An Implementation of the Solaris Doors API for Linux

Jason Lango (jal@cs.brown.edu)

Fall/Spring 1998

1 Introduction

The purpose of this paper is to describe the implementation of the Solaris Doors API on the Linux[1] operating system and to show the relative performance improvements that doors can offer versus some of the standard UNIX IPC mechanisms.

2 Motivation

Linux is a freely available UNIX-like[2] operating system for many varied hardware platforms. As of this writing, Linux is supported on the DEC Alpha, Intel x86, Sun SPARC and UltraSPARC, Motorola 68000 and PowerPC, Advanced RISC Machines ARM, and SGI MIPS processors. The list of supported hardware continues to grow at a rapid pace, mostly due to the strong programming support garnered by the open software development model and increased media and commercial support. Linux is used in a wide variety of applications, from the desktop to distributed-memory [8] and cluster-based [3, 4] supercomputing. The growing popularity of the Linux operating system, combined with the ability to freely publish source code modifications to the operating system, motivated its use in this project.

3 The Linux Architecture

Figure 1 shows a high-level view of the architecture of the Linux kernel.

Linux has the notion of virtualized file systems, files, inodes, directory cache, memory mapping objects, page tables, etc. The virtual file structure (struct **file**) is used for each

[1] At the time of writing the Doors implementation, Linux was at version 2.1.79 of the kernel. Presumably most of what is discussed here will be relevant until at least 2.3.x.

[2] While there do exist Linux distributions which have the official POSIX.1 certification, most do not due to the expense of obtaining such certification.

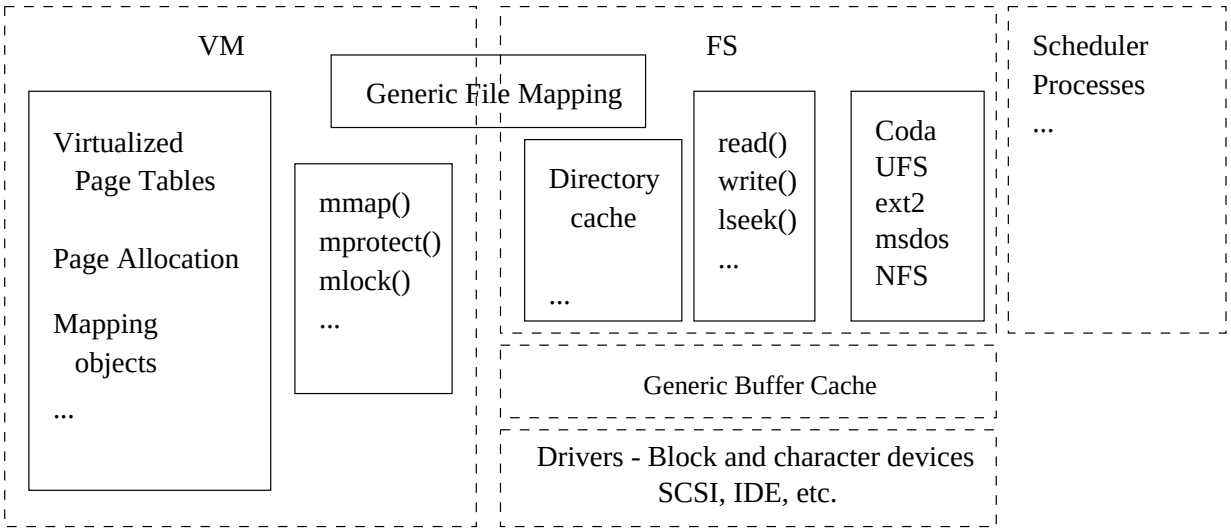

Figure 1: Linux Kernel Architecture

open file referenced by a particular process (including special files, such as pipes, sockets, etc.). The virtual inode (struct **inode**) is used for each representation of a shared file, much as the vnode is used in SVR4 and BSD. The virtual memory mapping object (struct **vm_area_struct**) is used to represent an instance of a particular type of memory mapping, such as an anonymous mapping (straight to physical memory), generic file mapping, shared memory mapping (POSIX and System V), etc. The virtual page tables (**pgd_t**, **pmd_t**, **pte_t**) are used to define a method for architecture-independent access to the actual memory management hardware through an abstract three-level page table. The directory cache is a global resource, whose entries can be customized by particular file system implementations (but generally are not).

