Enhanced Resource Sharing in UNIX

J. M. Barton and J. C. Wagner
Silicon Graphics Computer Systems

ABSTRACT: UNIX provides a programming model for the user which gives an illusion of multiprocessing. On uniprocessors, this illusion works well, providing communication paths, process security and a simple programming environment. Unfortunately, this model is not powerful enough to take full advantage of modern multiprocessor hardware. This results from a design which uses data queueing and preemption to provide the multiprocessing illusion.

Recent proposals for *lightweight processes* in UNIX show that other programming models may be effectively supported as well. These models provide for virtual address space sharing between tasks by breaking the process into several lightweight contexts that may be manipulated more quickly than a normal process. Unfortunately, such models raise questions about support for normal UNIX semantics, scheduling performance, and kernel overhead.

This paper describes a new programming model which goes beyond simple address space sharing, providing a selection of shared resources while

An earlier version of this paper was delivered at the Winter 1988 USENIX Technical Conference (Dallas).

retaining the key parts of the UNIX process model. The concept of a process is retained along with most of its semantics. The fundamental resource shared is the virtual address space. In addition, open file descriptors, user and group IDs, the current directory, and certain other elements of the process environment may be shared. This makes for easy construction of useful servers and simple concurrent applications, while providing high performance support for more computationally intensive applications.

This implementation, labelled *process share groups,* allows normal process actions to take place easily, such as system calls, page faulting, signalling, pausing, and other actions which are ill-defined in other models. The paper describes the philosophy which drove the design, as well as details of the implementation, which is based on, and upwardly compatible with, AT&T V.3 UNIX. Performance of normal UNIX processes is maintained while providing support for high performance parallel programming.

1. Introduction

The success of the UNIX kernel is owed in part to the effective way in which it hides the resource sharing necessary in a multiprogrammed environment. A UNIX process is a highly independent entity, having its own address space and environment. Communication paths are restricted to low bandwidth queueing mechanisms, such as pipes, sockets and messages.

When moved to a multiprocessor, this model is overly restrictive. We would like to allow several processors to work on a problem in parallel using high bandwidth communications (shared memory, for instance). Even though this explicit sharing of data is thought of most often when working with parallel programing, there are other resources that can be shared usefully. For example, a network server could share file descriptors with several children. The server would perform security checks and open a socket descriptor to the client, and then pass this descriptor to a waiting child with a simple message containing the descriptor.

In order to provide a conceptual framework for resource sharing among concurrent processes, we have introduced an additional level of functionality into the normal UNIX process model. A process may gain access to this additional level of functionality through an interface called *process share groups.* Members of a *share group* can potentially share many resources previously thought of as private to a single process.

This interface is first demonstrated in an AT&T System V.3 kernel, and relies on modified region handling abilities to share a virtual address space. New file opens can be propagated to all processes, as well as modifications to the environment of a process (such as the *ulimit(2)* values). By extending the semantics of the UNIX process in an upwardly compatible way, a powerful new programming model has been developed.

2. *The Road to Resource Sharing*

In order to provide a simple model for multiprogramming, the designers of UNIX chose a model of independent processes, shared access to a file system, and limited communication using queues, such as pipes or signals (Figure 1). Such a decision was appropriate, since a queueing model simplifies multiprogramming support. This gives the programmer a model where resource sharing is performed at an abstract level outside the program. For instance, UNIX programmers seldom concern themselves with running out of disk space or memory, even though these are shared with many other processes in the system. Soon the construction of multiprocess applications became necessary both to manage complexity and to allow for higher performance. This need was especially driven by the desire to program multiprocessors and distributed systems effectively. In response, new and explicit resource sharing capabilities were introduced into the system. The implementors

