MPSA

Memory Sharing and Isolation in a Single Address Space

Sean Lie

2002 6.033 Design Project 1

Revised October 30, 2002

Abstract

Multiple address spaces are typically used to achieve memory sharing and enforced isolation. However, the use of multiple address spaces results in high memory overhead for a large number of processes. To reduce memory overhead, a new memory management model was designed using a single address space with multiple permission tables. The MPSA (Multiple Permission table in a Single Address space) model achieves both memory sharing and enforced isolation while requiring much less memory than typical models that use multiple address spaces. Since implementing enforced isolation in a single address space can often result in a complicated and awkward system, simplicity and ease of integration were fundamental design goals of MPSA. As a result, MPSA demonstrates that implementing enforced isolation in a single address space does not necessarily entail a complicated system nor does it require significant changes to existing hardware and software structures.

Table of Contents

Introduction **3**
Design Overview **3**
Detailed Design Description **4**
 Virtual Address Space **4**
 Segment Table **4**
 Permission Table **6**
 MMU Translation and Permission Checking **7**
  Segment 0 PTR Mapping **9**
 Control **9**
 Sharing and Enforcing Modularity **10**
  Two instances of an application use the same code **10**
  Process A needs to pass data to process B (similar to UNIX pipe) **10**
 Context Switching and Kernel PTR Setting **10**
 System Calls **11**
  Allocate **12**
  GiveAccess **12**
  Run **13**
Discussion **13**
 Memory Overhead **13**
 Multi-level Tables **14**
 TLB **14**
 Context Switching **15**
Conclusions **15**
References **15**

List of Figures

Figure 1 – Virtual address bit convention **4**
Figure 2 – Segment table structure **5**
Figure 3 – Virtual Address translation algorithm **6**
Figure 4 – Permission table structure **6**
Figure 5 – MMU hardware interactions **7**
Figure 6 – Permission required for CPU instructions **8**
Figure 7 – MMU permission checking algorithm **8**
Figure 8 – MMU complete operation **8**
Figure 9 – An example of memory sharing and isolation using permission tables **10**
Figure 10 – Memory Overhead vs. Number of Processes **14**

Introduction

A standard method of implementing enforced memory isolation for individual processes involves providing each process its own address space [2]. Typically, each address space is implemented with one single-level page table. Then, by assigning each process its own page table, each process is given its own address space. However, single-level page tables are very large and assigning one per process can very quickly lead to high memory overhead [3]. The memory management model presented in this paper, MPSA, solves this problem by allowing all processes to operate in one single shared address space. A key feature of MPSA is that the ability to enforce memory isolation is not lost. Enforced isolation is accomplished by using one segment table shared by all processes and individual permission tables for each process. Since simplicity and ease or integration were fundamental design goals, integrating MPSA into a system requires minimal changes to the ISA, the CPU, the operating system kernel, and surrounding hardware. This paper describes the specifications of MPSA as well as some possible implementation and integration techniques. In addition, a high level comparison of MPSA to the traditional memory model (multiple address spaces) is provided.

Design Overview

MPSA requires specialized hardware and software for full functionality. The necessary hardware will reside between the CPU and the physical memory or cache. This hardware will be referred to as the Memory Management Unit (MMU). The necessary software will be provided by the operating system kernel. This paper will discuss in detail the specifications and functionality of the MMU. It will not present a specific software implementation for the operating system kernel. However, in some cases, possible software implementations are given as examples to illustrate certain functionality and integration techniques.

The MMU handles all memory accesses (loads, stores, instruction fetches, etc.) such that it is completely transparent to the CPU. Each memory address (virtual address) issued by the CPU is translated by the MMU into a physical address that is used to access the physical memory. Since the translation is abstracted away from the CPU, no instructions need to be modified or added to accommodate the MMU design.

The MMU uses one segment table to perform memory address translation. The segment table contains virtual-to-physical address mappings for the entire virtual address space. Therefore, every virtual address can be translated using the segment table.

The segment table provides a single address space but is insufficient to implement enforced memory isolation. Therefore, each process is assigned a permission table that contains information about the process's access rights to every segment in the virtual address space. The permission table dictates which part of virtual memory the process can write to, read from, and execute code in.

On all memory operations, the MMU uses the segment table to perform the virtual-to-physical address translation. In addition, it checks the virtual address against the permission table assigned to the current process. If the current process has sufficient access rights to perform the requested operation, the MMU presents the physical address to the physical memory and the

operation is performed. However, if the current process has insufficient access rights, the MMU does not allow the operation to be performed and an exception is signaled. The operating system kernel is responsible for handling the exception.

The MMU and all tables are completely abstracted away from user processes. Only the operating system kernel is aware of the MMU and only it has access to the segment and permission tables. For this reason, the kernel is also responsible for maintaining these tables appropriately. In addition, the kernel must provide an interface that user processes can use to modify these tables indirectly when appropriate.

