Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Simple Memory Protection for Embedded Operating System Kernels

Frank W. Miller [*]
Department of Computer Science & Electrical Engineering
University of Maryland, Baltimore County

Abstract

This work describes the design and implementation of memory protection in the *Roadrunner* operating system. The design is portable between various CPUs that provide page-level protection using Memory-Management Unit (MMU) hardware. The approach overlays protection domains on regions of physical memory that are in use by application processes and the operating system kernel. An analysis of code size shows that this design and implementation can be executed with an order of magnitude less code than that of implementations providing separate address spaces.

1 Introduction

This work presents the design and implementation of a memory protection subsystem for an operating system kernel. It may be useful for systems that do not or cannot utilize paging and/or swapping of memory to secondary storage. The types of computer systems in use today that have such a requirement are generally embedded. The primary goals of the design and implementation are simplicity and small code size.

There are a variety of commercial and academic embedded operating system kernels available today. Many do not implement memory protection of any kind. This work presents a general, page-based mechanism that can be used to add memory protection to these kernels. The initial design and implementation was performed in the *Roadrunner* operating system kernel, an embedded kernel similar to the VxWorks and pSOS kernels.

Virtually all modern operating system kernels that make use of MMU hardware utilize the address translation mechanism to implement virtual addressing and separate address spaces. The notable exceptions have been embedded operating system kernels. Bypassing address translation and making use strictly of the memory protection mechanisms provided by MMU hardware yields a simple memory protection design that can be implemented with an order of magnitude less code than designs that provide separate virtual address spaces. This code reduction is achieved even when a comparision is done excluding the code for paging and swapping.

The design uses page-based memory protection to 1) limit access by one process to the memory regions of other processes and 2) limit access by user processes to kernel memory. It does not utilize separate virtual address spaces or provide the illusion that physical memory is larger than it really is by using demand-paging and/or swapping.

The design provides protection portably. It can be implemented on a variety of different MMU hardware devices and the design implications associated with several MMUs are discussed.

The remainder of this work is organized as fol-

[*] Author's current address: *sentitO Networks, Inc.*, 2096 Gaither Road, Rockville, Maryland 20850 Email:`fwmiller@cornfed.com`

lows. Section 2 provides an introduction to the *Roadrunner* operating system in which the new memory protection mechanism is implemented. Section 3 provides the specifics of the design and implementation of the memory protection subsystem. Section 4 discusses the design points surrounding MMU designs found in three popular CPUs. Section 5 presents a set of measurements yielded by the initial implementation of this protection scheme in the *Roadrunner* operating system. Section 6 provides a brief survey of related work and Section 7 draws the work to its conclusion.

2 *Roadrunner* Design

There are three basic *Roadrunner* abstractions:

Processes: The basic unit of execution is the process. The *Roadrunner* kernel provides a POSIX-like process model for concurrency.

Files: A variety of I/O operations, including interprocess communications and device access are implemented using a multi-layered file system design.

Sockets: An implementation of the Internet protocols is available through a BSD Sockets interface.

These abstractions have an Application Programming Interface (API) that is exported through the kernel system call interface.

An important distinction between this design approach and that of separate address spaces is that user code must either be position-independent or have relocations performed at load time. The *Roadrunner* implementation performs relocation by default so position-independent code (PIC) is not required.

Memory Protection Operations

This work describes the design and implementation of the memory protection subsystem present in the *Roadrunner* operating system kernel [2, 4]. The basic design principle is that logical (or virtual) addresses are mapped one-to-one to physical addresses. This means that the value of logical address is the same as its corresponding physical address and that all processes reside in the same logical as well as physical address space. One-to-one mapping simplifies the memory management subsystem design dramatically. In addition, when the indirection represented by address translation is removed, the programmer can reason about the actual logical addresses as physical addresses and that can be useful for dealing with memory-mapped elements.

Protection is based on *domains*. A domain is a set of memory pages that are mapped using a set of page tables. In addition, a set of memory pages associated with the operating system kernel called the *kernel map* is kept. The kernel map is mapped into all domains but is accessible only in supervisor mode.

No address translation is performed. Only the protections attributes associated with page table entries are used. The basic operations that can be performed are listed as:

- *Insert the mapping of a page into a domain*
- *Remove the mapping of a page from a domain*
- *Insert the mapping of a page into the kernel map*
- *Remove the mapping of a page from the kernel map*
- *Update a domain to reflect the current mappings in the kernel map*

Table 1 lists the routines in the *Roadrunner* kernel that implement these basic operations.

