The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

Building Distributed Process Management on an Object-Oriented Framework

Ken Shirriff
Sun Microsystems

Building Distributed Process Management on an Object-Oriented Framework

Ken Shirriff
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043
`shirriff@eng.sun.com`

Abstract

The Solaris MC distributed operating system provides a single-system image across a cluster of nodes, including distributed process management. It supports remote signals, waits across nodes, remote execution, and a distributed `/proc` pseudo file system. Process management in Solaris MC is implemented through an object-oriented interface to the process system. This paper has three main goals: it illustrates how an existing UNIX operating system kernel can be extended to provide distributed process support, it provides interfaces that may be useful for general access to the kernel's process activity, and it gives experience with object-oriented programming in a commercial kernel.

1 Introduction

The Solaris MC research project[1] has set out to extend the Solaris® operating system to clusters of nodes. We believe that due to technology trends, the preferred architecture for large-scale servers will be a cluster of off-the-shelf multiprocessor nodes connected by an industry-standard interconnect. The Solaris MC operating system is a prototype distributed operating system that provides a *single system image* for such a cluster and will provide high availability so that the cluster can survive node failures. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same ABI/API as the Solaris OS, running unmodified applications.

As process management is a key component of an operating system, a single-system image cluster operating system must implement a variety of process-related operations. In addition to creating and destroying processes, a POSIX-compliant[13] operating system must support signalling a process, waiting for child process termination, managing process groups, and handing `tty` sessions. In addition, Solaris provides a file-based interface to processes through the `/proc` file system, which is used by `ps` and debuggers. Finally, cluster-based process management must provide additional functionality such as controlling remote execution of processes.

Our decision to base the Solaris MC operating system on the existing Solaris kernel influenced many of the architectural decisions in the process management subsystem. One goal was to minimize the amount of kernel change required, while making the changes necessary to provide a fully-transparent single-system image. Thus, the Solaris MC process subsystem can be distinguished on one hand from cluster process systems that run entirely at user level (such as GLUnix [22]) and on the other hand from systems that build distributed process management from scratch (such as Sprite[18]). By putting distributed process support in the kernel, we gain both better performance and more transparency than systems that operate at user level. By using most of an existing kernel, we reduce the development cost and increase the commercial potential. This is similar in concept to Locus [20] and OSF/1 AD TNC [25].

As well as describing an implementation of cluster process management, this paper describes interfaces into the kernel's process management. UNIX systems lack a good kernel interface allowing access to the low-level process management. In comparison, the file system has a well-defined existing interface between the kernel and the underlying storage system through vnodes [15], allowing new storage systems or distributed file systems to be "plugged in." The object-oriented interfaces described below are a first cut at providing similar access to processes.

This paper describes the process management subsystem of the Solaris MC operating system. Section 2 provides an overview of the implementation of distrib-

1. Solaris MC is an internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from
`http://www.sunlabs.com/research/solaris-mc`

uted process management. Section 3 discusses the interfaces that the Solaris MC process management uses. Section 4 gives implementation details, Section 5 gives performance measurements, Section 6 compares this system with other distributed operating systems, and Section 7 concludes the paper.

2 Distributed process management

2.1 Overview of the Solaris MC operating system

The Solaris MC operating system [14] is a prototype distributed operating system for a cluster of nodes that provides a single-system image: a cluster appears to the user and applications as a single computer running the Solaris operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX system and provides the same ABI/API as Solaris, running unmodified applications. The Solaris MC OS provides a single-system image through a distributed file system, clusterwide process management, transparent access from any node to one or more external network interfaces, and cluster administration tools. High availability is currently being built into the Solaris MC OS: if a node fails, the remaining nodes will remain operational, and the file system and networking will transparently recover.

The Solaris MC OS is built as a collection of distributed objects in C++, for several reasons. First, the object-oriented approach gives us a good mechanism for ensuring that components communicate through well-defined, versionable interfaces; we use the IDL interface definition language [21]. Second, a distributed object framework lets us invoke methods on objects without worrying if they are local or remote, simplifying design and programming. Finally, the object framework keeps track of object reference counting, even in the event of failures, and provides subsystem failure notification, simplifying the design for high availability.

We use an object communication runtime system [5] based on CORBA [16] that borrows from Solaris doors for interprocess communication and Spring subcontracts [10] for flexible communication semantics. Our object framework lets us define object interfaces using IDL [21]. We implement these objects in C++, and they can reside either at user level or in the kernel. References to these objects can be passed from node to node. A method on an object reference is invoked as if the object were a standard C++ object. If the object is remote, the object framework will transparently marshal arguments, send the request to the object's node, and return the reply. Thus, the object framework provides a convenient mechanism for implementing distributed objects and providing communication.

The Solaris MC proxy file system (PXFS) provides an efficient, distributed file system for the cluster. The file system uses client and server-side caching using the Spring caching architecture, which provides UNIX consistency semantics. The file system is built on top of vnodes, as shown in Figure 1: on the client side, the file system looks like a normal vnode-based file system, and on the server side an existing file system (such as UFS) plugs into the Solaris MC file system using the vnode interface to provide the actual file system storage. The components of the Solaris MC file system communicate using the object system.

2.2 Distributed process management

The process management component of Solaris MC was designed to satisfy several goals:

- Provide binary compatibility and POSIX semantics.
- Minimize changes to the kernel.
- Use an object-oriented design based on CORBA and IDL.
- Provide good performance: The system should avoid node-to-node communication for local process operations, for instance.
- Provide reliability in the face of failures: The system should be able to continue working if nodes fail or are added to the system.