Detailed Design Description

Virtual Address Space

The MMU uses a 32-bit virtual address space that is divided into segments. Of the 32 bits, the high-order 16 bits denote the virtual segment number and the low-order 16 bits denote the offset into that segment. This provides a total of 65536 segments with a maximum size of 64 Kbytes per segment. The bit structure of the MPSA virtual address is shown in Figure 1.

The number of segments resulting from this convention is sufficient for most applications. It is reasonable to assume that a system will run hundreds of processes simultaneously [1]. Therefore using only the high-order 8 bits to denote the virtual segment, resulting in a maximum of 256 segments, is clearly insufficient if each process requires one or more segments. Therefore, a 16-bit segment number (the natural step up from 8 bits) is used.

32-bit Virtual Address	
31 16	15 0
Segment Number	Offset

Figure 1 – Virtual address bit convention

The address issued by the CPU on memory operations is the virtual address. Without an MMU, this value would be used to address the physical memory directly. However, in MPSA model, the virtual address issued by the CPU is translated and checked for permissions by the MMU before any physical memory is accessed.

Segment Table

We assume that the physical memory is large enough to accommodate the entire virtual address space. If this is not the case, some virtual segments must be stored on disk and swapped in and out of physical memory as needed by the operating system kernel [2]. However, since this operation is not essential to the memory model, it will not be discussed in any detail. Thus, for illustration purposes and simplicity, we assume that the physical and virtual memory spaces are equal.

The MMU translates 32-bit virtual addresses to 32-bit physical addresses which are used to address the physical memory. To perform this translation, a data structure called the segment table is used. The segment table contains one entry per virtual segment (65536 in total). Each entry contains three fields: physical address, segment length, and valid. The *physical address*

field denotes the 32-bit physical address where the segment begins. The 16-bit *segment length* field denotes the length of that segment. The *valid* field denotes whether or not the segment has been allocated. Figure 2 illustrates the segment table structure and its relation to the physical memory.

