ChorusOS man pages section 5FEA: ChorusOS Features and APIs

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3338
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

NAME	This section gives the names of the commands or functions documented, followed by a brief description of what they do.
SYNOPSIS	This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘ "`filename` . . ." .
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are

	interdependent, such that everything enclosed must be treated as a unit.
FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
DESCRIPTION	This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.
OPTIONS	This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output - standard output, standard error, or output files - generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE	This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar
EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions.
FILES	This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

ChorusOS Features and APIs

NAME

Intro – ChorusOS APIs and features

DESCRIPTION

This manual contains the list of the different ChorusOS APIs and features.

Some of those APIs are always available in the system, such as the APIs provided as part of the ChorusOS core executive or of the pure library functions.

Other APIs depend on a specific feature value in the ChorusOS system configuration. To understand the relationship between features, use the *ews*(1CC) configuration tool (see the *ChorusOS 4.0 Introduction*).

The implementation of a feature API is done dynamically at system build:

- Either by adding a module implementing the feature to the ChorusOS system at link time
- Or by adding an actor implementing the feature in the list of actors to be loaded automatically at boot time

The ChorusOS: APIs and features are described in man pages grouped in section 5FEA. Features are identified by their feature name as known in the configuration tools. APIs, or modules implementing the APIs, are identified by a generic keyword, for example MEM for memory management APIs. Some of the features and APIs may be grouped in families and described in the same page.

For more information on a particular feature or API, see the manual page in this 5FEA section.

ACTOR_EXTENDED_MNG	extended actor management
ADMIN_CHORUSSTAT	print Chorus statistics
ADMIN_IFCONFIG	configure network interface parameters
ADMIN_MOUNT	mount file systems
ADMIN_NETSTAT	show network status
ADMIN_RARP	acquire local IP address through rarp
ADMIN_ROUTE	manipulate the routing tables
ADMIN_SHUTDOWN	close down the system
AF_LOCAL	support of AF_LOCAL domain for sockets
BPF	Berkeley packet filter
BSD	BSD compatible I/O system calls
CORE	basic core executive
DATE	time of day service

DEBUG_SYSTEM	system debug
DEV_MEM	memory device files
DYNAMIC_LIB	dynamic libraries
ENV	system environment
EVENT	event flag sets
FIFOFS	support of named pipes
FLASH	flash memory device
FS_MAPPER	support for swap in the IOM
GZ_FILE	uncompress executable files
HOT_RESTART	management of restartable actors and persistent memory
IDE_DISK	IDE disk device
IOM_IPC	list of ChorusOS features
IOM_OSI	include an IPC stack in the IOM
IPC	provide OSI stack entry points
IPC_REMOTE	inter-process communication
IPC_REMOTE_COMM	IPC remote communication
LAPBIND	built-in nameserver for local access points
LAPSAFE	safe mode for LAP invocations
LOCAL_CONSOLE	local console command interpretor
LOG	system logging
MEM	MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING
	memory management
MIPC	message queues
MON	system monitoring
MSDOSFS	MSDOS File system
MUTEX	mutexes
NFS_CLIENT	client side of the nfs protocol implementation

NFS_SERVER	server side of the nfs protocol implementation
PERF	performance support
POSIX-SEM	POSIX 1003.1b semaphores
POSIX-THREADS	POSIX 1003.1c pthread features
POSIX-TIMERS	POSIX 1003.1b clock / timer features
POSIX_MQ	POSIX 1003.1b message queue feature
POSIX_SHM	POSIX 1003.1b shared memory objects feature
POSIX_SOCKETS	POSIX 1003.1g compatible socket system feature
PPP	point to point protocol network interface
PRIVATE-DATA	per-thread and per-actor data
RAM_DISK	RAM disk device
RPC	RPC compatible I / O system calls
RSH	rsh command interpetor
RTC	realtime clock
RTMUTEX	real-time mutexes
SCHED	ROUND_ROBIN
	scheduler features
SCSI_DISK	disk device SCSI bus
SEM	semaphores
SLIP	point to point protocol network interface
SYSTIME	system time
TIMEOUT	interrupt-level timing
TIMER	system time
UFS	UNIX File System
USER_MODE	support for user actors
VTIMER	thread execution timing
VTTY	serial line support

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Name	Description
ACTOR_EXTENDED_MNGT(5FEA)	extended actor management
ADMIN_CHORUSSTAT(5FEA)	print Chorus statistics
ADMIN_IFCONFIG(5FEA)	configure network interface parameters
ADMIN_MOUNT(5FEA)	mount file systems
ADMIN_NETSTAT(5FEA)	show network status
ADMIN_RARP(5FEA)	acquire local IP address through rarp
ADMIN_ROUTE(5FEA)	manipulate the routing tables
ADMIN_SHUTDOWN(5FEA)	close down the system
AF_LOCAL(5FEA)	support of AF_LOCAL domain for sockets
BPF(5FEA)	Berkeley packet filter
BSD(5FEA)	BSD compatible I/O system calls
CORE(5FEA)	basic core executive
DATE(5FEA)	time of day service
DEBUG_SYSTEM(1M)	system debug
DEV_MEM(5FEA)	memory device files
DYNAMIC_LIB(5FEA)	dynamic libraries
ENV(5FEA)	system environment
EVENT(5FEA)	event flag sets
FIFOFS(5FEA)	support of named pipes.
FLASH(5FEA)	flash memory device
FS_MAPPER(5FEA)	support for swap in the IOM
GZ_FILE(5FEA)	uncompress executable files
HOT_RESTART(5FEA)	management of restartable actors and persistent memory

IDE_DISK(5FEA)	IDE disk device
IOM_IPC(5FEA)	include an IPC stack in the IOM
IOM_OSI(5FEA)	provide OSI stack entry points
IPC(5FEA)	inter-process communication
IPC_REMOTE(5FEA)	support for remote communication
IPC_REMOTE_COMM(1M)	IPC remote communication
LAPBIND(5FEA)	built-in nameserver for local access points
LAPSAFE(5FEA)	safe mode for LAP invocations
LOCAL_CONSOLE(5FEA)	local console command interpretor
LOG(1M)	system logging
MEM(5FEA)	memory management features
MEM_FLAT(5FEA)	See MEM(5FEA)
MEM_PROTECTED(5FEA)	See MEM(5FEA)
MEM_VIRTUAL(5FEA)	See MEM(5FEA)
MIPC(5FEA)	message queues
MON(1M)	system monitoring
MSDOSFS(5FEA)	MSDOS File system
MUTEX(5FEA)	mutexes
NFS_CLIENT(5FEA)	client side of the nfs protocol implementation
NFS_SERVER(5FEA)	server side of the nfs protocol implementation
ON_DEMAND_PAGING(5FEA)	See MEM(5FEA)
PERF(1M)	performance support
POSIX-SEM(5FEA)	POSIX 1003.1b semaphores
POSIX-THREADS(5FEA)	POSIX 1003.1c pthread features
POSIX-TIMERS(5FEA)	POSIX 1003.1b clock/timer features
POSIX_MQ(5FEA)	POSIX 1003.1b message queue feature

POSIX_SHM(5FEA)	POSIX 1003.1b shared memory objects feature
POSIX_SOCKETS(5FEA)	POSIX 1003.1g compatible socket system feature
PPP(5FEA)	point to point protocol network interface
PRIVATE-DATA(5FEA)	per-thread and per-actor data
RAM_DISK(5FEA)	RAM disk device
ROUND_ROBIN(5FEA)	See SCHED(5FEA)
RPC(5FEA)	RPC compatible I/O system calls
RSH(5FEA)	rsh command interpetor
RTC(1M)	realtime clock
RTMUTEX(1M)	real-time mutex
SCHED(5FEA)	scheduler features
SCSI_DISK(5FEA)	disk device SCSI bus
SEM(5FEA)	semaphores
SLIP(5FEA)	point to point protocol network interface
SYSTIME(5FEA)	system time
TIMEOUT(5FEA)	interrupt-level timing
TIMER(5FEA)	general interval timing service
UFS(5FEA)	UNIX File System
USER_MODE(5FEA)	support for user actors
VIRTUAL_ADDRESS_SPACE(5FEA)	See MEM(5FEA)
VTIMER(5FEA)	thread execution timing
VTTY(5FEA)	serial line support

NAME

ACTOR_EXTENDED_MNGT – extended actor management

FEATURE SUMMARY

The ACTOR_EXTENDED_MNGT feature provides extended management functions for ChorusOS actors, including dynamic loading and control of them. This feature also provides the underlying support for more advanced features, such as support of dynamically loadable libraries (DYNAMIC_LIB) as well as the uncompression of actors or libraries at load time (GZ_FILE). For general information see `intro`(2K).

API

The ACTOR_EXTENDED_MNGT feature API is summarized in the following table. These calls apply only to actors running under the extended actor management feature.