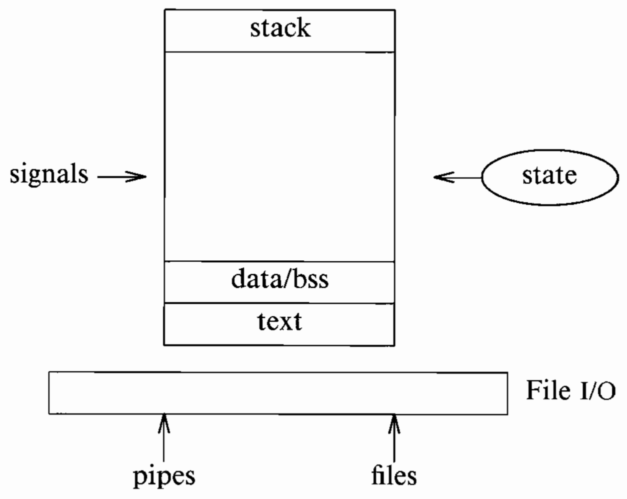
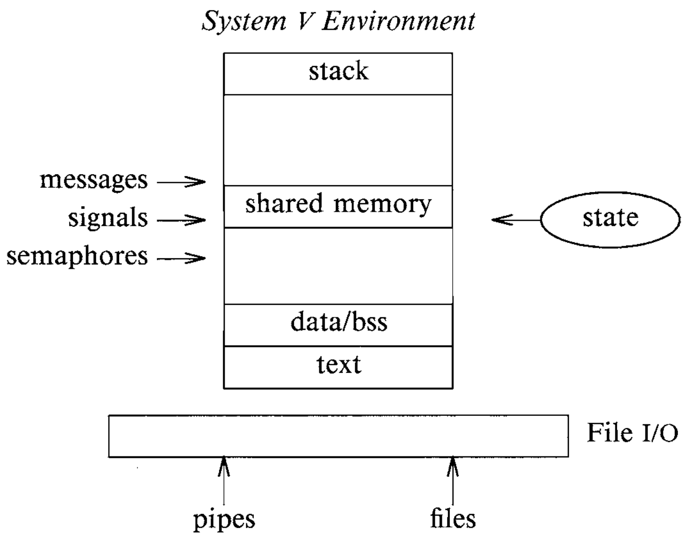


Figure 1: Version 7 Process Environment

of Berkeley UNIX were network oriented due to the desire to support Internet access in UNIX, and thus focused on distributed systems and advanced queueing schemes to support concurrent programming. This resulted in the *socket* interface, which supports well those applications whose synchronization requirements are not strict in the time domain, and where parallel execution is not necessarily required.

The implementors of System V, on the other hand, turned inward, focusing instead on local IPC mechanisms such as shared memory and semaphores, which support a more tightly coupled concurrent programming paradigm. Potentially, such work could avoid the limitations of queueing models and enhance parallel execution.

Although these models introduced important new interfaces, neither was adequate for support of tightly coupled parallel programming. The BSD model suffers from the inherent performance loss of any queueing and data copying model. The System V model suffers from synchronization mechanisms which require kernel interaction, which negates the impact of improved IPC mechanisms. In both models, the synchronization and data passing paths must be explicitly set up and managed by the application, which in many cases places awkward and unnatural



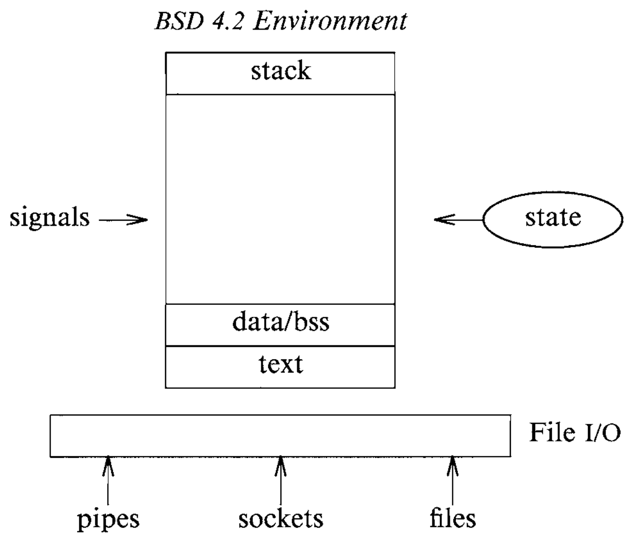


Figure 2: System V and BSD Process Environments

requirements on the programmer. The key weakness here is that kernel interaction is required for synchronization, which adds significant overhead for any tightly-coupled application.

To address the performance needs of tightly-coupled parallel programming, several *lightweight processing* models have been introduced into UNIX. These models provide intensive resource sharing by providing multiple threads of execution within a single process context. As an example, consider the Mach [Accetta et al. 1986] kernel, which introduced an implementation of lightweight processing labeled *threads* [Tevanian et al. 1987]. This model allows an address space to be shared among several threads of control. Each thread can execute independently in both kernel and user mode. Although independent execution adds additional cost for a threaded process, such as kernel context (the user area) and a kernel stack for each thread, a useful concurrent programming environment is provided (Figure 3).