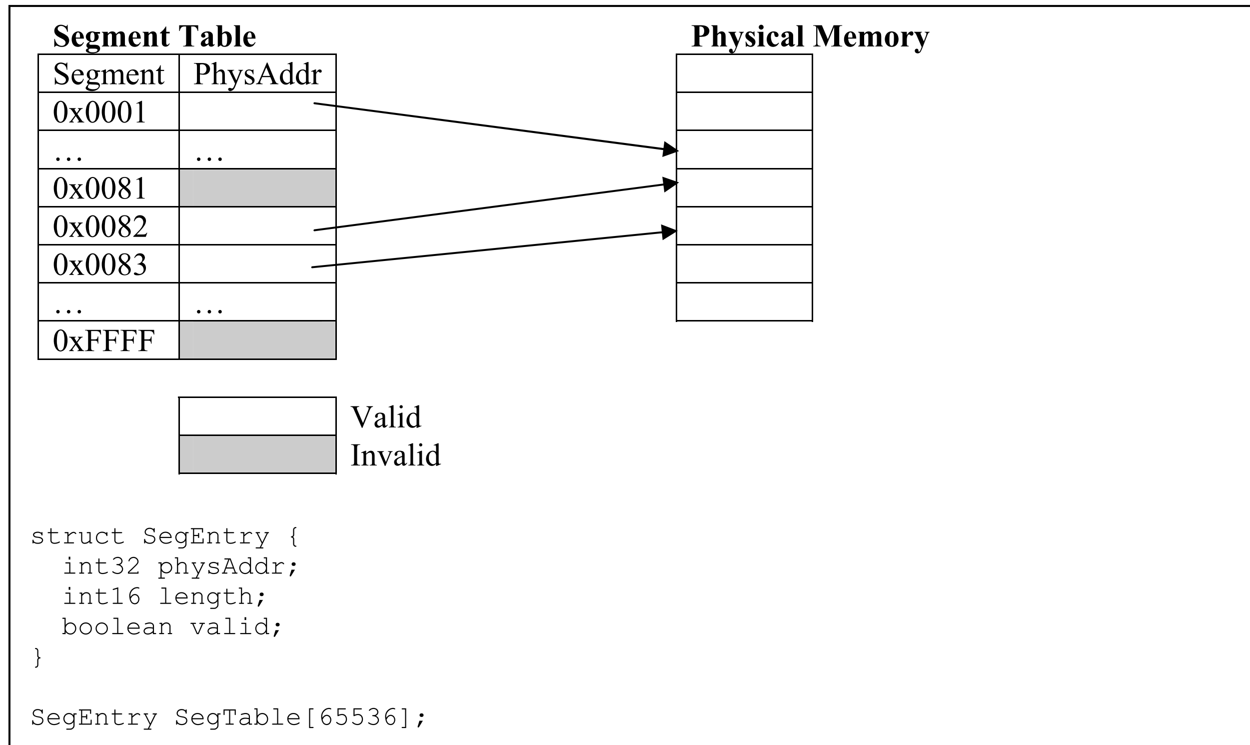
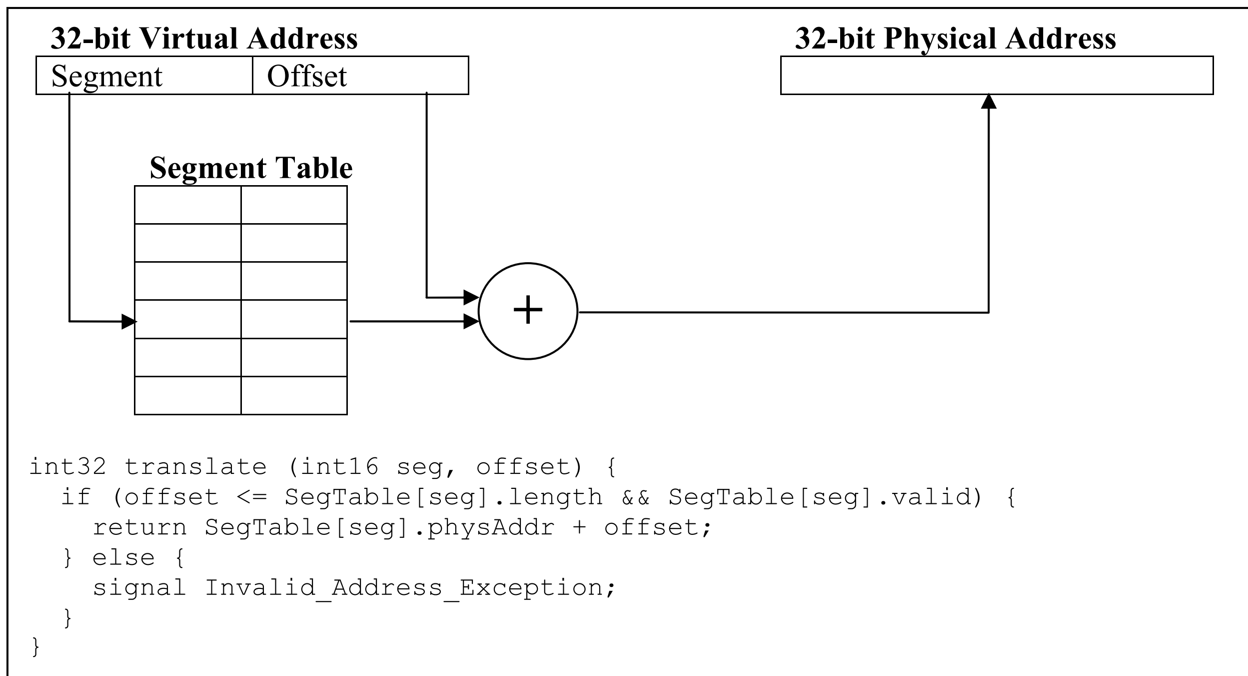


Figure 2 – Segment table structure

If the physical memory is smaller than the virtual address space, the segment table can be easily modified to contain disk addresses in addition to physical memory addresses.

Each segment table entry is 7 bytes in size. The entire segment table is 448 Kbytes. The table itself is stored in physical memory and accessed by virtual address. Therefore it will occupy 7 segments of the maximum length. By convention, the segment table will always reside in segments 1 through 7 in the virtual address space and at a static physical address that is known to the MMU.

Once given the segment number and offset by the CPU, the MMU uses the segment table in a translation algorithm to produce a physical address.



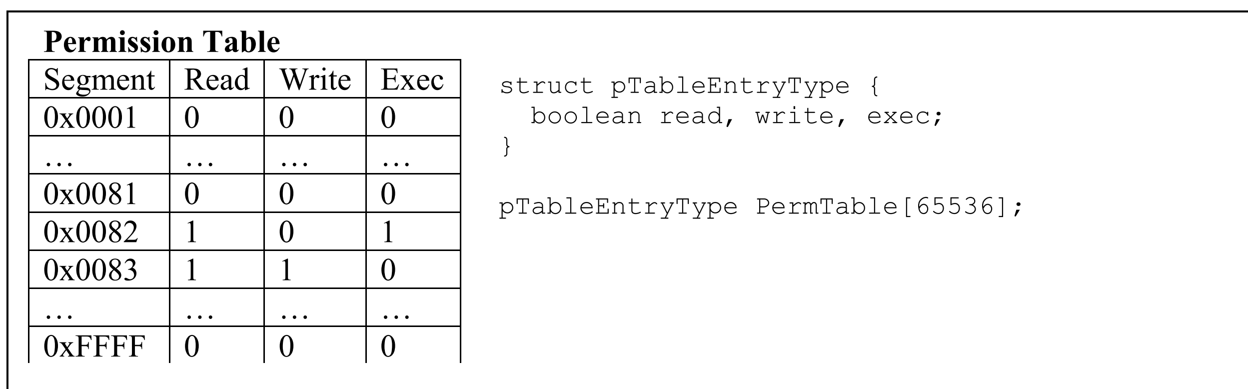

```
int32 translate (int16 seg, offset) {
  if (offset <= SegTable[seg].length && SegTable[seg].valid) {
    return SegTable[seg].physAddr + offset;
  } else {
    signal Invalid_Address_Exception;
  }
}
```