3 The *Roadrunner* Implementation of Memory Protection

There are three basic memory management data structures used in *Roadrunner*:

1. *Region Table:* an array of regions that track the allocations of all the physical memory in the system
2. *Page Tables:* each process belongs to a protection domain that is defined by the page tables that are associated with that process
3. *Kernel Map:* a list of mappings that are entered into all the page tables in the system but are accessible only when the CPU is in supervisor mode

Table 1: Page table management routines

`vm_map(pt, page, attr)`	Map a single page into a set of page tables
`vm_map_range(pt, start, len, attr)`	Map a sequence of contigous pages into a set of page tables
`vm_unmap(pt, page)`	Remove the mapping for a single page from a set of page tables
`vm_unmap_range(pt, start, len)`	Remove the mapping for a contiguous sequence of pages from a set of page tables
`vm_kmap_insert(entry)`	Insert a sequence of pages into the kernel map
`vm_kmap_remove(entry)`	Remove a sequence of pages from the kernel map
`vm_kmap(pt)`	Update specified page tables to reflect the current mappings in the kernel map

These three data structures are used in conjunction by the kernel memory management routines that are exported for use by the rest of the kernel subsystems and by user applications through the system call interface.

Regions

The basic unit of memory allocation is the *region*. A region is defined as a page-aligned, contiguous sequence of addressable locations that are tracked by a starting address and a length that must be a multiple of the page size. The entire physical memory on a given machine is managed using a boundary-tag heap implementation in *Roadrunner*. Figure 1 illustrates the basic data structure used to track the allocations of memory regions. Each region is tracked using its starting address, `start`, and length, `len`. Each region is owned by the process that allocated it originally, or by a process to which ownership has been transferred after allocation. The `proc` field tracks which process currently owns the region. Two double-linked lists of region data structures are maintained, using the `prev` and `next` fields, each in ascending order of starting address. The first is the free list, those regions that are not allocated to any process. The second is the allocated list, those regions that are being used by some process.

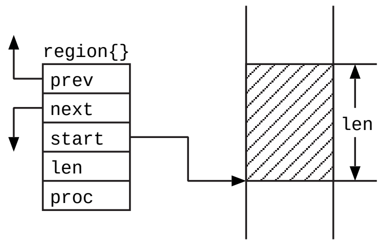

Figure 1: A region on one of the heap lists

Table 2 lists the routines used to manage the heap. The `valid_region()` routine provides a check for whether a pointer, specified by the `start` parameter, corresponds to the starting address of some region. It also serves as a general lookup routine for locating a region data structure given a starting address. The rest of the routines take a pointer to a region data structure like the one illustrated in Figure 1 as their first parameter. The `region_clear()` routine sets the fields of a region data structure to initialized values. The `region_insert()` routine inserts a region in ascending starting address order into a double-linked region list, specified by the `list` parameter. This routine is used to insert a region into either the free or allocated region lists. The `region_remove()` routine removes a region from the specified `list`. The `region_split()` routine takes one region and splits it into two regions. The `size` parameter specifies the offset from the beginning of the original region where the split is to occur.

Page Tables

The kernel keeps track of the page tables present in the system by maintaining a list of page table records. Figure 2 illustrates the page table record data structure and the associated page tables. The page table records are kept in a single-linked list using the `next` field. If multiple threads are executed within a single protection

Table 2: Region management routines

`valid_region(start)`	Check whether pointer corresponds to the starting address of a region
`region_clear(region)`	Initialize a region data structure
`region_insert(region, list)`	Insert a region into a double-linked region list
`region_remove(region, list)`	Remove a region from a region list
`region_split(region, size)`	Split a region into two regions

domain, the `refcnt` field tracks the total number of threads within the domain. The `pd` field points to the actual page table data structures. Note that there is a single pointer to the page tables themselves. This design implies that the page tables are arranged contiguously in memory. An assessment of current MMU implementations in several popular CPU architectures indicates that this is a reasonable assumption. More details on the page table structures of several popular processor architectures are given in Section 4.

