The Design of the Process Management Component of OSF/1 AD Version 2

OSF Research Institute

Simon Patience

Faramarz Rabii

{sp,rabii}@osf.org

November 15, 1993

Abstract

This paper describes the design of the components 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.

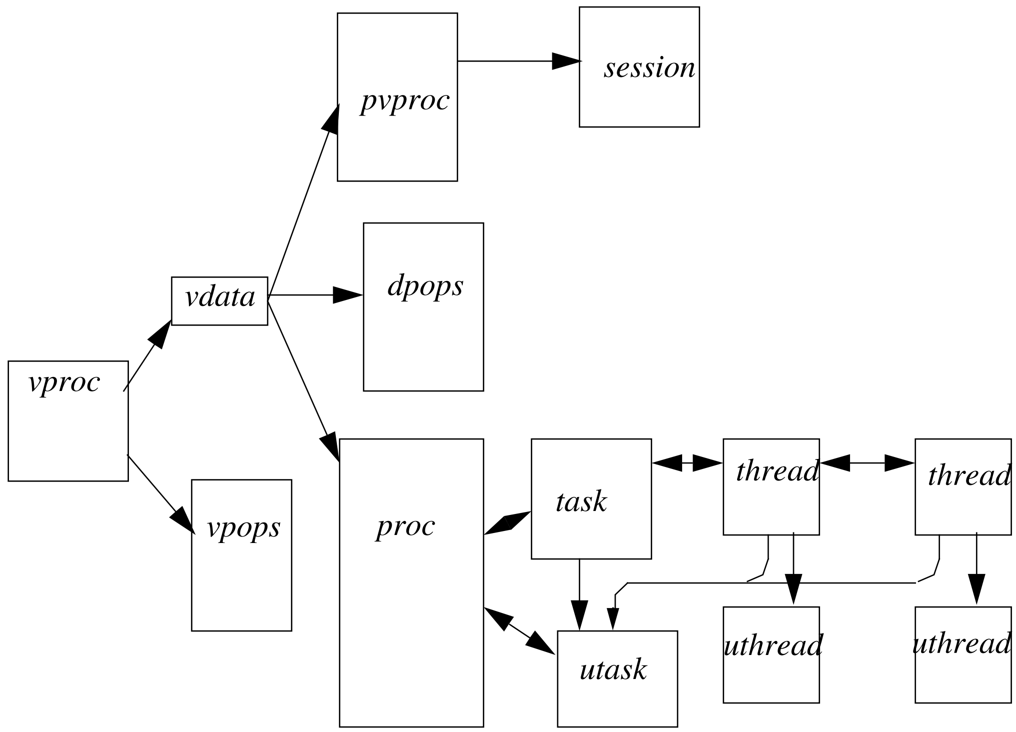
This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

1 Introduction

This document is intended to describe in detail the design of the process management part of OSF/1 AD version 2. It describes data structures and algorithms to be used in the implementation of the architecture described elsewhere[1].

2 Major Data Structures

In OSF/1, both the integrated kernel and the single server, the data structure used to describe a structure is the 'proc' structure. In OSF/1 AD Version 2, as with version 1, the basic structure is a 'vproc' which acts as a handle to the process. The vproc structure is needed to perform any operation on a process and, on some servers, also contains pointers to other data structures containing more information about the process. The figure below describes how all the structures describing a process are connected although not all data structures will be present in every task.



The vdata structure pointed at by the data field of the vproc structure is the implementation dependent structure which connects the structures together. In OSF/1 AD, this structure will contain two components, one for the pvproc and one for the proc. Each component contains a flag indicating if this structure is present and then either a pointer to the structure is present or the name of a port to the server that handles requests for the process. In addition, the vdata structure contains the a vector of operations, the dpops, which distribute the process management operations. These vectors are the way to distinguish between local and remote services and for remote services, which server is to provide them.

The pvproc structure contains all the process relationship information that used to reside in the proc structure (e.g. parent pid, group and session information, etc). In this way, the information describing the total state of a process can be distributed between servers. The vpops structure is a vector of functions that are to be called to perform operations on the process. These vectors enable third party distribution mechanisms, such as TNC, to easily replace the OSF/1 AD process management distribution.

In addition, there is a slight change in the organization of the pid_entry table. In the integrated kernel, the pid_entry table replaced the statically allocated proc table. Each active entry in the pid_entry table pointed to a valid proc structure, which had been dynamically allocated. Now that the proc structure is no longer the starting point for the description of a process, the pid_entry structures now contain a pointer to a vproc.

The pid_entry table is only found in servers which create processes, i.e. process management agents. In other tasks, vprocs are stored in a hash table using the pid (which is stored in the vproc) as the hash key. Thus in many PMAs, which both create processes and reference remote processes, both a pid_entry table and a vproc hash table will be present and both tables will have to be searched to locate a vproc. If the vproc is not found in either of these tables, then a vproc is allocated with its vdata indicating that it is remote and is then entered into the hash table.

3 System Layering

The process management functionality is implemented in three layers, called the virtual process (vproc), distributed process (dproc) and physical process (pproc) layers. Each layer performs a different task in the distribution of the process management function.

