Alcatraz: Better Living Through Segments

Michael Salib

March 21, 2002

Abstract

Alcatraz is a segmented memory system for the Amazing OS running on Beta microprocessors. Alcatraz includes a hardware Memory Mapping Unit in addition to the OS and CPU changes needed to support it. Segmented memory systems use virtual addresses that are verified and translated by an MMU before being passed to physical memory. Alcatraz uses a segment descriptor table containing the length and starting address of each segment as well as separate read, write, and execute permission bitmaps. Each permission table maps the currently executing segment and the segment of the requested address to a boolean value indicating whether the operation is permitted. These tables are created and maintained by the OS and are updated during each context switch. Alcatraz guarantees hard modularity while permitting sharing.

Contents

List of Figures

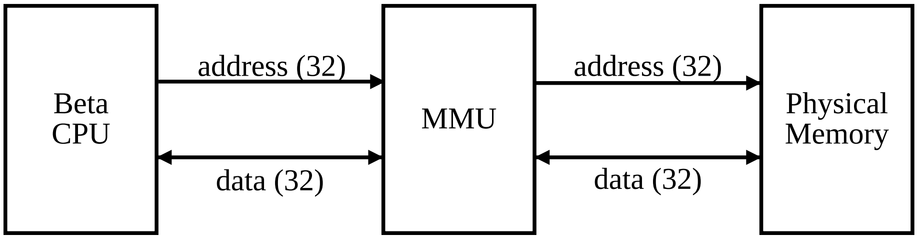

Figure 1: System Block Diagram

1 Introduction

This paper describes the design of a segmented memory system called Alcatraz. It includes a reference design for the MMU hardware and a description of the changes needed to support segmented memory in the operating system (OS) and CPU. Alcatraz's design was constrained by several requirements described below. We'll review these requirements before discussing Alcatraz's design. Afterward, we'll examine the rational for the design decisions Alcatraz embodies. Finally, we'll analyze Alcatraz and compare it to several alternative segmented memory systems.

1.1 Motivation

Despite excellent engineering, our computer systems currently exhibit serious reliability and stability problems. Most of these problems can be traced back to errors in customer software. Broken customer applications are able to corrupt system data as well as data belonging to other applications, leading to system crashes in the best case, and incorrect results in the worst case. As a result, Amazing systems appear unreliable to the customer. While we are not responsible for these problems, we are blamed for them.

Individual applications can corrupt the system because it does not enforce any kind of memory isolation. Any piece of executing code can access any area of memory it likes, regardless of who owns that memory. The solution to our customers' reliability problems and our own image problem is to modify our systems so that the hardware checks all memory accesses to ensure hostile applications cannot corrupt memory that does not belong to them. This will ensure that errors in one program will not propagate to other programs. In other words, we want to limit the "propagation of effects" in order to control complexity [3].

Traditionally, we have assumed that application programs will be well behaved and bug free, that they will behave like harmless, obedient children. However, experience has shown that user programs are often buggy, occasionally malicious, and much more like violent psychotics than obedient children. Accordingly, we need to change our mental model of the system from a nursery school for children to a prison for hardened criminals. By necessity, a prison has thick walls, bars, and gates that restrict access.

One common method to build these barriers is to use a Memory Management Unit (MMU). This device sits between the CPU and system memory and acts as an intermediary between the two. The MMU verifies that all memory accesses are restricted to memory that the currently running program owns. There are several memory management policies that

MMUs can support. We will focus exclusively on segmented memory systems.

In a segmented memory system, all system memory is divided into variable length segments. Applications only reference virtual addresses that contain a segment number and an offset. The MMU translates virtual addresses into physical addresses by consulting a segment descriptor table that indicates the location of the segment in physical memory as well as the segment length. The MMU stops any memory access that exceeds the length of the given segment. During this translation process, the MMU has the opportunity to check whether a given memory access is permitted.

1.2 Requirements

Hard Modularity User programs must be able to isolate their code and data from other programs. User programs must be prevented from reading or writing memory owned by other programs unless explicitly permitted to do so.

Sharing User programs must be able to selectively grant read, write or execute permissions for a specific segment to other programs.

Multiple Instance Support Several instances of the same program should be able to run concurrently while sharing a single copy of the program image. Instances must be isolated from one another.

Kernel Isolation The OS kernel must be implemented with segments. It must be able to read from and write to segments belonging to any program. The MMU should prevent all user programs from reading or writing kernel segments.

System Calls The MMU must support system calls.

Scheduling The MMU must support pre-emptive scheduling.

Process Creation User programs must be able to start new programs.

Generality Special purpose mechanisms to support the kernel are frowned upon. The MMU's design should not be closely tied to any particular OS.

Transparency The new segmented memory system should have as small an impact on the system API as possible. It should be transparent to programs that do not violate isolation rules.

2 Design Description

Alcatraz consists of a hardware MMU along with OS and CPU support. The MMU acts as an intermediary between the CPU and physical memory, intercepting, checking and translating all memory accesses (as shown in Figure 1). The MMU consults a series of tables located in main memory that describe the segments and their access permissions. These tables consist of a segment descriptor table followed by three segment permission tables (one for read,

```
def check(address, data, operation):
    if not(permission_tables[pc, address, operation]):
        raise SegmentPermissionFault

    if offset(address) > segment_desc_table[segment(address)].length:
        raise SegmentRangeFault

    new_address = (segment_desc_table[segment(address)].start +
                   offset(address))
    perform_memory_access(new_address, data, operation)
```

Figure 2: Simple pseudocode describing how the MMU hardware checks a single address.

write, and execute permissions). Figure 2 shows some simple pseudocode that describes the MMU's operation in relation to these tables.

