# The Process Management Architecture of OSF/1 AD Version 2

*OSF Research Institute*

*Faramarz Rabii*

*Simon Patience*
*{rabii,sp}@osf.org*

*September 20, 1993*

*Abstract*

This paper describes the architecture of distributed process management in OSF1/AD Version 2. This architecture provides a single system image, is based on the VPROC abstraction and allows for vendor specific enhancements. This design distributes all single process handling among equal process management agents while only confining inter-process handling to a single process management master.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

## 1 Introduction

Between 1991 and 1993, the OSF Research Institute developed the OSF/1 AD operating system for multicomputers [3]. Version 1 of OSF/1 AD became operational in 1992 and has served as the basis for Paragon OS as shipped on the Intel Paragon. Version 1 is also the basis for other vendors' technologies.

Presently, the OSF Research Institute is engaged in the design of OSF/1 AD Version 2. This version includes enhancements in the areas of cross-node Mach IPC, server co-location in the system address space, a vproc-based distributed process management, and a new vnode-based file system architecture.

Unlike in non TNC Version 1, the Version 2 process management provides a large degree of local process management functionality on each node. Process management is broken up into a number of subsystems, some handling all requirements of a local process and are distributed among all nodes, and others handling interprocess relationships such as groups and sessions, centralized in a single server for the whole system.

After providing an overview of OSF/1 AD, this document describes at a high level the new process management architecture of OSF/1 AD Version 2. More detailed information will become available in the future.

## 2 OSF/1 AD - A Multicomputer Version of OSF/1

OSF/1 AD extends OSF/1 to a multicomputer environment, providing a full *single system image*. Its architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. OSF/1 AD must allow all nodes and devices to be efficiently utilized and, in particular, must distribute OS functionality to avoid bottlenecks., improve fault reslience, reduce latency and reduce messaging.

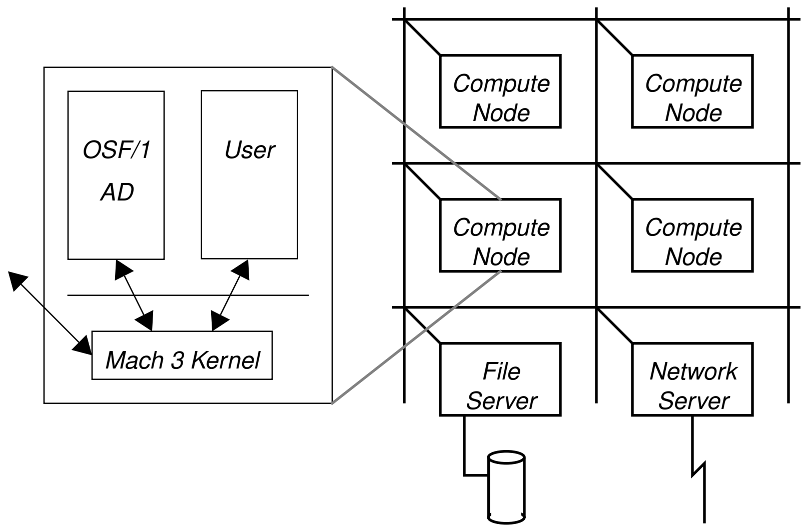



Figure 1. OSF/1 AD Version 2 Architecture

These considerations lead to the following model for the Version 2 architecture (see Figure 1):

The Mach kernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.

Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.

Disk and networking devices are managed by servers which are typically co-located on the same node with the device, although co-location is not mandatory.

Each application has a single server responsible for receiving all its system calls. This server is referred to as a *system call agent*. The binding of applications to system call agents, and the binding of system call agents to nodes is flexible, but typically, all applications running on a node use a local system call agent.

Process management functionality is distributed across system call agents. Process management system calls involving a single process are handled entirely by a local agent, while calls involving interprocess relationships or multiple processes require remote interaction.

File management system calls are handled by an applications's system call agent. This agent provides access to remotely managed files through a vnode-based distributed file system.

OSF/1 AD implements the entire OSF/1 Application Programming Interface (API). Although OSF/1 AD assumes no physical shared memory between nodes, it supports shared memory between processes residing on different nodes (via the Mach XMM subsystem). Transparent interprocess communication is provided by an extended Mach IPC to support cross-node communication (NORMA IPC).

The major differences between Versions 1 and 2 of OSF/1 AD include the absence of an emulator in each process's address space, the use of vnode technology to implement the distributed file system, more distributed process management and the system's ability to co-locate a server in the kernel address space [1]. The latter optimizes the interaction between application and system call agent during system call execution.

## 3 Process Management Requirements of OSF/1 AD Version 2

This section briefly describes what is to be achieved with the process management architecture.

