Mach Resource Control in OSF/1

David W. Mitchell
October, 1991

Abstract

All systems have inherent restrictions imposed by their hardware architecture and configuration; to ensure reasonable operation, these limitations must not be exceeded. Timesharing systems may also need to impose constraints on the consumption of an individual user to ensure that all users can get adequate access to system services. Systems layered upon Mach require a considerable amount of extra code to impose consistent and secure limits if Mach system services are also exported to users.

This paper describes the design and implementation of a general framework for controlling Mach resources which permits servers or other layers of kernel code to account for and control resource consumption in the Mach layer.

1. Introduction

All systems have inherent restrictions imposed by their hardware architecture and configuration. These include such things as support for only a finite amount of address space or physical memory, the ability to run a fixed number of processes, or to handle only a certain number of pending I/O requests. To ensure reasonable operation, these limitations must not be exceeded. At a minimum, the constraints imposed by hardware must be enforced on processes running on the system [1]. It is often desirable to further confine the consumption of any individual process in order to prevent one process from interfering with another. Commercial timesharing systems frequently provide mechanism for controlling the resources used by any single user to ensure that each user of the system can obtain access to the resources needed to make progress. Though workstation operating systems typically treat all users equally, in the commercial world users with more money get more resources. Some mechanism for controlling resource usage is necessary to make this feasible.

2. The Structure of OSF/1

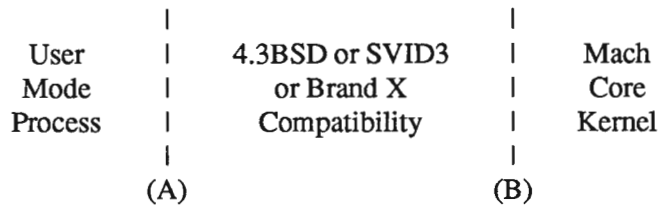
The OSF/1 operating system is an example of a layered operating system, consisting of two basic layers of kernel code. The core is derived from the Mach (2.5) operating system, which supports a small number of fundamental abstractions implemented by a communication oriented kernel which supports multiprocessing [2]. The basic abstractions exported are the task, thread, port, message, and VM object.

- A **task** comprises the execution environment in which threads may run, and is the basic unit of resource allocation, consisting of a virtual address space and access to various resources via ports.
- A **thread** encapsulates the state of a single run-time execution path within the task.

- A **message** is a typed collection of data objects, used for communication between threads. A message may be of any size, and is sent over a port.
- A **port** is a communication channel which is logically a queue of messages protected by the kernel.
- A **memory object** is a collection of data managed by a server which can be mapped into the address space of a task.

The outer layer of OSF/1 provides 4.3BSD and SVID3 functionality and has also been parallelized to enhance multiprocessing support [3].

Currently, these two layers are bound together and both run in supervisor mode on each hardware base. Ongoing research [4] is investigating the efficacy of restructuring these two layers so that only the Mach core ("micro-kernel") requires privileged mode. The 4.3BSD code and other compatibility layers would then run in user mode in one or more server processes.



Today, boundary 'A' is a "real" boundary in the sense that it is enforced by the memory management hardware, and boundary 'B' is one of software convention only. In the future, boundary 'B' may become the one enforced by hardware, with 'A' becoming simply the inter-process boundary between one or more server processes and other user-mode processes.

3. Motivation & Goals

The Mach core of OSF/1.0 has no notion of resource limiting, other than allocations failing due to complete exhaustion, though the Mach Kernel Interface Manual [5] lists resource limits among the capabilities contained within tasks. The compatibility layer partially enforces the BSD notion of limits, allowing a process control over:

- cpu time used
- maximum size of a file that can be created
- maximum size of the data segment (break value)
- maximum size of the stack
- maximum size of a core file
- maximum amount of physical memory used (resident set size)

Unfortunately, the traditional model of text/data/stack does not map well onto the more general virtual memory model of OSF/1, and in any case the data and stack limits are easily circumvented by direct use of the Mach primitives. Enforcement of limits on entities whose underlying implementation consumes system memory, such as tasks, threads, and virtual space is

nonexistent. An errant or malicious process may easily cause the kernel to allocate a large percentage of physical memory for internal use, leaving too little for user processes to run efficiently. There is a clear need for a general framework, usable by all subsystems, which will support accounting of, and limits to, the consumption of arbitrary Mach resources. This framework must also provide other layers of code with the ability to control and monitor these resources. It would also be useful if this framework was structured in such a way that it could be migrated easily into the micro-kernel research effort.

