Using a Segmented Memory System to Enforce Modularity between Programs

Rohit Rao
March 20, 2003

Contents

List of Figures

Abstract

The purpose of this project was to design a virtual memory system that met three basic design goals: enforced memory modularity, the capability of dynamic memory reallocation, and minimized power consumption. After analyzing a number of possible designs, a segmented memory system was chosen as the preferred implementation. By performing reads and writes in only one memory access and pushing the burden of computations to the dynamic allocation procedures (which are invoked very rarely), the design presented in this report met all design goals while minimizing energy consumption.

1 Introduction

Adding virtual memory to a computer system is an excellent way to achieve multiple design goals. A system with virtual memory is able to avoid addressing memory using physical addresses. Instead, a virtual memory manager assigns a virtual address to each physical location in memory and programs access memory using only these virtual addresses. This allows a system with virtual memory to abstract its actual memory hierarchy; on-board cache, system RAM, and hard drives are all accessed through virtual addresses. Interactions between programs and memory also become simpler. Instead of having to tell a program that it is not stored in contiguous physical memory, a virtual memory manager can assign the program a contiguous virtual address space. Finally, virtual memory provides an easy means of enforcing modularity between programs. Each program is given its own virtual address space, and therefore no program has the means of accessing another program's data.

The basic concepts of virtual memory continue to apply when a virtual memory system is used in an application such as the mote. By adding virtual memory to the mote, we can enforce modularity between individual programs and assign each program a separate address space. My design takes the form of a segmented memory system, in which the kernel maintains a table of starting locations for each process' memory allocation. A virtual address takes the form of a virtual page number, which is used in conjunction with a process ID to create a physical page number. The system can then use this physical address to access the desired data.

There are several major tradeoffs to consider when designing a virtual memory system for the mote. A good design will minimize power consumption during operation, and as a direct result, CPU usage and the number of memory accesses need to be minimized as well. This document will describe my chosen design, explain the tradeoffs it makes, and detail its performance when compared to other design alternatives.

2 Design Overview

A segmented memory system uses a "base and bound" approach to assign virtual addresses. The Memory Management Unit (MMU) allocates a contiguous block of memory for each individual process. Virtual addresses take the form of an offset within this allocated block. Therefore, each process believes that it has been assigned its own separate block of memory that begins at memory location 0x00. The main system kernel is responsible for maintaining a table that contains the starting page number of every process. (This is the address of the first page of allocated memory for that process.) Therefore, when a program attempts to read from a memory address, the MMU retrieves the starting page number, adds the virtual page number, and then

uses that to access the proper location in memory. The kernel also stores the amount of memory that has been allocated to each process. If a process issues a read or write command to a location that is outside its bounds, the MMU throws a segmentation fault and alerts the kernel of the illegal memory access.

This segmented memory system also includes procedures for allocating and destroying memory, though they require a large number of memory accesses. If a process makes a request for more memory, the system creates the requested amount of memory directly after the memory already assigned to the process. It does this by shifting everything else in memory down by the required amount. The variable that stores the amount of allocated memory for that process is also updated, as are the starting page numbers for all the relocated processes. Analogously, when the system destroys a process, it reclaims memory by moving all the other programs up by the reclaimed amount of space. Again, the starting page numbers and allocated space variables for all the affected processes are updated. Finally, when a new program is uploaded to the mote, the system allocates space for the program after all the already allocated memory and updates the starting page number and allocated space variables for the new process.

One important additional subsystem helps ensure that a program cannot access memory that the MMU did not assign it. Since the kernel has full access to the memory (it can access any location in memory, regardless of the segment's owner), a good design needs a way to differentiate between calls issued by the kernel and calls issued by user-level programs. I added a kernel/user control bit to assert this distinction. If the kernel bit is high, accesses to memory are sent directly to RAM without passing through the MMU. If the kernel bit is low (indicating user mode), all accesses to memory have their addresses translated by the MMU before the locations are passed on to RAM.

3 Design Description

The virtual memory system for the mote can be broken down into a number of subcomponents. At the heart of the system is the Memory Management Unit, which is responsible for converting virtual addresses to physical addresses. Overseeing the MMU is the operating system's kernel, which provides the MMU with enough information to perform translations. The OS also runs the software that controls what happens on memory allocations, memory destructions, and process context switches. Finally, a fault control system must be implemented to catch illegal memory accesses. The following sections describe in detail the chosen implementation for each subcomponent.