To support these goals, we implemented process management using the architecture shown in Figure 2. The process management globalization code is written as a C++ module that is loaded into the kernel address space. Conceptually, this module sits on top of the existing process code; system calls are directed into the

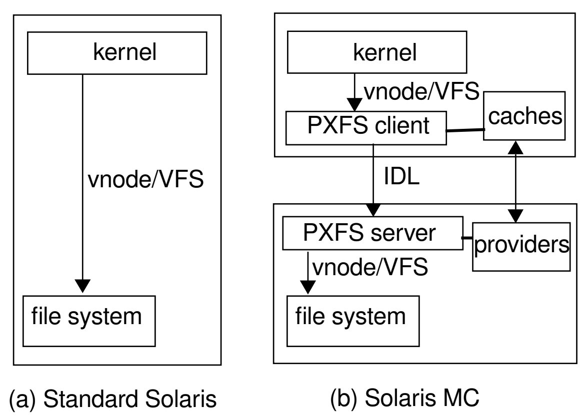

Figure 1: Structure of the Solaris MC proxy file system (PXFS). Solaris MC splits the file system across multiple nodes by adding PXFS client and PXFS server layers above the underlying file system. These layers communicate through IDL-based interfaces, as do the file system caches.

globalization layer. Much of the kernel's existing process management (*e.g.* threads, scheduling, and process creation) were used unchanged, some kernel operations were modified to hook into the globalization layer (*e.g.* `fork`, `exec`, and signals), and a few components of the kernel were largely replaced (*e.g.* `wait` and process groups). The distributed `/proc` file system is implemented independently, as will be discussed in Section 4.6.

We made the design decision to have a virtual process object associated with each process and a process manager object associated with each node. The virtual process object handles process-specific operations and holds the state necessary to globalize the process, while the process manager object handles node-specific operations (such as looking up a process ID or `pid`) and looks after the node-specific state. These objects communicate using IDL interfaces to perform operations.

A second key design decision was to keep most kernel process fields accurate locally, so most kernel code can run without modification. For instance, the local process ID is the same as the global process ID, rather than implementing a global `pid` space on top of a different local `pid` space. One alternative would be to do away with the existing kernel `pid` data structures and only use the globalized data. This approach would require extensive changes to the kernel, wherever these data structures were accessed. In our approach, the kernel needed to be modified only when the local and global pictures wouldn't correspond, for instance when children are accessed or a process is accessed by `pid`. Another approach would be to use separate local and global `pid`s; this would require mapping between them whenever the user supplies or receives a `pid`. This would reduce the kernel changes, but the mapping step would make all operations on `pid`s slower and more complicated.

The main objects used in the system are shown in Figure 3. The standard Solaris OS stores process state in a structure called `proc_t`. We added a field to this structure to point to the virtual process object; since existing Solaris code passes around `proc_t` pointers, we need some way to get from this to the virtual process object. The virtual process object manages the global parent/child relationships through object references to the parent and children (which may be on the same or different nodes). It also contains global process state information, `exit` status for children, and forwarding information for remotely execed processes. Additional objects (discussed in Section 4.2) manage process groups and sessions.

The process manager contains the node-specific data. In particular, it holds a map from `pid`s to virtual process objects for all local processes and processes originally created on this node. (This is analogous to the home structure in MOSIX [3].) It also contains object references to the process managers on the other nodes in the cluster. Thus, the process manager is used to locate processes, to locate other nodes, and to perform operations that aren't associated with a particular process (*e.g.* `sigsendset`).

To locate processes, we partitioned the `pid` space among the nodes, so the top digits of the `pid` hold the node number of the original node of the process (similar to OSF/1 [25] or Sprite [18]). Note that a process may move from node to node through the remote `exec` operation; since the `pid` can't change, the `pid` will specify the original node, not necessarily the current node. Thus, given a `pid` it is straightforward to determine the

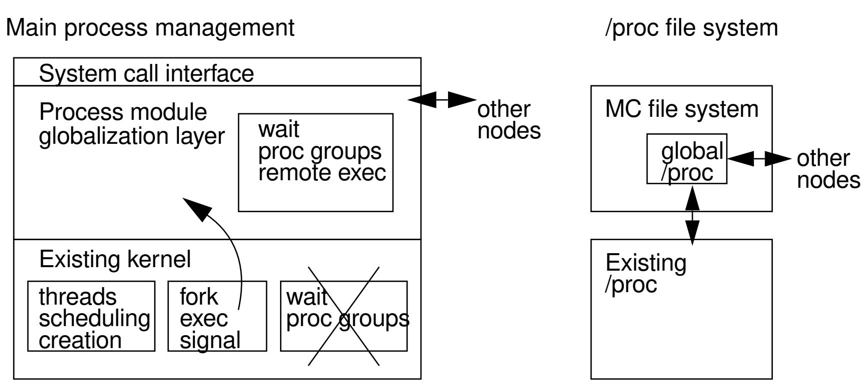

Figure 2: Structure of process management in the Solaris MC operating system. Process management is divided into two independent components. The main component, which supports UNIX process operations, is implemented as a module that interacts with the kernel to provide globalized process operations. The `/proc` component, which provides a global implementation of the `/proc` file system, is implemented as part of the Solaris MC PXFS file system.

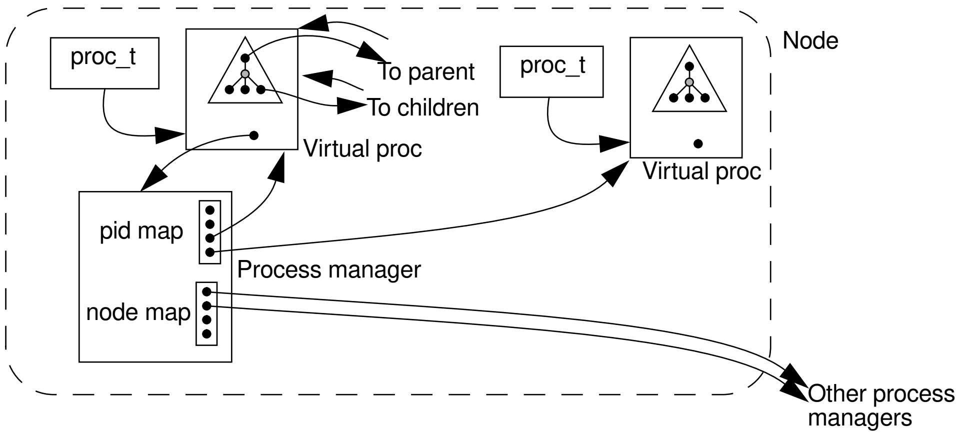

Figure 3: The objects for process management. For each local process, the existing Solaris `proc_t` structure points to a virtual process structure that holds the parent and child structure as well as other globalizing information. The process manager has a map from `pid`s to the virtual processes and from the node ids to the process managers. Solid circles indicate object references.

original node. The process manager on this node can be queried for an object reference to the virtual process object, and then the operation can be directed to that object. Instead of updating all references when a process changes nodes, we leave a forwarding reference in a virtual process object on the original node. When this is accessed, it sends a message to the requestor and the stale pointer is updated. Because the object framework does reference counting, the forwarding object will automatically be eliminated when there are no longer any references to it.