Function	**Comment**
`_exit`	Terminate an extended actor.
`acap`	Get an extended actor capability.
`aconf`	Get configurable system variables.
`acred`	Get / Set extended actor credentials.
`acreate`	Create a new actor.
`aload`	Load an actor.
`astart`	Start an actor.
`afexecve`	Create and load a new extended actor.
`afexecl`	Create and load a new extended actor.
`afexecv`	Create and load a new extended actor.
`afexecle`	Create and load a new extended actor.
`afexeclp`	Create and load a new extended actor.
`afexecvp`	Create and load a new extended actor.
`agetId`	Get an extended actor’s ID.
`akill`	Kill an extended actor.
`astat`	List all extended actors active on a site.
`await`	Wait for an extended actor to terminate or stop.
`awaits`	Wait for an extended actor to terminate or stop.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_CHORUSSTAT – print Chorus statistics

FEATURE SUMMARY

The ADMIN_CHORUSSTAT feature provides support for the built-in `chorusStat` command of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the chorusStat service, please refer to the `chorusStat` man page. Note that even though the ADMIN_CHORUSSTAT feature is not configured, one can still obtain statistical information on Chorus by running the `chorusStat`(1CC) command which is a "standalone" version of the C_INIT built-in command.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_IFCONFIG – configure network interface parameters

FEATURE SUMMARY

The ADMIN_IFCONFIG feature provides support for the built-in `ifconfig` command of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the ifconfig service, please refer to the `ifconfig` man page. Note that even though the ADMIN_IFCONFIG feature is not configured, one can still configure network interface parameters by running the `ifconfig`(1M) command which is a "standalone" version of the C_INIT built-in command. However, in order to be able to set up the network interface of the target system correctly at init time, the ADMIN_IFCONFIG feature is usually set..

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_MOUNT – mount file systems

FEATURE SUMMARY

The ADMIN_MOUNT feature provides support for the built-in `mount` and `umount` commands of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the mount service, please refer to the `mount`(1M) man page. This feature provides support to mount and umount UFS, MSDOS and NFS file systems. If this feature is not set, there will be no way to run a command to mount a file system within the target system. In this type of configuration, file systems will have to be mounted by user—provided applications embedded within the boot image using the *mount*(2POSIX) system call.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_NETSTAT – show network status

FEATURE SUMMARY

The ADMIN_NETSTAT feature provides support for the built-in `netstat`command of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the netstat service, please refer to the `netstat` man page. Note that even though the ADMIN_NETSTAT feature is not configured, it is still be possible get the network status by running the `netstat`(1CC) command which is a "standalone" version of the C_INIT built-in command.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_RARP – acquire local IP address through rarp

FEATURE SUMMARY

The ADMIN_RARP feature provides support for the built-in `rarp` command of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the rarp service, please refer to the `rarp`(1M) man page. The ADMIN_RARP feature enables the system to retrieve its local IP address using the RARP protocol, and to configure a network interface accordingly. This feature requires the ADMIN_IFCONFIG feature to be set, however, the reverse is not true.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_ROUTE – manipulate the routing tables

FEATURE SUMMARY

The ADMIN_ROUTE feature provides support for the built-in `route` command of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in command will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the route service, please refer to the `route` man page. Note that even though the `ADMIN_ROUTE` feature is not configured, one can still manage the routing tables of the ChorusOS system by running the `route`(1M) command which is a "standalone" version of the C_INIT built-in command, using the following syntax:

```
host% arun /bin/route arguments
```

However, in order to be able to set up the routing tables of the target system appropriately at init time, the ADMIN_ROUTE feature is usually set.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ADMIN_SHUTDOWN – close down the system

FEATURE SUMMARY

The ADMIN_SHUTDOWN feature provides support for the built-in `shutdown` and `reboot` commands of `C_INIT`(1M). If the feature is not configured, the C_INIT built-in commands will display an error message. This feature affects the contents of the `ADMIN` system actor. For more accurate information on the shutdown service, please refer to the `shutdown` man page. This feature allows you to stop part of the system or all of it, and potentially to reboot the system. Note that even though the ADMIN_SHUTDOWN feature is not configured, it is still be possible to stop the system by running the `shutdown`(1M) command which is a "standalone" version of the C_INIT built-in command.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

AF_LOCAL – support of AF_LOCAL domain for sockets

FEATURE SUMMARY

The AF_LOCAL feature provides support for the AF_LOCAL domain for sockets. It requires and complements the POSIX_SOCKETS feature which provides the AF_INET domain by itself. For general information on this feature, see `intro`(2POSIX), and the POSIX draft standard P1003.1g.

API

The AF_LOCAL feature API is the POSIX_SOCKETS feature API in addition to the following API:

Function	**Comment**
`pipe`	Create descriptor pair for interprocess communication.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

BPF – Berkeley packet filter

FEATURE SUMMARY

The BPF feature provides a raw interface to data link layers in a protocol—independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism. It must be configured for use by the ADMIN_RARP feature, or by the `dhcp_client`(1).

API

The BPF feature does not export any APIs itself, but allows access to the `/dev/bpfxx` devices.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

BSD – BSD compatible I/O system calls

FEATURE SUMMARY

The most important BSD library provides BSD compatible file I/O system calls. Note that this feature is simply a library which may or may not be linked with an application. It is not a feature which may be turned on or off when configuring a system.

API

The most important BSD library calls are summarized below. None of these library calls support multi-threaded applications.

Functions manipulating the termios structure:	`cfgetispeed`
	`cfgetospeed`
	`cfmakeraw`
	`cfsetispeed`
	`cfsetospeed`
	`cfsetpeed`

Database access methods:	`dbopen`
Get working directory	`getcwd` `getwd`
Get generic disk description by its name	`getdiskbyname`
Get or set the name of the machine	`gethostname` `sethostname`
Get information about mounted file systems	`getmntinfo`
Generate pathnames matching a pattern	`glob` `golbfree`
Gt or set system information	`sysctlbyname`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

CORE – basic core executive

FEATURE SUMMARY

The Core Executive implements the basic ChorusOS execution model and provides the framework for all other features which can be configured. Every system must include a Core Executive.

The Core Executive exports the basic set of kernel abstractions and services:

- The unit of application modularization (`actor`).
- The unit of execution (`thread`).
- Thread control operations.
- Exception management services.
- A minimal interrupt management service

No synchronization, scheduling, time, communication or memory management policies are wired into the core executive. These policies and services are provided by additional features, which the user may select depending on the particular hardware and software requirements.

Actors

The first abstraction exported by the Core Executive is the `actor`. The actor is the unit of program modularization for both applications and subsystems; subsystem servers and individual applications programs are examples of actors. The actor is also the unit of resource encapsulation. Through internal kernel interfaces, each kernel feature may attach particular resources to actors. For example, at the Core Executive level, threads are resources attached to actors. Memory management features will attach address spaces to actors.

In the core executive, actors may be created and deleted dynamically. When an actor is created, those kernel features which manage per-actor resources will be invoked by the core executive. At actor destruction time, these features will again be invoked in order to free their allocated resources.

Actors are designated by capabilities. An actor capability is an opaque identifier. The knowledge of the capability of an actor yields all rights regarding the actor (creating, deleting and modifying its resources, through the various feature interfaces). By default, only the creator of an actor knows the actor capability, though the creator can give the name to others.

The Core Executive offers a very basic framework for actor typing. In most cases, the Core Executive does not exploit this information; it acts as a repository of the typing information and makes it available to other kernel features when relevant:

- First, an actor may be either a `supervisor` actor or a `user` actor. This information defines the nature of the actor address space (for memory management features). Note that the actor privilege is not relevant for a memory management feature which does not support multiple protected address spaces. In this case, user actors are equivalent to supervisor actors.

- Second, an actor may be `trusted`. Trusted actors are also referred to as `system actors`. A supervisor actor is by definition trusted. When trusted, a user actor gets access to certain additional services.

Threads

The second important abstraction is the `thread`. The thread is the unit of execution, and represents a single flow of sequential execution of a program.

Threads execute at either `user` or `supervisor` privilege. Supervisor threads are *trusted* by the kernel. They execute in the privileged processor mode, with access to restricted processor instructions, and in the system address space. Certain kernel services are restricted to supervisor threads. User threads execute in non privileged mode and in isolated address spaces. However, user threads may execute temporarily with supervisor privilege. This can occur as a result of a trap or exception, as handlers for these events are necessarily created within supervisor actors, and in other cases.

A thread is created within a specific actor, but may execute in the context of several different actors during its lifetime. The current environment of an executing thread may involve two different actors:

HOME ACTOR	The actor in which the thread was created. The home actor of a thread is constant over the life of the thread.
EXECUTION ACTOR	The actor on behalf of which the thread currently executes. Any thread has the right to access and manage the resources of its execution actor. Local resources of an actor are named using `local identifiers` (handles), whose scope is limited to the local actor. When an operation applies to an actor other than the current thread's execution actor, the actor capability must be provided in addition to the local identifier of the resource.

Certain operations during the life of a thread depend on its execution actor. For example, an exception in a thread is managed according to the exception handling set for its execution actor.

When a thread is created, its home and execution actors are identical. During a local cross-actor invocation (see LAP(5FEA)), the execution actor changes for the duration of handler execution, then is reset to its original setting: the execution actor is set to the actor that owns the handler being invoked.

In this document the "current actor" refers to the execution actor unless otherwise noted. The interface constant K_MYACTOR and the `actorSelf` system call also refer to the execution actor of the currently executing thread.

When the home actor and execution actor of a thread are identical, it is said to be executing *internally*, or at application level. During periods when the execution actor is different from the home actor, the thread executes *externally*, or at server or kernel level. The distinction between internal and external is important in several areas of the API, within the Core Executive and in other features. Unlike the execution privilege, it allows uniform treatment of application actor execution regardless of whether the application actor is a user actor or a supervisor actor.

When the system is configured with MEM_FLM, only supervisor threads are supported.

Thread scheduling

The Core Executive only implements a very basic scheduling policy, and may be complemented by a scheduler feature. The interface between the Core Executive and a scheduler feature is a kernel internal interface.

From the Core Executive point of view, threads are independent entities. For example, a given scheduler may schedule multiple threads of an actor to run in parallel on a multiprocessor node.

Execution control

The Core Executive exports primitives allowing threads to coordinate their execution with other threads or with external events, as well as functions to asynchronously influence the execution of other threads.