3.1 Virtual Paged Memory

As described before, processes in the mote reference memory through virtual addresses, which consist of a virtual page number followed by an offset within that page. The MMU, when translating virtual addresses into physical addresses, first converts the virtual page number into a physical page number. It then concatenates the offset to the end of this physical page number, creating a physical memory address. My design divides the 32K of memory into pages of 256 bytes each. This page size was chosen because it provided a reasonable compromise between granularity and the number of control bits required. In other words, a page size of 256 bytes results in an acceptable amount of wasted (allocated but unused) memory but does not require an unreasonable amount of overhead and memory to maintain the internal tables. Since there are 128 pages total, seven bytes are required to encode each individual page. Another eight bytes are

required to encode the offset within each page, meaning that each physical address is 15 bits long.

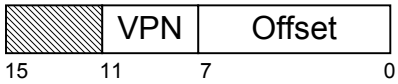

Figure 1: Format of a Virtual Address

The memory is segmented into pages as depicted in Figure 2. Each process has a starting page number ("base") and a total amount of allocated memory ("bound") associated with it. (Since programs are limited in size to 4K maximum, the OS only needs four bits to represent the bound variable.) Virtual addresses for a process take the form of a 4-bit offset within the process' allocated memory. When given a virtual address, the MMU adds the virtual address to the starting page number of the running process, creating the actual physical page number of the requested page. The MMU also compares the given virtual page number with the stored bound of the current process. If the virtual page number is larger than the bound, then the process is trying to access memory that does not belong to it, and the MMU must throw a fault.

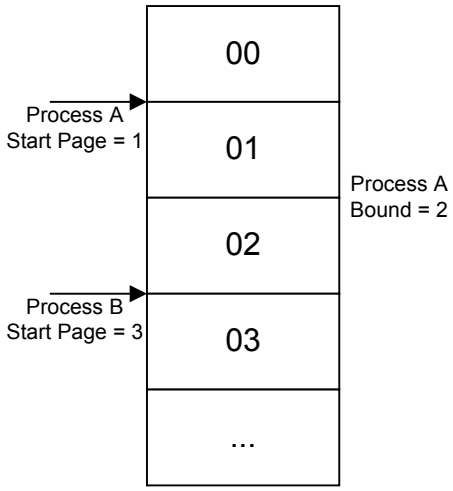

Figure 2: Diagram of pages in main memory, showing the base and bound pointers for two separate processes.

3.2 The Memory Management Unit

The MMU is the heart of the virtual memory system. It is responsible for adding a virtual page number and the stored starting page number, which creates the physical page number of the memory location that the program wants to access. It is possible to perform these steps in software, but that would require an extra CPU operation, wasting both time and power. In my design, I chose to incorporate a hardware adder in the MMU to perform this function. The adder takes in the virtual page number and the stored starting page number and outputs their sum. Since the adder runs in an amount of time that is negligible when compared to the 125ns period of the system clock, the translation from virtual addresses to physical addresses can take place during the same clock cycle as the original memory access call. Therefore, the MMU does not use any extra time or power during a read or write operation.

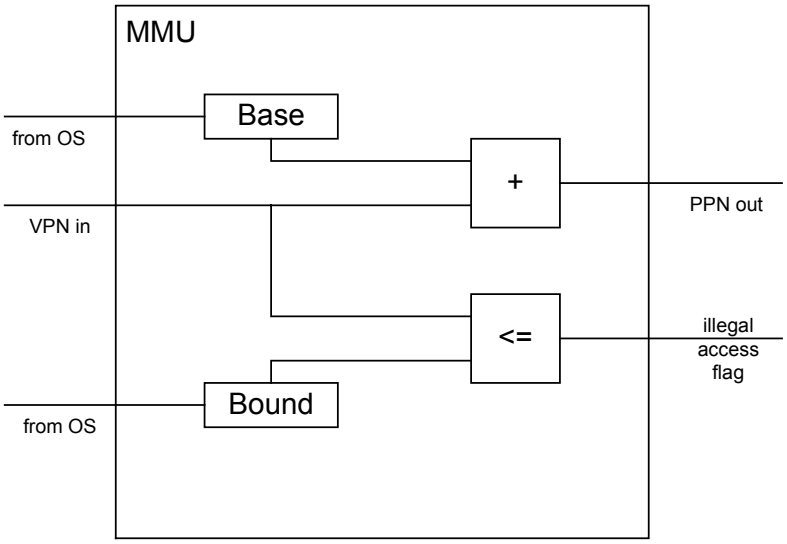

Figure 3: Block diagram of the MMU.