### 3.1 Requirements

- Single System Image (single pid space)
- Distributed Execution using rfork
- Adequate performance to support self hosting
- Scalable to a reasonable number of nodes (some bottlenecks acceptable)
- OSF/1 API compliance and application source compatibility
- Configurable Compute Nodes
- Failure Containment (node failure does not propagate)
- Support addition of LOCUS TNC code
- VPROCs as formal System Programming Interface

### 3.2 Goals

- Better distribution of process management functionality than Version 1
- Minimize forced cohabitation with other servers (tty, network, file system etc)
- Clean interface between process management and filesystem
- Clean implementation of interruptible system calls
- Better rejection of non-existent nodes

### 3.3 Non-goals

- rexec
- Failure Recovery
- Process/task Migration

## 4 Overview of the vproc layer

The vproc layer implements a generic interface allowing insertion of third party supplied vectors into the process management code in a similar way to file system's use of VNODES. Its purpose is to allow for the distribution of process management through simply replacing a vector of routines attached to the vproc structures (similar to vnode/file systems) and accessed through the already added VPOP hooks.

The vproc philosophy divides the traditional process management into two distinct layers. The layers involve both data structures as well as code paths. The first layer handles the inter-process relationships, and the second handles functionality involving a single process.

The first layer contains a generic structure called the vproc which holds the following information:

**Table 1: The vproc structure:**

| Field | Description |
|---|---|
| Pid | The traditional pid of a unix process |
| Data | An implementation dependent structure used for distributed process information |
| Hash links | Pointers used to hash the vproc structures. |
| Ops | Operation vectors for process management operations |
| Ref counts/locks | Miscellaneous reference counts, debugging and locking parts |

The vproc structure is common among all implementations, while the *data* part allows for specific implementations of (distributed) process management. A copy of the vproc structure is available to any system entity making a process management call but the *data* part and the operation vectors of the vproc depend on the entity in which the vproc is present. The information in the *data* part, along with the operation vectors, determine where and how a remote operation is to be performed.

The second layer contains the part of the traditional UNIX proc structure holding information for an individual process and is resident on the process management server responsible for a process. This layer is common across implementations and has no (or minimal) knowledge of remote entities.

All process management calls which potentially require remote operations are funneled through VPOP_XXX operations which implement the hook into the vproc layer. This layer determines which process management server(s) are to be contacted to perform actions on individual processes in order to satisfy the system call. Some process management calls are implemented completely locally and thus never have to go remote consequently these will not have corresponding VPOP_XXX operations. Such calls are handled in the same way as a non-distributed single server.

## 5 Process Management Architecture of OSF/1 AD Version 2

In Version 2, process management is distributed across all nodes in the multicomputer, and relies on VPROC technology for its implementation. Process management agents implement the client side providing local services for processes, while a single process management master provides services involving inter-process functionality. (See figure 2).

The remainder of the document discusses the particular aspects of the process management architecture.

## 5.1 Process Management Modules

Process management functionality is split into six logically separate components, although some of them reside in the same server. These components are the Process Management Master (PMM), the Process Management Agent (PMA), the Server Programming Environment (SPE), the Process Management Library (PML), the pid broker and the Process Placement Policy Module. The diagram shows schematically how the functional parts are distributed.

