it has to provide the following:

```
VirtualAddress = mem_allocate(size,permissions)
```

//the kernel allocates a new segment of length `size` to the process, and returns the Virtual Address of the beginning of the segment.

```
set_permissions(SegmentID,permissions)
```

//the current process requests that the kernel set the permissions of the segment `SegmentID`. It is up to the kernel to decide if the current process has permissions to make this change.

```
fork(permissions)
```

//creates another instance of the currently running program, and sets the permissions of the newly created segments to the ones specified by `permissions`. The `permissions` have to be equally or more restrictive than the current process's permissions.

```
exec(program, permissions)
```

//executes `program`, setting permissions on its segments. The `permissions` have to be equally or more restrictive than the current process's permissions.

The `permissions` structure is defined by the kernel.

3.2.2 Context-Switching

The kernel is responsible for making a context-switch. A context-switch occurs when control is passed from one running process to another. The way control passes from one process to another is by running code in the Kernel Entry Segment, such that control is passed to the kernel. The kernel then performs the context switch. In this memory system, it is important for the kernel to maintain three pointers for each process: the Return Location of the process, the Data Pointer and the Base Pointer. The Return Location is where control is passed when returning to the process. The Data Pointer is the starting address of the data segment of the process and the Base Pointer is the starting address of the stack segment of the process. The kernel is free to store these any way it likes. The current memory system does not place any restrictions to this.

4 Discussion

Given the above design, it is possible for multiple programs to share the single address space. This controlled by the kernel. The kernel can give read permissions to two or more processes for a certain segment. This allows sharing. Programs can allocate new segments by calling mem_allocate and a program can indicate whether other programs are allowed to read, write or execute its segments by using the permissions structure in the `mem_allocate` and `set_permissions` system calls.

Modularity is enforced by the fact that a process can request that it has sole permissions to read, write or execute a certain segment. The kernel sets these permissions, and the MMU ensures that this is done.

The kernel can set permissions on a certain segment such that two processes have execute permissions on that segment. This allows two programs to share an executable code segment.

The kernel sets the permissions for its own segments such that no other process can read, write or execute from them. The only exception to this, is the Kernel Entry Segment, which can be executed by any process.

When a program needs to run another program, it issues either a `fork()` or `exec()` as described in section 3.2.1.

5 Conclusions

The design described in this paper allows for a single address space and enforces modularity. I decided to move most of the "intelligence" to the kernel. This allows greater flexibility, because it is easier to change the kernel than it is to change the MMU. A different approach could be to treat the kernel just as any other program, and make the MMU control the permissions of every process. This would mean that the MMU would need to have knowledge about all processes running on the system as well as to decide on a permission policy for each process. I believe that this is not the correct approach because it would be very restrictive. Also, in a system, it is the kernel that has the most information about processes, so the kernel should decide on policies. By restricting the mechanism to the MMU and the policy to the kernel, it makes the system less complex and more flexible.

References

[1] http://www.cs.washington.edu/homes/levy/opal/sigops92.ps

[2] http://www.cs.washington.edu/homes/levy/opal/opal.html