There are several books available which contain greater detail about the architecture of the Linux kernel [5, 2]. This implementation of Solaris Doors on Linux implements a virtual file structure for the door, a device driver for Doors API method invocation, and uses the directory cache in order to implement a file-on-file mapping similar to the SVR4 STREAMS *fattach(3)* function (see below).

4 Introduction to the Solaris Doors API

The Solaris Doors API is essentially an RPC mechanism, which makes use of the UNIX notion of the filesystem as a universal name space and has built in support for multithreading.

The fundamental building block of this RPC mechanism is the *door*. Abstractly, the door can be thought of as a service procedure or object, upon which a thread can invoke a

method (or function call). The door is often referred to in the literature [6, 7] as "describing" a particular procedure in a remote server.

Doors are made visible to the applications programmer as standard UNIX file descriptors (or "door descriptors"). To make a door visible to other applications, a process may attach an existing door descriptor to an existing regular file in the UNIX file system, using a standard SVR4 mechanism for associating STREAMS file descriptors with files in the file system.

What follows is a list of the relevant library routines encompassing the Solaris Doors API, and the two related routines *fattach(3)* and *fdetach(3)*. Most detail is omitted, since it is covered at length in the Solaris documentation [7], but there should be enough to provide a general overview of the supported functionality.

- *door_create(3)* is used to create a new door. The function takes as its arguments a server procedure, data *cookie*, and flags. The data cookie is a value passed to the server procedure every time this door is the target of a *door_call(3)* method invocation. The result of the function call is a door descriptor, which can subsequently be used for *door_call(3)* invocations, attached to a file with *fattach(3)*, or passed directly to other processes (again via *door_call(3)*). The process which calls *door_create(3)* is the same process which handles *door_call(3)* invocations. Any running process can be a door server, simply by calling *door_create(3)*.

- *door_call(3)* is used to invoke a procedure call in the remote server process which created the door using *door_create(3)*. The function takes as its arguments a door descriptor, representing the procedure to call in the server, a memory block (containing the arguments for the procedure call), and a set of door descriptors which are to be passed to the server process. The door descriptor is the only required argument, the memory block and set of doors are each independently optional. The result of the function call is either an error code, or another memory block and set of door descriptors from the server process (again, both are optional, at the server's discretion). The client process may provide a memory buffer in which the results (if any) from the server will be placed. If the client-provided result buffer is too small to hold the results from the server, the operating system will allocate a new memory block large enough to hold the results of the call.

- *door_return(3)* is used within a server procedure to return values to the client process. It is only valid to call *door_return(3)* within a server procedure called by a *door_call(3)* invocation. The arguments that this function takes are a memory block and set of doors to be returned to the client process (as the results of *door_call(3)*). Both arguments are optional.

- *fattach(3)* is used to attach a door descriptor to a regular file in the UNIX file system.

The function takes as arguments the door descriptor and a pathname to the regular file.

- *fdetach(3)* is used to detach a door descriptor from a regular file (attached via *fattach(3)*). The function takes the pathname to the file as its argument.

- *door_revoke(3)* is used to revoke access to a door. All future *door_call(3)* invocations through any door descriptor referring to this door will fail.

- *door_info(3)* is used to get information about a door. This function returns the process id of the server which handles *door_call(3)* invocations, the server procedure (address in the server process), the data cookie for this door, a system-wide unique identifier for the door, and certain other attributes, such as whether the current process is the server for the door, whether the door has been revoked, etc.

- *door_cred(3)* can be used within a server procedure to get the credentials of the client process giving the current *door_call(3)* invocation. This function returns the user id, group id, and process id of the calling thread's client.

- *door_server_create(3)* is used to register a function which will be called in the server process whenever the kernel thinks that a new thread should be created. This allows the programmer the ability to control how many threads are actually created in the server and the initial parameters of the server threads (e.g. scheduling parameters, signal dispositions, etc.).

- *door_bind(3)* is used to bind the calling thread to a particular door. The calling thread will be one of a set of threads which will specifically handle invocations on this door. The door must have been created will the "private" attribute (via *door_create(3)*) indicating that it has a private pool of server threads.

- *door_unbind(3)* is used to remove the calling thread from a particular door's private server pool.

The Doors API was engineered with certain performance optimizations in mind. Server threads will be created in the calling process in proportion to the load on the server (at most one per concurrent request) [6]. The server may control how many threads are created via *door_server_create(3)*. Since the programmer doesn't necessarily provide the memory block to receive arguments to and from *door_call(3)*, the kernel may optimize the transfer of large arguments by mapping the underlying pages of memory into the target process and copying a page only if a process attempts to modify it. The kernel may also use handoff scheduling to optimize *door_call(3)* invocations [6].