Figure 3 – Virtual address translation algorithm

The translation algorithm is given in Figure 3. As the code illustrates, the MMU also checks that the segment has been allocated and that the offset is within the length of that segment. If it is, the MMU performs the translation normally. If it is not, the operating system kernel is notified (through an exception) that the current process is trying to access an invalid memory location.

Permission Table

Each running process is assigned a data structure called a permission table that specifies the process's access rights in every virtual segment. Once the MMU is given an address, it checks the entry in the current process's permission table corresponding to the segment number of the address. If the entry shows that the process has sufficient access to perform the operation requested, the MMU allows the operation. If not, the MMU signals an exception. Figure 4 illustrates the structure of the permission table for one process.



Permission Table

Segment	Read	Write	Exec
0x0001	0	0	0
…	…	…	…
0x0081	0	0	0
0x0082	1	0	1
0x0083	1	1	0
…	…	…	…
0xFFFF	0	0	0

```
struct pTableEntryType {
  boolean read, write, exec;
}

pTableEntryType PermTable[65536];
```

Figure 4 – Permission table structure

The permission table is indexed by segment number and each entry corresponds to the process's permissions within that segment. The *read* field indicates whether or not the process is allowed to read from the segment. The *write* field indicates whether or not the process is allowed to write to the segment. The *exec* field indicates whether or not the process is allowed to execute code in the segment.

Each entry in the permission table is 3 bits in size. Each permission table is 24 Kbytes. All permission tables are stored together in physical memory in one segment or series of segments. By convention, these segments do not contain anything else. There is a small memory penalty for using this data structure since there is one permission table per running process. As the number of running processes increase, the amount of memory needed increases as well. However, 24 Kbytes per process is reasonable when considering the number of processes and memory size of current systems. For example, if there are 100 running processes, then a total ~2.4 Mbytes of memory overhead is needed to maintain all the permission tables. Though it should be noted that some amount of physical memory is needed to maintain such a data structure, this memory overhead is acceptable for typical applications.

MMU Translation and Permission Checking

The MMU resides in between the CPU and the primary memory. Instructions involving memory operations are passed to the MMU for address translation and permission checking. The process is completely transparent to the CPU. The CPU does not need to be changed in any way to accommodate the MMU design. Figure 5 shows the CPU, memory, and MMU interconnection hardware.

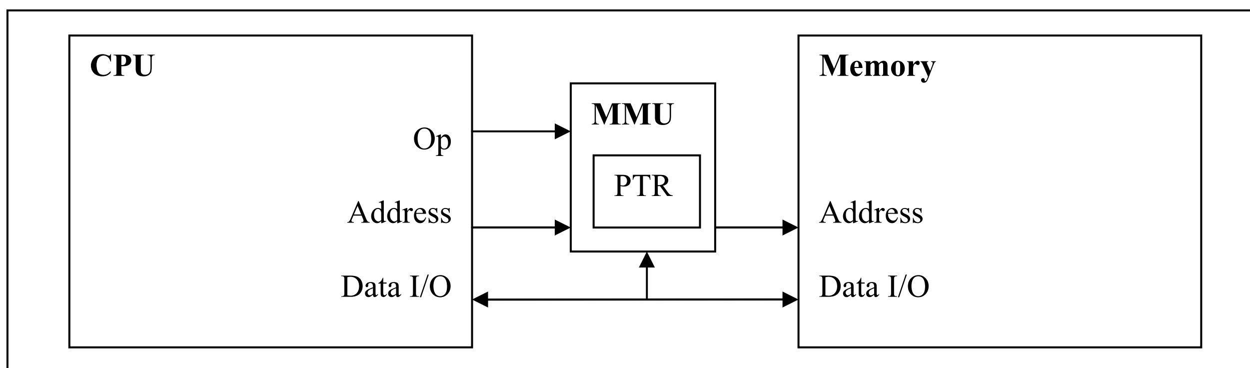


Figure 5 – MMU hardware interactions

There is a dedicated register inside the MMU that is a pointer to the current process's permission table. This register is named the Permission Table Register (PTR). The PTR contains the physical address of the permission table to be used by the MMU. This register can be accessed through segment 0. Writing to segment 0 (any offset) will result in a write to the PTR. This is the only virtual segment that does not map to a physical address. Only the kernel should have access to this segment and all processes permission tables should be set accordingly. Since the kernel has full access to each process's permission table, it can give any process access to the PTR. However, such actions by the kernel would violate the memory abstraction provided by MPSA and thus result in the loss of enforced isolation.