The virtual process object in Solaris MC can be compared with the global process structures in OSF/1 AD TNC [25]. In the OSF/1 system, the only process structure visible to the base server is the simple `vproc` structure, which is analogous to a `vnode` structure. The `vproc` structure contains the `pid`, an operation vector, and a pointer to internal data. This pointer references a `pvproc` structure, which holds the globalization information (location, parent-sibling-child relationships, group, and session), and another operation vector which directs operations to local process code or a remote operation stub that performs a RPC.

In both systems, the virtual process layer sits between the application's process requests and the underlying physical process code, providing globalization. An OSF/1 `vproc` structure is roughly analogous to a Solaris MC object reference, while the `pvproc` holds the actual data and roughly corresponds to the Solaris MC virtual process object. Thus, OSF/1 encapsulates process data by placing it in a separate `pvproc` structure, while Solaris MC encapsulates it inside an object. One key difference occurs on nodes that access a process that is located on a different node. In OSF/1, these nodes will have both a `vproc` and `pvproc` structure and the associated process state, while Solaris MC will only have an object reference. A second difference is that OSF/1 directs operations remotely or locally through an operation vector in the `pvproc` structure, while Solaris MC directs operations remotely or locally through the method table associated with the virtual process object reference, which is managed automatically by the object framework. Thus, the object framework hides the real location of a Solaris MC virtual process object. Finally, the marshalling and transport of requests between nodes is performed transparently by the Solaris MC object subsystem, while in OSF/1, the `vproc` system needed its own communication system, using the TNC message layer.

3 Process interfaces

Unlike the file system with its `vnode` interface, the process control in Solaris lacks a clean interface for extension. Adding a file system to UNIX used to be a difficult task. Early attempts at distributed file systems (*e.g.* Research Version 8 [1] and EFS [6]) used *ad hoc* interfaces, while NFS [26] and RFS [2] used more formal interfaces, `vnode`s and the file system switch respectively. These interfaces required extensive restructuring of the kernel, but the result is that adding a new file system is now relatively straightforward.

The same restructuring and kernel interface needs to be considered for the process layer in the kernel. The `/proc` file system provides one interface to manipulate processes, but it is limited in functionality to providing status and debugging control. The Locus `vproc` architecture [25] and the Solaris MC virtual process object can be considered steps towards a general interface analogous to vnodes.

One approach would be to provide distributed process support through extensions to the /proc interface; we didn't take this approach for several reasons. First, a file system based interface has a serious "impedance mismatch" with the operations that take place on processes; a simple process system call would get mapped to an open-write-close. Second, the overhead of opening files to manipulate processes is undesirable. Finally, a /proc-based interface would probably result in an assortment of ioctls more confusing than the current /proc ioctls; we desired a cleaner interface.

We broke the process interface into two parts: a procedural and an internal object-oriented interface. The procedural interface is used for communication between the existing kernel and the globalization layer; this interface is described in Section 3.1. The object-oriented interface is used by the global process implementation, and is described in Section 3.2.

3.1 Procedural interface

The procedural interface, given in Table 1, is a thin layer on top of the object-oriented interface for use by the existing C kernel, for both the system call layer and the underlying kernel process code to communicate with the globalization layer.

Most of the procedural calls are used to direct system calls into the globalization layer: VP_FORK, VP_EXEC, VP_VHANGUP, VP_WAIT, VP_SIGSENDSET, VP_SETSID, VP_PRIOCNTLSET, VP_GETSID, VP_SIGNAL, and VP_SETPGID. Note that operations such as getpid, nice, or setuid don't go into the globalization layer as they operate on the current (local) process.

The up-calls from the kernel to the globalization layer are VP_FORKDONE, at the end of a fork to allow the globalization layer to update its state; VP_EXIT, when a process exits, VP_SIGCLDMODE, when a process changes its SIGCLD handling flags; and VP_SIGCLD, when a process would send a SIGCLD to its parent.

These procedural calls are implemented by having VP_FORK, VP_EXIT, VP_SETSID, VP_VHANGUP, VP_WAIT, VP_SIGCLD, and VP_SIGCLDMODE call the virtual process object associated with the current process, while the remaining operations call the node object on the current node. This is an optimization; since the former set of operations act on the current process it is more efficient to send the operations to the virtual process object directly.

3.2 Internal interface

The object-oriented interface to the node object is given in Table 2. The node methods consist of operations to manage node state (addvpid, registerloadmgr, getprocmgr, addprocmgr, findvproc, findvpid, lockpid, unlockpid), operations that create new processes (rfork, rexec), and operations that act on multiple processes (sigsendset, priocntlset).

The interface to the process object is given in Table 3 and contains methods that act on an existing process. Most of these methods correspond to process system calls. The childstatuschange and releasechild methods are used for waits; they are described in more detail in Section 4.4. The childmigrated method keeps the parent informed of the child's location.

One potential future path is to merge the /proc file system with the object-oriented interface, allowing /proc operations to be performed through the object interface. The /proc file system would then be just a thin layer allowing access to the objects through the file system. This would make the process object into a single access path into the process system. It would also simplify the kernel code by combining all the process functionality into one place. Now, the Solaris /proc code is entirely separate from the virtual process code.