5 Summary of Work

The majority of the work involved in this project is in the implementation of a subset of the Solaris Doors API for Linux. *Omitted* from this implementation is maintaining private server pools for doors (via *door_bind(3)* and *door_unbind(3)*), server thread cancellation when a client thread is interrupted, using handoff scheduling to optimize *door_call(3)*, and sending a DOOR_UNREF message when only one open door descriptor remains pointing to the door. Memory mapping large arguments between client and server was implemented, which is not implemented in Solaris 2.6 [6].

Benchmarks were taken comparing the speed of doors on Linux versus other IPC mechanisms.

6 Implementation of the Doors API on Linux

6.1 Representing the door

The Linux kernel has a virtualized filesystem, similar to the SVR4 VNODE/VFS subsystem. The Linux **inode** structure is similar in spirit to the SVR4 **vnode**, in that it is an in-memory representation of an abstract inode.

The door is represented in the kernel code as a **door** structure and associated door inode. The door inode has special member functions for open and close operations, and refers to the underlying **door** structure via the filesystem specific information pointer (the *generic_ip* member).

When a new file descriptor refers to a door inode, the kernel will automatically call **door_open()**, which will increase the reference count on the door. When a file representing a door is closed, the kernel will call **door_release()**, which will decrease the reference count on the door. When the reference count goes to 1 and the door has the DOOR_UNREF attribute, a special message should be sent to the door's server process[3].

When a doors library call refers to a particular door via a door descriptor, the doors kernel code will look in the current process' file descriptor table, find the appropriate **inode** (indirectly through a **file** structure and then its **dentry** structure), then get the underlying **door** through the filesystem specific data pointer.

6.2 Sending a request: *door_call(3)*

In order to perform a *door_call(3)*, the client first gets the door inode from its file descriptor table, allocates a **door_message**, initializes the **door_message** with the door, current task, and arguments, enqueues the message on the door's message queue, then wakes a server and goes to sleep itself.

[3]Not currently implemented.

If the client is interrupted while waiting for the results of the *door_call(3)*, the client thread will first set the *dm_client* member of the **door_message** to NULL, then return an error code.

On a successful reply, the client thread destroys the **door_message** and frees any associated buffers.

6.3 Receiving and replying: *door_return(3)*

When the user library calls the kernel interface to *door_return(3)*, the calling server thread first checks to see if it has a request in progress. If so, it will copy its output arguments (if any) and wake the client thread. If the dm_client member of the door_message structure is NULL, the server knows that the client has been interrupted and destroys the message, freeing any reply buffers.

If the kernel mapped memory into the process address space during the previous request (the one to which we have just responded), the server thread now unmaps this memory.

The server thread then looks for a request to handle. If no requests are available, the server thread sleeps on a wait queue specific to its server process.

When the server thread gets a request to handle, it performs any argument copying that is necessary, caches the client credentials, frees any leftover buffers which aren't needed during the handling of the request, then returns to user-level.

At this point, the server thread appears to be returning from the last call to *door_return(3)*. The doors user library notices that the DOOR_RETURN ioctl has returned and does a *longjmp(3)* to the top-level call to *door_return(3)*, returning to the top of the stack.

6.4 Identifying the current server process

The function **find_server()** is used by most of the doors kernel routines to identify the *door_server* structure for the current task. This function is complicated by the fact that threads in Linux currently[4] don't share the same process id. The process id of the thread which created a door cannot uniquely identify all other threads within the same process.

Linux's threads are essentially variable weight processes which share system resources [1], such as the file descriptor table, virtual address space, etc. In the Linux kernel, a variable weight process is called a *task*.

In order to provide what is seemingly the most generality, **find_server()** uniquely identifies a process as the set of tasks which have the same file system structure (**fs_struct**).

[4] As of version 2.1.79.

6.5 Creating threads for incoming requests

In order to make as few assumptions as possible about the threads library, threads are not created from within the kernel. When the first *door_create(3)* is issued by a potential server process, a special thread is created in that process. This thread, considered internal to the doors user-level implementation, spends most of its time sleeping on a kernel synchronization object. When the kernel needs to create a thread, it wakes up this thread, which proceeds to call the user-level server creation function; possibly the one which the user installed via *door_server_create(3)* or the internal doors library function which creates a default pthread which immediately calls *door_return(3)*.