While there is a need to limit such potentially exhaustible system entities as:

- tasks
- virtual space
- threads
- ports, port sets, port names
- processor sets
- address map entries

on a per-user, per-session, or some other basis, it is also clear that this enforcement is most appropriately, efficiently, and securely done by code in the Mach core, though this layer has no knowledge of processes, users, or sessions. Hence, the approach taken here is to construct a general mechanism in the Mach core of OSF/1 which permits secure accounting of resource consumption on the basis of a task or arbitrary group of tasks with sufficient hooks and functionality to allow the outer layers of kernel code to control resource usage among groups of tasks on the basis of whatever abstractions that layer implements. This code was also designed so that it could be easily reusable in pure Mach, distributed environments and the micro-kernel. The interfaces exported from the Mach layer will be exported via MIG [6].

4. Structure & Function

To solve the problem of accounting for, and constraining resource consumption in a task, it seems natural to add an object which will encapsulate the required bookkeeping. In financial accounting, a ledger is a book containing accounts, to which debits (debt items) and credits (asset items) are posted from the original transaction records. Hence the metaphor for this new accounting object is that of a ledger, in which the transactions of the task's threads are posted in the form of debits and credits. For each Mach resource to be accounted for, there is a separate line item containing a balance and maximum. The balance tracks the amount of the resource consumed. There is a function which will debit the ledgers to account for resource consumption, and another to credit the ledgers when the resources are returned to the system pool. Each ledger has a maximum, setting a firm limit past which the ledger may not be debited. Those with backgrounds in accounting, noting the small loss of generality compared to an actual bookkeeper's ledger, may think of this new ledger object as a task's resource account with the system, containing an implicit right hand side of zero.

Similar functionality could have been obtained by having only a balance, which gets initialized to the maximum usage and is adjusted as resources are allocated and deallocated, never being allowed to go below zero. However, this would not permit the maximum to be adjusted once the ledger was in use without requiring implicit knowledge of the ledger's initial state.

A task has two ledgers attached to it: a private ledger which accounts for the resource usage of an individual task, and a shared ledger, which accounts for resources used among a group of tasks. All line items exist in both ledgers, and the routines which debit and credit items always modify both ledgers. Having two ledgers, one used per-task and another allowing a set of tasks to be controlled provides flexibility in implementing useful resource control on top of this facility. The per-task ledger is intended to provide constraints, which may be modified by the task. No privileged capability is required to access the per-task ledger, providing discretionary controls which are also useful in a pure Mach environment, without the intervening compatibility layers. The shared ledger is intended to enforce mandatory constraints set by the system administrator, and so requires that a certain capability (privileged host port) be held in order to modify the maxima.

Having a private per-task ledger also makes it possible to correctly account for a single process's consumption when it is necessary to detach the process from one shared ledger and attach to another. Since the private ledger details the resource usage accountable to each task, the change of shared ledgers is straightforward. Moving the task to a different shared ledger is necessary when the BSD code layer changes the process' real uid, such as during login.

Note that some maxima may be set to "don't care" or "infinite"; for example, limiting virtual space per user is not generally useful, and limiting tasks per task is inane.

This structure provides separation of mechanism and policy. While the Mach layer provides the hooks to account for threads, tasks, virtual space, etc., the BSD layer or server(s) may use the shared ledgers to implement resource limits on a per-user, per-session, or any other useful basis. Also, the per-task ledger may be set differently for each task. Providing for a list of ledgers was considered, but the increased generality was ruled out in favor of efficiency. Multiple shared ledgers gave little added functionality, since the outer layers of code can control the domain over which the sharing takes place and these layers are free to set the shared maxima on any basis desired, such as the lowest or highest usage allowed to any group of which the task is a member.

5. Implementation

A ledger is a simple data structure, containing a reference count, a lock, some number of line items each containing a balance and maximum, and a count of the number of line items.

Task_create() sets up each task with a private ledger of its own, with maxima inherited from its parent's private ledger. A task's shared ledger is inherited from its parent. If there is no parent, new ledgers are used, with maxima set to LEDGER_UNLIMITED.

Subsystems participating in resource control use the debit and credit functions to check whether further allocation is allowed, and to account for resources returned to the system, respectively.

The VM subsystem is one notable exception, in that it does not store the maximum and balance in the ledger structure, but forwards the data into the task's address map, for easy access by the lowest level VM routines. Therefore the VM subsystem does not use the debit and credit functions but maintains the balance as it always has, now validating against the maximum before allowing a change in size. The ledger code references the values in the task's address map when reading or writing the maximum or reading the balance.

The compatibility layer of OSF/1 currently uses the per-task ledger to implement discretionary controls such as the per-process address space limit of SVID3. The shared ledger is used to enforce mandatory limits such as the limit on processes per real uid mandated by POSIX 1003.1. This limit cannot be overridden by the user.

