Combining Real-Time and Time-Sharing Services on a Multiprocessor

Ziya Aral Ilya Gertner Dave Mitchell

Technology and Architecture Group
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, Ma. 01752-3004 *

Abstract

Multiprocessor systems offer a unique opportunity to provide general-purpose time-sharing services without sacrificing the deterministic behavior and minimal latencies required for real-time applications. It is possible to achieve this by partitioning the set of processors into two parts: (1) part is dedicated to time-sharing; (2) part is dedicated to real-time computations and control. Many existing approaches to real-time operating systems are based on modifying the base operating systems in order to meet the real-time constraints. The results is an environment that is both very costly to develop and maintain. Our approach combines the time-sharing and real-time services in a unique way; traditional time-sharing services continue to run as part of the operating system; while real-time services are implemented at a user-level that run on top of the dedicated set of processes called gangs. The result is a system that provides all traditional operating services (on System V)and still provides real-time services (for flight simulators).

Introduction

Overview

The growing size and complexity of real-time applications and advances in computer hardware and software have emphasized an ever increasing grey area between "real-time" and "general purpose" computing. Even as many commercial and technical applications have begun to stress real-time components, many real-time applications have spilled into the general purpose domain. The increasing complexity of real time tasks has increased the importance of advanced software development environments and sophisticated text and file management utilities. The concurrently expanding requirements for graphical display, program visualization, program monitoring, communications, and post-processing of data have added significant general purpose components to many real-time applications.

The standardization of many such services and applications in environments such as the UNIX operating system has created a gap between the current sophistication of mainstream scientific and technical computing based on standards and the more austere and typically proprietary environments once common to real-time computation.

Distributed architectures which combine real-time and general purpose components have provided only a partial solution via very loose interaction (typically NFS and TCP/IP) [17]. Unfortunately, the response time is such systems is inadequate for many applications that require a combination of real-time and general purpose services in a single, tightly coupled system (typically communicating via shared-memory).

Combining real-time and general purpose, standards based, time-sharing services on a single hardware platform remains an important problem. Attempts to extend a real-time environment to provide time-sharing services or modify time-sharing to provide real-time response results either in systems with inadequate real-time performance or in prohibitively complex and expensive systems which deviate significantly from the standards they were intended to implement [20, 13, 12].

With the emergence of the UNIX operating system and the availability of several "Real-Time UNIX"'s, standards efforts like POSIX P1003.4 appear to be promising at the interface definition level. Unfortunately, the most difficult part of the implementation has remained unspecified and the the fundamental problems has remain unsolved [10].

*This research was supported in part by the Defense Advanced Research Projects Agency (DoD) ARPA Order No. 5875. Monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

The difficulty lies in the inherently contradictory requirements for the real-time and time-sharing systems: real-time systems need exclusive access to all resources, while time-sharing systems need shared access to expensive resources (in order to reduce costs); performance is of essence to real-time systems, while protection and fair scheduling is important to time-sharing, etc. The need to mediate between these conflicting demands often results in systems that are either complex, expensive, and deviate significantly from the standards, or in systems that do not meet the real-time constraints.

The approach described in this paper is based on a very simple idea: instead of trying to fit real-time into the time-sharing framework we provide an environment where both systems can coexist. This is naturally accomplished on a multiprocessor where one set of processors is allocated for time-sharing and another one for real-time. The time-sharing system remains essentially unchanged. The real-time system requires some changes to the underlying kernel. The essence of those changes is to remove all intermediate software layers and provide direct hardware access to the real-time system that must control devices, memory, CPUs and own a scheduler. Communication between the real-time and time-sharing is easily supported with the client/server model: a client on behalf of the real-time issues a request to the time-sharing server which yields the request and returns results to the client.

The result is a hybrid system supporting a "guest operating system" running on a portion of the bare multiprocessor hardware, but also, because that guest retains a UNIX identity, a logical extension to UNIX, similar to a language runtime or a secondary scheduler.

The above has been implemented on nUNIX, an Encore enhancement of the UNIX System V operating system [6]. The Appendix contains the UNIX man pages for a few selected system calls. The communication between the real-time and time-sharing is supported via shared-memory, a very efficient method of communication. This paper describes the implementation of the system and discusses performance results and early experience.