Threads have serious limitations for many applications. An additional layer of complexity is added to the normal UNIX interface, requiring a totally new programming model and process behavior model. The programmer is saddled with two entirely different interfaces that must be managed, having little relation to each other. Although a parallel programming paradigm has been

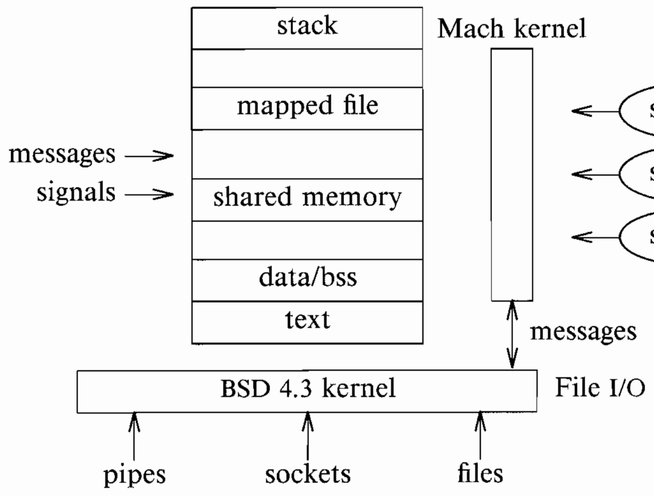


Figure 3: Mach Process Environment

added, inadequate interfaces are available for thread or process scheduling control and sychronization support.

Finally, although *threads* purport to reduce the overhead of task management, the resource overhead of extra stack and user area pages and indirect access to the user area makes this unlikely.

Although concurrency support in the UNIX model has improved dramatically over time, many of the proposed interfaces are clumsy or difficult to use. Ideally, a resource sharing interface should place little burden on the programmer while maintaining "expected behavior" of the system wherever possible.

So far, concurrency and parallel programming support have been presented as the driving forces behind improved resource sharing facilities. The following section delves into this area more deeply to give the reader a better understanding of why such changes are necessary.

3. Parallel Programming

To take maximum advantage of multiprocessors, it must be possible to create and execute parallel algorithms, and to get truly parallel execution of application code. Because of the data-sharing bandwidth necessary for applications to gain performance from parallelism, shared memory is usually the main data path. Sharing memory isn't enough, however. The sharing and sychronization mechanisms and other aspects of the environment must all work together to provide a useful programming model [Barton 1987].

A parallel programming model must take into account three important areas:

1. *Bandwidth.* The speed at which data can be passed between processes is a limiting factor in many algorithms. If the amount of data is small, and the rate of data passing low, then models such as sockets or pipes are useful. On the other hand, if the amount of data is large, or frequently accessed in parallel, then a shared memory model provides the highest bandwidth possible.

2. *Synchronization.* Synchronization penalties limit the performance of any parallel or concurrent application. Again, if long delays are tolerable, sockets or pipes provide a useful mechanism. If sychronization must occur quickly, some form of hardware supported lock is usually best.
3. *Environment.* The environment should support parallel programming easily. Powerful process scheduling control, ease of using shared resources and multiprocess debugging capabilites are all examples of the influence of the environment on ease of parallel programming.

Shared memory is seldom useful without a synchronization mechanism. For instance, pipes, System V messages or semaphores, sockets, or signals can be used to synchronize memory access between processes, but are all much lower bandwidth mechanisms than the memory itself, which is a liability in many applications. The best performance is obtained using some form of *busy-waiting* for synchronization. With *busy-waiting*, a lock is implemented which protects some resource. If the lock is free, the process immediately acquires it, and other processes that attempt to acquire the lock simply loop attempting to acquire the lock until it is freed. Hardware or software mechanisms are used to guarantee that only one process ever acquires the lock at a time. The assumption of such a scheme is that the resource only needs protecting for a short period of time. With hardware support for *busy-waiting*, synchronization speeds can approach memory access speeds.

The key component of the environment is the underlying operating system which implements the abstract programming model. Because a modern operating system such as UNIX must constantly react to internal and external events, the scheduling and context switching mechanisms are extremely important in insuring that parallel programs progress efficiently. Even if a multiprocess application limits itself to the number of processors available, the best performance will occur only if all tasks in a program are actually running in parallel for a significant amount of time. The kernel needs to take steps to insure that such tasks run in parallel whenever possible.

Dynamic creation and destruction of processes must be possible, and getting an existing process started on a new task must have minimal overhead. In a lightweight processing model, creation of a new task is often an order of magnitude faster than creation of a process. For instance, the Mach kernel can create and destroy threads at 10 times the rate of the *fork()* system call. This is irrelevant, however, since parallel programs tend to use a static number of tasks, and these tasks can be preallocated, which avoids dynamic startup costs. The scheduling model used in such applications is *self-scheduling*, in which an independent task waits for work to be queued, and competes for that work with other tasks. If normal processes are used [Beck & Olien 1987] instead of threads, then the speed penalties of process creation are eliminated by creating a pool of processes before entering parallel sections of code, each of which then self-schedules as work becomes available. If enough processes are not available, a new one may be dynamically created, but this problem can be tuned out of the application.