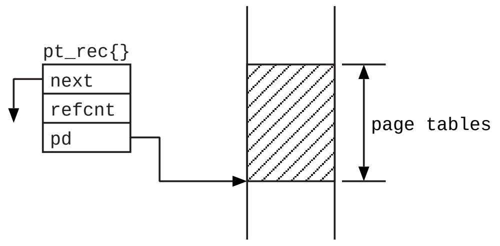

Figure 2: A page table record and its associated page tables

The first four routines in Table 1 implement the basic protection mechanism. They enter and remove address mappings to and from page tables, respectively. All four of these routines operate on a set of page tables specified by their first parameter, `pt`. The `vm_map()` routine provides the fundamental operation of inserting a mapping for a single page into a set of page tables. The page found at the location specified by the `start` parameter is inserted with the protection attributes specified by the `attr` parameter into the specified page tables. `vm_map_range()` is provided for convenience as a front-end to `vm_map()` to allow mapping a sequence of contiguous pages with a single call. The `start` parameter specifies the address of the first of a contiguous sequence of pages. The `len` parameter specifies the length, in bytes, of the page sequence. The initial implementation or the `vm_map_range()` routine makes calls to `vm_map()` for each page in the specified range. This implementation is obviously ripe for optimization.

The `vm_unmap()` routine balances `vm_map()` by providing the removal of a single page mapping from a page table. The `page` parameter specifies the starting address of the page that is to be unmapped. `vm_unmap_range()` is provided as a front-end to `vm_unmap()` to allow removal of a sequence of contiguous entries with a single call. `start` specifies the starting address of the page sequence and `len` gives the byte length of the page sequence to be unmapped. The `vm_unmap_range()` routine also make individual calls to `vm_unmap()` for each page in the specified range and can also be optimized.

The Kernel Map

In some virtual memory system designs that provide separate address spaces, the kernel has been maintained in its own address space. In the *Roadrunner* system, the memory used to hold the kernel and its associated data structures are mapped into all the page tables in the system. Kernel memory protection is provided by making these pages accessible only when the CPU has entered supervisor mode and that happens only when an interrupt occurs or a system call is made. The result is that system calls require only a transition from user to supervisor mode rather than a full context switch.

The kernel map is an array of kernel map entries where each entry represents a region that is entered in the kernel map. Figure 3 illustrates the structure of one of these kernel map entries and the region of memory that it represents. The `start` and `len` fields track the starting address and length of the region. The `attr` field stores the attributes that are associated with the pages in the region. This information is used when the pages are entered into a set of page tables by the `vm_kmap()` routine.

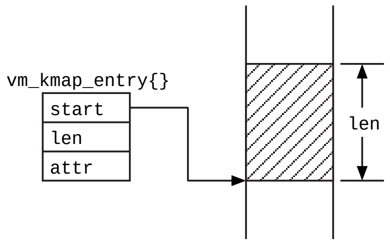

Figure 3: A kernel map entry and its associated memory region

The last three routines in Table 1 provide the API for managing the kernel map. The `vm_kmap_insert()` routine enters a kernel map entry, specified by the `entry` parameter, into the kernel map. The `vm_kmap_remove()` routine removes a previously entered kernel map entry, also specified by the `entry` parameter, from the kernel map. The `vm_kmap()` routine causes a set of page tables, specified by the `pt` parameter, to be updated with the current kernel map entries.

Page Faults

The most important function of page faults in a system using separate virtual address spaces is demand paging. Demand paging of user code can also be done using this approach under two additional conditions. First, all of the physical memory required to hold the program must be allocated when the program is started. Second, code relocation needs to be performed on-the-fly when sections of the program were loaded on-demand.

If demand paging of user code is implemented, page fault handling is similar to systems where separate virtual address spaces are used. When a page fault occurs, the appropriate kernel service determines whether the fault occured due to a code reference and if so, it loads the appropriated section of code and restarts the faulting process.

The initial *Roadrunner* implementation does not currently support demand paging of program code. As such, page fault handling is trivial, resulting in the termination of the process that caused the fault.

Kernel Memory Managment

Table 3 lists the routines that are used by kernel subsystems and by applications through the system call interface to allocate and free memory from the global heap.