Background

In this paper we describe work based on a unique combination of experiences: flight-simulator trainers and general-purpose parallel processing- the result of the recent merger of the Encore Computer Corporation and Gould Computer Systems.

Prior to the merger Encore has been engaged in DARPA-funded parallel processing research in developing a Gigamax, a 1000 MIPS multiprocessor [14]. In software, the researched focused on MACH, a parallel operating system [16, 11] and Parasight, a parallel programming environment [1, 4, 2, 5, 3].

Parasight is a unique performance monitor and debugger that utilizes the potential provided by the shared-memory multiprocessors: it uses shared memory to read the program status of the application; it uses multiple processors to run sophisticated monitors that non-intrusively monitor the behavior of the applications. It provides a natural fit for real-time where all variables and other statistics can be flexibly (with dynamic interaction with the user) and non-intrusively (from another processor) monitor the application while it is running. This is a unique capability, available only on shared-memory multiprocessors.

Another result of our operating system research was the development of nUNIX, which is an enhancement of the standard System V UNIX kernel [6]. nUNIX provides a extensible mechanism to support variable weight processes. This is achieved by redefining the process control block to be a structure of pointers to the resources. This capability turns out to be the crucial mechanism for implementing the environment to support both real-time and time-sharing systems on a single hardware platform. The following section describes those extensions in detail.

nUNIX: a Multiprocessor UNIX

The inadequacies of conventional UNIX kernels for multiprocessors are well known [7]. Attempts to solve those inadequacies have been centered around defining the notion of a light-weight thread that is a more efficient unit of computation than a UNIX process. There are basically two approaches for implementing light-weight threads: (1) kernel threads and (2) user-level threads [19].

Earlier attempts at implementing the light weight threads have been based on the user-level packages [9]. These packages typically support some number of "threads of control" mapped onto one or more UNIX processes. While quicker context switching along with an arbitrary number of execution units (threads in this case) can be obtained this way, there are some disadvantages. User-level synchronization can become inefficient when the OS swaps out one of the cooperating processes. It is quite difficult to have I/O operations such as open(2), close(2) and dup(2) made visible to all the participating processes; nor-

mally I/O requests are queued to a single process to avoid these complications, although other solutions are possible.

Additional problems with the user-level threads point in the direction of resource sharing. The problem is in precisely defining the semantics of the thread creation. In particular, how are resources distributed in a parallel program among threads? What happens if a thread opens a file? Is it open to all threads? If so, how does a thread open a private file? Resources sharing has become the major hurdle in the user-level implementations.

We have addressed the resource problem by making it the central issue in defining a thread. In fact, a thread is defined as a collection of resources that can be shared or duplicated among other threads. This definition is very general and already has been used to support either conventional UNIX process (which duplicates all resources upon the call to fork()) or MACH light-weight threads (which share ALL resources after the call to thread-fork()). In addition, the same mechanism can be used to implement a variety of flexible weight threads that meet the needs of the given application.

We have found this approach to be most suitable in our environment and implemented it at a lower-level in the UNIX kernel [6]. Here, creation of a UNIX process is done with a finer control of resources by explicitly specifying what resources are shared and what resources are duplicated.

The kernel has less work to do when creating a new process with shared resources, since this usually involves only incrementing a reference count rather than allocating and initializing new memory. With these resources orthogonal to the execution context, UNIX then provides support for a variable-weight process which spans the range from the traditional process with all resources private, to a light-weight process with an entirely shared address space resembling a kernel-supported thread; but without the problems typical of the user-level threads packages [10]:

- lack of external thread visibility,
- changed semantics of thread "exit", "fork",
- changed semantics of many system calls dealing with I/O,signals,etc.

The sharing of these resources also provides substantial power. Sharing portions or all of the address space results in inexpensive shared memory, free of artificial limitations, free of the overhead normally associated with creating and attaching shared regions, and useful without requiring complicated machinations in the source code.

Sharing the I/O resources allows several processes to use the I/O connections maintained by a single process, avoiding the overhead of initializing separately for each, and making asynchronous read-ahead and write-behind possible using an extremely light-weight process sharing the entire address space. The sharing of signal handlers, usage limits, user IDs, file descriptors, and statistics info, all make it possible for one process to manage any of those resources on behalf of all of its cooperating brethren.

Additional processes sharing a debugger's address space can nonintrusively monitor the activities of the target process; one parallel debugger, Parasight, has already been successfully implemented using this approach.