Lightweight processing models do have some advantages. Other resources which might be usefully shared are available to all of the processes, such as file descriptors. Unfortunately, these models often promote too much sharing, making it difficult for the programmer to manage his environment.

Ideally, then, we wish to pick and choose those resources which should be shared to match the application task at hand. If such sharing is integrated into the normal UNIX model, then the programmer can expect most facilities to work as expected, i.e., the "principle of least surprise." Signals, system calls, traps and other process events should happen in an expected way. By providing a control mechanism for sharing resources, a powerful programming model can be developed which combines the performance of lightweight processes with the rich functionality of the normal UNIX interface.

The IRIX kernel implements an explicit resource sharing mechanism which provides a simple programming model which is based on the normal UNIX interface, meeting the requirements above. The remainder of this paper describes this mechanism and how it is implemented.

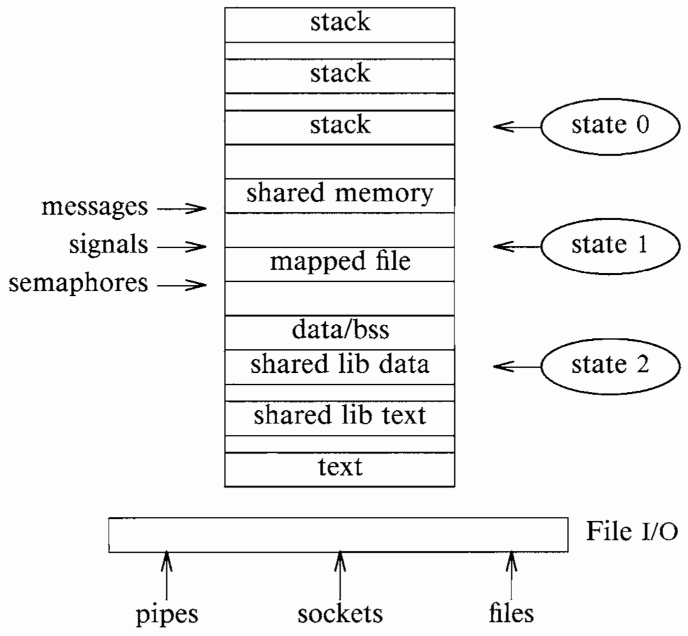


Figure 4: IRIX Programming Model

4. Share Groups

To address the failures and limitations of resource sharing when applied to multiprocessors, we added a layer of process management to the kernel that we call the *share group.* Such a group is a collection of processes which have a common ancestor and have not executed the *exec(2)* system call since being created. The parent-child relationship between processes is maintained, and forms the basis for a hierarchical resource sharing facility, where a parent can indicate what shared resources each child should share.

In general, the main resource shared is the virtual address space associated with the share group, although it need not be. Processes in the share group are free to pass pointers to data, and to use high performance synchronization and data passing techniques (mainly shared memory and spinlocks).

A small number of other resources may be shared in the initial implementation. Chief among these are file descriptors. When one of the processes in a group opens a file, the others will see the file as immediately available to them. The descriptor number[1] may be passed between processes or simply assumed as part of the algorithm. As an example, a user-level asynchronous I/O scheme could be implemented by sharing the memory and file descriptors. High level I/O calls are translated into an equivalent call in a child shared process, which performs the I/O directly from the original buffer and then signals the parent. Modifications to the file or device descriptor are automatically used by the child, since it shares the descriptor.[2]

The small set of shared resources was initially chosen to provide a wide range of applications while limiting the initial implementation to a reasonable amount of work. Shared file descriptors provide obvious advantages; the ability to change the working directory or root directory of an entire set of processes at once is useful in multiprocess applications. Process ID, *ulimit* and *umask* values are easily shared, and have potential for use in real applications.

5. *System Call Interface*

The share group interface is defined through two new system calls, *sproc()* and *prctl()*. The *sproc()* call is used to create a new process within the share group, and controls the resources which will be shared with the new process. The *prctl()* call is used to obtain information about the share group and to control certain features of the group and new processes.

1. The descriptor number is an index into the file table for a process, which holds pointers to open file table entries. For example, *standard input* is by convention descriptor number 0, while *standard output* indicates descriptor number 1.
2. This is the way most implementations of asychronous I/O are handled, only the extra process is inside the kernel instead of outside.