The MMU also requires some form of internal state to store the starting page number and bound of the current process. My design uses two sets of registers to store these variables. As mentioned before, these base and bound numbers can be stored using seven and four bits, respectively, so my MMU requires 11 bits of internal state to function. The burden of loading the proper values into these registers falls upon the operating system; when a context switch occurs, the OS must extract the proper values from memory and load them into the Base and Bound registers. Figure 3 depicts the internal organization of the MMU.

3.3 Fault Detection

The MMU is primarily responsible for ensuring that no program can access memory that has been allocated to another process. In the design described above, if a program issues a call trying to access a page that is beyond its actual page allocation, a simple adder will return a physical location that is in the memory space of another process. This illegal access can be solved by properly using the bound variable. This variable stores the number of pages that have already been allocated to the process in question. Therefore, if a process has been given 4 pages and it tries to read from virtual page number 5, it is obviously trying to access a page that is not in its address space. My design includes a simple less-than-or-equal-to comparator to test whether or not the physical page number created by the adder is valid.

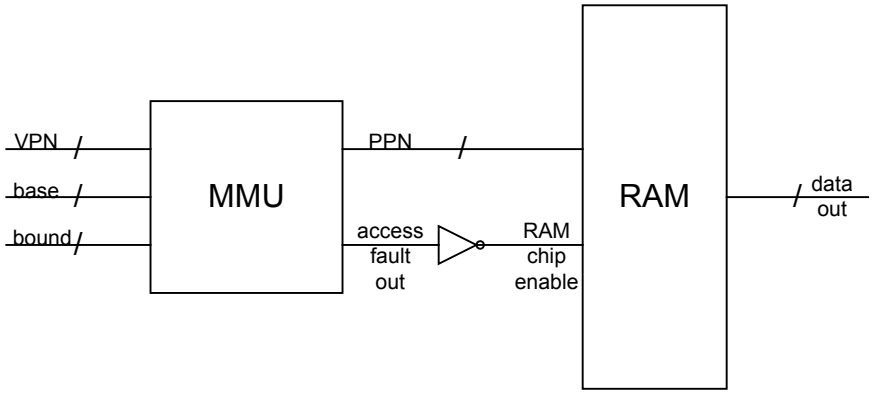

Figure 4: Interactions between the MMU and the RAM. When an illegal access fault is detected, the RAM is disabled, preventing any modification of data from occurring.

If the requested virtual page number is outside the allocated address space of the current process, the MMU outputs a fault flag. This in turn can set of an interrupt within the main kernel, causing it to jump to a section of code designed to handle illegal page access faults. The kernel can inform the program that it has tried to read from an invalid location in memory or terminate the erring process if it wishes. The MMU also needs to power down the RAM when an illegal access is detected. If the RAM was allowed to function normally, a program that tried to write to a location outside its bounds would actually succeed in getting the write through before the fault was detected by the kernel; this is not desired behavior, for it allows a program to modify data that belongs to another process. The interaction between the MMU and RAM is shown in Figure 4. Once again, my design pushes the burden for handling illegal access faults onto the kernel, though the MMU itself must initially detect the illegal access and prevent a read or write from occurring.

3.4 Context Switching

The OS in my design bears much of the burden of the virtual memory system. This section will detail what the OS must do on context switches. Every time the OS switches contexts, or changes the currently running process, it must perform a small amount of overhead. This is necessary to ensure that the MMU has enough information to correctly translate virtual addresses into physical addresses.

The OS must hold two tables for storing the starting page numbers and total allocated page numbers for each process. When the currently running process is changed, the OS must search through the tables, extract the proper values for the new process, and load these values into the two internal state registers of the MMU. The code below details the operation of the kernel's "on_switch" function.

```
int max_processes = 32;
array base_table[max_processes];
array bound_table[max_processes];

// This process (and the entire virtual memory
// system) assumes that new_PID is a valid PID
// that has been mapped to an actual process.  No
// guarantees are made for functionality and memory
// protection if this assumption is not held true.

boolean on_switch([in] new_PID)
{
   new_base <= basetable[new_PID];
   new_bound <= boundtable[new_PID];

   load new_base into the MMU's base register;
   load new_bound into the MMU's bound register;
   return true;
}
```

Figure 5: Pseudocode for the on_switch function. Note that the maximum number of processes is being fixed to 32 and that the input is assumed to be a valid PID.