The segment descriptor and permission tables are crucial to maintaining the isolation guarantees that Alcatraz provides. These tables are created and maintained by the operating system. The OS modifies the permission table entries just before scheduling a new process to run. In addition, the OS modifies the tables when starting new programs, establishing a shared segment between processes and whenever it allocates or deallocates memory on behalf of a process.

We'll describe Alcatraz in four stages. First, we'll explore the segment tables that are the cornerstone of Alcatraz's operation. Then we'll examine the MMU followed by a review of the CPU changes needed to support Alcatraz. Finally, we'll consider how the OS interacts with Alcatraz.

2.1 Segment Tables and Addressing

The core idea behind Alcatraz is address translation: code running on the CPU uses virtual addresses consisting of a segment and offset (as shown in Figure 3). The MMU hardware performs this address translation. Alcatraz uses 8-bit segment identifiers leaving 24-bits for the offset. This means that, at most, Alcatraz can support 256 segments with each segment having a maximum size of 16,777,216 bytes.

For each memory access, the MMU performs the following functions:

1. Verify that the access is within the segment's bounds,
2. verify that the segment currently executing is permitted to perform the requested memory operation, and
3. actually perform the address translation.

In order to perform step 1, the MMU references a segment descriptor table. This table is a linear array of 256 segment descriptor records indexed by the segment identifier. Each

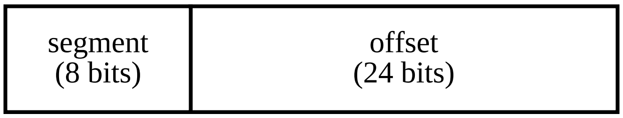

Figure 3: A virtual address.

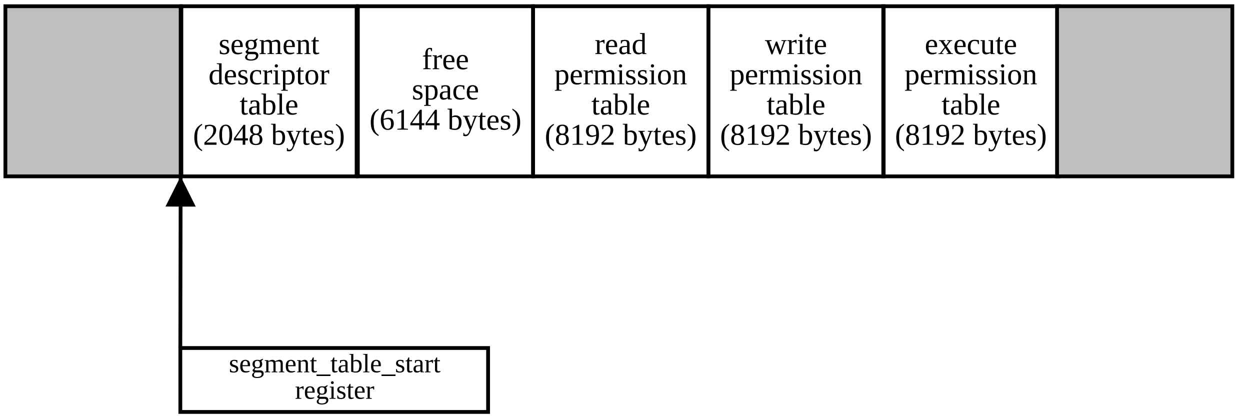
segment descriptor table (2048 bytes)
free space (6144 bytes)
read permission table (8192 bytes)
write permission table (8192 bytes)
execute permission table (8192 bytes)
segment_table_start register

Figure 4: This diagram indicates how the segment tables are located in physical memory. It shows the segment descriptor table starting at the address indicated by the segment_table_start register followed by come currently unused free space. Immediately following the free space are read, write, and execute permission tables.

segment descriptor is 8 bytes long and consists of two machine words: the segment's length and the address at which the segment starts in physical memory.

Step 2 relies on the segment permission tables. There is a separate permission table indicating read, write, and execute permissions. Each table is a 256 by 256 bitmap. The MMU indexes these bitmaps using a tuple made by combining the segment of the currently executing instruction (taken from the PC) together with the segment of the requested address. In effect, each bitmap serves as a function that maps a pair of segment identifiers to a single bit value indicating whether that memory operation is permitted.

As shown in Figure 4, the tables are arranged sequentially in physical memory starting at the address located in the segment_table_start register. This register will be described later in Section 2.3. Note that between the segment descriptor table and the segment permission tables lies an unused region of free space. This area is intended to be used as bookkeeping space by the OS. Its presence simplifies the MMU hardware by reducing the number of adders required.