6.6 File on file mounting: *fattach(3)*

Linux, unlike Solaris, does not have a built-in notion of STREAMS devices. Furthermore, there is no generic facility to accomplish file on file mounting, as in *namefs* in Solaris [6]. *fattach(3)* and *fdetach(3)* were implemented as wrappers around part of the doors kernel interface.

do_door_fattach() implements file on file mounting by using Linux's directory cache, much as the implementation of actual file system mounting works. Each **dentry** (directory cache entry) has a member *d_covers* and a member *d_mounts*. These members point to further **dentry** structures. A file on attached on top of another file will have a **dentry** whose *d_covers* field is non-NULL. A file with a file attached on top of it will have a dentry whose *d_mounts* is non-NULL. We raise the reference count on both **dentry** structures, such that neither will leave the directory cache until the file is unmounted (via *fdetach(3)*). *open(2)* will automatically scan the *d_mounts* member of a **dentry** hit in the directory cache, so when the regular file is opened it will automatically select the file mounted on it.

do_door_fdetach() simply reverses this operation, setting *d_mounts* and *d_covers* on both **dentry** structures to NULL, then lowering their reference counts.

6.7 Avoiding implementing system calls: */dev/door*

When writing the library and kernel code, it would be ideal to have an implementation which would be (1) *architecture independent* (that is, it should ideally simply require a recompile to run on any machine architecture that Linux supports, now and in the future) and (2) relatively *easy to program and debug*.

To satisfy (1), the decision was made to make the user library interface with the kernel via a character special device, namely "/dev/door". The entire user to kernel interface consists of *ioctl(2)*s on this character special device, essentially passing messages to the device driver which provides the entire kernel-level doors implementation. This has the advantage that we avoid having to write new system calls, which is architecture-dependent in Linux.

Implementing the doors kernel interface as a device driver has the additional advantage that it can be written as a loadable kernel module, satisfying (2) in that debugging time is decreased by not having to reboot the machine for each iteration of the code, compile, debug cycle and also being able to track the latest version of the kernel (since none of the kernel code proper needed to be modified).

The trade-off in this approach is that there is a small performance penalty associated with making all of the door calls *ioctl(2)*s on "/dev/door", but it should not be a difficult task to make them system calls in order to optimize them for a particular architecture.

6.8 Mapping arguments: door_vm_copy()

When either the result buffer in the client is too small to accept an output argument or set of door descriptors, or the input arguments to the server procedure exceed a predefined constant size, **door_vm_copy()** is called to map the arguments from the source process address space to the destination process address space.

There are a number of special cases which must be handled by this function and associated helper functions:

- *Unmapped regions* - if part of the source region is unmapped in the source process address space, the copy must fail.
- *Non-present pages* - if a page is not present (i.e. is on a swap device or resides in its backing file), it must be faulted in.
- *Mapped files* - if part of the source region in the source process address space is a memory mapped file object, the pages cannot be shared, since changes to the file would cause the server to witness changes in its input arguments.
- *Page-alignment* - if the source buffer doesn't start or end on a page boundary, the first and last pages must be explicitly copied such that no extra information is given to the target process, thus causing the copy to be a security hazard.

The algorithm that **door_vm_copy()** uses is fairly straightforward:

1. Find the first source mapping object, and verify that it is valid.
2. If the start address isn't page aligned, create a new page in the target process which contains the appropriate portion of the first source page, faulting in the source page if necessary.
3. For each subsequent full page in the source range, find the source mapping object. If the source mapping object is a file, fault in the page and copy it to a new page in the target process. If the source mapping is a regular page, map it copy-on-write into the target process.

4. If the end address isn't page aligned, create a new page in the target process which contains the appropriate portion of the source page, faulting in the source page if necessary.

To optimize the mapping of the middle pages, one may find the remaining contiguous pages within the source mapping object and map them all using the same flags and mapping method (e.g. copy-on-write or real copy).

7 Benchmark Results

A benchmark was run on a Digital ALPHAstation 255 (single 300 MHz Alpha processor) running Linux 2.1.79, comparing the speed of passing a message and waiting for a response using a door to the equivalent programs using pipes and ONC-RPC. Figure 2 shows the results of the benchmark.

An equivalent benchmark was run on a Sun Microsystems Ultra-1 (one 200MHz UltraSPARC processor) running Solaris 2.6 (figure 3) and on an Intel Pentium 100 running Linux 2.1.72 (figure 4).

The benchmark shows how the response latency varies with the size of the message passed from the client to the server. The latency of the door implementation on Linux is comparable to that of the pipe implementation, until we reach a message size of 16384 bytes, at which point the kernel door implementation switches to mapping its arguments between the client and the server, thus affecting a big performance improvement over the pipe implementation.