When the CPU requires access to memory, the MMU performs two tasks simultaneously. 1) It translates the virtual address requested by the CPU into a physical address as described above. 2) It checks the permission table located at the address stored in the PTR to ensure that the requested operation is allowed in that segment. Figure 6 illustrates the permissions necessary for all CPU memory operations.

	Permissions Required for Operation		
Operation	*Read*	*Write*	*Exec*
Inst. Fetch	X	X	1
Load	1	X	X
Store	X	1	X

1 = bit set in the permissions table
0 = bit not set in the permissions table
X = does not matter

Figure 6 – Permission required for CPU instructions

The MMU permission checking algorithm is given in Figure 7. The full MMU algorithm is given in Figure 8.

```
int32 PTR;
pTableType pTable = mem[PTR];

boolean validOp (opType op; int16 seg) {
  pTableEntryType pTableEntry = pTable[seg];
  if (op == Instruction_Fetch && pTableEntry.exec == 1) {return true; }
  elseif (op == Load && pTableEntry.read == 1) {return true; }
  elseif (op == Store && pTableEntry.write == 1) {return true; }
  else {return false; }
}
```

Figure 7 – MMU permission checking algorithm

```
int32 MMU (opType op; int16 seg, offset; int32 dataIn) {
int32 physAddr = translate(seg, offset);
  if (Invalid_Address_Exception signaled) {
    signal Invalid_Address_Exception;
  } else if (op == Store && seg == 0) {
    PTR = dataIn;
  } else {
    if (validOp(op, seg)) {
      return physAddr; //memory operation allowed to complete normally
    } else {
      signal Invalid_Permissions_Exception;
    }
  }
}
```

Figure 8 – MMU complete operation

If the address requested is valid and the current process has the appropriate permissions to perform the requested operation, the operation is allowed to proceed normally. If this is not the case, the MMU signals an appropriate exception and the memory operation does not complete. The operating system kernel is responsible for handling such exceptions.

Segment 0 PTR Mapping

As mentioned, a store operation to virtual segment 0 results in a write to the PTR. Access to segment 0 allows kernel code to read and write the PTR without any changes to the ISA. Alternatively, PTR access could have also been accomplished by adding a special CPU register [3] or instruction to the ISA and ensuring that only kernel code can use this register or instruction. However, mapping segment 0 to the PTR enables the ISA to remain unchanged and enables the already implemented permission checking mechanism to ensure only the kernel can change the PTR.

Access to the PTR through segment 0, however, does have a disadvantage. It results in an inconsistent address space. However, the inconsistency is only visible to the kernel since all user processes will not have access to segment 0. Thus, by changing the kernel slightly to deal with the small inconsistency, the problem is abstracted away from all user processes.

Control

The MMU can read from the segment table and current permission table. The MMU does not, however, have any control over the contents of these tables nor does it have the ability to write to the PTR. Only the kernel has this ability and thus the responsibility of maintaining these tables and the PTR.

The kernel, like any other process, is assigned its own permission table. However, the kernel table is unique in that it allows full access (read, write and execute) to every segment. Since the segment table and all permission tables are stored in memory, the kernel has full access to every entry in all of these tables. In addition, the kernel has full access to segment 0 allowing it to write to the PTR in the MMU.

The only permission table that allows full access to every segment is the kernel table. All other tables assigned to user processes have limited access. User process tables allow no access to segments 0 through 7 (the MMU PTR and the segment table) and all segments containing permission tables. Therefore, only the kernel process has the ability to read and modify the memory management mechanism that performs the translation and permission checking. User processes are restricted to provide a very strict enforced abstraction between the physical memory and user processes.

When kernel code is running on the CPU (when the CPU is in *kernel mode*), the code should have full access to every segment. Full access is accomplished by simply ensuring that the kernel's permissions table is used (and that the kernel's table has full access). However, there are several other ways that full access could have been accomplished. For example, a special kernel mode bit in the MMU would also give the same result (when the bit is set, the MMU gives full access to all segments). However, the use of a kernel permission table was chosen over

alternatives such as the special MMU bit since it reuses mechanisms already in place. The general purpose mechanisms used for permission checking is sufficient to implement the full access feature. Therefore, adding additional hardware to accomplish this task is wasteful.