3.1 The virtual process (vproc) layer

The vproc layer is similar to the vnode layer, which provides an object abstraction of a file, in that it provides an object abstraction for a process. The process abstraction allows for different kinds of process distribution implementations. The list of operations supported at the vproc layer, the VPOP interfaces, are well defined by Locus and have been freely given to OSF. The intention in OSF/1 AD V2 is to adhere to these definitions as closely as possible but they may have to be modified/augmented to support OSF/1 specific semantics but this will only be done if there is no feasible alternative. The VPOP macros call the dproc layer (either locally or remotely) using the vpop vector associated with each vproc structure. The list of VPOP interfaces as specified by Locus[3] are listed in Appendix A.

3.2 The distributed process (dproc) layer

The distributed process layer implemented the support for locating and calling the provider of given service for a specific process described by a vproc structure. If there is a possibility of the service being provided remotely, either because the target of the operation is a process which may not be the caller or because process relationships are affected, the dproc layer will call the service using the dpop vector of operations associated with the vdata structure pointed to by the vproc structure representing the target process. The functions in the vectors will either be local stubs layered on the pproc layer (which are also used by requests initiated remotely in the dproc layer of another server) or will be RPC stubs which send the request to the appropriate server.

3.3 The physical process (pproc) layer

The pproc layer implements the functions that directly manages the physical process (i.e. the task and threads) and access to the structures associated with it (i.e. proc, utask, uthread, task and thread). The set of operations in this layer has been defined by Locus[3] and their list is in Appendix B. As with the VPOP operations, the intention in OSF/1 AD V2 is to adhere to these definitions as closely as possible but they may have to be modified/augmented to support OSF/1 specific semantics but this will only be done if there is no feasible alternative.

4 The Server Programming Environment

The server programming environment provides any server with some basic services of server thread identity and synchronization primitives. This environment is very similar to that found inside the Mach 2.5 part of the OSF/1 IK and also the OSF/1 MK V5 single server. The figure below describes the data structures and the thread identity calls available to a server thread.

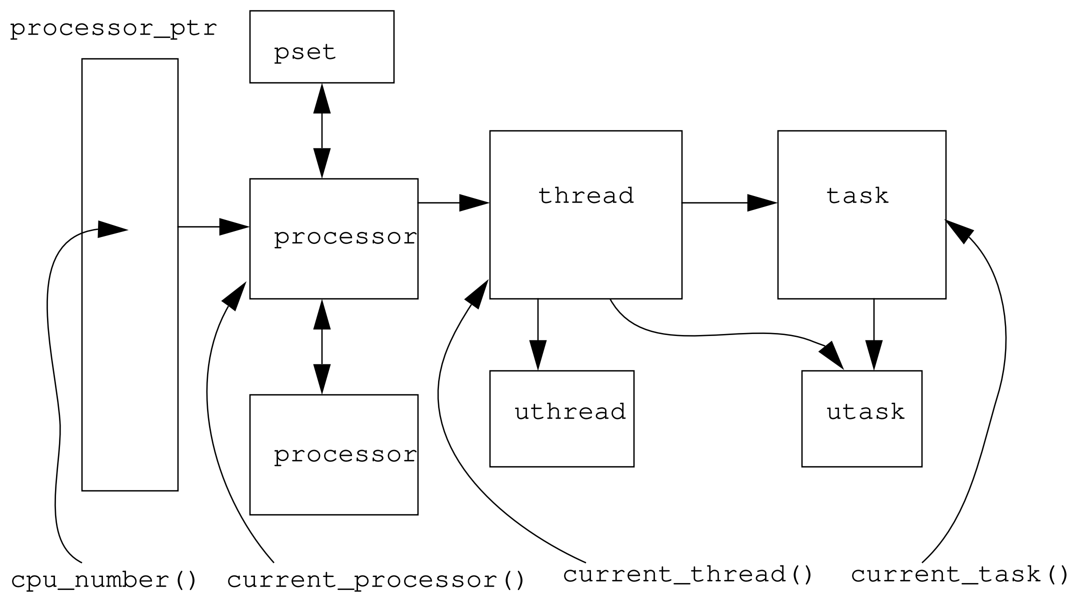


Figure 1. Server thread identity structures and functions

The concept of a cpu in the server environment is not a hardware cpu but is a user level thread, in our case a cthread but this environment allows the detachment of writing servers to the threads library used. It should never be necessary to use cthreads functions in normal server code.

Similarly, a processor really represents the server thread. In system call servers there may actually be two thread structures (and so identities) associated with a processor (the system call thread representing the client and an idle thread) but other server threads will only have one thread structure attached to the processor. A server thread will always see the same return value for cpu_number() and current_processor() for duration of its life.

The uthread and utask structures are really unix notions and in a perfect world would not be needed however they will present in the environment but a server thread can not expect the contents of either of them to be sensible unless it put the data there itself. The task will be the same (the server task itself) for all threads in the server except those handling system calls who have temporarily assumed the identity of the client.

There are a number of synchronization mechanisms available to servers. These break down into locking, events and sleep/wakeup.