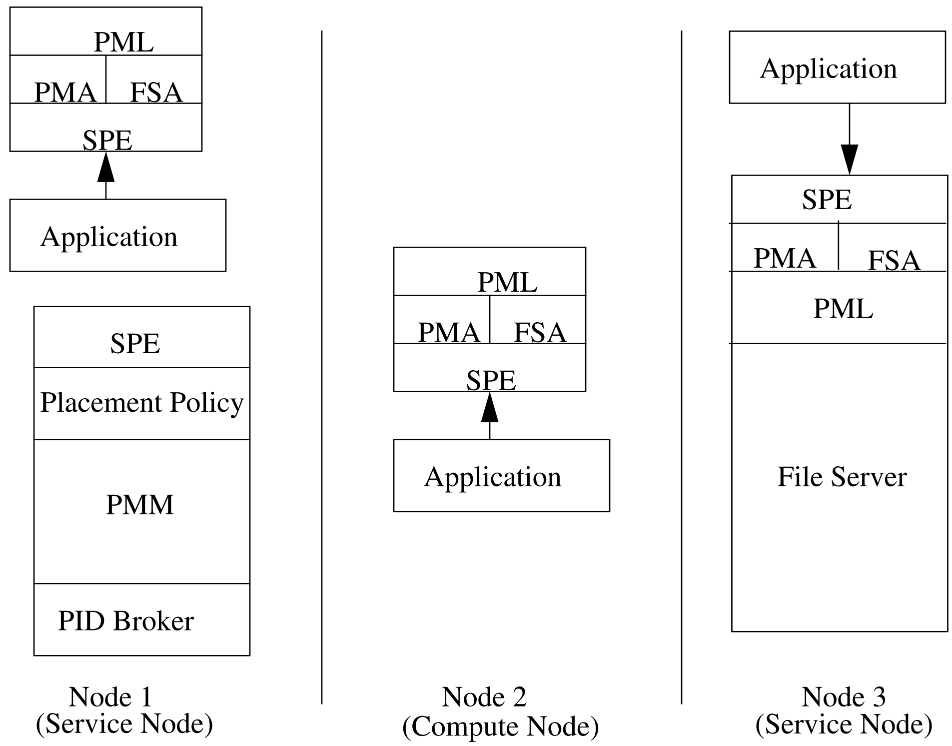



Figure 2. Distribution of Process Management Services

Each node supporting processes has a task consisting of a process management agent, a server programming environment and the process management library. This agent task also contains the local file system agent and potentially other local agents or servers.

The process management master is a separate Mach task from the per-node agent, even when they happen to be executing on the same node. There is only one process management master server in the system and it also contains the pid broker and the Process Placement Policy Module although these are logically separate.

The PML contains the client side of exported process management services. Consequently it is linked into any server wishing to use process management services (e.g. use vproc operations).

## 5.2 The Server Programming Environment

The Server Programming Environment (SPE) is not logically part of the process management code in that it does not provide any process management functionality. The function of SPE is to provide all servers with a set of general purpose functions providing the following functionality:

- A minimal runtime environment consisting of thread, task and uthread structures.

- Simple and complex locking.
- Events, thread blocking and wakeup.
- A managed pool of threads servicing messages arriving on a port set.
- Routines to insert a port into or remove it from this port set.
- Routines to register and de-register a local or forwarded service request for interruption.
- A way to interrupt a blocked local or remote service request.

The above functions require no knowledge of UNIX structures and provide basic services available to all servers.

#### 5.2.1 Message dispatching

The SPE provides a pool of threads whose job it is to listen for messages arriving on a set of ports. This set of ports consists of the ports inserted by the server using the routines also provided by the SPE. Servers linked with the SPE register servers, normally MiG generated, to be called to handle the messages that will be sent the the ports that this server has registered. A process management server handling system calls, for example, would register the task and thread exception ports with the SPE port set and the exc_server() as a server to handle the messages that will be sent to them.. When one of the server threads receives a message from the port set, it scans through the table of registered servers, calling each one in turn, until one of them recognizes and handles the message. The SPE will also arrange for the reply to be sent, if one is needed.

The pool of threads is managed by the SPE to ensure that there is no starvation (i.e. that there is always a thread to receive a message and that there are enough kernel threads to ensure forward progress) and to conserve resource consumption (i.e. that there are never too many of these threads sitting idle).

Some of this functionality might be eventually supplied by the server collocation project[1].

### 5.3 The Process Management Agent

The process management agent module is a distributed entity and is always linked with a file system agent. Its functionality includes;

- Handling of all process management services manipulating local processes.
- Provide services to the server programming environment to help provide other local agents with an execution environment identifying the client process, as requested in the filesystem design[2].
- Manage the interruption of system calls in progress using services provided by the server programming environment.

This entity implements a vproc layer to handle distribution of process management and a physical process layer providing services for calls involving local processes. Any process management service involving an individual process, no matter what the point of origin, will eventually be handled by the process management agent local to the target process.

As mentioned in the vproc section, the process management agent holds a vproc and a proc structure for each local process which are set up at (r)fork() time. These structures provide all the information required to process system calls involving the local process. The code in this entity closely resembles that in an undistributed single server. In addition to a pointer to the proc structre, the vproc structure's *data* section holds a handle to the process management master which is to be contacted whenever processing a call involving relationships with other processes. These include fork(), kill(), wait() and exit(). The process management master might be contacted for several reasons during calls involving the following:

- Resolution of a pid to a target process management agent when a call targeted to a specific pid is made.

- Setting up or tearing down inter-process relationships due to calls such as fork(), exit(), or those manipulating group or session relations.

- Traversal of inter-process relationships on behalf of calls such as signals to a process group.

In order to lower the burden on the process management master, care is taken to minimize its involvement in system call processing as much as possible. This can be achieved by recognizing that in general, calls requiring inter-process relationships can be divided into two types. Those where the target process is explicitly indicated, and those where the target process(es) are implicit. An example of the first type is a signal send to a particular pid via a kill() system call, while an example of the second type is a signal sent to a process group. In at least some of the first type, it is possible to by-pass interaction with the process management master and directly contact the target process management agent, if the pid to agent mapping is cached. Detailed examples of some of these calls are given in appendix A.