This solution provides completely uniform and consistent semantics, along with heterogeneous resource sharing, which is impossible with a user-level threads package. Lighter-weight processes allow the OS to optimize many operations for improved efficiency, and to consume less memory, allowing the support of a greatly increased number of processes.

The New Approach

This section begins with the description of the requirements for combining conventional time-sharing operating system (OS) and real-time (RT) services. (In this Section the acronym OS refers to the conventional time-sharing system while RT refers to the real-time application). Then, we proceed with the implementation of the system. Finally, we discuss our experiences in implementing and using the system.

Requirements

In the introduction we talked about the difficulties of combining time-sharing and real-time services which frequently have contradictory requirements. The key for RT is to achieve a complete ownership of devices; the key for time-sharing is to achieve "fair" sharing. This is a contradiction.

```
RT:                              OS:
 sole access to devices           shared access  to d
  (immediate, no latency)                (queued)

 occasional access to
  shared devices (queued)
```

For example, RT applications need very fast low latency access to hardware devices; in contrast time-sharing services attempt to provide a "fair scheduling" of devices at the expense of latency.

The problems are exacerbated even further for a RT application that needs to lower the costs of the device access for data backup. In addition, a complex RT application may need services that are normally offered by the conventional operating systems (like printing a report) which have no stringent timing requirements. It is the infusion of the general-purpose OS services into the RT environment that has greatly contributed to the complexity and costs of today's RT systems. The contradictory requirements for RT and OS at a more detailed level are:

```
RT:                    OS:
   own devices,          shared devices,
   own CPUs,             time-sliced sharing,
   own scheduler,       "fair" scheduler.
 ----------------       ------------------
   open system           closed system
```

RT wants to own devices such as the sensor devices, backup storage and CPUs in order to reduce latency; OS wants to share devices in order to reduce the costs. Likewise, RT wants to dictate its own rules of scheduling computations and resources; while OS wants to be "fair" to all users. In other words, RT wants to be an open system where all the resource scheduling decisions can be made or changed at will; while OS wants to be a closed system that makes most of the decisions at the initialization time and tries to be fair to all users.

Most of the existing RT systems can be classified as open systems where the system services and applications share the same address spaces and the same set of devices; all accesses to memory and devices are immediate via function calls. This is the most efficient for providing system services at the expense of protection: erroneous program may crash the entire RT application. In contrast, UNIX is closed system where all users and the system use separate address spaces and protection domains. Protection domain crossing occurs in thoroughly debugged boundaries with plenty of error checking ensuring that a user program does not crash the OS kernel. Clearly, ensuring protection comes at the expense of performance that is "wasted" (from a RT point of view) on error checking.

Implementation

This section describes the implementation of the main RT mechanisms and communication mechanisms between the RT and OS subsystems. The section is organized in three parts:

```
(1) Gangs, the ability to allocate a set
    of processors for certain tasks.
(2) Ability to communicate with the devices
    directly from the real-time programs
    (i.e. user programs), without any inter
    from any underlying systems software.
(3) Communications between the real-time
    and traditional OS.
```

The main mechanisms to implement the above are:

```
RT:
   mapped-in devices,
   gangs,
   wired VM,
   controlled cache.
```

Ownership of devices is achieved by mapping them into the user address space (just like in any kernel). In UNIX, this is accomplished with the calls "physmap(2)" and "intercept(2)". The actual code is very similar to the conventional kernel code. Following the mapping, the device is controlled by reading/writing to memory with certain conventions.

The interrupt handler had to be changed substantially from the conventional kernel code. In a conventional OS a hardware interrupt invokes the main interrupt handler (with the supervisor bit enabled) which validates the interrupt and passes it along as a user-level interrupt. This is unacceptable for RT applications that must handle interrupts within microseconds of its occurrence. We achieved the required response time by introducing a dual map scheme. The two maps define the same address space running in either the supervisor or user mode. A hardware interrupt invokes the handler with the supervisor bit enabled (a hardware requirement). Since the handler runs in the same address space with the application, the results are made immediately available to an application. In essence, the dual map is very similar to the "ring-crossing" approach that is common in a single-address space operating systems such as MULTICS.