VP_FORK(flag, pid, node)	VP_SIGSENDSET(psp, sig, local)
VP_EXIT(flag, status)	VP_SETSID(flag, rval)
VP_EXEC(fname, argp, envp, node)	VP_PRIOCNTLSET(version, psp, cmd, arg, rval, local)
VP_VHANGUP()	VP_GETSID(pid, pgid, sid)
VP_WAIT(idtype, id, ip, options)	VP_SIGNAL(pid, sigsend)
VP_SIGCLD(cp)	VP_SETPGID(pid, pgid)
VP_SIGCLDMODE(mode)	VP_FORKDONE(p,cp,mig)

Table 1: Procedural calls into the virtual process layer. These calls are just a thin layer between the C code of the Solaris kernel and the C++ code of the Solaris MC module

4 Details of process management

This section describes the main process functions in more detail. Sections 4.1, 4.2, 4.3, and 4.4 describe how we implemented signals, process groups, remote execution, and cross-node `waits`, respectively. Section 4.5 discusses what failure recovery will be provided. Section 4.6 explains how a global `/proc` file system was added to the Solaris MC file system. Finally, Section 4.7 discusses our experience with object-oriented programming in the kernel.

4.1 Signals

Several issues complicate signal delivery in a distributed system. Delivery of a signal to a remote process is straightforward; an object reference to the process is obtained from the `pid`, the signal method is invoked on that process object, and the object delivers the signal locally.

The more complicated cases are a `kill` to a process group, the `sigsend` system call, and the `sigsendset` system call. To signal a process group, the appropriate process group object is located, the `signal` method is invoked on this object, the process group object invokes the member process objects, and these object deliver the signals. Locking complicates this operation.

The `sigsend` and `sigsendset` system calls allow signals to be sent to complex sets of processes, formed by selecting sets of processes based on `pid`, group id, session id, user id, or scheduler class, and then combining these sets with boolean operations. To handle these system calls, the operation is sent to each node in the system and each node signals the local processes matching the set. This can be inefficient since all nodes must handle the operation; if performance becomes a problem, we will add optimizations for the simple cases (such as the set specifies a single process or process group) so they can be handled efficiently.

4.2 Process groups

POSIX process groups and sessions support job control: a `tty` or pseudo-`tty` corresponds to a session, and each job corresponds to a process group. The process management subsystem must keep track of which processes belong to which process groups, and which process groups belong to which sessions. Process group membership is used by the I/O subsystem to direct I/O only to processes in a foreground process group, and send `TTIN`/`TTOU` signals to background processes.

A POSIX-compliant system must also detect orphaned process groups (which are unrelated to orphaned processes). A process group is not orphaned as long as there is a process in the group with a parent outside the process group but inside the session. (The two prototypical orphaned process groups are the shell and a job that lost its controlling shell.) Orphaned process groups are not permitted to receive terminal-related stop signals, as there is no controlling process to return them to the foreground. In addition, if a process group becomes orphaned and contains a stopped process, the process group must receive `SIGHUP` and `SIGCONT` signals to prevent the processes from remaining stopped forever.

To support these POSIX semantics, the Solaris MC OS uses an object for each process group and an object for each session, with the session object referencing the

`addvpid(pid, vproc, update)`	`rfork(state, astate, flags)`
`registerloadmgr(mgr)`	`rexec(fname, argp, envp, state, flags, newvproc)`
`sigsendset(psp, sig, local, srcpid, cred, srcsession)`	`lockpid(pid)`
`priocntlset(version, psp, cmd, arg, rvp, local)`	`unlockpid(pid)`
`getprocmgr(nodenumber)`	`findvproc(pid, local, vproc)`
`addprocmgr(nodenumber, pm)`	`freevpid(pid)`

Table 2: Methods on the node object

`signal(srcpid, cred, srcsession, signum)`	`childmigrated(childpid, newchild)`
`setpgid(srcpid, pgid)`	`parentgroupchange(par_pgid, par_sid)`
`getsid(srcpid, pgid, sid)`	`childstatuschange(childpid, wcode, wdata, utime, stime, zombie, noparent)`
`getpid()`	`setsid(sess)`
`releasechild(mode, noparent)`	

Table 3: Methods on the virtual process object

member process group objects and each process group object referencing the member processes. For efficiency, we originally planned to include an additional object, the local process group object, which would reside on each node that had a process in the process group and would reference the local processes in the process group. The local process group object would be more efficient for signalling process groups split across multiple nodes and for determining orphaned process groups, since most messages would be intra-node, to the local process group object, rather than inter-node, to the top-level process group object. The added complexity of this approach, however, convinced us to eliminate the local process group objects.

The process group objects detect orphaned process groups by keeping a reference count of the number of controlling links to the process group (*i.e.* processes with a parent outside the process group but inside the session). Operations that create or destroy a link (changing process group membership or exiting of the parent or child) inform the process group object; if the count drops to zero, the process group is signalled as required.

4.3 Remote execution

The Solaris MC operating system provides a remote execution mechanism to create processes on remote nodes. The `rexec` system call is used for remote execution; it is similar to the UNIX `exec` system call except the call specifies a node where the new process image runs. The remote process then sees the same environment as if it were running locally, due to the global file system and other single-system-image features.

Process migration, which moves an existing process to a new node, is not yet implemented in Solaris MC, due to the additional complexity of moving the address space. The remote `exec` is much easier to implement and more efficient than migrating the address space of a running process (which could be partially paged out). We feel that most of the load-balancing benefit of remote processes can be obtained by positioning the process at execution time (as do the Plan 9 researchers [19], although Harchol-Balter and Downey [11] have found benefits from moving running processes). We plan to implement migration of running processes later, in particular to off-load processes from a node scheduled for maintenance; at this time we will provide `rfork` and migration (which are very similar, since a `rfork` is basically a `fork` and a `migrate`). The implementation of migration in Solaris MC will be similar to that in other distributed operating systems (*e.g.* Sprite [7], MOSIX [3], or Locus [20]), moving the address space to the new node.