It is possible to:

- Wait until an event is posted to the current thread.
- Stop (and restart) another thread.
- Stop (and restart) all of the threads of an actor.
- Abort another thread.

The semantics of these operations are the following:

Awaiting an event; signalling a waiting thread

CORE exports a simple thread synchronization interface built around the `threadSem` (*thread semaphore*) object. A `threadSem` is a user-defined object, logically equivalent to a binary semaphore, which is bound to a particular thread. The primitives `threadSemWait` and `threadSemPost` may be used to block the current thread on its `threadSem`, and to awaken a blocked thread, respectively. The binary semaphore state is effectively a "post pending" flag, which eliminates the possibility of a race condition in case a post operation executes before the corresponding wait.

This interface serves as a base for sleep (context switching) locks and other synchronization functions. CORE also exports non-sleep locks (see below).

Higher-level sleep synchronization functions are provided by the optional SEM and EVENT features.

The remainder of this section applies only to CORE.

Aborting a thread

A variety of kernel system calls may block the current thread when a request cannot be satisfied immediately or to await a particular event. Thread block and wakeup operations are performed by kernel features (for example synchronization features), through the invocation of a kernel internal interface.

It is possible to force a thread to exit the blocked state. Subsystems managers might need to awaken blocked threads prematurely, for example to allow them to process asynchronous signals.

When a thread is blocked, it may be ABORTABLE, depending on the blocking primitive it invoked and / or the arguments of this invocation.

The `threadAbort` primitive forces a thread which is blocked in an ABORTABLE state to be awakened. In general, the corresponding blocking call will return with a specific error code indicating the abort.

Because the invoker of `threadAbort` may not know whether the thread is currently blocked or not, the kernel also defines the behavior of `threadAbort` if the thread is active.

The effect of `threadAbort` thus depends on the state of the thread:

- ABORTABLE blocking operation, the thread is awakened (with the `K_EABORT` error code). The abort is said to *have been handled.*
- If the thread is not blocked in an ABORTABLE state, the abort event is recorded. The thread is said to be in the ABORTED state. The abort will be handled later.

When the thread is in the ABORTED state, it will handle the abort when one of the following situations occurs:

- The thread executes in its home actor, that is, in internal mode. In this case, the thread may execute an `abort handler`. Supervisor actors may attach abort handlers to other actors, in order to trap the entry of threads of these actors in the ABORTED state. If this type of handler is attached to the thread’s home actor, it will be invoked as soon as the thread executes in its home actor while in the ABORTED state.
- If the thread was executing in a different actor or a kernel call when aborted, and an abort handler is in effect, the handler will be invoked as soon as the thread returns to its home actor execution environment.

After execution of the abort handler, the abort is considered as having been handled.

- The thread invokes an ABORTABLE blocking primitive. In this case, this primitive will return immediately with the `K_EABORT` error code. The abort has been handled.
- The thread invokes the `threadAborted` kernel primitive. This primitive returns its current state (aborted or not) and clears it.

Asynchronous execution control operations (`threadAbort`) take effect immediately only if the target thread is currently running in internal mode, that is, if its current execution actor matches its home actor. In the case where the thread has changed its execution actor (through a kernel call, trap, or other) the operation will be deferred until the execution actor is reset to the home actor (for example, at return from kernel call or trap). Deletion of a thread is also considered an asynchronous control operation in this sense. Thus a thread is immune from deletion (except if it deletes itself) when its execution actor is changed for a trap or other cross-actor invocation. It will be deleted only when it returns to its home actor.

Thread and actor states

A thread, once created, may always be found in one of the following states:

ACTIVE	The thread is running or ready to run.
INACTIVE	The thread was just created and has not yet been activated (CORE only).
STOPPED	The thread (or the thread's actor) has been stopped (CORE only).
WAITING	The thread is blocked, waiting for a particular event or time interval. A WAITING thread may be ABORTABLE or NONABORTABLE.

A thread may be created either in the ACTIVE state or in the INACTIVE state. In the latter case, it must be activated with a `threadActivate` call before it can run.

Threads enter the WAITING state voluntarily, by invoking a blocking system call. The transition back to the ACTIVE state is normally triggered by a corresponding wakeup-style system call. In addition, a WAITING thread that is also ABORTABLE may be forced to exit its waiting state by the use of the `threadAbort` call. In certain cases this call also diverts the execution of an active thread.

Along with these four major states, there are several sets of substates which are orthogonal to the major states.

ABORTED / NON-ABORTED	The thread was in the ABORTED state when it was aborted using `threadAbort`, but the abort has not yet been taken into account by the thread. (An abort is "consumed" by a thread either when it is forced to return from a blocking call, or when it executes an abort handler, or when it calls `threadAborted` to explicitly query and clear the ABORTED state.)
INTERNAL / EXTERNAL	As described earlier in this section, a thread is in the INTERNAL state when it executes in its home actor, and in the EXTERNAL state when its execution actor is different from its home actor.

Thread context

The register context of a thread is initialized when the thread is created. The thread creation primitive permits portable parts of the context to be initialized, such as the program counter and the stack pointer. At thread creation time, machine—specific registers are set to machine—specific default values.

Once a thread has been created, the `threadContext` primitive allows all registers to be modified, including software registers (see below).

Per-thread and per-actor kernel data

The kernel modules often need to record data specific to actors and threads. For example, a communication module will have to store the set of ports attached to an actor. Similarly, an executive module will have to manage thread lists. For this purpose, the Core Executive provides an efficient mechanism for each kernel module to allocate its own portion of the thread and actor Core Executive structures, and to retrieve the address of its portion efficiently, given a thread identifier. This is a kernel internal service.

Applications and libraries also often need to allocate per thread data structures. The Core Executive also provides support for this function. The intention is to allow the programmer to logically associate an area within the actor address space with a specific thread, and to provide a way for a thread to access this data at any step of its execution, efficiently and without the need for special language constructs.

This mechanism facilitates the programming of pools of threads that share a code segment and share memory within their actor, but where each individual thread also wants to manage some private data, outside of its stack, in the global data section. No means are provided for protecting a thread's private data against access by other threads. This can be considered an advantage, as it allows flexible and low-cost communication policies between threads of an actor.

Two registers are associated with each thread, called `software registers` A software register may be considered as an extension of the hardware registers: it is part of the thread context, and is saved / restored at each context switch. By analogy with stack pointers on certain hardware architectures, two different registers are defined, one per execution mode (user or supervisor). Two kernel calls read and write the value of the software register associated with the current execution mode of the calling thread. These calls (`threadLoadR` and `threadStoreR`) are the new "machine instructions" that allow use of these registers. (On some targets, `threadLoadR` is available as a macro, thus bypassing the cost of a kernel call.)

A common way to use a software register is to initialize it with a pointer to a thread's private data structure when the thread is initialized, using `threadStoreR`. The pointer value may then be obtained at any step of the thread execution, using `threadLoadR`. Note that the software registers of a thread may also be initialized by the creator of a thread using `threadContext`. Typically, this is used at thread creation time in order to pass the address of the thread's private data structure to the newly created thread.

The PRIVATE-DATA feature provides a higher-level API on top of the software registers. Independent sets of per-thread or per-actor data may be declared by separate program units (or library functions) within a single program. When available, it is recommended that applications use this feature rather than interact with the software registers directly.

Interrupts

The Core Executive does not intervene within the interrupt processing path. Interrupt management modules connect their handlers directly to the interrupt vector. This ensures optimal performance for critical real-time applications.

Locks

The Core Executive provides two categories of non-sleep synchronization objects (on a multiprocessor, this is often referred to as "spinning" or "busy-waiting" synchronization).

`Spin locks` disable preemption on the current processor when they are held. While preemption is disabled, the current thread has access to only a few kernel system calls. These calls are indicated in the corresponding sections. Most of them are simple operations that awaken blocked threads, set timeouts, and so on. If any other system calls are invoked with preemption disabled, results are unpredictable.

`Masked locks` disable interrupts on the current processor when they are held.

Interrupt Level Execution Mode

While executing within an interrupt handler, or within a thread that has disabled preemption, one is not allowed to block. This condition also applies when a thread holds a spin lock, a masked spin lock or masks interrupts, since these implicitly disable the preemption. In this `interrupt handler` `execution mode`, the list of system calls permitted for invocation is restricted to the following:

```
semV
threadSemPost
eventPost
msgAllocate
msgPut
svMaskedLockGet
svMaskedLockRel
svMaskAll
svUnmaskAll
svUnmask
svPreemptable
svIntrLevel
svTimeoutSet
svTimeoutCancel
```

Note that the system does not enforce this restriction. However, if non-permitted system calls are issued, the system will hang non-deterministically.

The `lapInvoke` invoked in this execution mode is subject to the restriction that the invoked lap must have been created as a raw lap.

Exceptions

Three kinds of exceptions may be encountered within an operating system kernel context:

`Traps`	Generated voluntarily by the currently running thread, they are used to change the thread privilege level in order to execute system code.
`Software exceptions`	Generated involuntarily by the current thread, generally due to errors such as division by 0.
`Panics`	Explicitly generated by some kernel modules or supervisor actors, the panics correspond to software/hardware faults which are not recoverable at the application level.

The core executive provides basic services for subsystems to handle these three kinds of exceptions. For this purpose, the Core Executive exports an interface for subsystems to declare trap, exception, and panic handlers. Exception handlers are declared on a per actor basis, while trap handlers and panic handlers are declared on a site-wide basis.

Local invocation

The Core Executive provides a low-overhead mechanism for invocation of service routines in supervisor actors on the local site, by both user and supervisor actor callers. In addition, optional extensions provide safe on-the-fly shutdown of local service routines and a local name binding service (see the LAPSAFE and LAPBIND features).