### 5.4 The Process Management Master Architecture

The process management master module is a single entity running as an independent task on a single node. Its functionality includes:

- Handling of all inter-process relationships.

- Handling of process-session relationships and calls to get/set controlling ttys.

- Pid brokerage.

- Process placement policy

This entity contains all inter-process relationships, as well as session structures. Each live process has a vproc structure set up at fork time, with a *data* part pointing to a structure called a pvproc. The pvproc structure contains inter-process pointers pointing to vproc(s) of related processes, as well as the session structure when appropriate. The *data* part of the vproc structure also contains a handle to the process management agent handling the process. The process management master can only be reached through a process management agent or the process management library and is only contacted when an operation involving inter-process information is performed. As mentioned before, at fork time, a process management agent contacts the process management master asking it to set up the vproc for the new process and set up the pointers from the pvproc structure of the parent to the child's vproc, as well as those indicating group and session relationships. Every time a call is made that modifies the above relations, the process management agent processing the call will ask the process management master to update these pointers or to free the vproc/pvproc in case of exit().

Certain system calls, such as sending signals to a process group, will cause the process management master to traverse the group pointers calling into the appropriate process management agent at each hop in order to post the signal.

## 5.5 The PID Broker

Process IDs are managed by part of the process management master called the pid broker and by functions provided by the process management library. The pid broker is responsible for allocating pids to agents and providing a lookup service for clients to locate the agent managing a given pid. Note that the pid broker only provides a location service thus a pid may have been allocated to an agent but there may not necessarily be an active process associated with it and the client must contact the agent in order to determine if the process really exists. The pid broker is not visible to any non-process management code directly and is a process management internal service. It is used mainly inside the VPOP interfaces to locate servers when the process is remote to the requestor.

When a vproc operation is called for the first time on a vproc with a null data field or a vproc data structure containing a missing agent handle, the pid broker is contacted to locate the agent responsible for the process. The client side of the pid broker determines if the agent for that pid is already known. If so, the handle is stored in the vproc data structure and the vproc operation can continue. If not, the client pid broker requests a pid-to-agent lookup from the master side of the pid broker, which resides in the process management master. The master side responds with a send right of a port to the agent and the range of pids spanning the requested pid that this agent has also been allocated. The client side of the broker will cache this range to avoid future contact with the broker master when possible.

In OSF/1, proc structures are located by indexing into a pid_entry structure. The index into this structure is calculated from a pid in such a way that collision of two active pids onto one pid_entry is impossible. The pid space is divided up into chunks the size of the pid_entry table. This implicitely allocates a number of pids to each pid_entry slot (N * pid_entry table size + entry_index for N = 0 to PID_MAX/pid_entry table size) which cycles through them when a new process is created using that slot. This has the advantage that no locks are taken to find proc structures from pids and no complicated look-ups are performed (entry_index = pid %pid_entrytablesize) In order to preserve these features, the pid broker manages a logical pid_entry table whose size is the sum of all the agents pid_entry tables. The size of this logical table is configured at boot time and agents request allocation of sections of it from the pid broker when they initialize. This implicitly allocates ranges of pids that this agent may use for processes it manages in much the same way as the single server code. Agents do not allocate more pid_entry slots later but the architecture of the pid broker does not preclude this. In order to support a large number of nodes, the pid space is increase from a short to a long, forcing the few places which 'know' it is a short (mainly System V IPC) to be fixed. The broker notes which agents have been allocated which pid_entries (and thus pids) for future lookup requests. This allocation is also noted in the process management library broker so that the local agent may lookup local pids and find that it manages them itself without having to communicate with the pid broker.

If the process migrates, the pid broker will flush the range caches held in the process management library brokers so that future references will reload the caches with the fragmented pid ranges around the migrated process. As the first release of OSF/1 AD version 2 will not support process migration, this will not be an immediate issue.

## 5.6 Process Placement Policy Module

The standard policy module in OSF/1 AD Version 2 is not intended to perform any intelligent load balancing but is more a place holder for when intelligent load management can be provided. As process migration is not supported, the only use of the policy placement module is by fork(). Processes are created on other nodes in two ways, either using an explicit rfork() call or by a normal fork() call. Each process contains an attribute field in its proc structure which indicates which policy to use for its placement. The FORK_POLICY_LOCAL attribute can be set or unset using a special system call. The default policy is to use the policy placement module.

When executing fork(), the process management agent of the parent process determines which agent will manage the newly created child. In the case of rfork(), the parent agent looks up the process management agent controlling the specified node. If the parent process has the FORK_POLICY_LOCAL attribute then the parent's agent will also be the child's agent. Otherwise the parent agent contacts the placement policy module which returns a send right to an agent it has selected to manage the child.