The `rexec` call operates by packaging up the necessary state of the process (the list of open files, the `proc_t` data, and the list of children), and sending this to the destination node along with the `exec` arguments. The process is then started on the remote node (using a hook into low-level process creation, since the existing `fork` code won't work without a local parent) and the old process is eliminated. Finally, the parent receives an object reference to the new virtual process object to update its child list. Our process management code then transparently manages the cross-node parent-child relationships, as described in Section 4.4.

Remote execution takes advantage of the Solaris MC distributed file system to handle migration of open files across remote `exec`s. With the distributed file system, it doesn't matter what node is performing a file operation. For each open file, the offset and the reference to the file object are sent across to the new node. These are then used to create vnodes for the new process. File operations from the new process then operate transparently, with the Solaris MC file system ensuring cache consistency.

The Solaris MC operating system provides support for multiple load balancing policies. A node can be specified in the `rexec` call, or the location can be left up to the system. By default, `rexec` uses round-robin placement if a node isn't specified, but hooks are provided for an arbitrary policy; the `registerloadmgr` method allows a load management object to be registered with the node manager. For each remote execution, the node manager will then query the load management object for the destination node id, which can be generated by any desired algorithm. The load manager can use algorithms similar to the OSF/1 AD TNC [25] or Sprite [7] load daemons. Since the load management object communicates with the node manager through the Solaris MC object model, the load management object can be implemented either in the kernel or at user level, even on a different node. This illustrates the flexibility and power of our distributed object model.

4.4 Waits

The Solaris MC operating system supports the UNIX `wait` semantics, even when the parent and child are on different nodes.

Several approaches are possible for handling `wait`s. One model, the "pull" model, has the parent request information from the child when it does a `wait`. If the `wait` does not return immediately, the parent sets up callbacks with the children so that it is informed when an event of interest occurs. A second model, the "push" model, has the children inform the parent of every state change.

The pull model has the advantage that no messages are exchanged until the parent actually performs a `wait`. However, setting up and tearing down the callbacks can be expensive if the parent has many children, since many callbacks will have to be created and removed for each `wait`.

The Solaris MC operating system uses the push model. If child processes perform many state changes that the parent doesn't care about, this will be more expensive. Normally, however, the process will just exit, requiring one message. It would be interesting to compare the number of messages required for the push vs. pull models on a real workload, however.

Note that in Solaris an exiting child process normally goes into a "zombie" state until the parent performs a `wait` on it. It would have been simpler to do away with zombie processes altogether and just keep the child's exit status with the parent, but Solaris semantics require a zombie process which will show up on `ps`, for instance. In addition, POSIX requires that the `pid` isn't reused until the `wait` is done.

The Solaris MC operating system uses a simple state machine to handle `wait`s and avoid race conditions, as shown in Figure 4. When the child `exit`s, it informs the parent (through the `childstatuschange` method) and changes state to the right; when the parent `exit`s or `wait`s on the exited child, it informs the child (through the `releasechild` method) and the child changes state downwards. When the child reaches the final state, it can be freed from the system.

Thus, when a child exits, the parent's virtual process object keeps track of the state of the child (*i.e.* exited), its `exit` status, and the user and system time used by the child process (for use by `times(2)`).

4.5 Failure recovery

The Solaris MC operating system is designed to keep running in the event of node failures. The semantics for a node failure are that the processes on the failed node will die, and to the rest of the system it should look as if these processes were killed. That is, the normal semantics for `wait`s should apply, ensuring that zombie processes don't result.

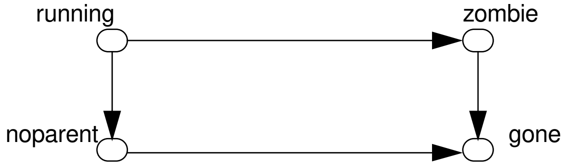

Figure 4: The state machine for processes. If a process exits, its state transitions to the right. If its parent exits, its state transitions downwards.

In the event of a failure, the system must recover necessary state information that was on the failed node. One type of information is migration pointers. If a process did multiple `rexec`s, it may have forwarding pointers on the failed node. After failure, it must be linked up with the parent. Also, after failure, process group membership must be checked to see if an orphaned process group resulted.

To support failure recovery, process management in the Solaris MC OS was designed to avoid single points of failure. For instance, there is no global `pid` server (this also helps efficiency). A remotely `exec`'d process can lose its home node through a failure; in this case, the next live node in sequence will take over as home node, providing a way to find the process.

Failure recovery in the Solaris MC operating system is still being developed and will be based on low-level failure handling in the object layer. The object layer will notify clients of failed invocations, will clean up references to the failed node, and will release objects that are no longer referenced from a live node.

4.6 The /proc file system

The `/proc` file system is a pseudo file system in Solaris [9] that provides access to the process state and image of each process running in the system. This file system is used principally by `ps` to print the state of the system and by debuggers such as `dbx` to manipulate the state of a process. Each process has an entry in `/proc` under its process id, so the process with `pid` nnnnn is accessible through /proc/nnnnn. The `/proc` file system supports `read`s and `write`s, which access the underlying memory of the process, and `ioctl`s, which can perform arbitrary operations on a process such as single stepping, returning state, or signalling. By providing a global `/proc`, the Solaris MC OS supports systemwide `ps` and cross-node debugging.

The `/proc` file system raises several implementation issues for a distributed process system. First, all processes on the system must appear in the `/proc` directory. Second, operations on a `/proc` entry must control the process wherever it resides. Third, `ioctl`s may require copying arbitrary amounts of data between user space on one node and the kernel on another. Finally, some `/proc` `ioctl`s return file descriptors for files that the process has opened; these file descriptors must be made meaningful on the destination node.

In Solaris MC, /proc is extended to provide a view of the processes running on the entire cluster, by merging together local /procs into a distributed picture, as shown in Figure 5. Thus, each node uses the existing /proc implementation to provide the actual /proc operations and a merge layer makes these look like a global /proc. To implement this, the readdir operation was modified to return the contents of all the local /procs, merged into a single directory. The new pathname lookup operation returns the vnode entry in the appropriate local /proc, so operations on the /proc entry then automatically go to the right node. If the process migrates away, the file system operation generates a “migrated” exception internally; the merging layer catches this and transparently redirects the operations.