4.1 Locks

Both simple locks and complex locks are supported. Simple locks are short term locks and not designed to be held for extended periods nor while blocked. Complex locks are multiple reader/single writer locks.

The list of calls available for simple locks are;

```
void simple_lock_init(simple_lock_t lock);
void simple_lock(simple_lock_t lock);
void simple_unlock(simple_lock_t lock);
boolean_t simple_lock_try(simple_lock_t lock);
```

The list of calls for complex locks are;

```
void lock_init(lock_t lock,
                boolean_t can_sleep);
void lock_write(lock_t lock);
void lock_read(lock_t lock);
boolean_t lock_try_write(lock_t lock);
boolean_t lock_try_read(lock_t lock);
void lock_write_to_read(lock_t lock);
void lock_done(lock_t lock);
void lock_read_done(lock_t lock);
void lock_write_done(lock_t lock);
void lock_set_recursive(lock_t lock);
void lock_clear_recursive(lock_t lock);
```

4.2 Events

A simple event facility is provided. The list of calls for events are;

```
int event_wait(event_t *event,
                boolean_t interruptible,
                int timeout);
int event_posted(event_t *event);
void event_clear(event_t *event);
void event_init(event_t *event);
void event_post(event_t *event);
```

4.3 Sleep/Wakeup

Basic thread to thread synchronization is achieved using the sleep/wakeup calls.

The list of calls for sleep/wakeup are;

```
void assert_wait(int event,
                boolean_t interruptible);
void assert_wait_mesg(int event,
                boolean_t interruptible,
                const char *message);
void thread_sleep(int event,
                simple_lock_t lock,
                boolean_t interruptible);
void thread_wakeup(int event);
void thread_wakeup_with_result(
                int event,
                int result);
void thread_wakeup_one(int event);
void thread_block(void);
void thread_yield(void);
void thread_set_timeout(int ticks);
void clear_wait(thread_t thread,
                int result,
                boolean_t interrupt_only);
```

4.4 Interruptible System Calls

When a local agent decides that a system call request should be forwarded to a remote server, the id of this server and a unique transaction id are registered with the SPE using the call remote_server_register(). The local agent must then call remote_server_deregister() when the remote operation is completed. At the remote server, the thread that is to service this request must first register itself and the same transaction_id with its SPE using server_thread_register().

```
void remote_server_register(ad_server_id_t remote_server_id,
                long transaction_id);
void remote_server_deregister();
```

Threads are able to be interrupted in the same way as in the single server, using the function clear_wait(). This function attempts to unblock the thread if the system call is still being handled locally but will send the interrupt RPC remote_clear_wait() to the registered server if the request has been forwarded. A server thread on the remote server will use server_thread_lookup() using the transaction id sent with the interruption request to find the server thread performing this transaction and will call clear_wait() for this thread. If the lookup fails then remote_clear_wait() will return UNKNOWN_TRANSACTION_ID.

```
kern_return_t
remote_clear_wait(mach_port_t server_port,
                long transaction_id,
                ad_server_id_t requesting_server_id,
                int result,
                boolean_t interrupt_only);
void server_thread_register(long transaction_id,
                ad_server_id_t requesting_server_id);
int server_thread_lookup(long transaction_id,
                ad_server_id_t requesting_server_id,
                thread_t *server_thread);
```

If the remote_clear_wait() function returns UNKNOWN_TRANSACTION_ID then the clear_wait() code will loop sending interruptions until either the interruption is acknowledged or the call is no longer registered as being remote.

The same mechanism is used at the remote server, thus allowing remote requests to be forwarded to other servers and still be interruptible. The remote server must call remote_server_register() before forwarding the request and the server receiving the forwarded request must also call server_thread_register() before processing the request.

The information triple thread/transaction_id/server_id is stored in a substructure of the thread structure of the thread calling server_thread_register(). This structure is also entered into a table hashed on server_id. This allows for a simple lookup of thread to transaction_id/server_id, needed to call remote_clear_wait(), and of server/transaction_id to thread, needed to implement remote_clear_wait(). Obviously, remote_clear_wait() cannot be called until the server port has been looked up from the server id.

5 The Process Management Agent

The Process Management Agent known as the PMA is the part of the process management which is most closely connected with user level processes. It receives system calls from processes and provides services for other agents such as the fileserver agent. The general architecture of this entity has been discussed in the Process Management Architecture document [1]. Here we will discuss detail of this entity.

5.1 Data Structures

In this section we discuss data structures in the PMA which are either new or have been modified from the non distributed server.

5.1.1 The pid_entry Table

The PMA keeps track of all local processes via a pid_entry table. This table as described in section 6.1 is similar to that used in the non distributed server and holds pointers to the 'vproc' structures for all local processes.

5.1.2 The 'vproc' Structure

The PMA contains a vproc structure for all local processes and any remote process being acted upon. This structure is allocated in two ways depending on the locality of the process.

- For a local process, the 'vproc' is allocated at fork time on the target PMA. A pointer to he structure is then inserted in the pid_entry table and can be found as described in section 2. In this case the 'vproc' operation vectors are set to local routines. The 'vdata' part of the 'vproc' structure holds a pointer to a real 'proc' structure and a handle to a remote entity (PMM) for the 'pvproc' structure.

