Simple Sharing and Enforced Modularity
Access Control in a Segmented Memory System

Jeff Bartelma

March 21, 2002

Contents

List of Figures

Abstract

All programs in a segmented memory system share the same address space. This arrangement makes sharing data very simple, but it also makes protecting data difficult. This paper presents the design of a memory management unit (MMU) that is able to enforce modularity within the single address space of a segmented system. The MMU draws on permission tables in main memory to act as an integrated virtual address translator and access control mechanism. The MMU's data structures and algorithms are demonstrated in the context of supporting standard O/S features such as memory allocation.

1 A Segmented Memory System

Most virtual memory systems provide each program module with a distinct address space, tied to physical memory that is invisible to other modules. By restricting each module's view of memory, this "standard model" of virtual memory provides excellent fault isolation.

However, to claim the advantage of modularity, a programmer adopting the standard model forfeits a convenient interface for sharing memory. Two programs attempting to access the same physical location must use the different virtual addresses specified in their respective page maps.

Sharing would be simpler if all processes ran in the same address space, that is, if any program could reference any piece of data, irrespective of the program that owns it. Ideally, we would like to retain isolation while providing this palatable interface for sharing.

One way to approach this problem is to use a *segmented memory system*: divide the single address space into a number of "segments" and interpret the b high-level bits of each CPU-issued virtual address as a segment number. The remaining bits of a virtual address are used as an offset into the segment. This bitwise translation means that any process referring to a particular virtual address V is talking about the same physical location, as we desire for a simple interface.

Retaining isolation in such a system requires a memory management unit (MMU) to intercept each virtually addressed memory request issued by the CPU and ensure that it is "legal" before translating and passing it to main memory.

A good MMU must enable a program to prevent all other programs from accessing its segments. That is, the MMU must be able to enforce modularity. At the same time, the MMU must facilitate useful sharing; for example, two instances of Emacs should be able to share the same segment of executable code while maintaining distinct, private data segments. Finally, the MMU must protect the kernel and provide support for its memory-related tasks, e.g., allocating segments or starting programs from within other programs.

This paper presents an MMU design to support segmented memory for the 32-bit Beta processor, and demonstrates its viability in light of the constraints mentioned above.

2 Conceptual Overview of the MMU

To achieve enforced modularity in a segmented memory system, each process must be able to protect its segments from other processes as it sees fit. The O/S must therefore maintain a database of permission information describing which processes are allowed to read from, write to, or execute in each memory segment. The kernel must update this permissions table whenever it services a system call to allocate a new data segment, spawn a new process, or change the permissions of a segment.

The MMU works by synthesizing queries to this permissions table from three pieces of information:

1. The process making a memory request.

2. The memory segment this process wishes to access.

3. The type of access (read, write, and/or execute) the process is asking for.

Subsequent sections will more closely examine the MMU's state, data structures, and address translation algorithm.

3 MMU Details and Design Decisions

The following subsections discuss the Beta MMU design in detail and explain the reasoning behind each design decision.

3.1 Process-Based vs Segment-Based Access Control

To design an MMU capable of enforcing modularity in every case, it is necessary to distinguish process-based access control from segment-based access control. An example will help to illustrate the difference.

Suppose two instances of MATLAB, A and B, are sharing executable code in segment X. Suppose further that instance A should be allowed access only to data segment C, while instance B must be allowed access only to data segment D. If the MMU had data specifying only that code in segment X should be allowed access segment C (segment-based access control), it would fail to stop instance B of MATLAB from illegally accessing A's private data. See Figure 1. To prevent this problem, the MMU must control access to segments based on the process, not segment, that generated the memory request.

3.2 State

The MMU uses two state registers. In accord with the preceding discussion, one of these registers—maintained by the kernel—must contain the ID number of the process that generated a memory request. To enable this processID to act as a table index, the kernel is required to sequentially number its list of active processes. The kernel itself should always be process 0.