Sharing and Enforced Modularity

In addition to not having access to the segment and permission tables, user processes have limited access to the rest of the virtual address space as well. Normally, user processes only have access to segments containing its own data. When a process is started, it is assigned a permission table by the kernel. The assigned table gives the process read and execute access to the segments containing the process's code. If the process requires more segments for a stack or a heap, it must request it (see the Allocate system call below). Once the additional segment has been requested, the kernel modifies the permission table to allow access to the process's data (stack and heap) as well as code. All other segments are marked as inaccessible.

When multiple processes are running, the kernel enforces memory isolation by ensuring that data belonging to each process is contained in segments that are accessible only to the owner process. Similarly, memory sharing is accomplished by granting multiple processes access to one or more segments. The specific type of access (read, write, or execute) depends on the nature of the relationship between the processes sharing the data. For illustration, consider the following two examples of data sharing:

1) Two instances of an application use the same code

The kernel will give both processes (instances) read and execute permissions for the segments containing the shared code. The read and execute fields of the appropriate entries in both permission tables (for each process) are set by the kernel. Since both processes have read and execute permissions to the same physical memory segment, they can run off of the same code in that physical segment. In addition, when combined with data isolation, processes can share code while maintaining entirely separate data segments. More details are presented in the Allocate system call section.

2) Process A needs to pass data to process B (similar to UNIX pipe)

To accommodate a *pipe* feature, the kernel will set aside a segment and give process A write access to that segment while giving process B read access. An example is given in Figure 9 to illustrate the permission table entries required to perform this *pipe* feature.

Process A's Permissions Table

Seg	R	W	E
F5	1	1	0
F6	0	0	0
F7	1	0	1
F8	0	0	0
F9	0	0	0
FA	0	1	0

Process B's Permissions Table

Seg	R	W	E	
F5	0	0	0	Process A's data
F6	1	0	1	Process B's code
F7	0	0	0	Process A's code
F8	1	1	0	Process B's data
F9	0	0	0	
FA	1	0	0	Process A & B's shared data

Figure 9 – An example of memory sharing and isolation using permission tables

Context Switching and Kernel PTR Setting

The kernel is responsible for keeping track of each permission table's address. When a context switch is required, the scheduler within the kernel chooses the next process to be run. Then the kernel loads the PTR of the MMU with the new process's permission table (in addition to everything else that normally occurs during a context switch). The MMU will use the new process's permission table for all future memory operations.

Before a context switch can begin, the system must switch from the running user process to the kernel scheduler process. The user processes do not have access to the MMU PTR (segment 0) and therefore cannot invoke the context switch as described above. To perform the initial switch from the user process to the kernel, a special hardware mechanism must be used. When a context switch is required, a hardware interrupt is signaled (as in typical context switches). Upon receiving the interrupt, the CPU will jump to the interrupt handler (scheduler) in the kernel. However, the MMU would not allow the kernel scheduler code to execute if the user process permission table is still in effect. Therefore, the interrupt must also cause the MMU to load the kernel permission table into the PTR. The address of the kernel permission table is static and predetermined. The kernel must ensure that its permissions table is located at that address for the MMU to function properly.

System Calls

In addition to context switches, there are cases when user processes need to perform operations that they have insufficient access to perform. For example, a running process may require more memory or a process may want to give other processes access to its data. Both of these examples require modification of permission tables, an operation user processes cannot perform directly in MPSA.

There are also cases when user processes need to run shared kernel code. These cases, however, are treated as simple code sharing as described above. The described method of code sharing, however, is insufficient when the code requires permissions beyond those of the current running process. Therefore, there exists a mechanism for a user process to request that a certain operation be performed by the kernel: a system call.

A system call requires three steps:
1) The user process provides the kernel with instructions
2) A context switch to the kernel is invoked and the kernel performs the desired operation
3) A context switch back to the user process is invoked

The three step process is given here in more detail: Information is passed between the user process and the kernel using the memory sharing mechanisms already discussed. A special segment in which the user process has write access to is allocated by the kernel. The user process writes data to that segment instructing the kernel to perform the desired operation. The user process then informs the kernel that it wishes to release control of the processor using standard multi-processing mechanisms (such as an interrupt). A context switch is then performed and the kernel gains control of the CPU. Once the context switch is complete, the kernel checks the shared data segment of the user process and performs the requested operation. The kernel then writes the completion status of the operation (informing the user process if the operation

completed as requested) and any other return values back to the shared data segment. A context switch is then performed and control is passed back to the user process. The user process can then check the shared data segment for completion status and other return values. In addition, when necessary, the kernel can pass return values back to the user process through a register.

Using the three steps just described, high level implementations of MMU-related system calls are provided below to illustrate some standard desired functionality.

System Call: int16 Allocate (int16 length; boolean read, write, exec)

If a user process requires more memory, it can use the Allocate system call. The user process informs the kernel of the amount of memory required through the *length* parameter. In this implementation, *length* is limited to one segment. In addition, the user process informs the kernel of the desired permissions for that segment through the *read*, *write*, and *exec* parameters.