Local service routines are used for two purposes. First, various subsystems and individual servers require a means of exporting an API to their clients on the local site. Second, supervisor actors connect handlers for certain hardware and software events (for example, exceptions and timeouts). Because the kernel is effectively the client in the latter case, a minimal local invocation mechanism is mandatory in any kernel instance implementing those handlers.

The *local access point*, or *lap*, is a generic software interface for kernel-mediated calls from one actor to another on the local site. A lap is a kernel object which represents a handler function that has been exported by a supervisor actor for invocation by client actors. Once a lap has been created, both server and clients refer to that lap via an opaque name called a *lap descriptor*. When the server creates a lap, the kernel generates and returns the lap descriptor. It can then be exported to clients, either directly to a specific client, via an established communication path, or for general use, via a nameserver. Any client that obtains a lap descriptor may invoke the lap, that is, call the handler function, via a kernel call. At invocation, the kernel performs certain actions to ensure a correct execution environment for the handler. In particular, the execution actor and host actor of the current thread are set to the handler's home actor for the duration of the handler invocation.

The basic or "raw" lap service provides a lightweight invocation with a minimum of functionality. A server creates a lap and obtains a lap descriptor using the `svLapCreate` system call, passing the handler entry address and a "cookie" value as arguments. At handler invocation, the cookie will be passed as an argument. This is useful when several lap objects use the same function entry point. The lap descriptor, once in possession of a client, may be used to invoke the handler. This is accomplished with the `lapInvoke` system call. No validity checking is performed either during the handler invocation or at the return to the calling code in the client actor. If the server withdraws its handler using `svLapDelete`, there is no protection for threads currently invoking the handler. Moreover, future invocations to the deleted lap are also not inhibited.

The lap descriptor is an opaque structure, with no application-visible fields. Functions are provided for optimized manipulations (clearing, testing validity, duplicating) of lap descriptors without introducing dependencies in an application on a particular descriptor format.

For efficiency reasons, the lap mechanism provides only a very simple argument passing scheme based on the assumption of direct access by a server to client memory. A single argument pointer is passed in the invocation, and the management of argument lists, types, and so on is left to individual server convention.

API

The Core Executive feature API is summarized in the following table.

Function	Comment
actorCreate	Create an actor
actorDelete	Delete an actor
actorSelf	Get the current actor's capability
lapDescDup	Duplicate a lap descriptor
lapDescIsZero	Check a lap descriptor
lapDescZero	Clear a lap descriptor
lapInvoke	Invoke a lap
lapResolve	Find a lap descriptor by name
threadActivate	Activate a newly created thread
threadContext	Get or/and set a thread's context
threadCreate	Create a thread
threadDelete	Delete a thread
threadDelay	Delay the current thread
threadLoadR	Get software register
threadName	Set/Get thread symbolic name
threadSelf	Get the current thread's LI
threadSemInit	Initialize a thread semaphore
threadSemWait	Wait on a thread semaphore
threadSemPost	Signal a thread semaphore
threadStat	Get thread information
threadStoreR	Set software register
svExcHandler	Set an actor's exception handler
svActorExcHandlerConnect	Connect an actor's exception handler
svActorExctHandlerDisconnect	Disconnect an actor's exception handler
svActorExctHandlerGetConnected	Get an actor's exception handler
svGetInvoker	Get handler invoker
svLapCreate	Create a lap
svLapDelete	Delete a lap
svMaskedLockGet	Disable interrupts and get a spin lock
svMaskedLockInit	Initialize a spin lock
svMaskedLockRel	Release a spin lock and enable interrupts
svSpinLockGet	Disable preemption and get a spin lock
svSpinLockInit	Initialize a spin lock
svSpinLockRel	Release a spin lock and enable preemption
svSpinLockTry	Try to get a spin lock and disable preemption
svSysCtx	Get the system context structure address
svSysPanic	Force panic handling processing
svSysReboot	Request a reboot of the local size
sySysTrapHandlerConnect	Connect a trap handler
sySysTrapHandlerDisconnect	Disconnect a trap handler
sySysTrapHandlerGetConnected	Get a trap handler
svTrapConnect	Connect a trap handler
svTrapDisConnect	Disconnect a trap handler
sysGetConf	Get ChorusOS module configuration value
sysRead	Read characters from system console
sysReboot	Request a reboot of the local site
sysWrite	Write characters from system console
sysPoll	Poll characters from system console

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

DATE – time of day service

FEATURE SUMMARY

The DATE feature maintains the time of day in Universal Time (UT, also known as GMT, Greenwich Mean Time). The notion of local time does not exist at the ChorusOS API level. Time zones and local seasonal adjustments are managed by libraries outside the kernel.

The time of day is obtained and manipulated using the `univTime`, `univTimeSet`, and `univTimeAdjust` system calls.

API

The DATE API is summarized in the following table:

Function	**Comment**
`univTime`	Get time of day.
`univTimeAdjust`	Adjust time of day.
`univTimeGetRes`	Get time of day resolution.
`univTimeSet`	Set time of day.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

DEBUG_SYSTEM – system debug

DESCRIPTION

The `DEBUG_SYSTEM` feature enables remote debugging of the ChorusOS operating system with the XRAY Debugger for ChorusOS. XRAY communicates through a dedicated debug server, named `rdbs` (see `rdbs`(1CC)), running on the host, which in turn communicates with the debug agent running on the target through the host debug API.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rdbs`(1CC)

ChorusOS 4.0 Introduction

NAME

DEV_MEM – memory device files

FEATURE SUMMARY

The `DEV_MEM` feature provides a raw interface to memory devices such as `/dev/zero`, `/dev/null`, `/dev/kmem` and `/dev/mem`.

API

The `DEV_MEM` feature does not export an API itself, but allows access to the devices listed above.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

DYNAMIC_LIB – dynamic libraries

FEATURE SUMMARY

The DYNAMIC_LIB feature provides support for dynamic libraries within ChorusOS. It requires and complements the ACTOR_EXTENDED_MNGT feature, so that actors may be linked with dynamic libraries. These libraries are loaded and mapped within the actor address space at execution time. Symbol resolution is performed at library load time. This feature also enables a running actor to ask for a library to be loaded and installed within its address space, and then to resolve symbols within this library. The feature takes care of dependencies between libraries.

API

The DYNAMIC_LIB feature API is summarized in the following table. These calls apply only to actors running under the extended actor management and dynamic libraries features.

Function	**Comment**
`dladdr`	Translate address into symbolic information.
`dlclose`	Close a dynamic library.
`dlerror`	Get diagnostic information.
`dlopen`	Gain access to a dynamic library file.
`dlsym`	Get the address of a symbol in a dymanic library.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ENV – system environment

FEATURE SUMMARY

The ChorusOS environment variables provide to users and applications the ability to define configuration parameters at various stages of the system construction and operation (boot and run time), and to applications to get the values of these parameters at run time.

These `dynamic` configuration parameters take the form of a string environment, that is, a set of string pairs (name, value).

API

The ENV API is summarized in the following table:

Function	**Comment**
`sysGetEnv`	Get a value.
`sysSetEnv`	Set a value.
`sysUnsetEnv`	Delete a value.

SEE ALSO

`configurator`(1CC), `ews`(1CC)

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

EVENT – event flag sets

FEATURE SUMMARY

The EVENT feature provides management of *event flag* sets.

An event flags set is a set of bits in memory associated with a thread wait queue. Each bit is associated with one *event*. In this feature, the set is implemented as an unsigned integer, therefore the maximum number of flags in a set is `8*sizeof(int)`. Within a set, each event flag is designated by an integer `0` and `8*sizeof(int)`.

When a flag is set, it is said to be `posted`, and the associated event is considered to have occurred. Otherwise, the associated event has not yet occurred. Both threads and interrupt handlers can use event flag sets for signalling purposes.

A thread can wait for a conjunctive *(and)* or disjunctive *(or)* subset of the events in one event flags set. Several threads may be pending on the same event. In that case, each of the threads will be made eligible to run when the event occurs.

Three operations are available on event flag sets: `eventWait`, `eventPost` and `enventClear`.

Event flag sets are data structures allocated in the client actors' address spaces. No kernel data structure is allocated for these objects. they are simply designated by the address of the structures. The number of these types of objects that threads may use is thus unlimited.

API

The EVENT API is summarized in the following table:

Function	**Comment**
`eventClear`	Clear event(s) in an event flag set.
`eventInit`	Initialize an event flag set.
`eventPost`	Signal event(s) to an event flag set.
`eventWait`	Wait for events in an event flag set.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

FIFOFS – support of named pipes.

FEATURE SUMMARY

The FIFOFS feature provides support for named pipes. It requires either the NFS_CLIENT or UFS to be configured, as well as POSIX_SOCKETS and AF_LOCAL.

API

The FIFOFS feature does not have its own API, but enables nodes created using `mkfifo` to actually be used as pipes.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

FLASH – flash memory device

FEATURE SUMMARY

The FLASH feature provides an interface to access a memory device. The flash memory may then be formatted, labelled and used to support regular file systems. The FLASH feature relies on the flash support based on the Flite 1.2 BSP, and is not supported for all target family architectures.

API

The FLASH feature does not itself export an API.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`flashdefrag`(1M), `format`(1M)

NAME

FS_MAPPER – support for swap in the IOM

FEATURE SUMMARY

The FS_MAPPER feature provides support for swap in the IOM. It requires either the IDE_DISK or SCSI_DISK to be configured, as well as VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING.

API

The FS_MAPPER feature exports the `swapon`(1M) system call.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

The *ChorusOS 4.0 File System Administration Guide* provides instructions on how to activate a swap partition.