3.5 Memory Reallocation and Process Destruction

In my design, memory allocations and process destructions are among the most expensive of operations. The simplest allocation operation to implement is new_protection_domain. When a new program is loaded onto the mote, the OS needs to check that there is enough free space to fit the program into memory. If there is, the OS can transfer the program into main memory and setup its associated variables. To simplify the check for enough free space, the OS maintains an internal variable that points to the end of the allocated memory (or, in other words, a pointer to the start of the free space). This EOM pointer is an 8-bit variable that is stored somewhere in the data portion of the memory allocated to the OS. When a program requests to be uploaded to the mote, the OS subtracts the EOM value from the total size of the physical memory, giving the total number of unallocated pages in memory. If there are enough unallocated pages to supply the new program with the amount of memory it requests, the OS assigns a unique PID to the new process, updates its base and bound values, and updates the EOM pointer. The pseudocode below outlines the new_protection_domain procedure.

```
// The default state of the EOMpointer is to point to page #8,
// which is after all the OS code and OS/MMU data.
int EOMpointer = 8;
int max_pages = 128;

boolean new_protection_domain([in]int req_size, [in]int PID,
                              [out]mem_region[] regions)
{
   // calculate the number of pages needed
   int rounded_size <= req_size rounded up to
                       the nearest multiple of 256;
   int rounded_pages <= rounded_size / 256;

   //verify that enough free pages exist to satisfy the request
   if ((max_pages - EOMpointer) <= rounded_pages)
      return false;

   // if there is enough free space, allocate it and
   // update the EOMpointer and the base/bound of the
   // requesting process
   base_table[PID] <= EOMpointer;
   bound_table[PID] <= rounded_pages - 1;     //pages are
                                                zero-indexed
   EOMpointer <= EOMpointer + rounded_pages;

   // format the output data -- must convert from
   // pages to actual bytes
   regions[0].start = base_table[PID]*256;
   regions[0].length = rounded_size;
   return true;
}
```

Figure 6: Pseudocode for the new_protection_domain method. Note the default value of the EOMpointer variable and the conversion from page numbers to physical addresses when returning a mem_region to the OS. The process that receives more memory is responsible for realizing that it may have received more memory than it asked for.

Though my design performs memory reads and writes very quickly, it uses a lot of processor time when it needs to allocate more memory to a process. When a process calls grow_protection_domain, the OS once again uses the EOM pointer to ensure that there is enough free space to meet the calling process' demand. If there is enough free space, the OS needs to undergo intensive memory copying; since memory is allocated in contiguous blocks, a large amount of data must be moved to make room for the newly allocated memory. The OS locates all the processes that were allocated memory after the current process. It then copies them to a location that is an appropriate number of pages further down in the memory. For example, if a process requests an additional 768K (3 pages) of memory, all the data that follows that process' data must be moved down three pages in memory. The pseudocode below details the steps in the grow_protection_domain procedure.

```
boolean grow_protection_domain([in]int PID, [in]int size)
{
   // calculate the number of pages needed
   int rounded_size <= req_size rounded up to the
                       nearest multiple of 256;
   int rounded_pages <= rounded_size / 256;

   // verify that enough free pages exist to satisfy the request
   if ((max_pages - EOMpointer) <= rounded_pages)
      return false;

   // int start_marker = base_table[PID] + bound_table[PID] + 1;
   // int end_marker = EOMpointer - 1;

   --> shift the appropriate segments in memory down by
       rounded_pages pages.  The section between start_marker
       and end_marker must be moved to memory locations
       (start_marker + rounded_pages) to (end_marker +
       rounded_pages).  To perform this shift without
       overwriting data, start at the bottom of block and work
       upwards.

   // update the base_table values and the EOMpointer
   bound_table[PID] <= bound_table[PID] + rounded_pages;
   EOMpointer <= EOMpointer + rounded_pages;

   for (p = 0; p < max_processes; p++)
   {
      if (base_table[p] > base_table[PID])
        base_table[p] <= base_table[p] + rounded_pages;
   }
}
```

Figure 7: Pseudocode for the grow_protection_domain function.