The kernel only allocates segments that are marked invalid in the segment table. The index of the corresponding entry in the table is the segment number being allocated. The kernel sets that entry to valid, assigns a physical memory address, and sets the length to that requested. The kernel also sets the corresponding entry in the user process's permission table to the requested permissions for that segment. The segment number is then passed back to the user process in a register.

When a process is started it only has read and execute access to its code. If the process requires more memory for a heap or stack, it must call the Allocate system call. When the process needs to access the segment, it calculates the virtual address by adding the desired offset to the value in the register containing the segment number. Generally, all data memory is accessed using the data segment register as a dynamic base to which static offsets can be added. Since registers are restored on context switches, multiple processes running the same code can use different data memory segments.

System Call: boolean GiveAccess (int16 seg; pidType pid; boolean read, write, exec)

If a user process wishes to give other processes access to its data, it can use the GiveAccess system call. GiveAccess informs the kernel that the user process wishes to give another process (*pid*) specific permissions (*read, write, and exec*) for a specific segment (*seg*). The GiveAcccess system call is necessary since processes do not have any direct access the permission tables.

To service the GiveAccess system call, the kernel first checks if the process making the request has sufficient permissions to perform the task. The process making the request must have at least the permissions it is requesting to assign. For example, if the process wishes to give another process write access, it must have write access itself. If the operation is allowed, the kernel assigns process *pid* the requested permissions by modifying *pid*'s permission table. An indicator of whether or not the operation was successfully is then sent back to the user process that made the system call.

System Call: boolean Run (appType app)

If a user process wishes to start another process, it can use the Run system call. The Run system call is the same mechanism the kernel uses to start new processes.

The kernel first creates a new permission table for the new process. The new permission table provides the new process with read and execute access to its code segment(s) only. If the process code already exists in a valid segment (it is being used by some other instance for example), the new process will share that code. If the code is not in a valid segment, the kernel will find a valid segment and move the code into that segment.

Once the kernel has given the new process a permission table, the process is added to the running-processes list (used by the scheduler) and the code can be executed when a context switch is performed to that process.

Discussion

Memory Overhead

To evaluate the memory overhead of MPSA, it will be compared to that of a hypothetical multiple address space model. We base the hypothetical model on the traditional scheme that uses single-level page tables [3]. The use of multi-level page tables will be briefly discussed later in this section.

The hypothetical model does not represent any memory system used today but is loosely based on the simple traditional multiple address space model on which most systems used are currently based [1]. Thus, a comparison of MPSA and the hypothetical model will give a relatively good indicator of MPSA's strengths.

The hypothetical model uses a 32-bit address space (same as MPSA), single-level page tables, a fixed page size of 8 Kbytes (same as the maximum segment size in MPSA), and the high-order 16 bits of the virtual address to denote the page number (same as MPSA). Assuming that only the physical page location (16 bits) is stored in each page table entry (this is a very conservative assumption since bits such as *valid* and *resident* are necessary in most systems), the hypothetical model will require 128 Kbytes (2^{16}x16 bits) for each page table. On the other hand, as shown previously, MPSA requires 448 Kbytes for the segment table and 24 Kbytes for each process permission table. Figure 10 shows how memory overhead scales as the number of processes is increased for both the hypothetical model (HM) and MPSA.