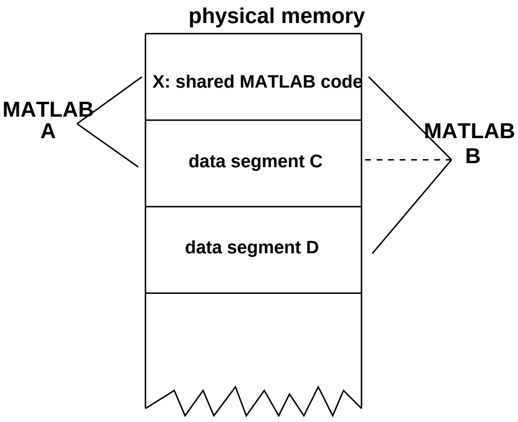


Figure 1: Two processes sharing a segment of executable code. The dotted line indicates unauthorized sharing allowed by segment-based access control.

The second MMU register, maintained by a few simple wires from the CPU, must contain the opcode of the instruction that generated the memory request. The opcode indicates whether a memory reference is a LD (read), ST (write), or JMP/BR (execute) request.

3.3 MMU Data Structures

3.3.1 Preliminary Considerations

An MMU needs data structures to store information about memory segments and permissions. However it is necessary to settle two preliminary design considerations before discussing suitable data structures.

First the MMU must know how to interpret the 32 bits of a virtual address. My design treats the high-level 8 bits of a virtual address as a segment number, and the remaining 24 as an offset. This arrangement permits a reasonable 256 memory segments, each of maximum physical size 2^{24} bytes. The maximum number of segments, we shall see, has implications for the way the MMU should store its data.

The second preliminary issue is to decide exactly what information about segments should be kept by the MMU to best meet its design criteria.

- To translate virtual addresses, the MMU needs to know the *physical address* corresponding to the beginning of each segment.
- To support segments of different physical size, the MMU needs to keep a *length* attribute for each segment. Then, isolating a small program in its own segment will not waste physical memory.
- To restrict access to uninitialized segments, the MMU should keep a *validity bit* for each segment. For instance, if the kernel has allocated only segments 1-35 to programs, a reference to an address in segment 212 should return an illegal address fault, because the physical address of this segment is undefined.

- Finally, to enforce modularity, the MMU must retain process/segment access permissions.

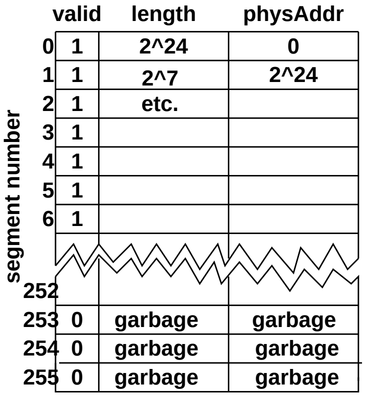
With these issues settled it is possible to consider the merits and drawbacks of various data structures.

3.3.2 Final Design and Justification

I will first present the final structure for the MMU data tables, then I will follow with a detailed justification.

Data about memory segments is split into two tables, a *SegmentTable* and a *PermissionsTable*. These tables reside contiguously in the above order at a hard-wired location. This location is accessible only to the kernel, which has permission to read and write the tables.

The differing purposes of the SegmentTable and PermissionsTable are best served by different organizations. The SegmentTable is indexed by segment and contains information pertaining to the physical aspects of each. The ith row contains the physical address, length, and validity of segment i. See Figure 2. The SegmentTable has a constant size of 256x3 fields.



segment number	valid	length	physAddr
0	1	2^24	0
1	1	2^7	2^24
2	1	etc.	
3	1		
4	1		
5	1		
6	1		
252			
253	0	garbage	garbage
254	0	garbage	garbage
255	0	garbage	garbage

Figure 2: Structure of the SegmentTable.

By contrast, the PermissionsTable is indexed by process. The jth row contains 256 access codes denoting process j's access level to each of the segments. See Figure 3. The access codes used in this table are shown in Table 1. The PermissionsTable has a variable size of Px256, where P is the number of processes currently running.

I arrived at the structural specifications above by breaking the design process into a sequence of questions and answering each in turn.

1. Should permissions be maintained in one large table or in a separate array for each process?

   Answer: One large table.