2.2 Memory Mapping Unit

As we have already described, the MMU hardware is responsible for checking and translating each memory operation. The MMU uses the segment permission tables to verify that the currently executing segment is permitted to perform the requested operation. It makes use of the segment descriptor table to verify that the requested address is within the range specified for that segment. Finally, the MMU uses the segment descriptor table to translate

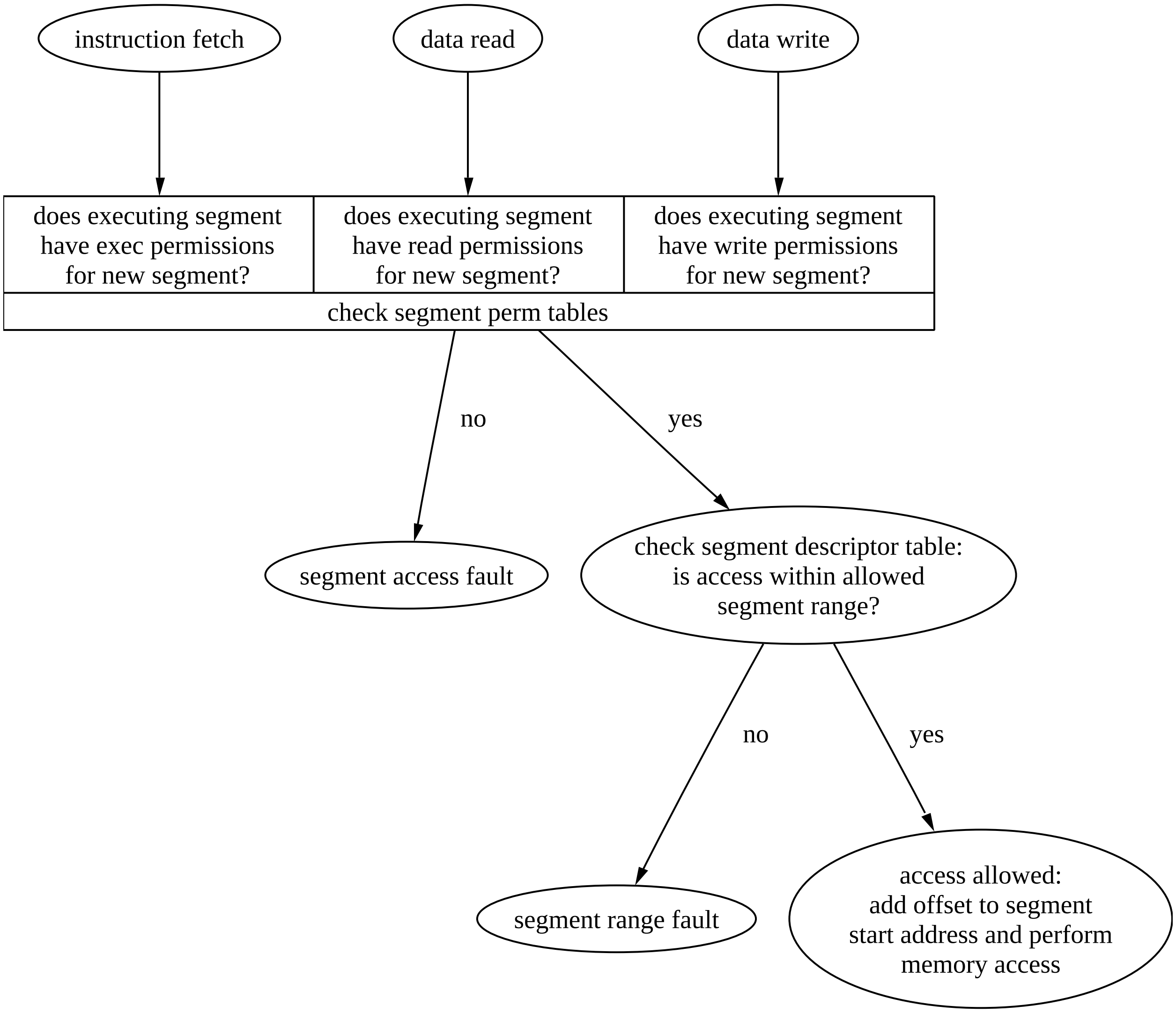
instruction fetch
data read
data write
does executing segment have exec permissions for new segment?
does executing segment have read permissions for new segment?
does executing segment have write permissions for new segment?
check segment perm tables
no
yes
segment access fault
check segment descriptor table: is access within allowed segment range?
no
yes
segment range fault
access allowed: add offset to segment start address and perform memory access

Figure 5: MMU Hardware Algorithm

the virtual address. After reviewing the CPU–MMU interface, we'll examine each of these steps in some detail. Figure 5 depicts the MMU's operation. In addition, Figure 6 contains a more detailed version of the pseudocode than was provided in Figure 2.