In the BSD layer, the first process has its ledger maxima set to the system defaults. When a new process is started, it inherits its private ledger maxima from its parent and uses the same shared ledger as its parent. Since only descendants of init will have their maxima set, any Mach system tasks will have unlimited access. These system defaults are included among the parameters which can be set by the system administrator at system boot time.

When a process changes its real uid, as happens during setuid(), it is necessary to change the shared ledger attached to the underlying task. The OSF/1 internal set_uids() routine, if it is changing the real uid, sets a flag in the process which is checked when a process tries to exec. During exec, set_per_user_limits() is called to attach the process to the shared ledger of another task belonging to the same user, if such a task exists. If there is none, set_per_user_limits() procures a new shared ledger to be used by all processes of that user. This is done by detaching the process from the default (system-wide, unlimited) shared ledger, attaching to the new one, and setting the maxima appropriately for the process's privilege. Processes belonging to superuser, or in security configurations, processes possessing the SEC_LIMIT privilege, have their maxima left unlimited. If an existing process belonging to that user is found, its shared ledger is attached to the new process provided that no system limit is exceeded, otherwise an error is returned. Calls that set only the effective uid, (e.g.: execve() and exec_load_loader()) do not need to do this, since accounting is attributed to the actual user, as indicated by the real uid.

This baroque structure was chosen to avoid adding a new failure scenario to the setuid(2) call, which would have had the nasty implication that a process might unwittingly be left running with high privilege after an attempt to lower its privilege. In practice, delaying the accounting check until the next exec call after the setuid operation does not allow a loophole, since setuid(2) is typically a one-way door out of privileged mode. For example, login would be unable to exec a shell and would fail. A user who attempts to exceed a limit by running a privileged program finds that another task cannot be started in which to run that program, since the process is accounted against the real uid.

Care was taken to layer the functions needed to implement this new facility, so that they could be moved easily into the micro-kernel environment. The routines which manipulate ledgers can be logically separated into three groups:

- internal to the Mach layer

ledger_create()	used by task_create
ledger_reference()	used by task_create
ledger_deallocate()	used by task_deallocate

- used by Mach subsystems participating in resource control

task_ledger_debit()	used by vm_allocate, thread_create, task_create, ...
task_ledger_credit()	used by vm_deallocate, thread_deallocate, task_deallocate, ...

- exported to BSD layer, server or other user code

task_private_ledger_read() task_private_ledger_write()	used by initialization code, setrlimit & others
task_shared_ledger_read() task_shared_ledger_write()	
task_shared_ledger_replace() task_shared_ledger_attach()	used by BSD layer to implement per-user limits

Of the routines exported out of the Mach layer, the latter four routines (which modify the shared ledger) all require the caller to supply a privileged port. Imposing this requirement prevents the process from overriding limits set up by the system administrator. The task's private limits may be changed at any time.

6. Unsolved Issues

This work provides the hooks necessary to account for, and control, the use of Mach kernel resources. It prevents an errant or malicious user from draining some key system resources and eliminates a source of covert channels. This work avoids the larger issues of how these limits can be set automatically in response to changes in configuration and load, and makes no attempt to solve the problem of providing each process or user with their fair share of system throughput under varying conditions [7].

7. Current Status

This work is part of a larger effort to commercialize OSF/1 by implementing controls on resource consumption, making fundamental system parameters modifiable either dynamically or at boot time, and eliminating system crashes due to resource allocation panics. The task_ledger facility has been used to implement the SVID3 per-task address space limit and per-user limits on tasks and threads with defaults which can be set by the system administrator at system bootstrap time. There has been no measurable performance impact. Other potentially exhaustible Mach kernel resources will soon be added into the framework.

References

1. S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, "The Design and Implementation of the 4.3BSD Unix Operating System", Addison-Wesley, Reading, MA (1989).
2. M. J. Accetta, et al., "Mach: A New Kernel Foundation for Unix Development", Proceedings of Summer Usenix, July 1986.
3. J. Boykin and A. Langerman, "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis", Workshop Proceedings, Usenix Workshop on Experiences with Distributed and Multiprocessor Systems, 1989.
4. D. Golub, R. Dean, A. Forin, R. Rashid, "Unix as an Application Program", Proceedings of the Summer 1990 USENIX Conference, pp. 87-96.
5. R. V. Baron, D. L. Black, et al., "Mach Kernel Interface Manual", April 1990, CMU.
6. R. P. Draves, M. B. Jones, M. R. Thompson, "MIG - the Mach Interface Generator", August 1989, CMU.
7. J. Kay, P. Lauder, "A Fair Share Scheduler", Communications of the ACM, January 1988, V.31 No.1, pp. 49-55.