NOTES

This release supports only one swap partition, which must be on a local disk.

NAME

GZ_FILE – uncompress executable files

FEATURE SUMMARY

The GZ_FILE feature enables dynamically loaded actors as well as dynamic libraries to be uncompressed at load time, prior to execution. This allows the repository space required to store these compressed files to be minimized. The GZ_FILE feature requires the ACTOR_EXTENDED_MNGT feature to be configured.

API

The GZ_LIB feature has no API. It is based on the gunzip tool. Thus, an executable file compressed on the host system using the gzip command (whose suffix is then ".gz") will be automatically recognized as a compressed executable file or dynamic library and uncompressed by the system at load time.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

HOT_RESTART – management of restartable actors and persistent memory

FEATURE SUMMARY

The HOT_RESTART feature provides support for rapidly reloading and reinitializing failed ChorusOS actors. Actors which benefit from this support are known as *restartable actors*. HOT_RESTART also provides all actors (not just restartable actors) with a means of storing persistent data.

The HOT_RESTART feature relies on the use of *persistent memory*, i.e., memory which can persist beyond a run-time instance of an actor. When HOT_RESTART is active, the initial text and data segments for restartable actors are stored in persistent memory. When the restartable actor fails, it is reloaded from this persistent memory, without accessing stable storage. Such an operation is known as *restarting* the actor, and is generally faster than reloading the actor from stable storage.

HOT_RESTART exports the following services:

- An actor restart mechanism which detects crashes in restartable actors and automatically restarts them from persistent memory, at the same addresses, with their code and data segments reset to their initial state.
- Named persistent memory allocation. Actors can allocate persistent memory as named blocks. These blocks can be used to store additional data which will persist beyond the lifetime of the run-time instance of the actor. Named persistent memory can be used by both restartable actors and non-restartable actors.
- A site restart mechanism to restart the kernel, boot actors and all restartable actors on a system without accessing stable storage.

The HOT_RESTART feature requires the ACTOR_EXTENDED_MNGT, LAPSAFE and ADMIN_SHUTDOWN features. For more information about hot restart, see the *ChorusOS Hot Restart Programmer's Guide*.

API

The HOT_RESTART feature API is summarized in the following tables. All functions are grouped in the `2RESTART` ChorusOS man page section.

Restartable Actor API

Function	**Comment**
`hrfexec()`	Spawn and load a hot restartable actor. This call can only be made from a hot restartable actor.
`hrGetActorGroup()`	Get the identifier of a given restartable actor's restart group.
`hrKillGroup()`	Kill (and do not restart) a group of restartable actors. This call is restricted to supervisor and trusted user actors.

The `sysShutdown`(2K) function, and the C_INIT commands `arun`(1M), `akill`(1M), `aps`(1M), `shutdown`(1M) and `restart`(1M) also provide support for restartable actors. For more information, see the corresponding manual pages.

Persistent Memory Management API

The persistent memory management API is available to all actors, both restartable and non-restartable.

`pmmAllocate()`	Allocate or retrieve a block of persistent memory.
`pmmFree()`	Free a block of persistent memory.
`pmmFreeAll()`	Free a group of persistent memory blocks.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

IDE_DISK – IDE disk device

FEATURE SUMMARY

The IDE_DISK feature provides an interface in order to access IDE disks. These "disks" may then be initialized and used as regular file systems. The IDE_DISK feature relies on the IDE bus support provided by the BSP to get access to disks connected on that bus.

API

The IDE_DISK feature does not itself export an API.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

IOM_IPC – include an IPC stack in the IOM

DESCRIPTION

The `IOM_IPC` feature provides support for the `ethIpcStackAttach`(2K) system call and the corresponding built-in `C_INIT`(1M) command, `ethIpcStackAttach`. If the feature is not configured, the built-in `C_INIT` command will display an error message.

If the `IOM_IPC` feature is set to `true`, an IPC stack is included in the `IOM` system actor. The IPC stack may be attached to an Ethernet interface.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`C_INIT`(1M), `ethIpcStackAttach`(2K)

NAME

IOM_OSI – provide OSI stack entry points

DESCRIPTION

The `IOM_OSI` feature provides support for the `ethOsiStackAttach`(2K) system call.

This call enables a user-provided actor to attach an OSI network stack to the IOM network engine

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ethOsiStackAttach`(2K)

NAME

IPC – inter-process communication

FEATURE SUMMARY

The IPC feature provides powerful asynchronous and synchronous communication services.

The IPC feature exports the following basic communication abstractions:

- The unit of communication (`message`).
- Point-to-point communication endpoints (`port`).
- Multicast communication endpoints (`groups`).

When the IPC_REMOTE feature is set, IPC services are provided in a distributed, location-transparent way, allowing applications distributed across the different nodes, or `sites`, of a network to communicate as if they were co-located on the same node.

BASIC CONCEPTS

The IPC feature allows threads to communicate and synchronize when they do not share memory, for example when they do not run on the same node.

Communications rely on the exchange of `messages` through `ports`.

Unique identifiers

The IPC location-transparent communication service is based on a uniform global naming scheme: communication entities are named using global *unique identifiers*. Two types of global identifiers are distinguished: `static` identifiers, which are provided to the system by the applications, and `dynamic` identifiers, which are returned by the system to the application.

Static identifiers are built in a deterministic way from *stamps* provided by the applications. On a site, only one communication object can be created with a given static identifier within the same communication feature. The maximum number of static stamps is fixed.

Network-wide dynamic identifiers, assigned by the system, are guaranteed to be unique across site reboots for a long time. The dynamic identifier of a new communication object is initially only known by the actor which creates the communication object. The actor can transmit this identifier to its clients through any application-specific communication mechanism (within a message returned to the client, for instance).

Messages

A message is an untyped string of bytes of variable but limited size (64 KB), called the `message body`. The sender of the message may optionally join a second byte string to the message body, the `message annex`. The message annex has a fixed size (120 bytes). The message body and the message annex are transferred with `copy` semantics from the sender address space to the receiver address space.

A `current message` is associated with each thread. The current message of a thread is a system descriptor of the last message received by the thread. The current message is used when the thread requires to reply to the sender of the message or acquire protection information about the sender of the message. This concept of current message allows the most common case, in which threads reply to messages in the order they are received, to be optimized and simplified. However, for other cases, the kernel provides the facility to `save` the current message, and `restore` a previously saved message as the current message.

Ports

Messages are not addressed directly to threads, but to intermediate entities called `ports`. Ports are named using `unique identifiers`.

A port is an "address" to which messages can be sent, and which has a queue holding the messages received by the port but not yet consumed by the threads. Port queues have a fixed maximum size (system parameter).

For a thread to be able to consume the messages received by a port, it is necessary that this port be `attached` to the actor that supports the thread. When a port is created by a thread, the thread attaches the port to an actor (possibly different from the one that supports the thread). The port receives a `local identifier`, relative to the actor it is attached to.

A port can only be attached to a single actor at a time, but can be successively attached to different actors: a port can `migrate` from one actor to another. This migration can be accompanied, or not, by the messages already received by the port and not yet consumed by a thread. The concept of port provides the basis for dynamic reconfiguration. The extra level of indirection (the ports) between any two communicating threads means that the threads supplying a given `service` can be changed from a thread of one actor to a thread of another actor. This is done by changing the attachment of the appropriate port from the first thread's actor to the new thread's actor.

When an actor is created, a first port is automatically attached to it: this port is called the actor's `default port`. The actor's default port may not be migrated nor deleted.

Groups of ports

Ports can be assembled into `groups` of ports. The concept of group extends port-to-port addressing between threads by adding a multicast facility. Alternatively, functional access to a service can be selected from among a group of (equivalent) services through use of port groups.

Creating a group of ports only allocates a name for the group. Ports may then be inserted into the group: the port group is built dynamically. A port can be removed from a group. Groups may not contain other groups.

Like an actor (see the description of the CORE feature), a group is named by a capability. This capability contains a unique identifier (UI), specific to the group. This UI can be used for sending messages to the ports in the group. The full group capability is needed in order to modify the group configuration (inserting ports in and removing ports from the group).

As for ports, messages are addressed to port groups by their UI. In the case of a group UI, the address is accompanied by an `address mode`. The possible address modes are:

- Broadcast to all ports in the group (`broadcast` mode).
- Addressing one of the ports of the group, arbitrarily selected (`functional` mode).
- Addressing one of the ports of the group, located on the same site as a given object designated by its UI (`associative functional` mode).
- Addressing one of the ports of the group, assuming that the selected port UI is on a different site from that of a given UI (`exclusive functional` mode).

Semantics of communication

The IPC services allows threads to exchange messages in either `asynchronous` mode or in `demand/response` (that is, `Remote Procedure Call (RPC)`) mode.

`Asynchronous mode:` The sender of an asynchronous message is blocked only during the time of local processing of the message by the system. The system does not guarantee that the message has been deposited at the destination location.

`RPC mode:` The RPC protocol allows the construction of client-server applications, using a demand / response protocol with management of `transactions`. The client is blocked until a response is returned from the server, or a user-defined optional timeout occurs. RPC guarantees at-most-once semantics for the delivery of the request; it also guarantees that the response received by a client is definitely that of the server and corresponds effectively to the request (and not to a former request to which the response might have been lost.). RPC also allows a client to be unblocked (with an error result) if the server is unreachable or if the server has crashed before emitting a response.

Finally, this protocol supports the propagation of abortion through the RPC. This mechanism is called `abort propagation`: when a thread which is waiting for an RPC reply is aborted, this event is propagated to the thread which is currently servicing the client request.