- For a remote process, the 'vproc' is allocated as the result of the initial call to VPROC_PID_-LOCATE for that processes pid. It is then linked in a hash queue for remote vprocs and can be found by looking through the appropriate hash queue. In this case the 'vproc' operations vectors are set to remote operations (RPC). The 'vdata' section holds a handle to a remote entity (a remote PMA) for the 'proc' structure and a handle to a remote entity (PMM) for the 'pvproc' structure.

5.1.3 The 'remote handles'

The handles for remote structures 'proc' and 'pvproc' are both represented in the same way. They consist of a target port to which the RPC will be directed and some debugging information including the node number of the target and a flag indicating whether the target is local or remote. In practice the type of operation vector will implicitly indicate the local or remote nature of the operation making all information but the target port optional used for debugging purposes only.

5.1.4 The 'proc' Structure

This structure pointed to by the 'vdata' part of the 'vproc' holds all information the non distributed 'proc' structure does except that of the process relations. Certain information in the 'proc' structure specifically those in the 'p_flag' and 'p_stat' fields will be kept in both the 'proc' (PMA) and the 'pvproc' (PMM) structures.

5.2 Entry Points to the PMA

The PMA acts as a server allowing entry via a server loop through two channels:

- System calls. When a system call arrives the call is examined and if classified as a PMA call, a PMA entry routine such as 'fork' is invoked. The PMA entry routine will then act in one of two ways. If the call can be fully handled by the local PMA a set of routines much like the non distributed server are invoked (This code is non vendor specific). If the call requires potential remote interaction, then entry routine will look up the appropriate vproc(s) and make a VPOP_XXX macro call. It is these macros which enable third party vendors to insert their own process management vectors thus replacing the OSF provided process management. In

OSF1/AD Version 2 (OSF code) these macros will in turn invoke one of two operations vectors. If the target node is local, a routine of type pma_xxx; otherwise a routine of type remote_pma_xxx is invoked. Details of both these routine types are given in later sections.

- Remote calls. The PMA can be invoked as a result of RPC(s) from remote entities including PML(s), remote PMA(s), and the PMM. In all such cases the calling entity invokes a MIG generated routine of the type c_pma_xxx which is received by the target PMA and passed to a routine of type s_pma_xxx. The routine s_pma_xxx sets up the appropriate environment and invokes a routine of type pma_xxx to do the real work. Details of all above routine types are given in sections below.

5.3 Major Types of Routines in the PMA

The following describes major types of routines in the PMA involved in distribution of process management functionality and their respective functions.

- Type pma_xxx. This type of routines implement the body of work for each system call or pseudo system call, where pseudo system calls are functions invoked on behalf of remote entities in the same manner as system calls; these functions however are not exported to user processes via the system call interface. These routines expect an environment as described in section 2 including a ‘vproc’ for the current process with the ‘vdata’ section holding a pointer to a valid ‘proc’ structure. Their general algorithm are as follows:

```
pma_xxx {
      Lookup of target vproc if needed
      VPOP_XXX   /* Call to nth (n = 1) potentially
                  * remote pma
      Intermediate local process handling
      Lookup of target vproc if needed
      VPOP_YYY   /* Call to nth (n = 2) potentially
                  * remote pma
      Intermediate local process handling
      .
      Lookup of target vproc if needed
      VPOP_ZZZ   /* Call to pmm
      Intermediate local process handling
      .
      Lookup of target vproc if needed
      VPOP_VVV   /* Call to nth (n = 3) potentially
                  * remote pma
      .
      Final local process handling and vproc release
}
```

- Type “system call entry”. These are system call entry points such as ‘fork’ and are invoked through the system call switch. They do some initial checks and copy in the arguments; invoke a routine of type pma_xxx and finally copy out the arguments and set return and error codes. Their general algorithm are as follows:

```
system_call_entry {
      Initial permissions check and copy-in of arguments
```

```
        Lookup of current vproc
        pma_xxx
        Release of current vproc
        Final cleanup and copy-out of arguments
        setting error and return codes
}
```

- Type VPOP_XXX. These macros on the PMA expand to pma_xxx for local process and to either remote_pma_xxx or remote_pmm_xxx for remote ones as directed by the vproc operations vectors.

- Type remote_pma_xxx. This types of routines are what VPROC_XXX for a process with a remote PMA will expand to. They expect a 'vproc' with the 'vdata' part holding a handle to a valid remote PMA. Their general algorithms are as follows:

```
remote_pma_xxx {
        Register for ISC
        c_pma_xxx
        Deregister for ISC
}
```

- Type c_pma_xxx. This type of routines are MIG generated client side of RPC(s) to a remote PMA.