5.1 The `sproc` *System Call*

The syntax of this call is:

```
int sproc(entry, shmask, arg)
   void          (*entry)();
   unsigned long shmask;
   long          arg;

returns -
   -1 - OS error in call
   >0 - new process ID
```

This call is similar to the *fork(2)* system call, in that a new process is created. The *shmask* parameter specifies which resources the child will share with the share group, each particular resource being identified with a bit in the parameter. A new stack is automatically created for the child process, and the new process is entered at the address given by *entry*. This new stack is visible to all other processes in the share group, and will automatically grow in size as needed. The single argument *arg* is passed to the newly created process as the only parameter to the entry point, and can be used to pass a pointer to a data block or perhaps an index into a table.

The first use of the *sproc()* call creates a *share group.* Whenever a process in the share group issues the *sproc()* call the new process is made part of the parent's share group. A new process may be created outside the share group through the *fork(2)* system call, and use of the *exec(2)* system call removes the process from the share group before overlaying the new process image, thus insuring a secure environment for the new program image.

All resource sharing is controlled through the *shmask* (*share mask*) parameter. Currently, the following elements can be shared:

```
PR_SADDR    - share virtual address space
PR_SULIMIT  - ulimit values
PR_SUMASK   - umask values
PR_SDIR     - current/root directory
PR_FDS      - open file descriptors
PR_SID      - uid/gid
PR_SALL     - all of the above and any future resources
```

When the child is created, the share mask is masked against the share mask used when creating the parent. This means that a process can only cause a child to share those resources that the parent can share as well, providing *strict inheritance* of those resources. The original process in a share group is given a mask indicating that all resources are shared.

Whenever a process modifies one of the shared resources, and its share mask indicates that it is sharing the resource, all other processes in the share group which are also sharing the resource will be updated.

To avoid forcing complex sychronization schemes on the programmer, the kernel uses a simple rule for modifications to the VM image: *by the time control is returned to the process making the VM modification, all other processes in the share group will also see that modification.* For instance, suppose that a subroutine is called in one process, which causes automatic stack growth. That subroutine then synchronizes with a client process and passes it a pointer to a local stack variable. If the VM image is not synchronized to the stack growth, the client may not be able to access this perfectly valid address.

If the virtual address space is not shared, the new process gets a *copy-on-write* image of the share group virtual address space. In this case, the new stack is not visible in the share group virtual address space.

Certain small parts of a process's VM space are not shared. The most critical of these is the process data area, or PRDA. This is a small amount of memory (typically less than a page in size) which records data which must remain private to the process, and is always at the same fixed virtual location in every process, allowing shared code to access private data. As an example, consider the *errno* variable provided by the C library. Since the data space is shared, if this variable were only in the data space it would be difficult for independent processes to make reliable system calls. The C library could locate a copy of *errno* in the PRDA for a process. The format and use of the PRDA is totally within the scope of the user program, and can be managed in any way appropriate. Libraries (such as the C library) which need some private data may use a portion of the PRDA, leaving a portion for direct use by the programmer.

5.2 The `prctl` *System Call*

The second system call is the *prctl()* call, which allows certain aspects of a share group to be controlled. Its syntax is:

```
int prctl(option [, value [, value2 ]])
   unsigned option;
   char     *value;
   char     *value2;

returns -
    -1 - error in OS call
   <>0 - request result
```

The following *option* values may be used:

```
PR_MAXPROCS      - return limit on processes per user
PR_MAXPPROCS     - return number of processes that the
                   system can run in parallel.
PR_SETSTACKSIZE  - sets the maximum stack size for
                   the current process.
PR_GETSTACKSIZE  - retrieves the maximum stack size for
                   the current process.
```

The PR_SETSTACKSIZE option allows the program to set the maximum stack size which it may have. This value is inherited across *sproc()* and *fork()* system calls, and indirectly controls the layout of the shared VM image.

6. Implementation

The implementation of share groups centered on four goals:

1. The implementation must work correctly in both multiprocessor and uniprocessor environments.
2. Synchronization between share group processes executing in the kernel must be able to proceed even though one or more of the processes are not available for execution.
3. The overall structure of the kernel must not be modified.

4. The performance for non-share group processes must not be reduced.

The multiprocessor requirement proved to be the major difficulty in the design, since there are no formal interfaces for synchronization between processes executing in the kernel. This is not usually a problem on uniprocessor systems since a process executing in the kernel cannot be preempted. Thus it may examine and modify the state of any other process, and know that the state will not change. This isn't true in a multiprocessor environment. The process being examined may be running on another processor, sleeping on a semaphore waiting for a resource (it could even be waiting for a resource that the examining process controls), or it may be waiting for a slow operation (e.g. read on a pty, performing the *wait(2)* system call, etc.).