A thread that attempts to receive a message on a port is blocked until a message is received or a user-defined optional time-out occurs. A thread may attempt to receive a message on several ports at a time. Among the set of ports attached to an actor, a subset of `enabled ports` is defined. A thread may attempt to receive a message sent to any of its actor's enabled ports. Ports attached to an actor may be dynamically enabled or disabled.

When a port is enabled, it receives a priority value: if several of the enabled ports hold a message when a thread attempts to receive on the enabled set of ports, the port with the highest priority is selected. The actor's default port may not be enabled.

When a port is not `enabled`, it is `disabled`. This does not mean that the port may not be used to send or receive messages. It only means that the port may not be used in multiple port receive requests. The default value is disabled.

Message handlers

As described above, the conventional way for an actor to consume messages delivered to its ports is to have threads explicitly expressing receive requests on those ports. An alternative to this scheme is the use of `message handlers`. Instead of explicitly creating threads, an actor may attach a handler (a routine in its address space) to the port. When a message is delivered to the port, the handler is executed within the context of a kernel-provided thread.

Message handlers and explicit receive requests are exclusive: when a message handler has been attached to a port, any attempt by a thread to receive a message on that port returns an error.

The use of message handlers is restricted to supervisor actors. It allows significant optimization of the RPC protocol when both client and server reside on the same site, avoiding thread context switches (from the kernel point of view, the client thread is used to run the handler) and memory copies (copy of the message into kernel buffers is avoided).

The way messages are consumed (threads or handler) is totally transparent for the client (the sender of messages). The strategy is selected by the server only.

Protection identifiers

The IPC feature provides a `Protection Identifier` (PI) to each actor and to each port. The structure of the Protection Identifiers is fixed, but the feature does not associate any semantics to their values. The kernel only acts as a secure repository for these identifiers.

An actor receives, when its IPC context is initialized, a PI equal to that of the actor that created it. A port also receives a PI equal to that of the actor that created it. A system thread can change the PI of any actor or port. Subsystem process managers are in charge of managing the values given to the PI of the actors and ports they control.

When a message is sent, it is stamped with the PI of both the sending actor and its port. These values can be read by the receiver of the message, which can apply its own protection policies and thus decide if it should reject the message. Subsystem servers may then apply the subsystem-specific protection policies, according to the PI semantics defined by the subsystem process manager.

Reconfiguration

The kernel allows the dynamic reconfiguration of services by permitting the `migration` of ports This reconfiguration mechanism requires both servers involved in the reconfiguration to be active at the same time.

The kernel also offers mechanisms permitting one to manage the stability of the system, even in the presence of failures of servers. The concept of port groups is used to establish the stability of server addresses. Note that a port group collects several ports together. A server that possesses a port group capability can insert new ports into the group, replacing the ports that were attached to servers that have terminated.

A client that *references a group UI* (rather than directly referencing the port attached to a server) can continue to obtain the needed services once the terminated port has been replaced in the group.

In other words, the lifetime of a group of ports is unlimited, because groups continue to exist even when ports within the group have terminated (logically, a group needs to contain only a single port, and this only if the server is alive). Thus clients can have stable services as long as their requests for services are made by emission of a message towards a group.

API

The IPC feature API is summarized in the following table:

Function	**Comment**
`actorPi`	Modify the PI of an actor.
`portCreate`	Create a port.
`portDeclare`	Declare a port.
`portDelete`	Destroy a port.
`portDisable`	Disable a port.
`portEnable`	Enable a port.
`portGetSeqNum`	Get a port sequence number.
`portLi`	Acquire the LI of a port.
`portMigrate`	Migrate a port.
`portPi`	Modify the PI of a port.
`portUi`	Acquire the UI of a port.
`grpAllocate`	Allocate a group name .
`grpPortInsert`	Insert a port into a group.
`grpPortRemove`	Remove a port from a group.

`ipcCall`	Send synchronously.
`ipcGetData`	Get the current message's body.
`ipcReceive`	Receive a message.
`ipcReply`	Reply to the current message.
`ipcRestore`	Restore a message as the current message.
`ipcSave`	Save the current message.
`ipcSend`	Send asynchronously.
`ipcSysInfo`	Get information about the current message.
`ipcTarget`	Construct an address.
`svMsgHandler`	Connect a message handler.
`svMsgHdlReply`	Prepare a reply to a handled message.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

IPC_REMOTE – support for remote communication

FEATURE SUMMARY

The IPC_REMOTE feature enables support for communication among multiple sites in a network, using a ChorusOS location-transparent communication feature such as IPC. Without this feature, IPC services may be used only within a single site.

See *IPC(5FEA)*.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

IPC_REMOTE_COMM – IPC remote communication

DESCRIPTION

If you set IPC_REMOTE, you can specify the communication method by setting the IPC_REMOTE_COMM feature. By default, this is set to EXT for external networking protocols. You can also set it to VME, and have the communication managed by the kernel directly.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

IPC_REMOTE(5FEA)

NAME

LAPBIND – built-in nameserver for local access points

FEATURE SUMMARY

Low overhead, same-site invocation of functions and APIs exported by supervisor actors may be done through use of *local access points*, or LAPs (see *CORE*(5FEA)). A LAP is designated and invoked via its LAP descriptor. A LAP descriptor may be directly transmitted by a server to one or more specific client actors, via shared memory or as an argument in another invocation. Alternatively, the LAPBIND feature provides a nameserver from which a LAP descriptor may be requested and obtained indirectly, using a static symbolic name which may be an arbitrary character string. Using the nameserver, a LAP may be exported to any potential client that knows the symbolic name of the LAP (or of the service exported via the LAP).

A server may optionally establish a name binding using `svLapBind`, and remove a binding using `svLapUnbind`. A client uses `lapResolve` to obtain a lap descriptor, given its symbolic name, optionally waiting if the name is not yet available.

API

The LAPBIND API is summarized in the following table:

Function	**Comment**
`lapResolve`	Find a lap descriptor by name.
`svLapBind`	Bind a name to a lap.
`svLapUnbind`	Unbind a lap name.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

NAME

LAPSAFE – safe mode for LAP invocations

FEATURE SUMMARY

The LAPSAFE feature does not export an API directly. It modifies the function and semantics of local access point (LAP) creation and invocation. In particular, it enables the K_LAP_SAFE option (see *svLapCreate*(2K)), which causes validity checking to be turned on for an individual LAP. If a LAP is invalid or has been deleted, `lapInvoke` will fail cleanly with an error return. That is, the `svLapDelete` call will block until all pending invocations have returned. Future invocations to the deleted lap through `lapInvoke` are also inhibited. This option allows a LAP to be safely withdrawn even when client actors continue to exist. This is useful for clean shutdown and reconfiguration of servers. The added control on the invocation of a SAFE laps w.r.t a RAW lap, has an impact on the performance and the size of kernel code.

The LAPSAFE feature is a prerequisite for hot restart (see the HOT_RESTART feature)man page.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

LOCAL_CONSOLE – local console command interpretor

FEATURE SUMMARY

The LOCAL_CONSOLE feature affects the configuration of the C_INIT actor, see `C_INIT`(1M). When configured, it starts running the C_INIT command interpretor on the local console of the target system forever. All C_INIT commands described within the `C_INIT`(1M) man page are available. This allows a ChorusOS system to be administered without the need for a host system and an NFS server being available. This feature is not exclusive to the C_INIT `RSH`(5FEA) feature. Both can be set, enabling the C_INIT command interpreter to be accessed either locally or remotely through the rsh protocol at the same time.

API

The LOCAL_CONSOLE feature does not have its own API. All commands defined by C_INIT may be typed in on the target console.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

LOG – system logging

DESCRIPTION

The `LOG` feature provides support for logging console activity on a target system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x / 750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object’s mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x / 750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object’s mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x/750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for "subregions" within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object's mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x / 750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object’s mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MIPC – message queues

FEATURE SUMMARY

The MIPC optional feature is designed to allow an application composed of one or multiple actors to create a *shared* communication environment (or `message space`) within which these actors can exchange messages in a very efficient way. In particular, supervisor and user actors of the same application can exchange messages using the MIPC service. Furthermore, these messages can be initially allocated and sent by interrupt handlers in order to be processed later in the context of threads.

BASIC CONCEPTS

Message spaces

The MIPC service is designed around the concept of `message space` which encapsulates, within a single entity, both a set of message `pools` shared by all actors of the application, and a set of message `queues` through which these actors exchange messages allocated from the shared message pools.

Each message pool is defined by a pair (message size, number of messages) provided by the application when it creates the message space. The configuration of the set of message pools depends on the communication needs of the application. From the application point of view, message pool requirements depend on the size range of the messages exchanged by the application, and the distribution of messages within the size range.

A message space is a temporary resource which must be explicitly created by the application. Once created, a message space may be opened by other actors of the application. A message space is bound to the actor which creates it and is automatically deleted when its creating actor and all actors which opened it have been deleted.

When an actor opens a message space, the system first assigns a private identifier to the message space. This identifier is returned to the calling actor and is used to designate the message space in all functions of the interface. The shared message pools are then mapped in the address space of the actor, at an address chosen automatically by the system.

Messages and queues

A message is simply an array of bytes which can be structured and manipulated at application level through any appropriate convention. Messages are presented to actors as pointers within their addressing spaces.

Messages are posted to `message queues`. Within a message space, a queue is designated by an unsigned integer which corresponds to its index in the set of queues. Messages can also have priorities.

All resources of a message space are shared without any restriction by all actors of the application which open it. Any of these actors can allocate messages from the shared message pools. In the same way, all actors have both send and receive rights on each queue of the message space.

Most applications only need to create a single message space. However, the MIPC service is designed to allow an application to create or open multiple message spaces. Within these types of applications, messages cannot be directly exchanged across different message spaces. In other words, a message allocated from [a message pool of] one message space cannot be sent to a queue of another message space.