The CPU interfaces with the MMU using 32-bit data, address and PC busses as well as single bit read/write, start and done signals. In addition, the MMU provides the CPU with two status signals indicating segment permission faults and segment range faults. The PC bus informs the MMU what instruction the CPU is currently executing. The read/write signal indicates whether the MMU should perform a read or write. The CPU uses the start signal to initiate new memory requests for the MMU. The MMU uses the done signal to indicate it has completed the requested operation. These two signals are needed to support asynchronous operation between the CPU and MMU. Given the tremendous speed disparity between physical memory and microprocessors, an asynchronous interface seemed most appropriate. The introduction of an MMU greatly enhances this disparity since each instruction will force

```
def segment(virtual_address):
    return virtual_address >> 24

def offset(virtual_address):
    return virtual_address & 0x0FFFFFFF


# operation is either 1, 2, 3 (0 is reserved for seg desc tables)

def check(address, data, operation):
    # check for access permissions
    op_offset = operation << 14
    perm_bits_addr = (segment_table_start +
                      (segment(pc) * 32) +
                      ((segment(address) >> 5) << 2) +
                      op_offset)
    perm_bits_mask = 0x1 << (segment(address) 0x1F)
    if not(physical_memory[perm_bits_addr] & perm_bits_mask):
        raise SegmentPermissionFault

    # check for range violations
    seg_desc_addr = segment_table_start + (segment(address) * 8)
    segment_length = physical_memory[seg_desc_addr]
    if offset(address) > segment_length:
        raise SegmentRangeFault

    # translate address and do the operation
    segment_start = physical_memory[seg_desc_addr + 4]
    if operation == read or operation == execute:
        return physical_memory[segment_start + offset(address)]
    else:
        physical_memory[segment_start + offset(address)] = data
```

Figure 6: Detailed pseudocode describing how the MMU hardware manipulates a single address.

the MMU to perform a minimum of four memory accesses (these performance issues will be described later in Section 4.1).

In addition to the interface described above, we also require that the CPU indicate whether it will treat the results of a read operation as an instruction or as data. The Beta generates instruction read requests when branching, jumping, and fetching the next instruction. All other read requests that the Beta generates are data requests. The Alcatraz MMU requires that the CPU indicate the type of each memory access, but since the original Beta specified separate instruction and data memories [1], it easily satisfies this requirement.

Now that we have scrutinized the CPU–MMU interface, we can discuss the MMU's operation. In order to check the segment permission tables, the MMU must construct the address of the relevant permission bit. In order to determine the address of the correct table, the MMU starts with the base segment table address located in the segment_table_start register. To this it adds an offset corresponding to the type of operation (read, write, or execute). The address of the row we're seeking can be found by adding the segment of the currently executing instruction multiplied by 32 since each row is 32 bytes long. Finally, the MMU adds the three most significant bits of the segment being requested. The result of all these machinations is the address of a 4-byte word that contains the relevant permission bit. In order to extract the single permission bit we care about, the MMU indexes the 32-bit word with the lowest 5 bits of the requested address's segment. If that bit is high, the MMU moves on to check for range violations. If that bit is low, the MMU indicates a segment permission fault to the CPU and waits for the next memory operation.

Following a successful permissions check, the MMU checks if the requested address's offset exceeds the length of the given segment. To do so it first calculates the address of the relevant segment descriptor. As before it starts with the value located in the segment_table_start register and adds an offset. The offset corresponds to the segment of the requested address multiplied by 8 since segment descriptors are 8 bytes in length. The resulting address is the start of the desired segment descriptor; the MMU loads the value at that address to discover the length of the segment. If the resulting length is less than the offset of the requested address, the MMU informs the CPU of a segment range fault and waits for the next memory request. Otherwise, the MMU continues processing the current request.

The last stage of processing a memory request corresponds to address translation. The MMU must add the requested address's offset to the segment's starting address. Starting addresses are loaded from the segment descriptor table; the MMU can thus use the address calculated for the segment length after adding 4 in order to discover the starting address of that segment. Once the MMU has translated the supplied virtual address into a physical address, it can perform the requested memory operation and pass the results back to the CPU.

2.3 CPU Support

The Beta CPU underwent several changes in order to support Alcatraz. These changes include the removal of the supervisor bit as well as the addition of a new register and two new instructions.

The new register is called the segment_table_start register and indicates the address in

Usage: STSS(Ra)

Opcode:

111001	Ra	*unused*

Operation: PC ← PC + 4
Reg[segment_table_start] ← Reg[Ra]

Set the contents of the segment_table_start register to be equal to the contents of register Ra. The segment_table_start register has a value of zero at system startup and cannot be changed once set to a nonzero value.

Usage: ENINT()

Opcode:

111010	*unused*

Operation: PC ← PC + 4
InterruptDisable ← 0

Enable interrupts. This instruction has no effect unless the interrupt disable bit has been set.

Figure 7: Instruction summary for set_segment_table_start and enable_interrupts.

physical memory where the segment tables begin. When this address is zero, the MMU is disabled and passes memory requests between the CPU and physical memory with no intervention. On startup, the CPU initializes the segment_table_start register to a value of zero. However, once this register is set to a nonzero value, it cannot be written to. It is effectively a write-once register.

The first new instruction is called SSTS (set_segment_table_start) and is used to load the segment_table_start register with a value from another register. Figure 7 show the instruction synopsis for SSTS. It is important to realize that segment_table_start is not a regular register; it cannot be read from and it can only be written to once by using the SSTS instruction.

The second major change to the Beta CPU is the elimination of the supervisor bit; it has been replaced by an interrupt disable bit. Jump instructions are free to switch control to any address, subject to the isolation constraints imposed by the MMU. Exceptions and traps function just as they did before; the only difference is they don't set the supervisor bit. Instead, exceptions and traps set the interrupt disable bit. When set, interrupts are disabled. We've added a new processor instruction, ENINT that enables interrupts again. ENINT is described in Figure 7.