- Type s_pma_xxx. This type of routines are hand written server side of RPC(s) from a remote client to a target PMA. Their general algorithms are as follows:

```
s_pma_xxx {
        start_pma_server_op         /*
                                    Lookup the target vproc
                                    Setup the environment
                                    Generate debugging info
                                    Start stat accumulation
                                    */
        pma_xxx
        end_pma_server_op           /*
                                    Generate debugging info
                                    End stat accumulation
                                    Tear down environment
                                    */

}
```

5.4 Performance Considerations

It is of paramount importance that the process management have acceptable performance characteristics. This includes the requirement that the number of RPC(s) in each invocation of a pma_xxx routine be minimized to one per remote entity. Performance is achieved through the above requirement and a number of other methods described below:

- Code restructuring. In some cases code accessing inter-process relations is sprinkled in between code manipulating single process information. In such cases code re-structuring may be required to bundle all access to inter-process relations in one call to the PMM. Similar restructuring may be required in access to remote PMA(s).

- Information duplication. Certain information such as those held in 'p_flag', and 'p_stat' are required on both PMA and PMM. Since these fields are always set on the PMA immediately before or after a call to the PMM, it is possible to keep them coherently on both entities thus avoiding extra communications.

- Passing of extra information. In certain calls such as those where a process on one PMA manipulates a process on another (remote) PMA, information required for permissions checks may be passed in the RPC to the target PMA thus eliminating the need for the target PMA to call the originating PMA for permissions checks.

- Holding of locks. It is important that global locks not be held across RPC(s), this may require code restructuring or introduction of new locks.

- Init. Having an 'init' process inheriting all orphans and waiting for them causes a bottleneck and impedes scalability well as fault tolerance. Since in the current design the PMM already is a bottleneck and all waits have to go through it anyway, 'init' waiting for all processes is not expected to cause serious additional problems. However in any design distributing the PMM it may be desirable to change the inheritance so that each PMAs orphaned processes can be handled locally.

5.5 Miscellaneous Issues

- Uses of vprocs. VPROC macros are used whenever the target of a process management request may be remote. They allow third party vendors to insert their own code for distribution process management as well as creating a uniform programming interface for all components of process management.

- Vendor independent code. In OSF1/AD Version 2, all of PMA code is vendor independent except for VPROC macros with PMM as the target. Therefore the 'proc' structure and all inter-PMA communications are considered common code. It is however possible for a third party to totally replace the OSF1/AD PMA by replacing relevant system call entries in the system call switch with their own as long as appropriate external interfaces are exported.

6 The Process Management Master

The Process Management Master known as the PMM is a single entity responsible for the handling of all inter-process relationships as well as the controlling tty issues in OSF1/AD Version 2. The architecture of this entity is describe in the Process Management Architecture document already [1]. Here we discuss details of this entity.

6.1 Data Structures

The PMM contains data structures responsible for inter-process relationships as well as the group and session information. Data structures responsible for inter-process relationships are different from those in the non distributed server and discussed below, while the group and session structures are identical to those in the non distributed server and are not discussed.

6.1.1 The pid_entry Table.

The PMM keeps track of all processes via a process table. This table is similar to that on the PMA(s) in the sense that it's entries contain pointers to the 'vproc' structures of the processes. Since the PMM knows and is involved in the creation of all processes its process table is a super-set of the process table on all PMA(s) as describe in section 6.1

6.1.2 The 'vproc' Structure

The PMM contains a 'vproc' structure for each process existing in the whole system. All such structures have the same nature and are created during 'fork' as a side effect of the PMA contacting the PMM. The structures are pointed to by the pid_entry table and have 'vdata' sections holding handles to the PMA responsible for the 'proc' structure and pointers to the real 'pvproc' structures which are local to the PMM.

6.1.3 The 'remote handles'

These structures are identical in nature to those residing in the PMA.

6.1.4 The 'pvproc' Structure

The 'pvproc' structure is the main data repository for processes on the PMM. It contains the interprocess relationships as well as pointers to the group and session structures. It is this structure that is used most often by PMM routines in a similar way to the 'proc' structure in the non distributed server.

6.2 Entry to the PMM

The PMM acts as a server for RPC(s) from PMA(s) and PML(s) and can only be reached via such calls. When the PMM is entered a hand written routine s_pmm_xxx will accept the call. It will then set up the environment and call pmm_xxx which does the actual work.

6.3 Major Types of Routines in the PMM

The following describes major types of routines in the PMM.

- Type pmm_xxx. This type of routines implement the body of work done in the PMM. They expect an environment as described in section 2 containing a 'vproc' with the 'vdata' part holding a pointer to a valid 'pvproc' structure. The routines will in general act as follows:

```
pmm_xxx {
      Initial local PMM handling
      Lookup of target vproc if needed
      VPOP_XXX   /* Call to nth (n = 1) potentially
                  * remote pma
      Intermediate pmm handling
      Lookup of target vproc if needed
      VPOP_YYY   /* Call to nth (n = 2) potentially
                  * remote pma
      Intermediate PMM handling
      .
      Final local PMM handling and vproc release
}
```

- Type VPOP_XXX. These macros on the PMM always expand to a call to remote_pma_xxx.