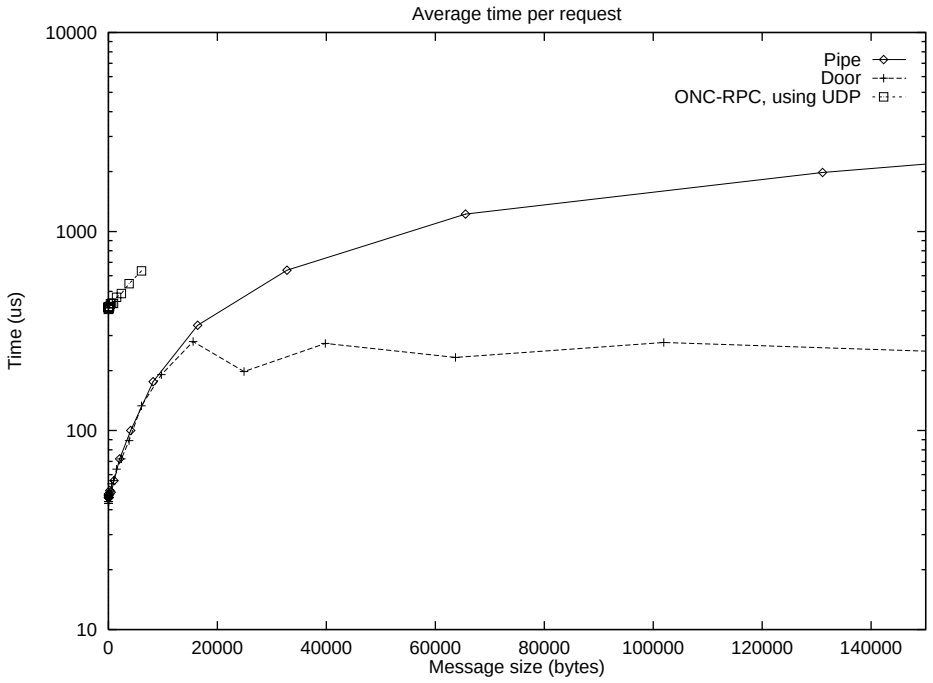

Figure 2: Doors versus standard UNIX IPC, Linux 2.1.79 on ALPHAstation 255

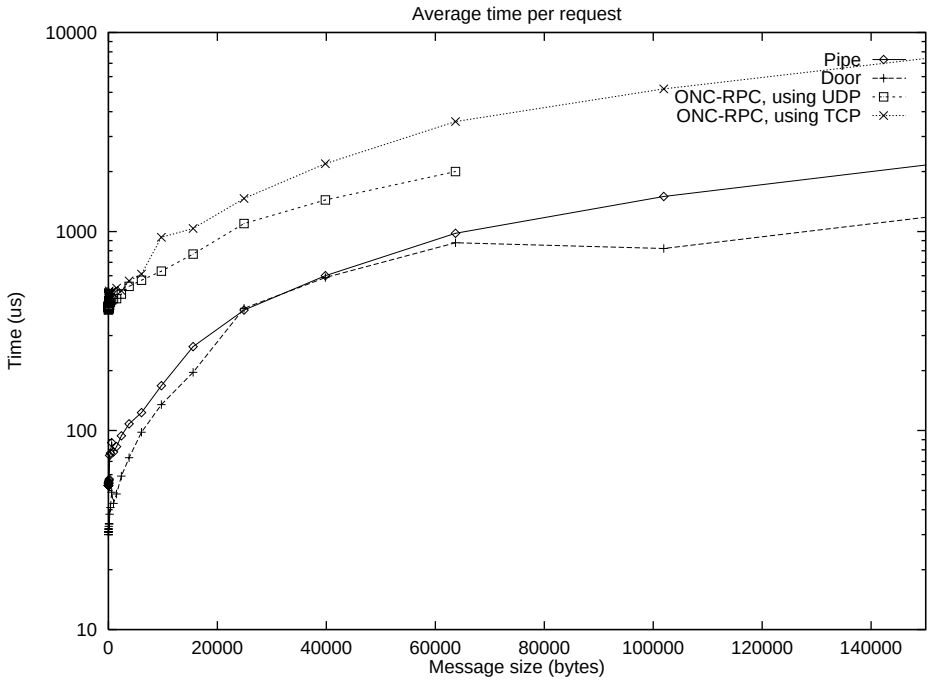

Figure 3: Doors versus standard UNIX IPC, Solaris 2.6 on Sun Ultra-1

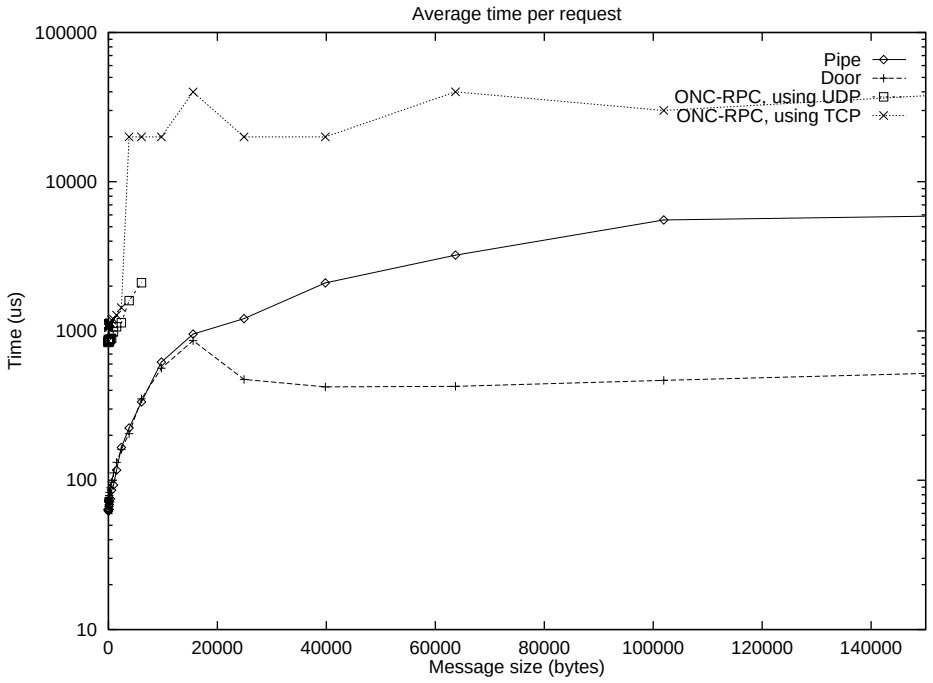

Figure 4: Doors versus standard UNIX IPC, Linux 2.1.72 on Intel Pentium 100

8 Conclusions

The Solaris Doors API is an interesting new IPC mechanism. Its chief advantages are in the optimizations which can be made in the kernel implementation and how it automates the job of thread creation (in the general case) for the server programmer.

The benefits of such an interface are compelling enough to motivate the implementation of the Doors API on operating systems platforms other than Solaris.

9 Future Work

Some features and optimizations which might be added to the Doors API for Linux follow.

- Explicitly passing shared memory objects via doors. Passing mutexes and semaphores via shared memory would be an option, as well.
- Offering an option similar to MAP_PRIVATE for the input and output arguments, i.e. the recipient could witness changes to the source memory pages, but modifications would cause the recipient to make a private copy of a page.