6.1 Data Structures

For each share group, there is a single data structure (the shared address block) that is referenced by all members of the group:

```
typedef struct shaddr_s {
   /* the following are all to handle pregions */
   preg_t       *s_region;  /* processes' shared
                               pregions */
   lock_t       s_acclck;   /* lock on access
                               to shared block */
   sema_t       s_updwait;  /* wait for update lock */
   short        s_acccnt;   /* count of readers */
   ushort       s_waitcnt;  /* count of waiting
                               processes */

   /* generic fields for handling shared processes */
   struct proc  *s_plink;   /* link to shared
                               processes */
   ushort       s_refcnt;   /* # processes in list */
   ushort       s_flag;     /* flags */
   lock_t       s_listlock; /* protects s_plink */

   /* semaphore for single threading open file
      updating */
   sema_t       s_fupdsema; /* wait for opening file */
   struct file  **s_ofile;  /* copy of open file
                               descriptors */
```

```
    char          *s_pofile;  /* copy of open file
                                 flags */
    struct inode *s_cdir;     /* current directory */
    struct inode *s_rdir;     /* root directory */

    /* lock for updating misc things that don't need
       a semaphore */
    lock_t        s_rupdlock; /* update lock */

    /* hold values for other sharing options */
    short         s_cmask;    /* mask for file
                                 creation */
    daddr_t       s_limit;    /* maximum write address */
    ushort        s_uid;      /* effective user id */
    ushort        s_gid;      /* effective group id */
} shaddr_t;
```

This structure is dynamically allocated the first time that a process invokes the *sproc(2)* system call. A pointer in the *proc* structure points to this, and the *s_plink* field links all processes via a link field in the *proc* structure. To protect the linked list during searching, the lock *s_listlock* is used. A reference count is kept in *s_refcnt*, and the structure is thrown away once the last member exits (Figure 5). The rest of the fields are used to implement resource sharing between group members and are explained in the following sections.

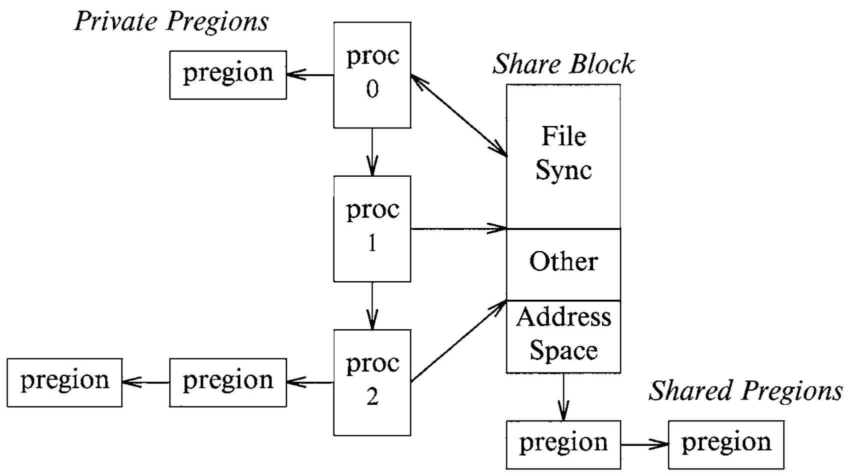


Figure 5: Share Block Data Structure

6.2 Virtual Space Sharing

The kernel in which this was implemented is based on System V.3, and therefore uses the *region* [Bach 1986] model of virtual memory. This model consists of 2 main data structures – *regions*, which describe contiguous virtual spaces (and contain all the page table information etc.) and *pregions*, which are linked per-process and describe the virtual address at which a region is attached to the process as well as other per-process information about the region of memory. This model is designed to allow for full orthogonality between regions that grow (up or down), and those that are shared.

The most interesting (and difficult) part of resource sharing to implement is sharing of the VM image, although the job is simplified by using *regions* as a basis. The *sproc* system call shares code with the standard *fork* call, the only difference being region handling. If address space sharing is not indicated when the *sproc()* call is made, then the standard *fork()* operations are performed, providing *copy-on-write* access to the VM image. If VM sharing is specified, any private pregions of the parent process are marked as *copy-on-write* in the child, while all other regions are shared (the debugger, for instance, may create a private text region for a particular process when planting breakpoints). A point to remember is that a *fork()* or non-VM sharing *sproc()* call leaves any visible stack or other regions from the share group as *copy-on-write* elements of the new process.