Ownership of the processors is achieved with the new "gang(2)" system call that allocates the specified processor to the given context. The process is permanently assigned to a specific CPU within the GANG; once within the GANG, the value returned from get_cpuid() should remain constant. These processes are not subject to normal time slice interrupts or scheduling algorithms. Similarly, the CPUs do not field ordinary I/O interrupts such as I/O completion, clock, or rescheduling (i.e. time slice) interrupts. User level signals are delivered immediately. Further predictability is achieved by locking the virtual memory and controlling hardware caches.

The RT multi-tasking scheduler is running as a user program (as opposed to other systems where the scheduler must be part of the system kernel). This allows use of several schedulers at the same time on the same hardware platform. (For quite some time we have been running the mix consisting of the standard UNIX scheduler, and Parallel Ada scheduler [15]; the RT priority-based scheduler has been the most recent addition.)

Minimal latency is guaranteed to RT devices which are allocated in an exclusive mode. Other devices which do not require minimal latency are shared by both RT and OS systems. One example is a disk drive that can be configured either as a dedicated RT device or as a shared-device as part of the time-sharing OS services.

For accessing shared-devices, we adopted a client-server model frequently used in distributed systems. The server program runs as a regular UNIX program (OS server) subject to the conventional scheduling and interrupts (though typically run at a high priority for minimal latency); the client program is installed as a library as part of the RT application. A library call for system services is diverted by the client software and queued in the shared buffer that is later accessed by the server. To support efficient communications the OS server shares the entire memory with the RT application. In addition, the OS server runs at a high (UNIX) priority.

This approach significantly simplifies the RT programs which do not deal at all with the conventional OS services. This is accomplished with a simple remote procedure call interface (client library) which marshals requests into command buffers that are passed along to the OS server. In some cases, the overhead for calling UNIX services can be further reduced by temporary lending one of the ganged CPUs to run the OS server. (A similar optimization has already been used successfully to implement Parallel Ada run-time [15]).

Preliminary Experience

In this section we evaluate the preliminary experience in implementing and using this paradigm for combining RT and OS. We evaluate the results in terms of the complexity of the implementation (approximate number of lines changed or added) and in terms of performance.

Gangs were fairly straightforward to implement: the system calls for creating, destroying, adding to, removing from the gang required less than ten pages of C code. The total number of changes in the OS scheduler and interrupt handler required only a few lines. Mapping in devices was also straightforward: UNIX calls physmap(2) and intercept(2) as used in the same way as they are used out of the UNIX kernel. The dual map scheme, while complex in design, was quite straightforward to implement and it did not impact the device specific interrupt handlers.

Wired-VM or VM with locked pages was based on the System V version that already comes with what is called "lazy locking" which simply marks pages locked and returns. A page faults occurs only the first time when that page is accessed; once in memory, the page remains in memory forever (or until it is unlocked). We corrected this behavior by having pages locked and in-memory upon return from the new system call.

RT also controls hardware caches. This is necessary in order to check for the performance boundaries in the application's execution. Although it has been demonstrated that there are cases when disabling a cache may actual speed-up the program's execution [8], in our experience most of the application do run much faster with the cache enabled.

Complexity of the communication between the RT and OS is similar to the complexity of any distributed program running at user-level. The OS server is a user program that reads requests from shared memory and converts them to UNIX system calls. It follows the classic distributed server philosophy by waiting in an infinite loop for work that is queued in shared memory, calling UNIX services to do the work, and returning results. The server is about a ten page C program. Clearly, with more optimization, the size of the program may double to twenty pages but it will remain a small program and we do not expect that it could ever exceed one hundred pages.

The client stub is also a classic distributed client program that simplify marshals function call requests in to a shared memory buffer [18]. The important fact is that normally the client never gives up control and returns immediately to the RT scheduler. The only exception is the optimization when the client "lends"

its processor to the server to perform a system call.

Conclusions

This paper describes a new paradigm for combining the services of real-time and time-sharing systems. Previous approaches which have been based on modifying either the time-sharing system to provide real-time services or modifying real-time system to provide time-sharing services have produced very complex and expensive to maintain systems. In contrast, our approach is based on the coexistence of the two environments on the same hardware platform.

Fine control of resources is the key mechanism for providing real-time services that always need an exclusive, minimal latency access to some devices such as the real-time sensors and occasionally need access to shared devices such as the printer. Exclusive access is provided by allocating devices, memory and CPUs. This is naturally accomplished on a multiprocessor where a certain set of processors are "ganged" together to provide real-time services. Access to shared-devices is provided via a distributed model: a client on behalf of the real-time system which is under the control of the real-time scheduler issues a request to the time-sharing server which is under the control of time-sharing. This is a very simple and very powerful mechanism that provides all UNIX services to an RT application.