2.4 Operating System Support

We will next examine how the OS interacts with the MMU at a few of the most critical interfaces such as bootstrapping, system calls, and scheduling. We will also sketch how an OS can implement critical services such as process creation, program loading and memory allocation. Finally, we will describe how Alcatraz enables the OS to efficiently implement shared memory.

As mentioned in Section 2.3, on reset and power up, the MMU is disabled. During the boot process, the OS creates an initial set of segment tables in kernel memory. The initial segment descriptor table includes descriptors for all the segments the kernel expects to use. The initial permission tables have all access disabled except for columns corresponding to access from kernel segments. This ensures that the kernel can access all segments.

System calls work just as they did before. Applications execute a specific undefined instruction after placing the system call arguments on the stack. The CPU traps the resulting exception and transfers control to the OS exception handler [2]. Along the way, the CPU sets the Exception Pointer (XP) register with the value $PC + 4$ [1].

Like the mechanics of system calls, OS scheduling is largely unchanged. Just as in the original Beta, the OS scheduler is called as a result of either a timer interrupt or an application call to yield(). And just as before, the OS saves the system state (i.e., the registers) into a process structure associated with the previously running process [4]. Alcatraz introduces one new wrinkle into the scheduling protocol. Before the OS can transfer control to the next process scheduled to run, it must reset the segment permission tables. In particular, it must replace them with the initial segment permission tables that only contain entries for the kernel. After that, the OS should write out permission bits appropriate for the new process based on state stored in the corresponding process structure. This entire process is shown in Figure 8.

It is important to understand that Alcatraz treats processes and segments as completely separate entities. In particular, the MMU knows nothing about processes: it has no idea what process is currently running, how processes are created, or even what a process is. The MMU deals only with segments. A process is an abstraction created by the OS. The OS associates a set of segments with each process. The OS changes the set of segments associated with a particular process in response to memory allocation and deallocation, shared memory requests, and kernel/user space data transfers (such as pipes, sockets, and zero copy IO).

For the purposes of discussion, we will assume the presence of a unix-like operating system. Such an OS will keep a proc structure for every process running on the system. It will also maintain a list of these structures which we will call the process list. proc structures live in kernel space and are used by the kernel to store information about individual processes. For example, when a process's time slice is complete, the first thing the kernel will do is to save the state of the registers and program counter for that process into it's proc structure. Later on, just before scheduling that process to run again, the kernel will restore the system state from the process's proc structure [4].

A unix-like OS might implement process creation in the following manner. Given an executable file to run, the kernel would allocate a new segment. This new segment would be equal in size to the executable. The kernel would then copy the contents of the executable file into the newly allocated segment. After creating a stack segment for the new process, the kernel would create a proc structure containing all the relevant information about it. At that point, the kernel schedules the new process for execution by adding it's proc structure to the process list.

The scheduler will eventually select it for execution. There is one important optimization that will prove vital to satisfying the multiple instance requirement: reusing executable text segments. The kernel keeps a table of executable file to segment mappings. When starting a new process, the kernel checks to see if the program file being started has already been

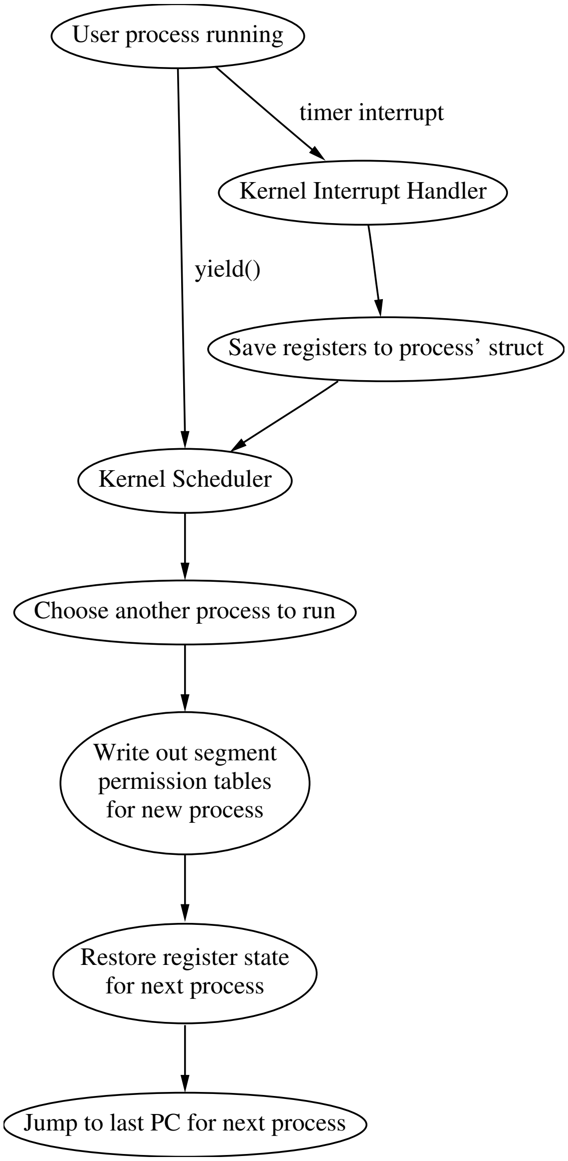
User process running
timer interrupt
Kernel Interrupt Handler
yield()
Save registers to process' struct
Kernel Scheduler
Choose another process to run
Write out segment permission tables for new process
Restore register state for next process
Jump to last PC for next process