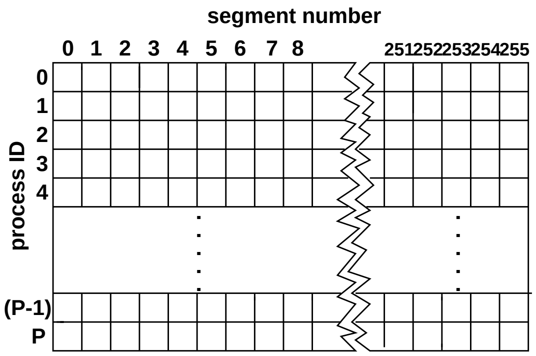


Figure 3: Structure of the PermissionsTable.

access code	**permission granted**
0	no access
1	read-only
2	read, write
3	execute
4	rd, wr, ex

Table 1: PermissionsTable access codes and the corresponding interpretations.

In the latter case, starting a new process entails finding a safe place to store its array of access permissions. To absolutely guarantee the integrity of this array we should allocate it a private segment, but this is a terrible waste of virtual address space. Moreover, the system must keep track of the location of every such array, requiring a large, variable number of MMU state registers.

On the other hand, if the data is kept in one large chunk that never moves we may avoid adding MMU state by hardwiring its starting address. Also, we can protect all the permissions data together by placing this large chunk in a single protected segment; we don't need to waste a significant fraction of the virtual address space storing permissions.

2. Should permissions data be organized by process or segment?

   Answer: By process.

   The choices are as follows.

   (a) For each segment, keep a list of the processes that are allowed access.

   (b) For each process, keep a list of the segments it can access.

   In organization (a), the permissions table can have only as many rows as segments, but it is as wide as the number of processes. This is actually a terrible disadvantage, because adding a new process corresponds to widening every row of the table by one entry. This in turn requires readdressing all but the first row of the table! A constant offset from the location marking the beginning of the MMU data tables may correspond to row 4 of the table, or row 40, depending on the number of processes running.

Organization (b) requires separating data about a segment's physical attributes from data about permissions, because physical segment information is indexed by segment, and we are indexing permissions by process. However, this is a minor drawback. The real news is that adding a new process does not require the whole table to be readdressed. Rather than widening each row of the table, adding a process simply tacks a new row on the end. This is an enormous advantage over organization (a).

3. Should the width of each PermissionsTable row be fixed at maximum (256, the number of segments), or should each row be allowed a different length?

   Answer: The width should be fixed at 256.

   Since the average row of the PermissionsTable will contain mostly zeroes (i.e., the average process will have no access to most segments), we could save space by keeping explicit access codes for only those rare segments granting non-default access. However, this "optimization" creates table rows of varying length. Again this makes reliable offset addressing into the table from a fixed location impossible. It only makes sense to bear the additional complexity of this scheme if the system has an extraordinarily small memory or such a large number of segments that it is impractical to store an explicit access code for each process/segment combination.

3.4 Virtual Address Translation

Because all memory requests must pass through the MMU's virtual address translation procedure, this is a natural place to implement access control. The translation procedure queries the SegmentTable and PermissionsTable described in the previous section using the virtual address, as well as the opcode and processID gathered from the MMU's two registers. Although the translation procedure would be implemented in hardware, its duties may be represented conceptually as follows:

```
#define SEGMENTMASK 0x000000FF
#define OFFSETMASK 0x00FFFFFF

int translate (int virtual, int opcode, int processID) {
  int segmentNum = (virtual>>24) & SEGMENTMASK;
  int offset = virtual & OFFSETMASK;

  if(SegmentTable[segmentNum].valid==1){ // is segment valid?
     struct SegmentDescriptor segment = SegmentTable[segmentNum];}
    else return ILLEGAL_ADDRESS_FAULT;

  int accessCode=PermissionTable[processID][segmentNum];
  /* case analysis based on access code and opcode */
  bool accessAllowed=caseAnalysis(accessCode, opcode);

  if ((offset<segment.length) && accessAllowed){
    return segment.physAddr+offset;}
  else return ILLEGAL_ADDRESS_FAULT;
  }
```

4 MMU Design Viability

The following sections demonstrate the viability of the MMU design by outlining how the O/S and user applications can use it to realize the design goals of a segmented memory system.