Table 3: Kernel memory management routines

`malloc(size)`	Allocate a region of memory in the calling process's protection domain
`free(start)`	Free a region of memory previously allocated to the calling process
`kmalloc(size)`	Allocate a region of memory for the kernel
`kfree(start)`	Free a region of memory previously allocated to the kernel

The `malloc()` routine performs an allocation on behalf of a process by performing a first-fit search of the free list. When a region is found that is at least as large as a request specified by the `size` parameter, it is removed from the free list using the `region_remove()` routine. The remainder is split off using the `region_split()` routine and returned to the free list using the `region_insert()` routine. The region satisfying the request is then mapped into the protection domain of the calling process using the `vm_map_range()` routine.

The `free()` routine returns a previously allocated region to the heap. After obtaining the region corresponding to the specified `start` parameter using the `valid_region()` lookup, the `region_insert()` routine is used to enter the region into the free list. The inserted region is then merged with its neighbors, both previous and next if they are adjacent. Adjacency means that the two regions together form a contiguous sequence of pages. Merging is done to reduce fragmentation.

The `kmalloc()` routine allocates some memory on behalf of the kernel. After obtaining a region from the heap in a manner similar to the `malloc()` routine based on the specified `size` request, an entry is placed into the kernel map using the `vm_kmap_insert()` routine. This action records the new region as an element of the kernel map. Subsequent calls to `vm_kmap()` will cause the new region to be accessible as part of the kernel when a process is running in supervisor mode.

The `kfree()` routine first removes the kernel mapping for the region specified by the `start` parameter using `vm_kmap_remove()`. The region is then placed back on the free list using `region_insert()`.

4 Assessing Various Memory Management Architectures

The *Roadrunner* memory protection system is designed to be portable. It makes few assumptions about the underlying hardware and can be ported to a variety of architectures. In this section, some of the details encountered when implementing this memory protection mechanism on several processors are presented. The initial implementation effort has focused on the IA-32 architecture but some design elements necessary for two other CPU architectures are also discussed.

These details are hidden by the interface given in Table 1. The interface to the memory protection subsystem remains the same when *Roadrunner* runs on different processors but the underlying implementation of the interface is different.

Intel IA-32

The initial implementation effort has been focused on the Intel IA-32 (or x86) Instruction Set Architecture (ISA) [3]. This architecture provides hardware support for both segmentation and paging. The segmentation features are bypassed completely. This is done by initializing all the segment registers to address the entire 4 Gbyte address space.

The hardware has a fixed page size of 4 Kbytes and a two-level hierarchical page table structure is used. The first level is a single page called the page directory. Each entry in the page directory contain a pointer to a second-level page table. Each entry in the page table contains a pointer to a target physical page.

Since the *Roadrunner* memory protection mechanism maps logical addresses one-to-one to physical addresses, the kernel need only maintain in the worst case, the page directory and enough pages to cover the physical memory in the machine. As an example, each second-level page table addresses 4 Mbytes of physical memory so a machine with 64 Mbytes of main memory requires 1 page for the page directory and 16 pages (or 64 Kbytes) for second-level page tables or a total of 17 total pages for each set of page tables.

The current implementation allocates page tables statically using this worst-case formulation. Future enhancements will provide demand-allocation of second-level page table pages. Demand-allocation occurs when a `vm_map()` operation is executed and the page directory indicates that no second-level page table has been allocated to handle the address range in which the specified address to be mapped falls.

Motorola PowerPC

The Motorola PowerPC CPU [5] is a Reduced-Instruction Set Computer (RISC) ISA that is popular in the embedded world. This architecture provides three hardware mechanisms for memory protection, segmentation, Block Address Translation (BAT), and paging. Segmentation is bypassed in a manner similar to that of the IA-32 ISA. BAT is intended to provide address translations for ranges that are larger than a single page. This mechanism, included to allow addressing of hardware elements such as frame buffers and memory-mapped I/O devices, is also bypassed by initializing the BAT array such that no logical addresses match any array element.

The PowerPC also uses a 4 Kbyte page size but instead of a two-level hierarchy, implements a hashed page table organization. Some bits of the logical address are hashed to determine the page table entry that contains the corresponding physical address. Page Table Entries (PTEs) are organized into Page Table Entry Groups (PTEGs). There are two hashing functions, primary and secondary, that are used to determine which PTEG a logical address resides in. The primary hash function is applied first and the resulting PTEG is searched for the matching PTE. If the search fails, the secondary function is applied and a second PTEG is searched. If either search succeeds, the resulting PTE yields the corresponding physical address. If both searches fail, a page fault occurs.