Algorithmically, the private regions for a process are examined first when demand paging or loading a new process, followed by examination of the shared regions. This provides the *copy-on-write* abilities of a non-VM sharing share group member. It also provides a basis for future enhancements to the manner in which the VM is shared. For instance, it could be possible to share part of the VM image and have *copy-on-write* access to other parts of the image.

sproc() allocates a new stack segment in a non-overlapping region of the parent's virtual address space. The new process starts on this stack and therefore does not inherit the stack context of the parent (though the child can reference the parent's stack).

The child process starts executing at the address given in the *sproc()* call.

With shared access to data regions, it must be possible to grow and shrink such regions as needed, for instance if the user calls *sbrk(2)*. Unfortunately, the stock (V.3) region implementation does not support growing or shrinking shared regions. The main problem is that although regions have a well defined locking protocol, a basic assumption is made that only the process owning a private region (one that has only a single reference) will ever change it. Using this assumption, various parts of the kernel hold implicit pointers into the region (i.e. pointers to pages) even though they no longer hold a lock on the region. If a process comes in and shrinks such a region, then any implicit pointers are left dangling. Lack of adequate region locking also comes into play when scanning a pregion list attempting to match a virtual address reference with a region of actual memory (during a page fault, for example). This list is not usually protected via a lock since only the calling process can change it. With growable shared regions, information that is used during the scan could change.

The obvious solution to this is to use a lock on the pregion list. Since all share group processes must obtain this lock each time they page fault, the lock was made a *shared read lock* – any number of processes can scan the list, but if a process needs to update or change the list or what it points to, it must wait until all others are done scanning. Unfortunately, to guard against the "implicit" references mentioned above, the lock could not be hidden in the existing region lock routines, but had to be placed around the entire sections of code that reference the virtual address.

The shared read lock consists of a spin lock *s_acclck* which guards the counters: *s_acccnt* and *s_waitcnt*. *s_acccnt* is the number of processes reading the list (or -1 if someone is updating the list); *s_waitcnt* counts the number of processes waiting for the shared lock. *s_updwait* is a semaphore on which waiting processes sleep. The shared pregion list is protected via the shared lock in all places that the pregion list is accessed. In most cases the only code change was to put calls to the shared lock routines around the large sections of code that reference the pregion or region. Since there is only one list, if one process adds a pregion (say through a *mmap(2)* call) all other share group members will immediately see that new virtual region. Since operations that require the update lock are relatively rare (fork, exec, mmap, sbrk, etc.) compared to the operations that scan (page fault, pager) the shared lock is almost always available and multiple processes do not collide.

When a process first creates a share group (by calling *sproc(2)*) all of its sharable pregions are moved to the list of pregions in the shared address block. Some types of regions are not currently sharable. For instance, private text regions may be created for debugging – that way breakpoints may (but are not required to) be set in an individual share group member.

An interesting problem occurs when a process wishes to delete some section of its virtual space either by removing or shrinking a region. In this case it is important that the actual physical pages not be freed until *all* share group members have agreed to not reference those pages. Also important is that the initiating process not have to wait for each group member to run before completing its system call (some members may not run for a long time). To solve this problem, we use the fact that the TLB (translation lookaside buffer) is managed by software for the target processor (a MIPS R2000 [MIPS 1986]). Thus before shrinking or detaching a region, we synchronously flush the TLBs for ALL processors, while holding the update lock for the share group's pregions. Thus if any group members are running, they will immediately trap on a TLB miss exception, come into the kernel and attempt to grab the shared read lock and block. Since the lock is held for update, the process will sleep until the lock is released. Fortunately, regions are shrunk or removed only rarely.

6.3 Other Attribute Sharing

Unlike virtual memory, other process resources are not visible outside of the kernel, thus it is only important that they be synchronized whenever a group member enters the kernel. As with VM synchronization, we don't want to force the calling process to wait until all other group members have synchronized. To implement this, we keep a copy of each resource in the shared address block: current directory inode pointer, open file descriptors, user ids, etc. This also allows immediate updating of this data, as most of it is kept in the user area and therefore inaccessible to other than the owning process.