4.1 Sharing Executable Code

One potential advantage of a segmented memory system is the ability for two instances of a program to share executable code. If the program is large, this tactic can save a significant amount of memory.

Recall the MATLAB code-sharing example used to motivate process-based access control in section 3.1. Having examined the MMU design in detail, we may take a closer look at how the MMU provides the capability to share code.

Thus, consider once more two instances of MATLAB, A and B, again with respective private data segments C and D, sharing executable code from the same segment X. When instance A is executing, the code running in segment X must store its variables in segment C, and when instance B is executing, the code in X must use D for storage. While this situation appears confusing to a human, to the MMU, the MATLAB instances A and B are as different as any other pair of processes. The MMU prevents process A from accessing process D using the same method as always: indexing the PermissionsTable using the current processID, and checking permission codes for the segments being requested.

While these checks prevent each instance of MATLAB from corrupting the other's private data, they do not explain how two processes running identical code know to ask for their respective data segments in the first place. When a context switch occurs making A the current process, the kernel restores A's saved stack pointer and other registers. Because A's stack pointer points to a location in C, A asks for and finds its mutable data in segment C.

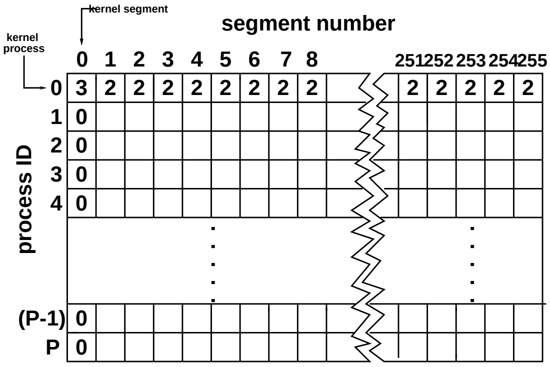
4.2 MMU-Kernel Interactions

In order to show that the MMU can enforce modularity, it is first necessary to examine the ways the kernel and MMU interact. The following sections explore how the MMU protects the kernel, and how the kernel modifies the MMU's data structures in the course of common O/S functions.

4.2.1 Protecting the Kernel

A good segmented memory system should be able maintain kernel integrity using the same mechanisms that protect user programs. We can use the MMU to restrict access to the kernel as follows. First, the kernel process should be permanently fixed in segment 0, and its row of permissions set to [3 2 2 ... 2]. Then the kernel can execute in its own segment, and read and write every other segment. Whenever the kernel starts a new process (more on this shortly), it must set the first bit of the new process' permission vector to value 0 (no access). See Figure 4. Then regardless what process ID is in the MMU's register, the kernel code in segment 0 can be read, written, or executed by no other process.

In particular, provided the kernel's instruction for updating the process ID register is located inside segment 0, access to this instruction is limited by the MMU to the kernel process. With this access restriction already in place, a supervisor bit is not needed to further protect the kernel, and it can be eliminated from the Beta specification.[1]



	0	1	2	3	4	5	6	7	8		251	252	253	254	255
0	3	2	2	2	2	2	2	2	2		2	2	2	2	2
1	0														
2	0														
3	0														
4	0														
					⋮								⋮		
(P-1)	0														
P	0														

Figure 4: Protecting the kernel. The column of zero bits at left indicates that no process besides the kernel has access to the kernel's segment; the row of twos at top indicates that the kernel has read/write access to every other memory segment.

[1] Of course, interrupts would have to be handled by an alternative mechanism.

Of course, programs must still be able to make system calls requesting services from the kernel. How does execution get into the kernel? Just as in the virtual memory system described by Saltzer and Kaashoek[1], system calls should directly invoke a processor-provided hardware mechanism. However, this mechanism (which the Beta must now supply) executes a slightly different set of atomic actions:

1. Save the value of the MMU's processID register, and change it to 0, the kernel's fixed process number.
2. Save the program counter, and reload it with the address of the kernel's entry point, 0x00000000.

Similarly, to safely exit the kernel, the following actions must be executed atomically:

1. Reload the MMU's processID register from the value saved upon entering the kernel. Since the kernel is always process 0, we needn't save this register first.
2. Reload the program counter from its saved location.