Figure 8: Scheduling Protocol

```
def create_process(path):
    if not(program_image_map.has_key(path)):
        # we need to load the program image first...
        program_image = open(path)
        program_seg = get_new_segment(program_image.size)
        if not(program_seg):
            # something went wrong. either:
            #  there wasn't enough memory for the program image,
            #  there aren't any segment IDs left, or
            #  memory is too fragmented to allocate a contiguous block that large.
            raise AllocationError
        copy_file_to_segment(program_image, program_seg)
        program_image.close()
    else:
        # we already have this program loaded...
        program_seg = program_image_map[path]
        stack_seg = get_new_segment(DEFAULT_STACK_SIZE)
        if not(stack_seg):
            # something went wrong...
            raise AllocationError
        new_process = process(path = path,
                              program_seg = program_seg,
                              stack_seg = stack_seg)
        process_table.append(new_process)
        return new_process.pid
```

Figure 9: Pseudocode for the create_process system call

loaded into memory. If it has, the kernel simply reuses the existing executable text segment. This entire procedure is outlined in Figure 9, which shows pseudocode for a hypothetical create_process system call. For security reasons, shared executable segments should not be writable by default. If a shared segment is writable, a malicious instance could modify the executable image in memory and thereby gain access to other instances' data.

Memory allocation is handled in a straightforward manner. The application (or more likely the standard C library) issues a system call requesting a certain amount of memory. The kernel keeps a free list and a used list of segments. In response to a memory request, the kernel picks the first available segment from it's free segment list. It then determines whether it has a free region of memory with the requested size. If it does, the kernel fills the appropriate entry in the segment descriptor tables and modifies the process's proc structure. The proc structure modification involves adding the new segment to a list of segments that process is permitted to access. When the kernel next schedules that process to run, it will rewrite the segment permission tables based on the permission lists stored in the process's proc structure. Should any of these operations fail, the kernel returns an error code signalling allocation failure to the application.

Shared memory and IPC are implemented in a similar manner. Upon receiving an application request to access another process's memory, the kernel simply adds the appropriate segment to the requesting application's proc structure permission list. Of course, the kernel only does so if the other application indicated that it wanted that segment to be shared with the requesting process. It is important to realize that applications ask the kernel to share segments with another *process*, not segment. The OS is responsible for translating the concept of process level isolation which applications discuss down to the level of segment isolation which the MMU understands. Conversely, the OS is responsible for removing shared segment permissions when the segment is freed.

3 Design Rational

Now that we understand how Alcatraz works, we can evaluate it according to the criteria we specified in Section 1.2. Our goal here is to explain how different aspects of Alcatraz's design interact to satisfy the different requirements.

Hard Modularity Assuming that the segment permission tables have been correctly setup by the OS and that the OS has placed the segment tables in memory inaccessible to ordinary processes, Alcatraz guarantees that no process can access memory owned by another process. The segment permission tables are the key to making this guarantee.

Sharing Since Alcatraz maintains three different segment permission tables (one for read, write and execute permissions), it can easily ensure that only authorized executing segments can access a shared data segment. The OS translates the notion of process permission down to segment permission by rewriting the permission tables before each process runs.

Multiple Instance Support Since Alcatraz reuses loaded program segments as described in Section 2.4 and Figure 2, multiple instances of a single program will share the same executable segment. The isolation requirement is also satisfied since all instance state is stored either in the instance's proc structure (in kernel memory) or in the instance's stack segment. During context switches, the kernel rewrites the segment permission tables, taking care to ensure that any instance of a program is only given access permissions to its own data segments and read/exec permissions for the shared executable segment.

Kernel Isolation As long as the kernel sets up the segment permission tables so that all user segments are forbidden access to kernel segments, Alcatraz can guarantee that user space programs will be unable to touch the kernel. As long as the kernel writes permissions so that it can read and write any segment in the system, Alcatraz guarantees that the kernel can access all system memory. Since the kernel is the first program to run on the system and the only program that has access to the segment permission tables, it can easily meet these requirements.

System Calls Because Alcatraz does not change the standard exception handling components of the Beta CPU, system calls work as they did before. Applications simply issue an undefined instruction and the CPU transfers control to the kernel during the resulting exception.

Scheduling Scheduling also works as before. Alcatraz's isolation guarantees rely on OS permission table processing before each context switch.

Process Creation Alcatraz allows user programs to start new applications by asking the kernel to do so on their behalf. Since Alcatraz treats the kernel as a trusted intermediary, starting a new program does not violate modularity constraints.

Generality Alcatraz does not have any special support for the kernel beyond what the original Beta provided. In fact, Alcatraz makes the system more general by mandating the removal of the supervisor bit. Although Alcatraz was designed with a unix-like OS in mind, it can accommodate microkernel and exokernel designs since the kernel can (in theory) grant any executing segment permission to modify the segment tables.

Transparency User programs do not need any special support in order to work with Alcatraz. In fact, as long as they interface properly with the new kernel, they shouldn't require any modifications at all.