API

The MIPC API is summarized in the following table:

Function	**Comment**
`msgAllocate`	Allocate a message.
`msgFree`	Free a message.
`msgGet`	Get a message.
`msgPut`	Post a message.
`msgRemove`	Remove a message from a queue.
`msgSpaceCreate`	Create a message space.
`msgSpaceOpen`	Open a message space.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

NAME

MON – system monitoring

DESCRIPTION

The `MON` feature provides a means to monitor the activity of kernel objects such as threads, actors, and ports. Handlers can be connected to the events related to these objects so that, for example, information related to thread-sleep / wake events can be known. Handlers can also monitor global events, affecting the entire system.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MSDOSFS – MSDOS File system

FEATURE SUMMARY

The MSDOSFS feature provides POSIX-compatible file I/O system calls on top of the MSDOSFS file system on a local disk. Thus, it requires a local disk to be configured and accessible on the target system: at least one of the following features: RAM_DISK, IDE_DISK or SCSI_DISK must be configured. It is usually embedded in any configuration which needs to use a file system as part of the boot image of the system.

API

The MSDOSFS feature API is summarized in the following table. It is identical to the API exported by the NFS_CLIENT feature. However, some system calls within this API will return with error codes since the underlying file system layout does not allow support all of these operations, for example: `symlink`, `mknod`,... For general information on the API provided by this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993). Note that some of the calls listed are also included in other features.

Function	**Comment**
`access`	Check access permissions.
`chdir`, `fchdir`	Change current directory.
`chmod`, `fchmod`	Change access mode.
`chown`, `fchown`	Change owner.
`chroot`	Change root directory.
`dup`, `dup2`	Duplicate an open file descriptor.
`fcntl`	File control.
`flock`	Apply or remove an advisory lock on an open file.
`fpathconf`	Get configurable pathname variables.
`fsync`	Synchronize a file's in-core statistics with that on disk.
`getdirentries`	Get directory entries in a file system independent format.
`getfsstat`	Get list of all mounted file systems.
`ioctl`	Device control.
`link`	Make a hard file link.
`lseek`	Move read/write file pointer.
`mkdir`	Make a directory file.

`mkfifo`	Make fifos.
`mknod`	Create a special file.
`mount`, `umount`	Mount or unmount a file system.
`open`	Open for reading or writing.
`read`, `readv`	Read from file.
`readlink`	Read a value of a symbolic link.
`rename`	Change the name of a file.
`rmdir`	Remove a directory file.
`stat`, `fstat`, `lstat`	Get file status.
`statfs`, `fstatfs`	Get file system statistics.
`symlink`	Make a symbolic link to a file.
`sync`	Synchronize disk block in-core status with that on disk.
`truncate`, `ftruncate`	Truncate a file.
`umask`	Set file creation mode mask.
`unlink`	Remove a directory entry.
`utimes`	Set file access and modification times.
`write`, `writev`	Write to a file.

The following library calls do not support multi-threaded applications:

`close`	Close a file descriptor.
`opendir`	Open a directory.
`closedir`	Close a directory.
`readdir`	Read directory entry.
`rewinddir`	Reset directory stream.
`scandir`	Scan a directory for matching entries.
`seekdir`	Set the position of the next readdir() call in the directory stream.
`telldir`	Return current location in directory stream.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MUTEX – mutexes

FEATURE SUMMARY

The ChorusOS system provides mutual exclusion locks, which are sleep locks called `mutexes`. When using mutexes, threads sleep instead of spinning when contention occurs.

Mutexes are data structures allocated in the client actors' address spaces. No kernel data structure is allocated for these objects, they are simply designated by the addresses of the structures. The number of these types of objects that threads may use is thus unlimited.

API

The MUTEX API is summarized in the following table:

Function	**Comment**
`mutexGet`	Acquire a mutex.
`mutexInit`	Initialize a mutex.
`mutexRel`	Release a mutex.
`mutexTry`	Try to acquire a mutex.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

NFS_CLIENT – client side of the nfs protocol implementation

FEATURE SUMMARY

The NFS_CLIENT feature provides POSIX-compatible file I/O system calls on top of the NFS file system. It provides only the client side implementation of the protocol, and thus requires a host system to provide the server side implementation of the NFS protocol. The NFS_CLIENT feature can be configured to run on top of either Ethernet or PPP or SLIP. The NFS_CLIENT requires the POSIX_SOCKETS feature to be configured.

API

The NFS_CLIENT feature API is summarized in the following table. For general information on the API provided by this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993). Note that some of the calls listed are also included in other features.

Function	**Comment**
`access`	Check access permissions.
`chdir`, `fchdir`	Change current directory.
`chmod`, `fchmod`	Change access mode.
`chown`, `fchown`	Change owner.
`chroot`	Change root directory.
`close`	Close a file descriptor.
`dup`, `dup2`	Duplicate an open file descriptor.
`fcntl`	File control.
`flock`	Apply or remove an advisory lock on an open file.
`fpathconf`	Get configurable pathname variables.
`fsync`	Synchronize a file's in-core stats with that on disk.
`getdirentries`	Get directory entries in a file system independent format.
`getfsstat`	Get list of all mounted file systems.
`ioctl`	Device control.
`link`	Make a hard file link.
`lseek`	Move read / write file pointer.
`mkdir`	Make a directory file.
`mkfifo`	Make fifos.

`mknod`	Create a special file.
`mount`, `umount`	Mount or unmount a file system.
`open`	Open for reading or writing.
`read`, `readv`	Read from file.
`readlink`	Read a value of a symbolic link.
`rename`	Change the name of a file.
`rmdir`	Remove a directory file.
`stat`, `fstat`, `lstat`	Get file status.
`statfs`, `fstatfs`	Get file system statistics.
`symlink`	Make a symbolic link to a file.
`sync`	Synchronize disk block in-core status with that on disk.
`truncate`, `ftruncate`	Truncate a file.
`umask`	Set file creation mode mask.
`unlink`	Remove a directory entry.
`utimes`	Set file access and modification times.
`write`, `writev`	Write to a file.

The following library calls do not support multi-threaded applications:

`opendir`	Open a directory.
`closedir`	Close a directory>
`readdir`	Read directory entry.
`rewinddir`	Reset directory stream.
`scandir`	Scan a directory for matching entries.
`seekdir`	Set the position of the next readdir() call in the directory stream.
`telldir`	Return current location in directory stream.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

NFS_SERVER – server side of the nfs protocol implementation

FEATURE SUMMARY

The NFS_SERVER feature provides the services to provide an NFS server on top of a local file system: most commonly UFS, but possibly MSDOSFS. It provides only the server side implementation of the protocol, the client side being provided by the NFS_CLIENT feature. The NFS_SERVER requires the POSIX_SOCKETS and one of UFS or MSDOSFS features to be configured.

API

The NFS_SERVER feature API is summarized in the following table. For general information on the API provided by this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993). Note that some of the calls listed are also included in other features.

Function	**Comment**
`getfh`	Get file handle.
`nfssvc`	NFS services.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x / 750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object’s mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

PERF – performance support

DESCRIPTION

This module provides an API to share the system timer (clock) in two modes:

- The free-running mode causes the timer to overflow after reaching it's maximum value and continue to count up from its minimum value. This mode can be used for fine grained execution measurement. This deactivates the system clock.
- In the periodic mode, the system timer is shared between the application and the system tick. The timer will generate an interrupt at a constant interval. The application hander will be invoked at the required period. This mode can be used by applications such as profilers.

The PERF API closely follows the timer(9DDI).

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`timer`(9DDI)

NAME

POSIX_MQ – POSIX 1003.1b message queue feature

FEATURE SUMMARY

The POSIX_MQ feature is a compatible implementation of the POSIX 1003.1b real-time message queue API. For general information on this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993).

API

The POSIX_MQ feature API is summarized in the following table:

Function	**Comment**
`fpathconf`	Return value of configurable limit (same as for regular files)
`mq_close`	Close a message queue.
`mq_getattr`	Retrieve message queue attributes.
`mq_open`	Open a message queue.
`mq_receive`	Receive a message from a message queue.
`mq_send`	Send a message to a message queue.
`mq_setattr`	Set message queue attributes.
`mq_unlink`	Unlink a message queue.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

POSIX-SEM – POSIX 1003.1b semaphores

DESCRIPTION

The POSIX-SEM API is a compatible implementation of the POSIX 1003.1b semaphores API. For general information on this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993). Note that this feature is simply a library which may or not be linked with an application. It is not a feature which may be turned on or off when configuring a system.

API

The POSIX-SEM API is summarized in the following table. Note that some of the calls listed are also included in other features.

Function	**Comment**
`sem_init`	Initialize a semaphore..
`sem_destroy`	Delete a semaphore.
`sem_wait`	Wait on a semaphore.
`sem_trywait`	Attempt to lock a semaphore.
`sem_post`	Signal a semaphore.
`sem_getvalue`	Get semaphore counter value.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

POSIX_SHM – POSIX 1003.1b shared memory objects feature

FEATURE SUMMARY

The POSIX_SHM feature is a compatible implementation of the POSIX 1003.1b real-time shared memory objects API. For general information on this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993).

API

The POSIX_SHM feature API is summarized in the following table. Note that some of the calls listed are also included in other features.