Those resources which have reference counts (file descriptors and inodes) have the count bumped one for the shared address block. This avoids any races whereby the process that changed the resource exits before all other group members have had a chance to synchronize. Since there always exists a reliable available copy of the data, all that remains is to synchronize the data on each member's entry to the kernel. To make this efficient, multiple bits in the proc structure *p_flag* word are used. When a group member changes a sharable resource, it first checks its *p_shmask* mask (the kernel version of the share mask) to see if it is sharing this particular resource. If so, the share block is locked for update, the resource is modified (open the file, set user id, etc.), a copy is made in the shared address block, each sharing group member's *p_flag* word is updated, and the lock is released. When a shared process enters the system via a system call, the collection of bits in *p_flag* is checked in a single test; if any are set then a routine to handle the synchronization is called. Other events that must be checked on entry to the system were also changed to this scheme, thus lowering the system call overhead for most system calls.

The above scheme works well except if two processes attempt to update a resource at the same time. The lock will stop the second process, but it is important that the second process be synchronized prior to being allowed to update the resource. This is handled by also checking the synchronization bits after acquiring the lock.

7. Analysis

The resource sharing mechanism described above was implemented and tested on a MIPS R2000 based multiprocessor. As expected, the time for a *sproc()* system call is slightly less than a regular *fork()*. The overhead for synchronizing virtual memory is negligible except when detaching or shrinking regions. In practice this only happens if a process shrinks its data space (fairly rare) or if a process does considerable VM management activity (e.g. mapping or unmapping files).

As expected from the design, normal UNIX processes experience no penalty for the addition of share group support.

8. Future Directions

Although the set of features included in the first release is adequate for many tasks, there are many other interesting capabilities that could be added.

For example, the ability to selectively share regions when calling *sproc()* could be a useful facility, somewhat along the lines of Mach shared memory, thus allowing some parts of the address space to be accessed *copy-on-write* while others are simply shared. This ability is a simple extension to the current scheme, as it only requires proper management of the private pregion list and the shared pregion list.

There are also other resources that could be usefully shared. For instance, the scheduling parameters of a process could be shared among the members of the share group. Since the shared address block is always resident, it provides a convenient handle for making scheduling decisions about the process group as a whole. In a multiprocessor example, the programmer could specify that at least two of the processes in the share group must run in parallel, or the group should not be allowed to execute at all. The priority of the whole group could be raised or lowered, or a whole process group could be conveniently blocked or unblocked.

Currently, calling *exec(2)* breaks the association with the share group. By modifying the concept of pregion sharing to handle a unique address space, it could be possible to have a group of unrelated programs managed as a whole for file sharing or scheduling purposes. If this is possible, then we can also consider allowing an unrelated process to join a share group dynamically, which has many interesting implications. From another point of view, such sharing blurs the line between the *fork()* and *exec()* system calls, with the difference determined by the amount of resource sharing involved.

Finally, it might be useful to allow a process to *stop* sharing a resource. For instance, the *fork()* primitive already performs this for the virtual address space if used within a share group. Addition of "stop sharing" requests for other resources is under investigation.

9. Conclusion

This paper has presented a new and unique interface for the System V kernel, *share groups,* that allows an extremely high level of resource sharing between UNIX processes. The programmer has control of what is being shared, including whether the VM image is shared or not. Familiar UNIX mechanisms for managing processes are used for share group processes as well, providing compatability and expected behavior.

Using these resource sharing abilities, it is possible to construct simple yet powerful applications which use multiple processes, or parallel applications which can take full advantage of modern multiprocessors.

This interface has been implemented within a production kernel, and meets the promise of its design. Processes not associated with a share group experience no penalty for the inclusion of share group support, while processes within a share group may take advantage of a common address space and automatic sharing of certain fundamental resources.

This interface also allows for a large degree of future extensions, and shows how an additional layer of process management may be added to the UNIX kernel without penalizing standard programs.

References

Mike Accetta et al., Mach: A New Kernel Foundation For UNIX Development, *USENIX Association Conference Proceedings,* Summer, 1986.

Maurice J. Bach, *The Design of the UNIX Operating System,* Prentice Hall, New Jersey, 1986.

J. M. Barton, Multiprocessors: Can UNIX Take the Plunge?, *UNIX Review,* October, 1987.

Bob Beck and Dave Olien, A Parallel Programming Process Model, *USENIX Association Conference Proceedings,* Winter, 1987.

MIPS R2000 Processor Architecture, Mips Computer Systems, Inc., Mountain View, CA, 1986.

Avadis Tevanian et al., Mach Threads and the UNIX Kernel: The Battle for Control, *USENIX Association Conference Proceedings,* Summer 1987.

J. C. Wagner and J. M. Barton, Threads in System V: Letting UNIX Parallel Process, *;login:*, Vol. 12, No. 5, September/October 1987.

[*submitted May 10, 1988; accepted May 24, 1988*]