Should the kernel need to perform a context switch during a system call it has only to replace the value of the processID register with the number of the process it is switching to, then load the program counter with the entry point of that process.

4.2.2 Allocating Data Segments

We would like for a program to be able to allocate new segments to hold data. This is also done using a system call, perhaps of form

```
int systemAllocateNewSegment(int length, int[] permissions)
```

The kernel handles such a call by looping down the SegmentTable, looking for the first row with a valid bit of 0. The segment corresponding to this row is the first unallocated segment. If there is no such row in the SegmentTable, the system has already allocated all 256 segments and must return a fault. The kernel checks to make sure the length request is fine (that is, less than 2^{24} or the amount of remaining physical memory, whichever is smaller). If this test checks out okay, the kernel changes the segment's valid bit to 1, sets the length as specified, and assigns an appropriate physical address. Immediately thereafter, the kernel replaces all but the topmost (kernel's) entry of the corresponding column of the PermissionsTable with the values specified in the system call. This is *not* a security hazard: the parent process is simply specifying which processes may access its data segment. Normally, we expect the parent process to endow itself alone read/write access, but these permissions are purely discretionary. Finally, the system call returns the first virtual address of the new segment to the program.

4.2.3 Starting Programs

Similarly, we would like for a running program to be able to start other programs. For instance, typing the name of an executable in a shell should start that program. The first step of this procedure should be to allocate a new segment to hold the program's code. But unlike allocating a new data segment, starting a new process raises potential security pitfalls.

For example, if Emacs creates a new data segment and decides to make it writable by all other processes, it is a curious situation but the program's prerogative. On the other hand, if the user starts Netscape, Emacs, and an xterm window using the shell command line, the shell should not have write access to the memory of all these processes! To enforce modularity, the kernel must therefore initialize the new segment's column in the permission table with all zeroes except for the first entry, which must be a 2. This is so process 0, the kernel, retains read and write access to the new data segment as always.

The next step is to start the new process. The kernel copies the program text to the newly allocated segment and adds a line for the new process to the bottom of the PermissionsTable. This line is required to be zero everywhere except for the location corresponding to the newly allocated segment. This entry of the PermissionsTable should be a 4, such that the program has read, write and execution access to its segment.

It is absolutely crucial that whenever the kernel adds a new row to the PermissionsTable, every entry is 0 except that of the new segment. Otherwise the new process has been given unauthorized access to all segments with nonzero entries, and modularity cannot be enforced.

In pseudocode form, the system call to start a new program might look like

```
int systemStartNewProgram(int length, programText){
  addr=systemAllocateNewSegment(int length, int[] allZeroesButFirst);
  copyProgTexttoNewSeg(addr, programText)
  addLinetoPermissionTable();
  }
```

4.2.4 Changing Permissions

But what if the new process really does want to share its segments? The system call above does not allow for this possibility. Finally, the kernel must provide a system call by which a properly authorized process may grant other processes access to its segments. It is assumed that the kernel has some facility for interprocess communication or other means by which one process may know of another and obtain its processID.

4.3 Enforced Modularity

We have seen how each common MMU-kernel interaction, e.g., allocating data segments, can be made to preserve enforced modularity using careful MMU table modifications as detailed above. In other words, the MMU and its kernel interface provide a way for programs allocating new data segments or starting other programs to ensure their segments will not be written, read, or executed by other processes.

The argument that the MMU design is sufficient to enforce modularity in general is, loosely speaking, inductive. Given that the kernel starts off isolated in segment 0 and that allocating new data segments, starting new programs, etc. can be done in such a way as to enforce modularity, the MMU design can be said to enforce modularity. This is notable because all processes reside within a single address space; each process can see, but still not access, virtual addresses it does not own.

5 Summary

The primary motivation for creating a segmented memory system is to simplify data sharing. However, because such a system allows any process to reference any piece of data, even in another program, careful access controls must be enacted. I have presented a memory management unit (MMU) design to support segmented memory for the 32-bit Beta processor. I have also argued that the design is sufficient to support common O/S tasks and enforce modularity even though all processes run in a shared address space.

References

[1] J. Saltzer and F. Kaashoek. *Topics in the Engineering of Computer Systems*, draft January 2002.