- Type remote_pma_xxx. This is exactly as in the case of PMA (see section 4.3)

- Type s_remote_pmm. This type of routines implements to server end of RPC(s) made to the PMM. They set up the environment and then call into the pmm_xxx routines. Their general algorithms act as follows:

```
s_remote_pmm {
      start_pmm_server_op         /*
                                  Lookup the target vproc
                                  Setup the environment
                                  Generate debugging info
                                  Start stat accumulation
                                  */
      pmm_xxx
      end_pmm_server_op           /*
                                  Generate debugging info
                                  End stat accumulation
                                  Tear down environment
                                  Release target vproc
                                  */
}
```

6.4 Performance Considerations

Performance issues for the PMM are similar to those of PMA. Here we will mention the major issues expanding only one the differences between those of PMA and PMM.

- Code restructuring. As in the PMA multiple RPC(s) to the same entity are to be avoided in a single execution of a pmm_xxx type routines through code restructuring.

- Information duplication. As in the PMA.

- Passing of extra information. As in the PMA.

- Holding of locks. The problem here is similar to that of PMA(s), however since there is only one instance of the PMM the problem is much more aggravated. Global locks such as the process relation lock currently held across loops in ‘wait’ must not be held across RPC(s) since they will present particularly severe bottlenecks.

In general the single nature of the PMM makes performance considerations particularly important, and in any event the true solution to acceptable performance by the PMM will lie in its distribution.

6.5 Miscellaneous Issues

- Use of VPROC(s). Vproc macros are used by the PMM when the target of a call can be a remote part of process management as described in section 4.

- Vendor independent code. There is no vendor independent code in the PMM.

7 PID Broker

The pid broker is a service internal to the process management code. It provides a lookup service returning a port to the PMA managing the process of a given pid. The pid broker is a centralized service with distributed caching (i.e. it is a master/slave design). The pid broker also manages pid allocation to PMAs. The pid broker is present in every server either using or providing process management services, including the PMA and the PMM.

7.1 The pid_entry table

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 logically divided up into chunks the size of the pid_entry table. This implicitly allocates a number of pids to each pid_entry slot which are cycled through when a new process is created using that slot. The set of pids allocated to a slot, whose index into the pid_entry table is *entry_index*, can be described by the following equation and by figure 2;

```
pid = N * sizeof(pid_entry table) + entry_index
for all N from 0 to (PID_MAX/sizeof(pid_entry table))
```

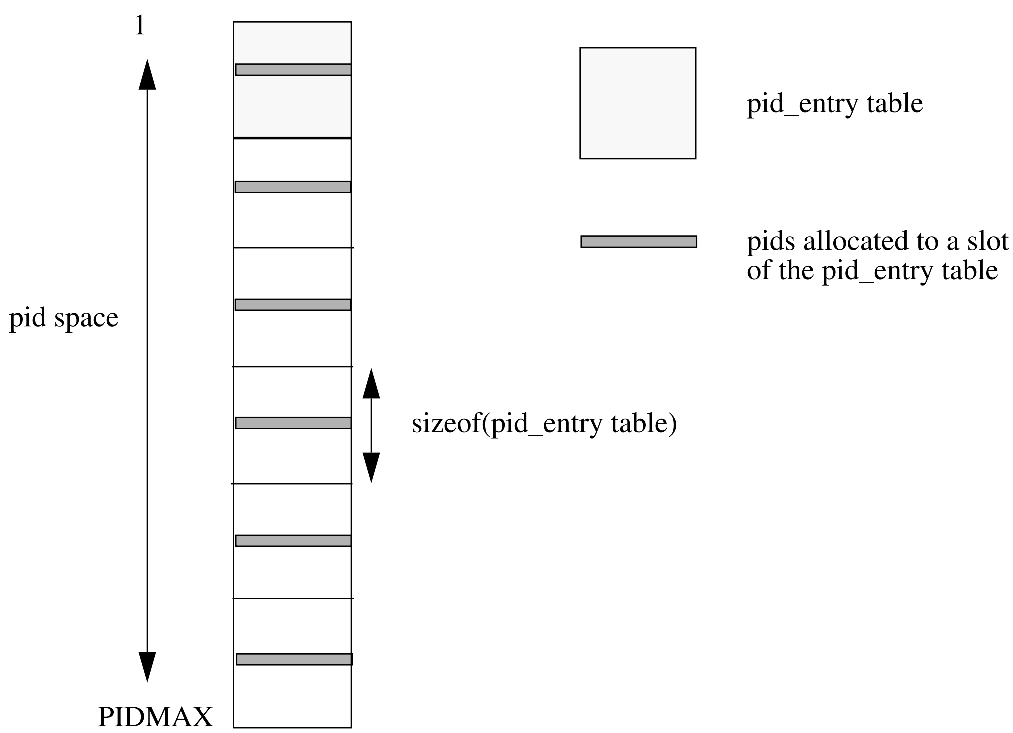


Figure 2. Allocation of pids to an entry in the pid_entry table.

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 %sizeof(pid_entry table) 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 shaded pid_entry table in figure 2 would now represent this logical pid_entry table). 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 sets of pids that this agent may use for processes it manages in much the same way as the single server code.