Preliminary experience with the dual services model has been encouraging. The UNIX changes have been incorporated into our standard UNIX System V distribution. Real-time system scheduler (running at a user-level) has also been implemented and is the process of being tested by several flight simulators.

Acknowledgments

The authors wish to acknowledge Greg Schaffer, Jeff Russo, and others who have been implementing the real-time system that generated the need for the changes in UNIX described in this paper. Dave Webber contributed to the clarity of the final version of this paper.

References

[1] Z. Aral and I. Gertner. High-level debugging in Parasight. In *ACM Workshop on Parallel and Distributed Debugging*. University of Wisconsin-Madison, May 1988.

[2] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. In *ACM/SIGPLAN PPEALS 1988 — Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, Connecticut, July 1988.

[3] Z. Aral, I. Gertner, J. Grier, and G. Schaffer. Performance monitoring on shared-memory multiprocessors. In *HICSS23, Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, January 1990.

[4] Z. Aral, I. Gertner, and G.Schaffer. Parasight: An architecture for high-level debugging and profiling. In *ACM 1988 International Conference on Supercomputing*, Saint Malo, France, July 1988.

[5] Z. Aral, I. Gertner, and G.Schaffer. Efficient debugging primitives for multiprocessors. In *ACM/SIGPLAN ASPLOS 1989— Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, May 1989.

[6] Z. Aral, I. Gertner, and A. Langerman. Variable weight processes with flexible resources. In *USENIX Conference Proceedings Winter*, June 1989.

[7] J. Barton and J. Wagner. Beyond threads: Resource sharing in UNIX. In *Winter 1988 Usenix Conference Proceedings*, February 1988.

[8] CACM Vol. 12,No.6. *An Anomaly in Space-Time Characteristics of Certain Programs*, 1969.

[9] T. Doeppner. Threads - a system for the support of concurrent programming. Computer Science Technical Report CS-87-11, Brown University, June 1987.

[10] IEEE Technical Committee on Operating Systems. *Threads Extension for Portable Operating System*, 1990.

[11] S.Loverso J.Boykin, A.Langerman. *The Parallelization of MACH / 4.3BSD: Design Philosophy.* USENIX Workshop on Distributed and Multiprocessor Systems, Fort Lauderdale, Florida, 1989.

[12] MASSCOMP, Concurrent Company. *RTU: a real-time UNIX operating system*, 1990.

[13] MODCOMP, an AEG Company, Fort Lauderdale, Florida. *REAL/IX- Fully Preemptive Real-Time UNIX*, 1990.

[14] I. Nassi. A preliminary report on the Ultramax: A massively parallel shared memory multiprocessor. *DARPA Workshop on Parallel Architectures for Mathematical and Scientific Computing*, July 1987.

[15] I. Nassi and N.Habermann. Efficient implementation of ada tasks. Technical Report CMU-CS-80-103, Carnegie Mellon University, 1980.

[16] R. Rashid. Threads of a new system. *Unix Review*, August 1986.

[17] Ready Systems Inc., Mountain View, CA. *VRTX User Manual*, 1989.

[18] Sun Microsystems Inc. *Remote Procedure Call Protocol*, 1986.

[19] A. Tevanian, R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. Mach threads and the unix kernel: The battle for control. In *USENIX Conference Proceedings Summer 1987*, June 1987.

[20] UNIX World, Vol. 4, No. 11. *UNIX Overcomes Real-Time Limitations*, 1987.

APPENDIX

ESCAPES(2)

NAME
escapes - introduction to escapes from normal Unix conventions

DESCRIPTION
This section describes a group of Multimax extensions that address shortcomings of standard Unix implementations. Provided are hooks for alternative scheduling mechanisms, avoidance of typical Unix-imposed overhead, sharing of such resources as address space and file descriptors, and closer access to and control over the hardware. These extensions allow the Multimax to support user-level threads packages, processes which field interrupts and provide what are normally viewed as system services, as well as real-time and diagnostic programs which need closer control over various hardware facilities.