One side effect of the dividing memory up into pages is that it becomes nearly impossible to provide a program with the exact amount of memory that it requested. My design rounds all allocation requests up to the nearest multiple of 256 bytes and allocates that much to the process. (Overallocation is acceptable since it provides the process with at least as much memory as it wants.) As a consequence, my design requires the process to realize that it has been allocated more memory than it asked for. If a process sends a request to grow its virtual address space by

one byte, it should realize that it will receive 256 bytes of memory. Therefore, the process should use that overallocated memory instead of sending another grow request to allocate an additional one byte of memory.

The delete_protection_domain works analogously to grow_protection_domain. When a process is deleted from the mote, all the data that followed that process in memory is "compacted", or shifted up, creating one large block of allocated memory instead of two smaller separated blocks. For example, if a process with seven pages of allocated memory is destroyed, the OS shifts all the other data in memory up by seven pages, as shown in Figure 8. The pseudocode below details the operation of the delete_protection_domain procedure.

```
boolean destroy_protection_domain([in]int PID)
{
   int removed_pages <= bound_table[PID] + 1;


   // update the base_table values and the EOMpointer
   EOMpointer <= EOMpointer - removed_pages;

   for (p = 0; p < max_processes; p++)
   {
      if (base_table[p] > base_table[PID])
        base_table[p] = base_table[p] - removed_pages;
   }
}
```

Figure 8: Pseudocode for the destroy_protection_domain method. Note that there is no need to explicitly reset the values stored for the base and bound of the destroyed PID. When a new process is given that PID, the old values will be overwritten anyways.

3.6 The Kernel/User Bit

The virtual memory system also requires some additional protection from user-level programs, which my design provides for through the inclusion of a kernel/user bit. The basic operation of the kernel/user bit is similar to other systems: the bit is set to high when the system is running in kernel mode and set to low when it is running user-level programs. The two registers that store the base and bound values must be protected from accidental (or malicious) overwriting by user-level programs. Since only the kernel is allowed to write to these registers, my design requires the kernel bit to be high when writing to the base and bound registers. If a write is attempted when the kernel bit is set to user mode, a fault should be thrown.

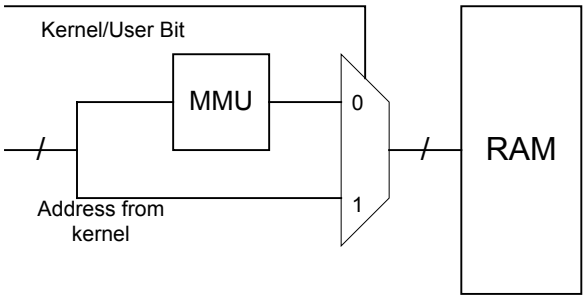

Figure 9: Block diagram of the MMU/Multiplexor setup.

A good design should also allow the kernel to directly access any location in memory without having to first be translated by the virtual memory system. My design allows for this by user the kernel/user bit to multiplex the address inputs of the RAM. When the system is running in user mode, the MMU assumes that all address are virtual addresses and translates them into physical addresses. When the system is running in kernel mode, however, the addresses outputted by the kernel are treated as physical addresses and passed directly into the RAM. The diagram above shows the multiplexing that occurs.

3.7 Performance Analysis and Assumptions

The main benefit of this design is that it enforces modularity with almost no extra cost where memory reads and writes are concerned. When a process issues a call to read from memory, the MMU translates the virtual address into a physical address using only a hardware adder. Since this translation is done in a fraction of a clock cycle, no additional CPU time or memory accesses are needed to retrieve data from memory. The tradeoff in this design comes in reallocating memory. Reallocating memory and destroying processes are extremely costly operations in this segmented memory system, since a large amount of data must be physically copied to a different location in memory. The exact cost of a reallocation or a destruction is dependent on the number of processes and the total amount of allocated memory at the time; in the worst case, an destruction could result in the movement of up to 29K of memory, which would require 29 thousand memory reads and 29 thousand writes. The worst case scenario for a reallocation is similar, resulting in the movement of 28.75K of data.