The algorithm for the placement policy module is initially simply round robin among the service nodes. No compute nodes are used (an explicit rfork is needed to launch processes on these nodes) and no attempt at load balancing will be made. The algorithm (or lack of it) is not be visible to the agent and so more sophisticated placement policies may be used at a later date.

## 5.7 The Process Management Library

The process management library defines the client side exported interfaces of the process management sub-system, together with many macros defined in header files. It contains any functions needed to implement the VPROC operations used by clients of process management and many support functions implementing things like caching of pid-to-agent lookup, lazy vproc data initialization, etc. The main difference between this component of the rest of the process management code is that this is not a server and will not have its own threads, but provides process management services to servers and is called by their server threads.

## 5.8 VPROC operations

In OSF's implementation of OSF1/AD Version 2, process management will involve co-operation among the PMM, PMA and PML described above. Each of these hold a vproc structure per process being acted upon. The *data* and operation vectors however will differ on each entity. (See figure 3).

The *data* part points to an intermediary structure which in turn has either local or remote representations of two structures, one holding the inter-process relationships and the other the proc structure. The local representation of inter-process relationships is a structure called the pvproc. It resides only on the process management master and contains the following;

**Table 2: pvproc structure**

| Field | Description |
|---|---|
| pvp_flag | pvproc attributes |
| pvp_head_childl | First child of parent |
| pvp_childl | Next child of parent |
| pvp_head_foster_childl | First foster child of parent |
| pvp_head_pgrpl | Pointer to first pgrp member |
| pvp_pgrpl | Next pgrp member |
| pvp_sessionl | Pointer to pgrp leaders in session |

**Table 2: pvproc structure**

| Field | Description |
|---|---|
| pvp_ppid | Parent process id |
| pvp_foster_ppid | Pid of process that will inherit accounting stats when this proc is reaped |
| pvp_pgid | Process group leader id |
| pvp_jobc | vproc(s) contributing to non-orphan status (valid on group leader vproc only) |
| pvp_session | Pointer to session structure |

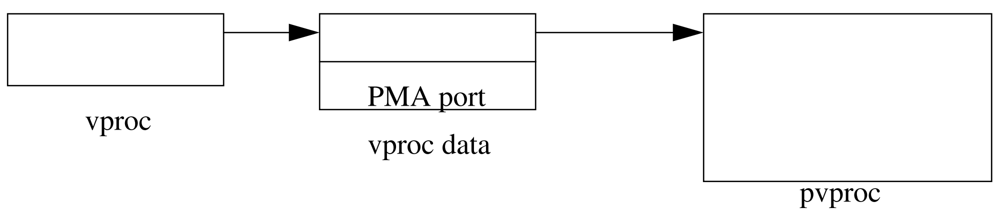
The remote representation of the pvproc structure consists of a port to the process management master and is found on all other entities.

The local representation of a process is the proc structure which only resides on the process management agent handling the process. This structure is a traditional UNIX proc structure without inter-process relationship pointers. The remote representation consists of a port to the appropriate process management agent and is found on all other entities.

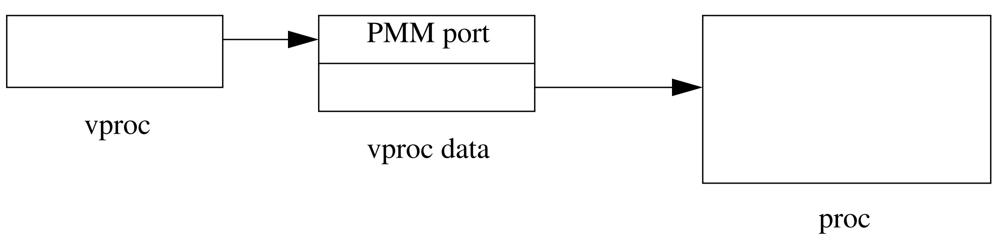
Similarly the operations vectors for operations either on processes or those involving inter-process relationships, point to functions invoking RPC(s) to the remote service if the corresponding *data* representation is a port. Otherwise the operation vectors point to the local functions that implement the service directly.

The organization of vproc structures in each entity is shown in the figure below.

A) Process Management Master




B) Agent where the process is local



C) Server/Agent where the process is remote

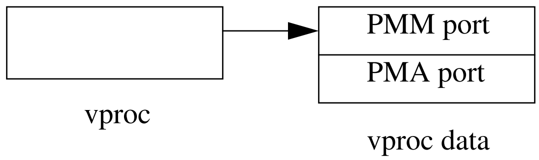



Figure 3.

All calls that could potentially be remote are made as VPROC operations. In order to make VPROC calls, a vproc structure representing the target process is required. This structure is obtained via a LOCATE_VPROC_PID call. This structure must be released when the operations on that process are complete using VPROC_RELEASE. Consequently the only data needed to be able to invoke operations on processes is the pid.