- Passing generic file descriptors via *door_call(3)*.
- Hand-off scheduling.

References

[1] Aral, Z., Bloom, J., Doeppner, T., Gertner, I., Langerman, A., and Schaffer, G., "Variable Weight Processes with Flexible Shared Resources," *Proceedings of the Winter 1989 USENIX Conference*, San Diego CA, January 1989, pp. 405-412.

[2] Beck, M., et al. Linux Kernel Internals. *Addison-Wesley*, Nov. 1997.

[3] Becker, D., Sterling, T., Savarese, D., Dorband, J., Ranawak, U., Packer, C. BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION. *Proceedings, International Conference on Parallel Processing*, 1995.

[4] Ridge, D., Becker, D., Merkey, P., Becker, T., Merkey, P. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. *Proceedings, IEEE Aerospace*, 1997.

[5] Rusling, D. The Linux Kernel. *Not yet published.* http://www.linuxhq.com/guides/TLK/index.html

[6] Voll, J. "Doors" in Solaris: Lightweight RPC using File Descriptors. *Sun Developer NEWS*, Vol. 1, No. 1, Fall 1996.

[7] Sun Microsystems, Inc. Solaris Manual Pages: Library Routines. *Solaris 2.6 Reference Manual AnswerBook*, 1997.

[8] Tridgell, A., Mackerras, P., Sitsky, D., Walsh, D. AP/Linux - A modern OS for the AP1000+. *Proceedings, Sixth Parallel Computing Workshop*, 1996.