The design proposed in this paper remains feasible because it is assumed that process destructions occur very infrequently. It stands to reason that processes are only created and destroyed when the mote is being programmed. Therefore, these operations can have inefficient implementations, since performance is not an important consideration while the mote is still in the lab. The only remaining operation is memory reallocations. Since each process is limited to 4K maximum, they can only request reallocations 12 times. This results in a weak upper bound of 180 total reallocations over the operation life of the mote, using the maximum of 32 processes. When compared to the amount of reads and writes that occur, the grow_protection_domain method is called very rarely. In addition, when processes are initially loaded onto the mote, they request enough space to hold all their code and data structures. If programs are written and analyzed well, there should be little to no need for dynamic reallocation. Therefore, it is justifiable to place a large burden on reallocations in order to reduce the memory and CPU consumption of simple memory reads and writes.

3.8 Specification Changes

The design I propose in this paper contains a number of changes to the original hardware and software specifications. Since this design divides the main memory into 256-byte pages, memory is always allocated in 256-byte blocks. Therefore, when a process requests more memory, it is allocated memory in units of pages, not bytes. Each process is responsible for knowing that it may receive more memory than it asked for, and it should use this knowledge to avoid asking for memory when it has already been allocated enough. (For example, a process should not make a second reallocation requests for 1 byte if it receives 256 bytes from the first request.)

The memory used in the mote is also byte-addressed with no alignment restrictions, which could cause problems in some situations. For example, if a process issued a request for

the last byte in its address space, the memory would also return the first byte of the next address space, a violation of the enforced modularity that we seek. For this reason, I chose to ground the lowest address bit of my RAM, forcing the RAM to be word-aligned. Programs written for use on the mote are responsible for knowing that their calls to memory will automatically be word-aligned.

3.9 Hardware Limitations

My design also has a number of limitations that are imposed by the hardware of the system. A few examples include the maximum size of the RAM, the maximum number of pages that can be allocated to a process, and the maximum number of processes that can run on the mote at any given time. The first two of these examples are caused by hardware limitations; the registers that store the base (VPN) and the bound of a process are limited to 7 and 4 bits, respectively. By adding one more bit to each of these registers (and adding one more bit to each value stored in the corresponding software tables), the maximum RAM size and the maximum number of pages per process can be doubled. I chose to limit the capability of the mote to the size of the specified hardware because it was never made clear how likely mote hardware expansion was. If hardware expansion is found to be a likely possibility, it would be a simple matter to add the extra bits to the bus. The maximum number of processes is, one the other hand, purely a software limitation. To allow for more processes, one only needs to change the value of the max_processes variable. The main limitation is that the size of the base and bound tables is directly proportional to the maximum number of processes; the more processes that can exist, the more memory the tables require.

3.10 Other Possible Design Extensions

There are a number of ways to improve on the basic design presented in this report. These improvements either optimize or add functionality to the existing design, but were not incorporated into the original design because it was felt that the added benefits did not justify the increased development costs. One improvement is to lower memory reallocation and destruction costs by not compacting memory when processes are destroyed. This idea breaks the memory into separate chunks of contiguous allocated memory that have unallocated space between them, meaning that the free space in this design is distributed throughout the RAM and not grouped at the end. When a program requests more memory, the nearest free page is (on average) physically closer than it would be in the original design. Therefore, less data needs to be copied to a different location in memory. This idea is demonstrated in Figure 10 below.

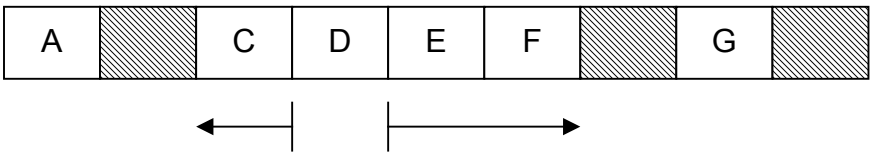

Figure 10: Diagram of memory organization in a non-compacting system. The arrows point out the closest unallocated blocks to the allocated space for process D.

Using this scheme to organize memory, however, requires a careful redesign of the reallocation algorithm. The algorithm can no longer simply look at the end of the allocated memory to find free space. Instead, the new algorithm needs to start at the boundaries of the allocated space of the current process and search outwards until it locates enough free pages to fulfill the reallocation demand. This search pattern ensures that the nearest free pages will be

used, meaning that the smallest amount of data needs to be displaced. If the search algorithm reaches the boundaries of the whole memory without locating enough free space, then it returns saying that there is not enough space to comply with the request.