7.2 PID Allocation

Before a PMA can create processes, it must request an allocation of the global pid_entry table from the pid broker. This implicitly allocates a number of pid ranges from the global pid space to the requesting PMA. The pid broker registers each of these ranges, along with a port to the PMA, separately into its database for future lookups. This allocation is also noted in the PMA side of the broker so that it may lookup local pids and find that it manages them itself without having to communicate with the central pid broker. The size of a pid_entry table, which is in fact the maximum number of concurrently active process, on a given node is specified by a boot time configuration variable of the form;

```
DEFAULT_MAXPROCS=1000
MAXPROCS=<0..10>10,<21..1000>5
```

which would place a limit of 10 processes on node numbers 0 to 10 inclusive and 5 on node numbers 21 to 1000 inclusive. All other nodes would have the default limit of 1000 processes per node.

Agents do not allocate more pid_entry slots beyond the initial allocation at boot time but the design of the pid broker does not preclude this. In order to support a large number of nodes, the pid space is increased from a short to a long, forcing the few places which 'know' it is a short (mainly System V IPC) to be fixed.

The interface for pid allocation is as follows;

```
kern_return_t
pid_allocate_request(
      unsigned   request_size,              /* IN */
      mach_port_tpma_port,                  /* IN */
      unsigned   *global_pid_entry_size,    /* OUT */
      unsigned   *global_pid_entry_offset   /* OUT */
      );
```

The code to allocate the next pid for a new process is unchanged with respect to the single server. However the values of PID_MAX and npid are different in the following ways;

```
#define PID_MAX LONG_MAX
npid = global_pid_entry_size;
```

Thus the code to calculate the next pid to use for the pid_entry slot, unchanged from the single server, would be;

```
pid_wrap = ((PID_MAX / npid) * npid);
pe->pe_pid = (pe->pe_pid + npid) % pid_wrap;
```

The calculation to locate the index of a process in the pid_entry table is modified slightly by subtracting the logical offset of the pid_entry table for this PMA into the global pid_entry table.

```
entry_index = ((pid) % npid) - global_pid_entry_offset;
```

7.3 PID Lookup

The pid lookup service is distributed in a master/slave organization. There is a master database in the PMM but this location is required by the design. The mapping of every usable pid and the PMA handling that pid is stored in this database. In addition, every server that uses the pid broker will have a subset of this database cached. As pids are allocated in ranges, the database contains pid range to PMA port mappings. This makes the local caching more effective as a single lookup involving the master results in a cache entry which will satisfy many pids, typically about 3K for a service node.

The intention is to use the same code for both the master pid broker, where the database will be large and the entries are entered in ordered groups, and for the slave brokers, where the database is comparatively small and the entries are entered randomly. Consequently the data structure to be used is a splay tree[2] as it has the advantages of an efficient lookup operation for both small and large amounts of data and the structure itself is small, needing only two pointers per element (i.e. pid range/PMA port). If the pid is not contained in any range in the database, a message is sent to the master pid broker to search the master database, if this not already the master broker.

There is a single interface to the lookup part of the pid broker;

```
mach_port_t
pma_locate_pid(pid_t pid);
```

There is however an addition interface used between the master and slave parts of the pid broker;

```
kern_return_t
pma_master_locate_pid(
      pid_t       pid;            /* IN */
      pid_t       *base_pid;      /* OUT */
      unsigned    *range;         /* OUT */
      mach_port_t*pma_port        /* OUT */
      );
```

This call returns the range of pids spanning the given pid and a send right to the port of the PMA managing that part of the pid space. A successful lookup in the pid broker does not mean that the process actually exists as the pid broker only provides a location service; the returned PMA must be contacted to determine whether the process is alive or not.

8 The Process Placement Policy Module

The process placement policy module is part of the process management master task. It is called by fork() when the process does not have the FORK_LOCAL attribute in order to determine where the child process should be created. This module's policy will initially be a round robin allocation among nodes specified at boot time configuration as timeshare nodes. The boot configuration variable takes the form;

```
TIMESHARE_NODES=1..5,10..30
```

which will cause the process placement policy to round robin allocate nodes 1 to 5 inclusive and 10 to 30 inclusive.

In line with the much of the process management sub-system, the call to the process placement policy module returns send-rights of ports to process management agents rather than node numbers.

In future releases of OSF/1 AD, it is expected that a more informed approach to process placement will be implemented.

9 System Wide operations

There are a several system calls which have system-wide effect. These are implemented using the VPSOP macros. The target for these calls in OSF/1 AD Version 2 will always be the PMM (acting a handy central repository of information). There is nothing in the interface that forces the PMM to be the target or even that the target is the same on every node. The following table lists the supported operations;

Table 1: VPSOP operations

Operation	Function
VPSOP_HOSTID	Get or set the host id
VPSOP_HOSTNAME	Get or set the host name
VPSOP_DOMAINNAME	Get or set the domain name
VPSOP_REBOOT	Halt and reboot the entire machine

In addition to these commands, a number of other operations are supported to implement the table system call. As table is not really a single system call, the call is in a separate section where each command is considered separately.