The implementation of the distributed /proc file system illustrates the advantages of an object-oriented file system implementation. The merging /proc file system was implemented as a subclass of the standard Solaris MC distributed file system that redefines the readdir and lookup operations. The remainder of the file system code is inherited unmodified. Thus, the object-oriented approach allows new file system semantics to be implemented while sharing most of the old code.

The /proc file system uses two techniques for handling cross-node ioctls. One solution would be to modify the /proc implementation code to explicitly copy the argument data between the two nodes. This would, however, require modifications to the /proc source. The solution used in Solaris MC extends the technique used by the Solaris MC file system; the low-level routines to copy data to and from the kernel (copyin, copyout, copyinstr, copyoutstr) were modified so that if they are called in the course of an ioctl, the data is transferred from the remote node. A few ioctls, however, required special handling because they return the numeric value of a file descriptor corresponding to an open file. For these ioctls, the open file is wrapped in a Solaris MC PXFS file system object and the object reference is transferred to the client node. A new file descriptor is opened on the client corresponding to this object. Then, any operations on this file descriptor will be sent through PXFS back to the original open file.

4.7 Experience with objects

Our experience with object-oriented programming in the kernel is generally positive. Objects provided a cleaner design because of the encapsulation of data structures and the enforcement of well-defined interfaces. The object framework simplified implementation because of the location-independence of object invocations. This obviates having to keep track of which processes are local and which processes are global. It also makes remote procedure calls entirely transparent.

One major difficulty we found with object-oriented programming for the kernel is that the C++ tools aren’t as well developed as for C. We encountered several compiler and linker difficulties, and templates remain a problem. In addition, our C++ debugging environment is rather primitive. For instance, we had to modify kadb to return demangled C++ names.

A second issue with object-oriented programming is efficiency; our problems generally arose from over-enthusiastic use of C++ features. One problem the Solaris MC project encountered was that C++ exceptions were costly in our implementation, both when they were thrown and even when they weren’t (since the additional code polluted the instruction cache). As a result, we decided to remove C++ exceptions from the Solaris MC implementation and use a simple error return mechanism instead. A second problem is that object-oriented programming makes it easy to have an excessive number of classes implementing nested data

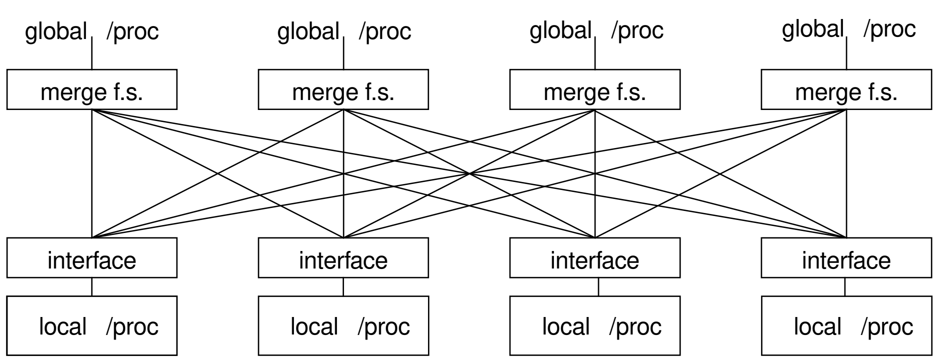

Figure 5: The distributed /proc file system merges together the local /proc file systems so each node has a /proc that provides a global view of the system.

abstractions and multiple levels of subclasses to provide specialization. Unfortunately, this can result in a very deep call graph, with the associated performance penalty. We are currently restructuring the implementation of our transport layer, where performance suffers due to this problem.

5 Performance measurements

Tables 4 and 5 give performance measurements of the Solaris MC process management implementation.[2] These measurements were taken on a cluster of four two-processor SPARCstation 10's, three with 50 MHz processors and one with 40 MHz processors, each with 64 MB and running Solaris 2.6 with Solaris MC modifications. The interconnect is a 10 MByte/sec 100-Base T using the SunFast interface, and the transport layer uses the STREAMS stack. Note that Solaris MC is a research prototype and has not been tuned for performance. Thus, these numbers should be viewed as very preliminary and not a measure of the full potential of the system.

5.1 Micro-benchmarks

Table 4 gives several performance measurements comparing the unmodified Solaris OS, the Solaris MC OS performing local operations, and the Solaris MC OS performing remote operations.

The first three sets of measurements are from the Locus MicroBenchmarks (TNC Performance Test Suite). The first line shows the time for a process to `fork`, the child to `exit`, and the parent to `wait` on the child. There is some slowdown in Solaris MC due to the additional overhead of going through the globalization layer and creating the virtual process object. The second line shows the time taken for a process that does multiple `exec`s, either on a single node or between nodes. The Solaris MC overhead from local `exec`s is minimal, since the virtual process object remains unchanged. There is additional overhead, however for a remote `exec`, due to the process state (open file descriptors, etc.) that must be transmitted across the network, and the virtual process object that must be created on the remote node. The third line shows performance for a process that performs a cycle of `fork`, `exec` (local or remote), and `exit`. Again, Solaris MC has some overhead even for the local case due to the `fork`, and much more overhead for the remote `exec`. These measurements show that Solaris MC could use tuning of the virtual process object to improve `fork` performance, and remote `exec` performance is limited by the inter-node communication cost, but the globalization layer adds negligible cost to a local `exec`.