Since the implementation of the page tables is irrelavent behind the generic interface, this organization does not provide any functional difference to that of the IA-32 two-level hierarchy. However, it does have implications for physical memory usage. For the hardware to locate a PTEG in main memory, all pages in the page tables must be laid out contiguously in physical memory. This constraint does not exist in the IA-32 design, i.e. the location of second-level page table pages can be arbitrarily placed in physical memory.

For separate, demand-paged, virtual address spaces, the hash table organization has the advantage of allowing the overall size of the page tables to be varied with respect to the desired hit and collision rates. Table 7-21 in [5] discusses the recommended minimums for page table sizes. In general, these sizes are larger than those required for the IA-32 CPU, e.g. 512 Kbytes of page table is recommended for a main memory size of 64 Mbytes. Note however, that these recommendations are tailored to providing a separate 32-bit logical address space to each process.

In the *Roadrunner* mechanism, a different calculation is required. For a given amount of main memory, the page table should be given a *maximum* size that allows all of the physical memory pages can be mapped simultaneously. This value is the same as the size quoted for the IA-32 implementation. In a manner analogous to demand-allocation of second-level page tables, the size of the hashed page table can be tailored dynamically to focus on the actual amount of memory allocated to a given process, rather than the entire physical memory.

MIPS

Another popular RISC ISA is the MIPS core [8]. There are a number of CPUs that are based on the MIPS ISA but they fall into two broad categories, based on either the MIPS R3000 and R4000 cores. There are variations on the specifics of the PTEs for these two cores but the basic principles are the same in both.

While the IA-32 and PowerPC architectures provide hardware mechanisms for loading a PTE into a Translation Lookaside Buffer (TLB) entry when a page fault occurs, the MIPS architecture requires the operating system kernel software to perform this function. If an address is asserted and a corresponding TLB entry is present, the physical address is aquired and used to access memory. If no matching TLB entry is present, a page fault is signaled to the operating system kernel.

The MIPS PTE is divided into three basic elements, the Address Space Identifier (ASID), the Virtual Page Number (VPN), and the offset within the page. In *Roadrunner*, the ASID can be used to reference the protection domain, the VPN references a page within the domain and the offset completes the physical address reference. The VPN is the upper bits of the logical address and it is used as a key to lookup the corresponding physical page address in the page tables.

The MIPS architecture places no requirements on the page table structure in main memory. The operating system developer can tailor the page table structure as desired. In *Roadrunner*, a system targeted at resource-constrained applications, the goal is to minimize main memory usage.

Management of the TLB is explicit and since domains contain unique addresses, identifying entries to be discarded when the TLB is updated due to a context switch is made easier.

5 Measuring Memory Protection Performance

The major advantage of the memory protection design presented in this work is its simplicity. The *Roadrunner* design requires roughly an order of magnitude less code to implement the same function. The memory management code in the 2.2.14 Linux kernel consists of *approximately* 6689 lines of C found in the `/usr/src/linux/mm` and `/usr/src/linux/arch/i386/mm` directories as shipped with the Redhat 6.2 distribution [9]. The count for the Linux implementation explicitly excludes code that performs swapping. The *Roadrunner* memory protection implementation consists of 658 lines of C source code in a set of files found in the the `roadrunner/sys/src/kern` directory and 74 lines of IA-32 assembly found in the `roadrunner/sys/src/kern/asm.S` source file.

The advantage of this approach would be unimportant if the performance of the design was unacceptable. A series of measurements presented in Table 4 demonstrate that this approach presents extremely good performance. The set of measurements was obtained using an Intel motherboard with a 1.7 GHz Pentium 4 CPU, 512 Kbytes of second-level cache, and 256 Mbytes of RDRAM. All measurements are the average of a large number of samples taken using the Pentium timestamp counter, which runs at the resolution of the processor clock.

Table 4: Performance of *Roadrunner* Memory Protection Operations

Operation	Average Execution Time (μsec)
`vm_map()`	0.15
`vm_map_range()`	4.22
`vm_unmap()`	0.11
`vm_unmap_range()`	0.41
`vm_kmap_insert()`	1.27
`vm_kmap()`	123.0
`malloc()`	1.45
`free()`	0.91
`kmalloc()`	3.24
`exec()`	1710.0
Context switch	1.71

The basic memory management routines presented in Table 1 provide the building blocks for other routines and as such, need to operate very quickly. The all-important `vm_map()` and `vm_unmap()` both exhibit execution times between 100 and 150 *nanoseconds*. Even with the naive implementation of `vm_map_range()` where each page table entry requires a call to `vm_map()`, an average of 256 pages (for these measurements) can be mapped in an average time of approximately four microseconds. Clearing page table entries is faster. Setting up entries requires several logic operations to set appropriate permissions bits. Clearing simply zeros entries yielding the approximately 410 *nanoseconds* to to clear the entries for an average of 196 pages (for these measurements).