4. Feasibility

The main design alternative that I considered was the page table-based virtual memory system. A page-table based system uses tables to store the data needed to translate virtual addresses into physical addresses. When more memory is allocated to a process, a new entry is added into the page table, and when a process is destroyed, its entries are removed. The main attraction of a page-table based system is that allocation and destruction of processes are very simple. In my design, an allocation requires a large amount of data to be moved to a different location in memory. However, in a page-table based design, allocating new memory is reduced to simply adding one more entry into the page table.

Despite the apparent advantages of a page-table based design, I chose to go with a segmented memory system. The drawback of a page table design is that it requires two memory accesses to read or write to a location in memory. The virtual memory system needs to access the memory once to lookup the physical address, and then it accesses the memory a second time to actually retrieve the desired data. As a result, a page-table based system effectively doubles the number of memory accesses that occur. A TLB cache can be added to speed up memory reads and writes, but this makes the virtual memory system unnecessarily complex. A page-table based system also requires much more memory than a segmented memory system. With the page sizes and total memory size outlined in this paper, a page-table based system would require well over 512 bytes of memory to hold its page tables, while my base and bound design only uses 64 bytes of memory.

With the advantages and disadvantages of each system in mind, I decided that a segmented memory system would best meet the outlined goals. A page-table based system would offer much more flexibility, but the majority of that flexibility is unnecessary and excessive for this particular application. If it is true that memory reallocation and process destruction occurs only very rarely, then a segmented virtual memory system would offer significantly better performance with less overhead.

5 Conclusions

After considering a variety of design alternatives, I decided that a segmented virtual memory system was the best design for use on the mote. I divided the physical memory into pages of 256 bytes each and created an MMU to convert the virtual addresses used by individual processes into physical addresses that can be fed into the RAM. The final design required only one memory access to perform a read or write operation, making it significantly more attractive than a page-table based design. Dynamic allocations took a significant amount of processor time, but this was not a concern because dynamic allocations were assumed to occur only very rarely. All things considered, the base and bound virtual memory system presented in this paper is the simplest and most efficient system that meets all of the required design goals.

6 Acknowledgements

I discussed a number of the design ideas detailed in this paper with Isaac Cambron and Rupesh Kanthan, and it is partially due to these conversations that my design was able to take this shape. Office hours provided by Chris Lesniewski-Laas were also helpful.

Appendix A: Block Diagram

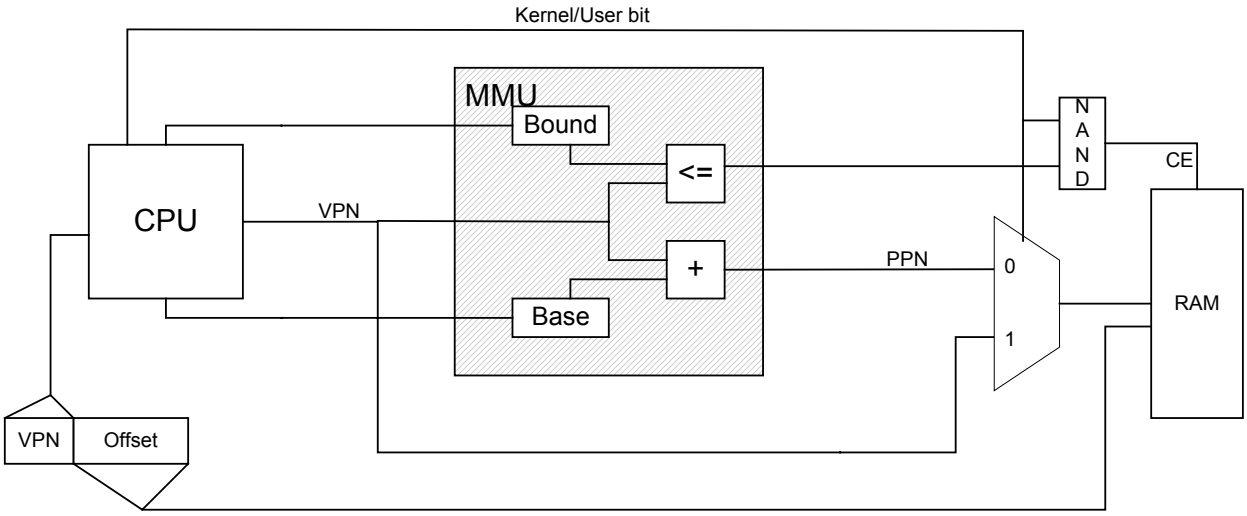

Figure A-1: A block diagram of all proposed hardware.