An expected list of VPROC operations is given below.

**Table 3: List of VPROC operations**

| Operation | Description |
|---|---|
| LOCATE_VPROC_PID | Get a vproc given a pid |
| VPROC_RELEASE | Release a vproc |
| VPOP_FORK | Unix fork |
| VPOP_EXIT | Unix exit |
| VPOP_WAIT | Unix wait |
| VPOP_PROC_NICE | Nice a process |
| VPOP_PGRP_NICE | Read/write process nice value for process group |
| VPOP_SIGPROC | Signal a process |
| VPOP_SIGPGRP | Signal a process group |
| VPOP_SETSID | Set session id of a process and create a new session |
| VPOP_SETPGID | Set session id of a process and create a new session |
| VPOP_GET_ATTR | Get relationship attributes for process |
| VPOP_SETPINIT | Converts process to a child of init |
| VPOP_SESSION_GETTY | Get the tty vnode handle of the controlling terminal for the caller’s vproc |
| VPOP_SESSION_SETTY | Set the tty vnode handle of the controlling terminal for the caller’s vproc |
| VPOP_PTRACE | Process tracing functions for a child process |

**Table 3: List of VPROC operations**

| Operation | Description |
|---|---|
| VPOP_TERMINAL_SIGPGRP | Sends terminal access control signal to process group |
| VPOP_RFORK | Unix fork but to a target node |
| VPOP_RFORKMULTI | Unix fork but to a set of target nodes |

### 5.9 Interruptible System Calls

System calls are interrupted when the arrival of a signal needs to unblock the thread that is suspended during the system call. Threads are registered as having system calls in progress when the system call arrives. When a local agent decides that the request should be forwarded to a remote server, a port to this server and a unique transaction id are also registered. The process management agent interrupts the system call using clear_wait(), as in the integrated kernel. This function attempts to unblock the thread if the system call is still being handled locally and will send an interrupt RPC to the registered server if the request has been forwarded.

The interruption request sent to the server will contain the transaction id that the local agent registered with the server programming environment. The remote server may reject any request for a transaction id that is not active in the server, which means that the server does not need to remember which transactions have already been processed. Consequently, clear_wait() code on the agent must be prepared for the interrupt to fail if either the interruption has overtaken the system call request or if the server has completed the request before the interruption arrives, In either case, the process management agent will loop sending interruptions until either the interruption is acknowledged or the call is no longer registered as being remote.

### 5.10 System Wide Operations

Certain operations such as setting host name or halting the system do not affect a specific process but the whole system. Therefore a new set of operations called VPSOP_XXX have been defined. These calls may be initiated on any agent in the system and will have the same result despite the PMA of origin. The system call paths for these operations are similar to VPROC based calls, but will go through the PMM in all cases as long as initial permissions and other checks are successful. When the PMM is contacted on behalf of one of these calls, it will contact all active PMA(s) and ask each of them to complete the call. Once all PMA(s) have completed the call, the PMM will return to the calling PMA with either success or an appropriate error code. Since the PMM must potentially contact a very large number of PMA(s) efficient algorithms may be required.

### 5.11 Lightweight Process Management Agent

One of the requirements of our design is that we support a lighter-weight agent. However, close examination of the file system agent requirements shows that much of the full process management code needs to be present in any lightweight server for two reasons. In order to have a functioning file system agent, we need the Process Management Library, and all the associated code. In addition, in order for the fileserver agent to have acceptable performance, we need to have considerable caching of process related information; therefore at this stage it is not clear what advantage will be gained in breaking up the actual process management agent's code beyond that which we have already decided. It is however possible to reduce the size of pre-allocated resources in a process management agent simply through configuration variables. In the first revision we intend to implement lightweight server through the latter approach and leave any separation of the process management agent into subsystems for future study.

### 5.12 Node Failure Containment

OSF/1 AD Version 2 supports minimal failure handling. When a node goes down, the process management master will obtain a dead name notification for the process management agent's main port. The process management master then proceeds to cleanup any structures involving processes on the dead node as if all the processes handled by that agent had just called exit. This allows the rest of the system to proceed while denying access to any processes under control of the failed agent. This design needs to be accompanied with similar cleanup code in other non process management subsystems in order to achieve full failure contaminant.