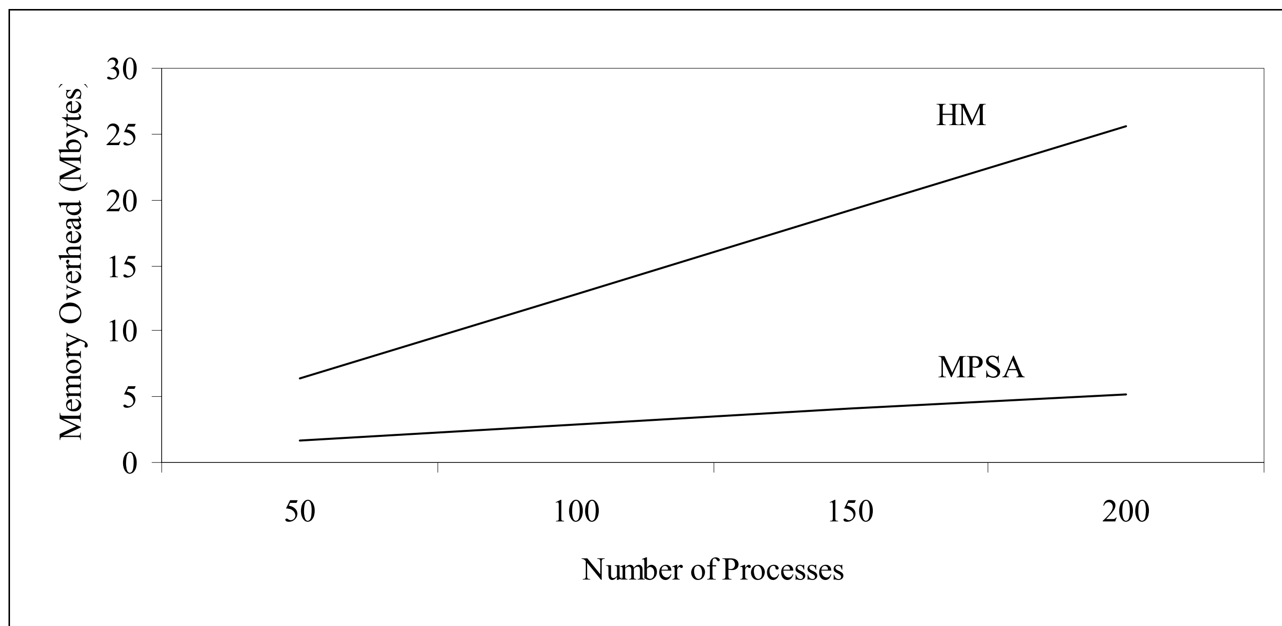


Figure 10 – Memory Overhead vs. Number of Processes

As Figure 10 illustrates, the total memory overhead of MPSA is significantly less than that of the more traditional multiple single-level page table model [3] for a large number of processes. This result is expected since the fundamental goal of MPSA was to decrease the memory overhead of traditional models without the loss of functionality.

Multi-level Tables

Most multiple address space models used in current systems use multi-level page tables to reduce memory overhead [1]. Multi-level page tables are effective since most of the address space is not normally needed by any one process. This multi-level technique, however, is not unique to multiple address space models. It can be applied to MPSA as well to achieve even more memory overhead savings. Specifically, the MPSA permission table can be made into a multi-level structure with very little change to the overall memory model. The specifics for multi-level permission tables will not be provided in this paper however the concept is identical to that of standard multi-level page tables. Therefore, the gain from using multi-level permission tables will be similar to that from multi-level page tables. Single-level tables are used in MPSA since they are simple and, in a 32-bit address space, are not too large. However, the possible use of multi-level tables is being mentioned here to note that even more memory savings can be achieved by adding some complexity.

TLB

This paper has ignored many hardware components present in current systems. The simple CPU model used in this paper does not have a TLB for example. All current CPUs have many performance enhancing features such a TLB and it is important to note that MPSA is easily integrate-able into such systems as well. The key feature that makes MPSA easy to integrate is its ability to present an interface that very closely resembles current memory models. For example, the MPSA segment table serves the same role (translation) and has essentially the same

format as the page table. Thus, the TLB can interact with the MPSA segment table in the same way as it does, in current systems, with the page table.

Context switching

Since a fundamental assumption behind the design of MPSA is that systems run many concurrent processes, it is important to consider the performance effects of MPSA on context switching. In multiple address space models, a context switch is performed by changing the current page table. In MPSA, it is performed by changing the current permission table. Both models require only one change to one register. However, context switches in current systems often require much more work. Most of the extra work is required because a fundamental assumption of this paper is not usually true in current systems. We assumed that the physical memory is always large enough to accommodate the entire virtual address space. However, most current systems have virtual address spaces much larger than the physical memory. Thus, context switches often require that the new page table be pulled into physical memory from the hard drive before the next process is permitted to run. Similarly, if MPSA were used on a current system, often the new permission table will need to be pulled into physical memory from the hard drive before it can be used. However, since permission tables are significantly smaller than page tables (24 Kbytes vs. 128 Kbytes), context switches will be faster in MPSA than in multiple address space models even under conditions when parts of virtual memory are stored on disk. This result was not expected since MPSA was not designed with context switching performance in mind. This favorable result is simply a side effect of MPSA's simplicity and overall memory savings.

Conclusions

The MPSA memory management model presented achieves both memory sharing and enforced isolation using a single address space. MPSA achieves the same functionality as the traditional multiple address space model but saves substantially on memory overhead. Memory savings are achieved with virtually no gain in hardware or software complexity. The MPSA design has demonstrated that memory overhead savings gained through the use of a single address space can be achieved with minimal changes to the ISA, the CPU, the operating system kernel, and surrounding hardware. Enforced isolation in a single address space can be achieved through a simple and easily integrate-able design.

References

[1] Hennessy, J. L. and Patterson, D. A. [1996]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, California, 439-461.

[2] Saltzer, J. H. and Kaashoek, M. F. [2002]. *Topics in the Engineering of Computer Systems*, MIT 6.033 class notes, draft release 1.14, Cambridge, Massachusetts, 2.21-2.32

[3] Ward, S. A. and Halstead, R. H. Jr. [1990]. *Computation Structures*, MIT Press, Cambridge, Massachusetts, 486-496, 544-548