The next measurements show performance of the `PIOCSTATUS` `ioctl` on the `/proc` file system, which returns a 508 byte structure containing process status information. This measurement illustrates the performance of the globalized `/proc` file system and of the cross-node `ioctl` data copying. In the local case, there is a slowdown of about 30µs due to the overhead of going through the `/proc` globalization code and the PXFS `ioctl` layer. The remote case is considerably slower due to the network traffic. As discussed in Section 4.6, the `copyout` of `ioctl` data to user level is done through modified low-level copy functions. Thus, two network round-trips are required: one to invoke the remote `ioctl`, and a second to send the data back. Originally, an `address-space` object was created for every `ioctl` to handle data copying, but the overhead of object creation made even local `ioctl`s take about 1 ms, an unacceptable overhead. However, by using a per-node object rather than a per-operation object, and by using the standard copy functions for local `ioctl`s rather than the modified ones, the overhead was substantially reduced.

The final measurements in Table 4 shows performance for signalling between processes. In this test, two processes are started and send signals back and forth: the parent process sends a `USR1` signal to the child process, the child process catches the signal and sends a `USR1` signal to the parent, the parent catches the signal, and the cycle repeats. The time given is the average one-way time (i.e. half the cycle time). Again there is some

2. The programs used for these tests are available from `http://www.sunlabs.com/research/solaris-mc/process.html`

Operation	Solaris	Solaris MC	Solaris MC, remote
`fork/exit (TNC #1)`	9 ms	14 ms	n/a
`(r)exec (TNC #5,#6)`	32 ms	33 ms	58 ms
`fork/(r)exec/exit (TNC #7,#8)`	35 ms	42 ms	67 ms
`PIOCSTATUS` `ioctl` on `/proc`	0.05ms	0.08 ms	3 ms
Signals	0.14 ms	0.27 ms	2 ms

Table 4: Performance of various operations performed on standard Solaris, on Solaris MC locally, and on Solaris MC across nodes.

Operating system, file system, process location	Sequential	Parallel
Solaris, local UFS file system, local processes	26 sec (33 sec)	18 sec (22 sec)
Solaris, remote NFS file system, local processes	36 sec (38 sec)	23 sec (27 sec)
Solaris MC, local UFS file system, local processes	26 sec (32 sec)	20 sec
Solaris MC, local PXFS, local processes	27 sec (31 sec)	19 sec
Solaris MC, remote PXFS, local processes	29 sec (35 sec)	21 sec
Solaris MC, PXFS, processes running throughout cluster	36 sec	15 sec

Table 5: Times for the compilation phase of the modified Andrew benchmark. Numbers in parentheses were measured on a 40 MHz node. The parallel measurements in the first five rows run multiple processes on the two processors on a node; the final parallel measurement runs processes on all nodes in the cluster.

slowdown for the local case in Solaris MC and a considerable overhead for between-node communication.

5.2 Parallel make performance

One interesting measure is how well system performance scales for parallel tasks on multiple nodes. Table 5 gives performance measurements for the system on the compilation phase of the modified Andrew benchmark [17], compiling a sequence of files sequentially and in parallel across the cluster under various conditions. The first two lines show compilation time for a single node running Solaris, accessing the files from a local disk or NFS, and compiling with sequential `make` or parallel `make` (which takes advantage of the dual processors on a node). Unfortunately, one of the nodes had 40 MHz processors, while the others had 50 MHz processors, which makes cluster measurements slightly harder to interpret; numbers in parentheses show results on the slower node. The next three lines show performance of a single node running Solaris MC, using a local disk running the UNIX file system, a local disk running the Solaris MC proxy file system (PXFS), or a remote node running the Solaris MC file system. Finally, the last line shows performance of Solaris MC when processes are executed around the cluster, either sequentially or in parallel.

Table 5 shows a speedup for parallel compilation on the Solaris MC cluster compared to a single Solaris node (15 sec vs. 18 sec), but this speedup isn't as dramatic as one would expect when going from 2 processors to 8 processors. Several factors account for this. First, the benchmark compile ends with a link phase that must be done sequentially and takes about 5 seconds. Thus, even with perfect scaling, the entire compile would take about 8 seconds. Second, there is a performance penalty in accessing PXFS files from a remote node (although comparing the PXFS measurements with the Solaris measurements shows that locally, PXFS is comparable to the native file system, and remotely it is faster than NFS). Thus, the compile is slowed down due to cross-node file accesses. Third, the node with 40 MHz processors slows down the processes that are migrated to that node. Fourth, the parallel `make` uses a simple round-robin allocation policy, which yields non-optimum load balancing, especially when there is one slow node. Finally, comparing the last two lines of Table 5 shows that there is a significant performance penalty when processes are executed on remote nodes (increasing the time from 29 to 36 seconds for sequential execution). The effects of these factors can be reduced, however. Most significantly, with a faster interconnect and tuning of the object transport layer, the overheads due to communication between nodes will be reduced, improving file system and remote execution performance. With a faster interconnect and a balanced system, the performance improvement from parallel execution would be considerably better.

5.3 Performance discussion

Note that the Solaris operating system has been extensively tuned for performance, while Solaris MC is an almost entirely untuned research system. We expect these numbers to improve significantly with tuning, but there will still be some performance penalty due to the longer code path through the globalization layer.

Remote process operations are considerably slower than local operations because of the additional network transport time. The performance of distributed process management depends strongly on the performance of the underlying object system and the cluster network. We are in the process of optimizing the Solaris MC object framework to provide faster inter-node communication.

6 Related work

Several research and commercial operating systems provide distributed process management, such as Unisys Opus (Chorus) [4], Intel XP/S MP Paragon, OSF/1 AD TNC (Mach)[25], DCE TCF (Locus)[20], Sprite [7], GLUnix [22], and MOSIX [3]. Solaris MC uses many of the concepts from these systems.

Process management in the Solaris MC operating system differs from previous systems in several ways. First, many of the previous systems build distributed process management from scratch; the Solaris MC OS demonstrates how distributed process management can be added to an existing commercial kernel while minimizing kernel changes. On the other hand, the Solaris MC OS provides a stronger single-system image than systems such as GLUnix, which build a globalization layer at user level on top of an existing kernel.