Unix processes provide an elegant abstraction of the underlying hardware, embodying a "virtual machine." The operating system maintains the illusion that each program has this machine all to itself, and the emphasis has been primarily on separation, or keeping one process from interfering with another. In this model of communicating sequential processes, all interprocess communication is mediated by the operating system, imposing substantial overhead. A better model is that of a multi-threaded parallel machine, where various resources are arbitrarily shared among the multiple threads of execution. The res_ctl(2) call provides the ability to share certain resources (notably address space and file descriptors) between related processes. This makes possible multi-threaded applications running in the same address space, as well as efficient implementations of inter-process communication built on shared memory. Routines are provided (see brk(2)) to grow shared data, and fork_set_stack(2) is provided to handle the details of process duplication (or creation of another thread) when the stack is shared. set_index(2) and get_index(2) supply one long integer's worth of truly private storage. This allows processes with a totally shared address space to store one item of data in a way that is less expensive to access than a normal system call such as getpid(2).

Threads packages and parallel applications require different scheduling than the pre-emptive time-slicing normally provided. The gang(2) system call allows one or more processes to remain locked down to the CPUs they are running on, exempt from being rescheduled. Once locked down, these CPUs avoid many of the normal sources of system overhead, such as the handling of clock and I/O completion interrupts, performance of working set scans, and interrupt-driven rescheduling. All but one of the CPUs in the system may have processes latched onto them. The yield(2) and block(2) calls allow a process to voluntarily give up it's current time slice, allowing another process to be run, and unblock(2) is complementary to block. These allow user scheduling of (non-gang) processes.

Several routines provide closer access to and control over system hardware. cachectl(2) allows a program to disable or enable any hardware caching of ranges of memory. This allows a real-time program to establish worst case memory access timings, or run with completely deterministic access times. It also allows a program to indicate it's intention to execute modifiable data.

Further determinism can be gained by setting a bit in the header of the executable image (see /usr/include/aouthdr.h). The U_BATC bit requests

GANG(2)

NAME
gang - gang scheduling operations

SYNOPSIS
```
#include <sys/types.h>
#include <sys/gang.h>

int gang_create (nprocs, flags)
int nprocs;
int flags;

int gang_enter (gang_id)
int gang_id;

int gang_exit ()

int gang_destroy (gang_id)
int gang_id;
```

DESCRIPTION
A GANG is a set of processes which are scheduled as a unit to yield maximum performance and predictability for fine-grained parallel processing algorithms. The number of processes which can enter a GANG is limited by the number of CPUs assigned to the GANG. The processes will be permanently assigned to a specific CPU within the GANG; once within the GANG, the value returned from getcpuid(3C) should remain constant. These processes will not be subject to normal time slice interrupts or scheduling algorithms. Similarly, the CPUs will not have to field ordinary arbitrated I/O interrupts. User level signals will be delivered immediately.

The process which creates the GANG must either be super-user or super-group (see intro(2)). Processes which enter the GANG must have a matching user ID. Super-group processes also have special permissions for the nice(2), setpriority(2), and plock(2) system calls (note that while executing within a GANG, the process priority is irrelevant).

gang create is used to create a new GANG, the return value is the unique GANG identifier (gang id) to be used later in calls to gang enter and gang destroy.

nprocs specifies how many processors are to be allocated to the new GANG. If processors (CPUs) are already allocated to existing GANGs (processor scheduling classes), it may not be possible to immediately satisfy the request. In this case, the gang create call will return immediately with an EAGAIN error if (flags & GANG_WAIT) is "false", or be suspended interruptibly until the resources are available.

flags is used to specify options for the created GANG as follows:

GANG_WAIT — If the user is willing to wait (interruptibly) for CPU resources to become available, GANG_WAIT should be used.

GANG_KEEP — The system normally destroys a GANG when the last member process exits (see exit(2)) from the GANG.

GANG_KEEP retains the GANG for future gang enter requests. This option should be used sparingly since the processors may be unavailable for other use in the interim.

GANG_EXITSIG — Causes the signal SIGHUP to be sent to existing GANG members when a process exits the system without having done an explicit gang exit.

GANG_RMIDSIG — Causes the signal SIGKILL to be sent to existing GANG members when a GANG is explicitly removed with gang destroy.

gang create will fail without creating a GANG if one or more of the following are true:

[ENOSPC] nproc is greater than the system-imposed maximum. The limit is determined by the number of actual processors in the system less the system configuration parameter reservecpu.