### 5.13 Distributed Process Management Master

The current design out-performs the model in OSF1/AD Version 1 in the sense that the Process Management Master is only invoked for a subset of process management system calls, and even then it hands most of the work to process management agents.Despite that OSF1/AD Version 2 does suffer from a bottleneck in its process management subsystem, as it has a single process management master. The general architecture however does not preclude distribution of the master. In fact we fully expect third parties to distribute this functionality thus achieving a system with no bottlenecks in the process management. The ease of the above goal is due to the fact that the process management master can be replaced as a module with a distributed version by simply replacing the VPROC vector pointers much like the way it is done with TNC in OSF1/AD Version 1. In fact it will be possible for any recipient of this technology including OSF to implement their own distributed and/or replicated process management master with whatever degree of distribution/replication desired.

### 5.14 Process Migration

The non TNC OSF1/AD Version 2, will not implement process migration in the initial phase. However as in the case of distribution of process management master, it will be possible to implement and we fully expect TNC to do so. The general (oversimplified) algorithm will involve step similar to those mentioned below. The source process management agent will suspend the process, package its process and fileserver client information, call mach to migrate the task (assuming a task migration service exists), and call the target process management agent. The target process management agent will then (as in rfork) set up the local process information, call the local fileserver agent and setup the open files information, call the pid broker telling it the new process management agent, and finally resume the task. The decision of which node to migrate to will be made by a more complex Process Placement Policy Module than that implemented in the initial versions of OSF/1 AD Version 2, possibly based along the lines described in Milojicic et al[4].

## 6 Alternative Designs and Implementations

The above design was done using the VPROC interfaces in order to allow third party enhancements. Such a technique was used in OSF1/AD Version 1, where two distinct implementations exist both using the same VPROC layer, one from OSF with a single process manager, and the other the TNC system from Locus Computing Corp which provides a fully distributed peer to peer process management system.

In OSF1/AD Version 2 process management code is divided into two types, the implementation neutral code expected to be shared by all implementations and the implementation specific code expected to depend on a particular design.

The implementation neutral code consists of the following:

- All process management system call paths which are handled fully in the local PMA and do not require the VPROC layer.

- The upper code path of the system calls consisting of the SPEs server loop, the exception server and the upper layer of the PMA up to where the VPROC macros are called.

- The bottom layer of the PMA which is contacted by the vproc layer to achieve single process operations local to the PMA.

The implementation specific code consists of the following:

- All the vproc layer implementing distribution, that is communication between PMAs, PMLs and the PMM.

- All of the PMM including the pid broker and the process placement policy module.

- All of the PML.

Both types of code are provided by OSF and replacement or enhancement of the latter layer will allow distribution and/or replication of the PMM functionality allowing for peer to peer distribution as well as process migration.

## 7 Process Management Filesystem Interactions

There are a number of services provided by the process management system to the filesystem and vice vera. The process management services are enumerated below:

- Generic non UNIX services. These consist of the services provided by the SPE, as described earlier, and are provided to all servers including those of the filesystem.

- The VPROC layer. These services are provided by the PML as described earlier and will allow any server including those of the filesystem to make process management requests through a uniform set of interfaces, regardless of the location of the target processes, using the pid as only form of identification of the process.

- A UNIX runtime environment. At the time a system call is received, the exception server will call into the PMA in order to obtain a full UNIX environment for the target agent which may be the file system agent. This environment will then allow the target agent to access the real UNIX structures such a the proc structure of a process (minus the fields moved into the vproc layer) in the same way as in a non distributed single server.

Next are the services provided by the filesystem to the process management:

- General filesystem routines. The fileserver agent will allow the co-resident PMA to call certain filesystem routines. There will include pathname lookup (namei) and VOP routines as done in the non distributed single server.

- Tty representation. The PMM will keep an opaque handle to controlling ttys in its session data structures. This handle is provided by the filesystem to the PMM and will be retrieved by the file system code for functions like uprintf(). The file system must provide a method for the PMM to be able to extract device information from this handle. Due to this, there is no longer a necessity for the PMM and the tty code to be in the same server, as there was in Version 1.

## 8 Configuration Issues

There are two types of configuration information needed for the process management system; boot time and run time.

The boot time configuration consists of answers to two questions, on which node is the process management master to run and on which node is ‘init’ to be run. This is solved by the microkernel storing configuration strings passed to it at boot time using platform dependent techniques. Servers then call into the kernel to retrieve these strings.

The run time configuration needs to convert a node number and the name of a service to a port to a server providing that service for that node. This is needed so that PMAs can get send rights to a PMM port and so that both the PMM and PMAs can get send rights to other node’s PMAs. The problem is solved by running a name server per node, which will make itself available to all servers in the norma domain via a norma_set_special_ports() call to its local microkernel. Both the PMM and all PMAs register their service and their service ports with their local name server, under unique names (for that node) represented by ascii strings.

So at boot time, all PMAs will read the boot configuration strings to find the node locations of the PMM and init. The PMA running on the init node uses norma_get_special_ports() to find the name server for the node where the PMM is running. It will then lookup the “PMM” service with this name service which returns a port to the process management master. The PMA now has all it needs to be able to allocate some pid space and then create init.