The kernel memory management routines presented in Table 3 fall into two categories. `malloc()` and `free()` are called very often on behalf of applications. These routines need to perform better than the corresponding kernel routines, `kmalloc()` and `kfree()`, which are only used within the kernel to allocate data areas for kernel subsystems. All the routines have latencies between 1 and 2 microseconds except for `kmalloc()`. This operation is expensive since it requires a call to `vm_kmap_insert()`.

The latencies associated with `vm_kmap()` and `exec()` represent the most expensive operations presented here. The prior operations represents approximately 40 calls to `vm_unmap_range()`. The latter operation is a hybrid system call representing a combination of the semantics associated with `fork` and `execve` in typical UNIX-like systems. `exec()` loads a program and starts it running as a new process in its own protection domain. This operations requires a variety of initializations in addition to setting up the protection domain. The value measured here excludes the load time of the program code.

6 Related Work

The design of the *Roadrunner* memory protection subsystem and the structure of this paper were influenced most heavily by Rashid, *et. al.* in [7]. However, the Mach VM system provides separate, demand-paged, virtual address spaces as compared to the single address space in *Roadrunner*. Also, the *Roadrunner* design does away with the address map abstraction. The single address space allows the use of the generic interface given in Table 1, which is closer to the Mach pmap interface, to be substituted for the hardware-independent address map data structure and routines.

There is a significant body of work in the area of Single-Address-Space Operating Systems (SASOS) that is typified by the Opal system developed at the University of Washington [1]. These systems generally seek to provide a single *virtual* address space, that is, they still perform a translation between the logical and physical address.

The EMERALDS micro-kernel developed at the University of Michigan [10] is an example of a kernel designed specifically for the resource-constrained environments in small-to-medium sized embedded systems. The kernel includes a traditional multi-threaded, process-oriented concurrency model. The system maps the pages associated with the operating system kernel into the page tables of every process and uses the user/supervisor transition to implement system calls in a manner that is analogous to the use of the kernel map in the *Roadrunner* design.

The Mythos microkernel [6] was developed as a threads based system to support the GNAT (Gnu Ada 9X translator) project. The kernel provided a Pthreads-based interface for concurrency to applications through the kernel API in a manner similar to the *Roadrunner* kernel. The system did not provide memory protection however.

7 Conclusion

This work has presented the design and implementation of a new memory protection mechanism that provides traditional levels of protection using significantly less code than designs that perform address translations. This design can be added to existing embedded operating system kernels that do not currently provide memory protection easily due to its small implementation effort. In addition, the design is portable among CPU architectures, which makes it even more attractive for use in these kernels, since they tend to be available for several different processors.

The *Roadrunner* operating system, in which the first implementation was performed, is free software, available under the GNU General Public License (GPL), on the World-Wide Web at `http://www.cornfed.com`.

References

[1] Chase, J., *et. al.*, "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems, 12,* 4, 1994, pp. 271-307.

[2] Cornfed Systems, Inc., *The* Roadrunner *Operating System*, 2000.

[3] Intel Corp., *80386 Programmer's Reference Manual*, 1986.

[4] Miller, F. W., "pk: A POSIX Threads Kernel", *FREENIX track, 1999 USENIX Annual Technical Conference*, 1999.

[5] Motorola Inc., *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, 1997.

[6] Mueller, F., Rustagi, V., and Baker, T., *Mythos – a micro-kernel threads opertating system*, TR 94-11, Dept. of Computer Science, Florida State University, 1994.

[7] Rashid, R., *et. al.*, “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, *Proc. of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 1987.

[8] Sweetman, D., *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.

[9] Redhat, Inc., *Redhat Linux 6.2*.

[10] Zuberi, K. M., Pillai, P., and Shin, K. G., “EMERALDS: a small-memory real-time microkernel”, *17th ACM Symposium on Operating System Principles (SOSP99)*, 1999.