4 Analysis

We will begin by analyzing Alcatraz's performance. Afterward, we will introduce some alternative designs and compare them with Alcatraz.

4.1 Performance Analysis

There are several different performance metrics which can be used to evaluate a given design. We'll start by discussing how some of Alcatraz's core design decisions impact performance.

Later, we'll examine latency, memory space consumed, and how memory requirements scale as the segment size grows. Finally, we'll briefly mention how naive implementations of performance enhancements can violate Alcatraz's isolation guarantees.

One core idea behind Alcatraz is the use of the kernel as a trusted intermediary. Consequentially, many memory operations require kernel involvement. For example, starting new programs, allocating and freeing segments, and marking a segment for Inter Process Communication (IPC) all require system calls. But since Alcatraz implements system calls using exceptions, system calls are relatively slow. If nothing else, they involve a context switch and often have deleterious impacts on the cache. At first glance then, it appears that Alcatraz's design is rather slow.

However, on closer examination, one can see that system calls are relatively infrequent. Most processes will allocate only one or two segments in their lifetime. Applications that use IPC will spend far more time manipulating shared data than changing the access rights on shared segments (which will usually happen only once). Infrequent events, even very slow ones, have little impact on overall system performance. By performing these operations in kernel space, Alcatraz takes a slight performance hit, but that allows us to simplify (and speed up) the hardware, so the far more common case is much faster. This is a good example of Amdahl's law [2] [3] at work. Seen in this light, Alcatraz is designed for high performance.

Because system calls are slow, good performance depends on making sure system calls are infrequent. For example, application memory allocation can be very slow since applications may need to allocate space for millions of objects in their lifetime. Naive applications that ask the kernel to make a new segment for each and every object created will exhibit terrible performance due to the system call overhead. As a result, most applications will make use of an intelligent user level allocator (like the one in libc) that batches memory requests to ensure that kernel memory allocation really is an rare event.

Since Alcatraz uses 8-bit segment numbers and 8-byte segment descriptor records, the segment descriptor table takes 2048 bytes of memory. Each segment permission table takes 256^2 bits of space. Since there are 3 permission tables, Alcatraz uses a total of 26,624 bytes of memory for it's tables. However, if 256 segments proves too limiting we can increase the size of the segment field at the expense of the size of the offset. This raises the question of how Alcatraz's memory consumption scales as the size of the segment field increases. Repeating our analysis for segment descriptors N bits long, we discover that M, the memory consumed by Alcatraz, grows as the square of N:

$$M = (8bytes)N + \frac{3N^2}{8bits/byte} = \Theta(N^2)$$

Overhead memory consumed is only one metric. A more important one is memory operation latency. For load and store operations, Alcatraz requires five physical memory accesses. One memory access each is needed for the instruction fetch, permission check, range check, start address retrieval, and the actual memory operation. All other instructions require four memory operations. If the operation triggers a permission or range fault, the latency increases tremendously since exceptions require kernel processing.

The large number of extra memory accesses dramatically reduces performance, but the highly localized nature of these additional memory reads suggests that the use of a cache could alleviate most of extra latency imposed by the MMU. However, while caching is vital for good performance, it must be carefully tailored so as not to defeat Alcatraz's isolation guarantees. In particular, if a cache is placed between the CPU and MMU, a malicious application may be able to read data or execute code for which it is not authorized. This exploit works by loading addresses belonging to the process (or kernel) that was previously running. Since that data is already in the cache, it would be provided to the CPU without passing through the MMU at all. The solution is to either insure that caches are not placed between the CPU and MMU, or, if a cache must be placed before the MMU, ensure that the cache gets flushed on each context switch.

4.2 Comparative Analysis

The design of Alcatraz involved choosing from among several options for many different questions. Different choices could have made Alcatraz very different from what it is today; they certainly would have impacted the cost, performance, and requirements. For a few important design decisions, we will examine and compare alternatives below.

4.2.1 System Call Mechanism

We originally considered implementing system calls as direct procedure calls into the kernel. System calls would be faster since they would not trigger exceptions and the whole system would be more general since calling kernel functions would become just like calling library functions.

Unfortunately, this makes it very difficult to meet the isolation requirements. Specifically, once an application jumps into the kernel, the resulting code is operating with the kernel's privileges. However, there is no guarantee that the application will jump to a kernel function entry point. In fact, because Alcatraz has no mechanism in place to specify and restrict segment entry points, an application can jump to any kernel instruction. Since hostile programs can tailor system state (registers and stack values) before entering kernel code, they could execute kernel functions with arbitrary arguments while using kernel privileges.

A related problem is that the kernel must trust the application to tell it where to return to; this could be exploited to enter into another program's code. With exception based system calls, the kernel can rely on the CPU to set the XP register so it knows authoritatively exactly who called it.

In any event, the CPU needs to have an exception mechanism in order to support device interrupts, preemptive scheduling and instruction errors (like arithmetic overflow). Thus, implementing system calls as direct procedure calls does not allow us to reduce or simplify our hardware at all.

4.2.2 Permission Table Structure

Early in the design process, we elected to implement the permission lookup tables using a dense matrix representation. This means that each table includes space for all possible combinations of executing and requested segments, even when most of those entries are empty. An alternative approach would be to implement the permission tables with some other data structure better suited to sparse data such as a linked list.