Similarly when one agent needs to contact another, it uses norma_get_special_ports() to find the name server for the node that the target agent manages. It then looks up the “PMA” service with that name service.

Note that the PMM will never lookup a port for a PMA but waits for PMAs to contact it, which they only do when necessary, thus alleviating a boot time bottleneck.

A design for a sophisticated configuration manager has been proposed, however since an implementation of this service will not be available soon, it has been decided that OSF1/AD Version 2 will initially use the simpler method described above. It meets the basic needs OSF1/AD Version 2 and can be adopted to use more sophisticated methods when they become available.

## 9 Acknowledgments

Thanks to David Black for his guidance, Paul Roy for his input and letting us steal the Frame file for the file system architecture to use as the template for this document, and to Bill Bryant for his suggestions of improvement for readability and catching our stupid errofs.

## 10 References

[1] Condict, M., Mitchell, D. Optimizing Mach-based Systems By Server Co-location. OSF Research Institute internal document. August 1993.

[2] Roy, P., Noveck, D., Netterwala, D. OSF/1 AD Version 2 File System Requirements. OSF Research Institute internal document.

[3] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*.

[4] Milojicic, D., Zint, W., Dangel, A., Giese, P.Task Migration on the top of the Mach Microkernel. In *Proceedings of the USENIX Mach Symposium*, April 1993.

# Appendix A

# Analysis of Selected System Calls

This appendix describes how the various components in the process management system interact during the execution of selected representative system calls.

## sigaction()

The system call request is redirected to the process management agent. As the process is managed locally by this agent, the signal disposition is set into the utask or uthread structures. The process management agent then returns to the system call redirector which then returns to the application.

## fork()

The system call request is redirected to the process management agent. The PMA identifies the agent where the process it to be created by either a) looking up the agent managing the specified node, in the case of rfork, b) using itself if the parent process has the FORK_LOCAL attribute or c) contacting the Process Placement Policy Module.

The PMA asks the file system agent for the inheritable file state and collects the inheritable process management state. The parent PMA then sends this state to the target PMA with the request to create the child. The target PMA creates the new process and asks the file system agent to set the file state and sets the process management state sent by the parent. The PMM is informed of the child and its relationship to its parent. At this point the process becomes visible to the rest of the system. Finally the child is resumed by the target PMA which also replies to the parent PMA with the child's pid. Finally the parent PMA returns to the system call redirector which then returns to the application.

## kill()

The system call request is redirected to the local process management agent. The local PMA looks up with the pid broker which agent manages the process whose pid was specified in the call and sends the kill request to the target PMA.The target PMA makes the signal pending and attempts to deliver the signal in basically the same way as in the single server. Before attempting delivery, the PMA will attempt to unblock the target thread if it has been blocked during a system call using clear_wait().

To interrupt a blocked system call, the clear_wait() code determines if the thread has a system call in progress,. If there is a system call in progress, and it is still local, clear_wait() wakes the thread as in the single server. If the system call has been sent to a remote server, then the PMA will loop sending an interrupt message to the remote server until either the remote server confirms the interrupt as successful or the system call is no longer registered as remote.

## killpg()

The system call request is redirected to the local process management agent. The VPOP_SIGPGRP macro sends a mesasge to the PMM. The PMM finds the process group leader and then calls VPOP_SENDSIG to every process in that group. The action of the agent receiving this message is the same as that in kill() above.

# Appendix B

## Analysis of locality of process management system calls

The following appendix examines various process management system calls as to how they will be handled by the OSF1/AD Version 2 process management. The goal is to determine whether a call can be handled by a local PMA, through PMA - PMA interactions or there is a need for PMM intervention. The first entry will name the system call or class of calls, and the second entry will point out the type of interaction required to handle it. A letter (c) indicates the call is a representative for a class of calls.

**Table 1:**

| syscalls | Interaction Types |
|---|---|
| killpg | Local PMA - PMM - remote PMAs |
| setpgrp | Local PMA - PMM |
| killpid | Local PMA - remote PMA |
| fork (c) | Local PMA - PMM - remote PMA |
| wait | Local PMA - PMM |
| setsid (c) | Local PMA - PMM |
| exit | Local PMA - PMM |
| table | Each command handled differently |
| mmap (c) | Local PMA |
| profil | Local PMA |
| setrlimit (c) | Local PMA |
| sigaction (c) | Local PMA |
| setpriority (c) | Local PMA - remote PMA |
| getpid (c) | Local PMA |
| umask | Local PMA |
| getgroups (c) | Local PMA |
| setgroups (c) | Local PMA |
| obreak (c) | Local PMA |
| exec (c) | Local PMA |