[EPERM] The requesting process does not have super-group privileges (see intro(2)).

[EAGAIN] The available number or the configuration of the unallocated processors (CPUs) is not sufficient to fill the request.

[EINTR] A signal was received while waiting for resources to become available.

RES_CTL(2)

NAME
res_ctl - control inheritance of resources

SYNOPSIS
```
#include <sys/types.h>
#include <sys/res_ctl.h>
int res_ctl(numres, resvecptr, resop)
int       numres;
res_vec_t *resvecptr;
int       resop;

RES_SET(  res, &rset)
RES_CLR(  res, &rset)
RES_ISSET(res, &rset)
RES_ZERO( res, &rset)
int res;
res_vec_t rset;
```

DESCRIPTION
res_ctl controls the inheritance of various resources when a process performs a fork(2) operation. The first numres resources will be marked as inheritable, uninheritable or will be made private, depending on resop.

The resource sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such resource sets:

RES_CLR(res, &rset)	Removed res from rset.
RES_ISSET(res, &rset)	Indicates if res is a member of rset. If res is a member, RES_ISSET is nonzero. Otherwise, RES_ISSET is 0.
RES_SET(res, &rset)	Includes a particular resource res in rset.
RES_ZERO(&rset)	Initializes a resource set rset to the null set.

resop may be one of the following:

RESOP_SHARE
This operation causes the resources specified to be shared rather than copied over fork(2), between the parent and child processes. This operation has no effect on the current state of those resources.

RESOP_NO_SHARE
This operation is the inverse of RESOP_SHARE; it causes the resources specified to be copied rather than shared over fork(2). It also has no effect on the current state of those resources.

RESOP_PRIVATE
This operation severs any sharing of the resources specified, giving the process its own private resource. This might be used by a parent process to disassociate itself from shared resources set up solely to be passed to and shared among a group of recently spawned processes.

Resources are specified by setting bits in the resource sets whose addresses are passed in resvecptr.

RES_STACK_BIT	OS-supported stack area
RES_DATA_BIT	data space (.data + .bss)
RES_TEXT_BIT	instruction space (shared by default)
RES_IO_BIT	file descriptors and I/O state
RES_SIG_BIT	signal handling
RES_PERM_BIT	permissions
RES_LIM_BIT	user limits
RES_STAT_BIT	per-process statistics (when such exist)

RETURN VALUE
res_ctl replaces the given resource set with the set of resources currently marked to be shared with processes resulting from fork(2); bits set to one indicate that a resource will be shared over fork(2). Zero is returned upon successful completion; otherwise, it returns -1 and the global variable errno is set to indicate the error.

res_ctl will fail if oneor more

[EINVAL] Invalid parameter or unimplemented operation [ENOMEM] Insufficient resources to complete request [EAGAIN] Insufficient resources to complete request

INTERCEPT(2)

NAME
intercept - allow direct handling of hardware traps and interrupts

SYNOPSIS
```
#include <sys/types.h>
#include <sys/escapes.h>

int intercept(n_intrs, intrs)
int n_intrs;
struct interrupt *intrs;
```

DESCRIPTION
intercept is a Multimax extension which requests that certain interrupts and traps be vectored directly to code in the calling process. n_intrs specifies the number of traps and interrupts which will be intercepted, and what code will handle which traps is specified by the members of the interrupt array: intrs[0] intrs[1], ... iov[n_intrs-1].

The interrupt structure is defined as

```
struct interrupt {
    int trap_offset;
    caddr_t trap_handler;
};
```

Each interrupt entry specifies the hardware specific byte offset of the trap in the system vector table, and the base address of the handler in the program's address space.

Traps and interrupts indicated are then handled in supervisor mode by the calling process.

SEE ALSO
aoutdump(1), intro(2), escapes(2)

DIAGNOSTICS
Upon successful completion, zero is returned. Otherwise, a -1 is returned and errno is set to indicate the error.

intercept fails and no traps and interrupts are intercepted if one or more of the following are true:

[EPERM] the caller is neither a member of the supergroup nor superuser.

[EINVAL] Invalid trap offsets are specified, or an address points outside of the program's address space.

[EINVAL] The program is not memory resident and was not loaded with the U_HOSTED attribute allowing it to provide co-resident operating system services by being dual-mapped into both supervisor and user space.