The Solaris MC OS also differs from previous systems in that it is built on an object-oriented communication framework, rather than a RPC-based framework. One key difference is that the object-oriented framework transparently routes invocations to the local or remote node as necessary, compared to RPC-based systems which require explicit marshalling of arguments and calling to a particular node. In addition, the object-oriented framework provides a built-in object reference counting mechanism; this avoids *ad hoc* mechanisms that are typically required in an RPC-based system to clean up state after node failures. The Solaris MC OS also illustrates how object-oriented programming can be added to an existing monolithic kernel.

The Solaris MC OS also presents new object-oriented interfaces to the process subsystem. These interfaces may be useful to applications that require more control over the process subsystem.

Finally, the Solaris MC OS shows how the `/proc` file system can be extended to a cluster to provide file access to process state throughout a cluster. Unlike the VPROCS [24] implementations of `/proc`, Solaris MC uses object inheritance to provide a distributed `/proc` through subclassing of the PXFS file system implementation.

7 Conclusions

Process management in the Solaris MC research operating system provides a distributed view of processes with a single `pid` space across a cluster of machines, while preserving POSIX semantics. It supports the standard UNIX process operations and the Solaris `/proc` file system as well as providing remote execution.

Process management is implemented in an object framework, with objects corresponding to each process, process group, and system node. This object framework simplified implementation of process management by providing transparent communication between nodes, failure notification, and reference counting.

Unlike the file system with the `vnode` interface, process management in UNIX generally lacks an interface for extending the system. While the `/proc` interface provides some control over processes, it is limited to status and process control operations. Process management in the Solaris MC operating system extends the access to the operating system's process internals by providing an object-oriented interface. In the future, the `/proc` interface and the object-oriented interface could be merged to provide a single extensible interface to the operating system's process management.

Process management was designed to allow most local operations to take place without network communication; there is no central server that must be contacted for process operations. There is some performance penalty due to the overhead of the globalization layer and due to creation of the virtual process object. With tuning, however, we expect this overhead to be reduced. Remote operations suffer a performance penalty due to the interconnect bandwidth and the overhead of the Solaris MC transport layer.

The main components in Solaris MC process management remaining to be implemented are process migration and load balancing, although we have remote process execution. Support for failure recovery and for full process group semantics also need to be implemented.

In conclusion, process management in the Solaris MC operating system illustrates how an existing monolithic operating system can be extended, with relatively few kernel changes, to provide transparent clusterwide process management. It also provides a set of object-oriented interfaces that can be used to provide access to the process internals.

Acknowledgments

This work would not have been possible without the work of the entire Solaris MC team in building Solaris MC. The author is grateful to Scott Wilson, Yousef Khalidi, the anonymous referees and especially the paper "shepherd" Clem Cole for their helpful comments on the paper.

References

[1] AT&T, *Research Unix Version 8,* Murray Hill, NJ.

[2] M. J. Bach, "Distributed file systems", *The Design of the UNIX Operating System,*" Prentice-Hall, Englewood Cliffs, NJ, 1986.

[3] A. Barak, S. Guday, and R. G. Wheeler, "The MOSIX Distributed Operating Systems," *Lecture Notes*

in Computer Science 672, Springer-Verlag, Berlin, 1993.

[4] N. Batlivala, *et al.*, "Experience with SVR4 Over CHORUS," *Proceedings of USENIX Workshop on Microkernels & Other Kernel Architectures*, April 1992.

[5] J. Bernabeu, V. Matena, and Y. Khalidi, "Extending a Traditional OS Using Object-Oriented Techniques", 2nd Conference on Object-Oriented Technologies and Systems (COOTS), June 1996.

[6] C. T. Cole, P. B. Flinn, and A. Atlas, "An Implementation of an Extended File System for UNIX," *Proceedings of Summer USENIX 1985*, pp. 131-144.

[7] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software—Practice & Experience*, vol. 21(8), August 1991.

[8] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, 4(4):353-384, Fall 1991.

[9] R. Faulkner and R. Gomes, "The Process File System and Process Model in UNIX System V," *Proceedings of Winter Usenix 1991.*

[10] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A Flexible Base for Distributed Programming," *Symposium on Operating System Principles*, 1993, pp. 69-79.

[11] M. Harchol-Balter and A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *Proceedings of ACM Sigmetrics '96 Conference on Measurement and Modeling of Computer Systems*, May 23-26 1996, pp 13-24.

[12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, 6(1), February 1988, pp. 51-81.

[13] IEEE, *IEEE Standard for Information Technology Portable Operating System Interface,* IEEE Std 1003.1b-1993, April 1994.

[14] Y. Khalidi, J. Bernabeu, V. Matena, K. Shirriff, M. Thadani. "Solaris MC: A Multicomputer Operating System", *Proceedings of Usenix 1996,* January 1996, pp. 191-203.

[15] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of Summer USENIX Conference 1986*, pp. 238-247.

[16] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Revision 1.2, December 1993.

[17] J. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" *Proceedings of Summer USENIX 1990*, pp. 247-256

[18] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, February 1988.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey., and P. Winterbottom, "Plan 9 From Bell Labs," *Computing Systems*, Vol. 8, No. 3, Summer 1995, pp. 221- 254.

[20] G. Popek and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[21] Sun Microsystems, Inc. *IDL Programmer's Guide*, 1992.

[22] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson, *Efficient, Portable, and Robust Extension of Operating System Functionality*, UC Berkeley Technical Report CS-94-842, December, 1994.

[23] B. Walker, J. Lilienkamp, J. Hopfield, R. Zajcew, G. Thiel, R. Mathews, J. Mott, and F. Lawlor, "Extending DCE to Transparent Processing Clusters," *UniForum 1992 Conference Proceedings,* pp 189-199.

[24] B. Walker, R. Zajcew, G. Thiel, *VPROCS: A Virtual Process Interface for POSIX systems*, Technical Report LA-0920, Locus Computing Corporation, May, 1992.

[25] Roman Zajcew, *et al.*, "An OSF/1 UNIX for Massively Parallel Multicomputers," *Proceedings of Winter USENIX Conference* 1993.

[26] R. Sandberg, D. Goldberg, D. Walsh, and B. Lyon, "The Design of the Sun Network File System," *Proceedings of Summer USENIX 1985*, pp 119-131.