10 Table

The following table lists the table commands and indicates how they are implemented;

Table 2: Table commands

TBL command	Where/how implemented
TBL_U_TTYD	PMM using VPOP
TBL_UAREA	PMA of target process using VPOP
TBL_LOADAVG	PMA of calling process
TBL_INCLUDE_VERSION	PMA of calling process
TBL_ARGUMENTS	PMA of calling process
TBL_MAXUPRC	PMA of calling process
TBL_PROCINFO	PMM using VPSOP and PMA using VPOP
TBL_ENVIRONMENT	PMA of target process using VPOP
TBL_SYSINFO	PMA of calling process
TBL_DKINFO	????
TBL_TTYINFO	????
TBL_MSGDS	PMM using VPSOP
TBL_SEMDS	PMM using VPSOP
TBL_SHMDS	PMM using VPSOP
TBL_MSGINFO	PMM using VPSOP
TBL_SEMINFO	PMM using VPSOP
TBL_SHMINFO	PMM using VPSOP

Table 2: Table commands

TBL command	Where/how implemented
TBL_INTR	????
TBL_PGINFO	VNODE pager
TBL_VERSION	PMA of calling process
TBL_MALLOCS	PMA of calling process
TBL_MALLOCT	PMA of calling process
TBL_MAXUTHREADS	PMA of calling process
TBL_THREAD_MAX	PMA of calling process
TBL_TASK_MAX	PMA of calling process
TBL_NGROUPS_MAX	PMM using VPSOP
TBL_KERNEL_ARG_MAX	PMA of calling process
TBL_CLK_TCK	PMA of calling process
TBL_DEFAULT_STACK_SIZE	PMA of calling process
TBL_DEFAULT_DATA_SIZE	PMA of calling process
TBL_DEFAULT_ADDR_SPACE	PMA of calling process
TBL_PAGEDAEMONPARAM	PMA of calling process
TBL_STR_PARMS	????
TBL_MAJNUMSHIFT	????
TBL_GETMODEL	PMA of the calling process
TBL_KERNEL_VERSION	PMA of the calling process
TBL_VMSTATS	PMA of the calling process

11 Node Failure

Node failure can be tolerated with certain restrictions which are; All the processes on the failed node (or more precisely, handled by the PMA on the failed node) will be terminated from the perspective of all other processes in the system. If the failed node has either the PMM or init running on it, the whole system fails.

Node failure is handled in different ways by the PMM and other servers. The PMM uses dead-name notification for agents that have contacted it. When this notification arrives, the PMM searches its process tree to find processes which have that PMA as their agent and treating each one as if it had called exit().

The PMA also holds ports to other PMAs but does not use dead-name notification. Instead, operations continue as normal until the sending of a message to another PMA returns with MACH_-SEND_INVALID_DEST. When this happens, the pid broker is purged of any cache entries for that PMA and the original request fails with ESRCH as if the target process had already terminated.

12 Configuration

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 two strings are;

```
PMM_NODE=#
PMA_INIT_NODE=#
```

where # is the node number where they should be executed.

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_get_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.

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.

13 References

[1] Rabii, F. Patience, S. The Process Management Architecture of OSF/1 AD Version 2. OSF Research Institute internal document. September 1993.

[2] Sleator, D. Tarjan, R. Self-Adjusting Binary Search Trees. Journal of the Association for Computing Machinery, Volume 32, No. 3, July 1985.

[3] Locus Computing Corp. Vproc Interface Specification. Locus internal document number LA-0073-93, July 15, 1993.

Appendix A

List of VPOP Operations

Table 1:

Operation	Description
VPOP_FORK	fork a process
VPOP_EXIT	terminate a process
VPOP_WAIT	wait for a child process to terminate
VPOP_PROC_NICE	get or set process scheduling priority
VPOP_PGRP_NICE	get or set process group scheduling priority
VPOP_SIGPROC	send a signal to process
VPOP_SIGPGRP	send a signal a process group
VPOP_TERMINAL_SIGPGRP	send a job control signal to a process group
VPOP_REPORT_STATE	report a change in run state of a process
VPOP_SETSID	create a new session
VPOP_SETPGID	set process group id
VPOP_GETATTR	get process attributes
VPOP_SETPINIT	kernelize a process
VPOP_PTRACE	trace the execution of a child process
VPOP_SETCTTY	set controlling terminal for a process

Appendix B

List of pproc operations

Table 1:

Operation	Description
pproc_exit	terminate a physical process
pproc_reap	get status information for a physical process
pproc_wakeup	wakeup a physical process
pproc_assert_sleep_wait	declares intention to sleep on a physical process structure awaiting an event
pproc_sleep_waiting	sleep on a physical process structure awaiting an event
pproc_add_rusage	accumulate resource usage statistics for a physical process
pproc_fork	fork a physical process
pproc_get_nice	get scheduler priority information for a physical process
pproc_set_nice	set scheduler priority information for a physical process
pproc_chksigmask	check is a signal is masked or ignored by a physical process
pproc_psignal	send a signal to a physical process
pproc_getattr	get attributes for a physical process
pproc_setattr	set attributes for a physical process
pproc_ptrace	perform a trace request on a physical process
pproc_setctty	set a processes controlling tty information
pproc_getctty	get a processes controlling tty information
pproc_pinfo	get information about a process
pproc_setsigact	set sigaction for a physical process
pproc_sigaddset	add a signal to the processes signal set
pproc_pidgen	allocate a process id