Sparse data structures would reduce the total memory required in the average case, but introduces several other problems. For starters, the MMU would need a more complex lookup algorithm. In addition, sparse representations generally require some layer of indirection. Extra layers of indirection require additional memory accesses. But since memory is so incredibly slow, these additional accesses dramatically reduce performance.

Even if we did use a sparse representation, it is far from clear which sparse structure is best since there are several important common cases: most applications will have permissions for only two segments, but the kernel will need to have permissions for all. Since a good fraction (30%) of CPU time is spent in the kernel [4], it would be very difficult to find a scheme that supports both common cases efficiently, without introducing a special purpose kernel handling mechanism and thereby reducing system generality.

Finally, it is worthwhile to consider just how much sparse tables actually save us. Alcatraz currently uses about 26 KB of memory. Given current RAM prices ($0.50/MB), this memory costs less than a penny. Consequentially, sparse tables are a non-solution to a non-problem.

4.2.3 Permission Placement

Another critical design question was whether the segment tables should be centralized in one area controlled by the kernel or whether they should be distributed so that each segment could write its own permission table directly. A distributed approach would allow applications to indicate to the MMU which executing segments could access their segments. Effectively, this approach breaks a segment permission table into columns and places each column at the front of the segment it represents.

Distributed permission tables should be faster since applications would no longer need to make system calls simply to change segment permissions. Unfortunately, this has a negligible impact on overall speed since changing segment permissions is an infrequent event.

The decentralized approach also makes it difficult to meet our kernel access requirements since an application could simply deny the kernel permission to access it's segments. This particular problem could be solved by making the MMU support some special purpose mechanism that would exempt the kernel from permission checks. However, this kind of special purpose mechanism would greatly reduce system generality.

The most serious problem with distributed permission placement is that it destroys abstraction barriers. Specifically, it requires that applications set permissions in terms of segments, since the MMU only deals in terms of segments and knows nothing about processes. But there is no other reason that applications need to have access to the kernel's process–segment map. Forcing applications to understand so much information about how segments are associated with processes greatly reduces transparency. Alcatraz allows a process to specify it's segments' permissions in terms of other processes. This approach maintains the abstraction barrier.

An alternative approach to making decentralized permissions work would be to make the MMU know about processes. This approach not only obliterates an abstraction barrier (albeit a different one), but also ties the MMU to one operating system's notion of what a process is. The end to end argument suggests that binding the MMU so tightly to one process model is very undesirable. The end to end argument states that higher level functionality (like a process model) should not be implemented in lower level infrastructure [3].

4.2.4 Verification Location

One of the most important design questions we faced was where should memory checking occur. Alcatraz implements permission and range checking in hardware, but an alternate solution would have been to perform such checks in software.

A software memory checker would work something like this: each user level memory access would trigger a CPU fault. The resulting exception would transfer control to the kernel. The kernel could then perform the necessary permission and range checks by consulting the segment tables. Memory accesses that occur when the supervisor bit is set would not fault, thus allowing the kernel to function.

By moving functionality into software, we can make the hardware much simpler. The system is also more flexible since it can be changed simply by rewriting a piece of the OS rather than building new hardware. Unfortunately, this approach is extremely slow. Each application instruction would require the CPU to process many overhead instructions.

One potential solution to the speed problem is to compromise by doing more work in hardware. This technique improves performance at the expense of simplicity and flexibility. Such a compromise solution might permit all memory accesses that occur within the same segment. The accesses would take place entirely in hardware. Attempts to access memory outside the currently executing segment would cause exceptions just as before. This system is optimized for a common case, but that case is simply not common enough. Instruction fetches would not need kernel intervention, but all data accesses would still go through the kernel.

Worse than than the performance is that the current system does no range checking. We can implement range checking in hardware, but that means building hardware with about the same complexity as Alcatraz currently has while suffering greatly reduced performance.

5 Conclusions

We have just completed a world wind tour of Alcatraz. Along the way we covered why Amazing Computers needs hard modularity in their computers, how segmented memory systems work, and what the exact requirements are for such a system. We also examined how Alcatraz works by focusing on the MMU hardware design in addition to the role played by the OS. We analyzed Alcatraz's performance and scalability and compared it to several alternative designs to better understand the tradeoffs it embodies.

In closing, we would like to review some key lessons that Alcatraz relies on:

1. Treat the kernel as a trusted intermediary. This allows us to exploit the Principal of Least Privilege [3].

2. Use the current PC to discover which segment is currently running.

3. Hierarchy is good. The MMU should only deal with segments and processes should only think in terms of processes. The OS should reconcile these two world views.

References

[1] *6.004 Beta Documentation.* Massachusetts Institute of Technology, Cambridge, MA, 1997.

[2] David A. Patterson and John L. Hennessy. *Computer Architecture and Design: The Hardware Software Interface.* Morgan Kaufmann, Cambridge, MA, 1997.

[3] Jerome H. Saltzer and M. Frans Kaashoek. *Topics in the Engineering of Computer Systems.* Massachusetts Institute of Technology, Cambridge, MA, 2002.

[4] Uresh Vahalia. *Unix Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, NJ, 1996.