Function	**Comment**
`close`	Close a file descriptor.
`dup`	Duplicate an open file descriptor.
`dup2`	Duplicate an open file descriptor.
`fchmod`	Change mode of file.
`fchown`	Change owner and group of a file.
`fcntl`	File control.
`fpathconf`	Get configurable pathname variables.
`fstat`	Get file status.
`ftruncate`	Set size of a shared memory object.
`mmap`	Map actor addresses to memory object.
`munmap`	Unmap previously mapped addresses.
`shm_open`	Open a shared memory object.
`shm_unlink`	Unlink a shared memory object.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

POSIX_SOCKETS – POSIX 1003.1g compatible socket system feature

FEATURE SUMMARY

The POSIX_SOCKETS feature provides POSIX-compatible socket system calls. For general information on this feature, see `intro`(2POSIX), and the POSIX draft standard P1003.1g. However, the POSIX_SOCKETS only provides support for the AF_INET domain. The AF_LOCAL domain support is provided by the AF_LOCAL feature.

API

The POSIX_SOCKETS feature API is summarized in the following table. Note that some of the calls listed are also included in other features.

Function	**Comment**
`accept`	Accept a connection on a socket.
`bind`	Bind a name to a socket.
`close`	Close a file descriptor.
`connect`	Initiate a connection on a socket.
`dup`	Duplicate an open file descriptor.
`dup2`	Duplicate an open file descriptor.
`fcntl`	File control.
`getpeername`	Get name of connected peer.
`getsockname`	Get socket name.
`setsockopt`	Set options on sockets.
`getsockopt`	Get options on sockets.
`ioctl`	Device control.
`listen`	Listen for connections on a socket.
`read`	Read from a socket.
`recv`	Receive a message from a socket.
`recvfrom`	Receive a message from a socket.
`recvmsg`	Receive a message from a socket.
`select`	Synchronous I/O multiplexing.
`send`	Send a message from a socket.
`sendto`	Send a message from a socket.
`sendmsg`	Send a message from a socket.

`shutdown`	Shut down part of a full-duplex connection.
`socket`	Create an endpoint for communication.
`socketpair`	Create a pair of connected sockets.
`write`	Write on a socket.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

POSIX-THREADS – POSIX 1003.1c pthread features

DESCRIPTION

The POSIX-THREADS API is a compatible implementation of the POSIX 1003.1c pthread API. For detailed information on this feature, see `intro`(3POSIX). Note that this feature is simply a library which may or not be linked with an application. It is not a feature which may be turned on or off when configuring a system.

API

The POSIX-THREADS API is summarized in the following table:

Function	**Comment**
`pthread_attr_init`	Initialize a thread attribute object.
`pthread_attr_destroy`	Destroy a thread attribute object.
`pthread_attr_setstacksize`	Set the stacksize attribute.
`pthread_attr_getstacksize`	Get the stacksize attribute.
`pthread_attr_setstackaddr`	Set the stackaddr attribute.
`pthread_attr_getstackaddr`	Get the stackaddr attribute.
`pthread_attr_setdetachstate`	Set the detachstate attribute.
`pthread_attr_getdetachstate`	Get the detachstate attribute.
`pthread_attr_setscope`	Set the contention scope attribute.
`pthread_attr_getscope`	Get the contention scope attribute.
`pthread_attr_setinheritsched`	Set the scheduling inheritance attribute.
`pthread_attr_getinheritsched`	Get the scheduling inheritance attribute.
`pthread_attr_setschedpolicy`	Set the scheduling policy attribute.
`pthread_attr_getschedpolicy`	Get the scheduling policy attribute.
`pthread_attr_setschedparam`	Set the scheduling parameter attribute.
`pthread_attr_getschedparam`	Get the scheduling parameter attribute.
`pthread_cond_init`	Initialize a condition variable.
`pthread_cond_destroy`	Destroy a condition variable.
`pthread_cond_signal`	Signal a condition variable.

`pthread_cond_broadcast`	Broadcast a condition variable.
`pthread_cond_wait`	Wait on a condition variable.
`pthread_cond_timedwait`	Wait with timeout on a condition variable.
`pthread_condattr_init`	Initialize a condition variable attribute object.
`pthread_condattr_destroy`	Destroy a condition variable attribute object.
`pthread_create`	Create a thread.
`pthread_equal`	Compare thread identifiers.
`pthread_exit`	Terminate the calling thread.
`pthread_join`	Wait for thread termination.
`pthread_key_create`	Create a thread-specific data key.
`pthread_key_delete`	Delete a thread-specific data key.
`pthread_kill`	Send a deletion signal to a thread.
`pthread_mutex_init`	Initialize a mutex.
`pthread_mutex_destroy`	Delete a mutex.
`pthread_mutex_lock`	Lock a mutex.
`pthread_mutex_trylock`	Attempt to lock a mutex without waiting.
`pthread_mutex_unlock`	Unlock a mutex.
`pthread_mutexattr_init`	Initialize a mutex attribute object.
`pthread_mutexattr_destroy`	Destroy a mutex attribute object.
`pthread_once`	Dynamically initialize a library.
`pthread_self`	Get the identifier of the calling thread.
`pthread_setschedparam`	Set the current scheduling policy and parameters of a thread.
`pthread_getschedparam`	Get the current scheduling policy and parameters of a thread.

`pthread_setspecific`	Associate a thread-specific value with a key.
`pthread_getspecific`	Retrieve the thread-specific value associated with a key.
`pthread_yield`, `sched_yield`	Yield the processor to another thread.
`sched_get_priority_max`	Get maximum priority for policy.
`sched_get_priority_min`	Get minimum priority for policy.
`sched_rr_get_interval`	Get time quantum for SCHED_RR policy.
`sysconf`	Get configurable system variables.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

POSIX-TIMERS – POSIX 1003.1b clock / timer features

DESCRIPTION

The POSIX-TIMERS API is a compatible implementation of the POSIX 1003.1b real-time clock / timer API. For detailed information on this feature, see `intro`(3POSIX). Note that this feature is simply a library which may or may not be linked with an application. It is not a feature which may be turned on or off when configuring a system.

API

The POSIX-TIMERS API is summarized in the following table:

Function	**Comment**
`clock_settime`	Set clock to a specified value.
`clock_gettime`	Get value of clock.
`clock_getres`	Get resolution of clock.
`nanosleep`	Delay the current thread with high resolution.
`timer_create`	Create a timer.
`timer_delete`	Delete a timer.
`timer_settime`	Set and arm or disarm a timer.
`timer_gettime`	Get remaining interval for an active timer.
`timer_getoverrun`	Get current overrun count for a timer.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

PPP – point to point protocol network interface

FEATURE SUMMARY

The PPP feature allows serial lines to be used as network interfaces using the point-to-point protocol. This feature needs to be configured in order to fully support the PPP network interface provided by the ChorusOS system.

API

The PPP feature does not export any APIs itself. It simply adds support of the PPP ifnet within the IOM.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

PRIVATE-DATA – per-thread and per-actor data

DESCRIPTION

The PRIVATE-DATA API implements a high-level interface for management of private per-thread data within the actor address space. It also provides a per-actor data service for supervisor actors only.

API

The PRIVATE-DATA API is summarized in the following table:

Function	**Comment**
`padGet`	Return actor-specific value associated with key.
`padKeyCreate`	Create an actor private key.
`padKeyDelete`	Delete an actor private key.
`padSet`	Set actor's key specific value.
`ptdErrnoAddr`	Return thread-specific errno address.
`ptdGet`	Return thread-specific value associated with key.
`ptdKeyCreate`	Create a thread-specific data key.
`ptdKeyDelete`	Delete a thread-specific data key.
`ptdRemoteGet`	Return a thread-specific data value for another thread.
`ptdRemoteSet`	Set a thread-specific data value for another thread.
`ptdSet`	Set a thread-specific value.
`ptdThreadDelete`	Delete all thread-specific values and call destructors.
`ptdThreadId`	Return the thread ID.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

RAM_DISK – RAM disk device

FEATURE SUMMARY

The RAM_DISK feature provides an interface to chunks of memory which can be seen and handled as disks. These "disks" may then be initialized and used as regular file systems, although their contents will be lost at system shutdown time. This feature is also required to get access to the MSDOS file system which is usually embedded as part of the system boot image.

API

The RAM_DISK feature does not export any APIs itself.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SCHED, ROUND_ROBIN – scheduler features

FEATURE SUMMARY

A scheduler is a feature which provides *scheduling policies* . A scheduling policy is a set of rules, procedures, or criteria used in making processor scheduling decisions. Each scheduler feature implements one or more scheduling policies (also known as scheduling classes), interacting with the Core Executive according to a defined kernel internal interface. A scheduler is mandatory in all kernel instances.

Schedulers may base scheduling decisions on thread priorities, task deadlines, quality of service objectives, time sharing response, fairness among groups of threads, or any other paradigm. The Core Executive places no constraints on the type of policy, nor on the data structures used to implement a policy. Run queue structure, for example, is left to the scheduler.

All schedulers manage a certain number of per-thread and per-system parameters and attributes, and export an API for manipulation of this information or for other operations. Several system calls may be involved. `threadScheduler` is implemented by all schedulers, for manipulation of thread-specific scheduling attributes. Scheduling parameter descriptors defined for `threadScheduler` are also used in the *schedparam* argument of the `threadCreate` system call (see the description of the CORE feature).

`schedAdmin` is supported in some schedulers for site-wide administration of scheduling parameters.

Any scheduler may export additional system calls for specific operations.

The default basic scheduler implements the CLASS_FIFO scheduling class, which provides simple preemptive scheduling based on thread priorities. The optional ROUND_ROBIN scheduler feature enables THE additional CLASS_RR scheduling class which is similar to CLASS_FIFO but adds round-robin time slicing based on a configurable time quantum.

More detailed information about these scheduling classes is found in the `threadScheduler` description.

API

The SCHED features API is summarized in the following table:

```
Function        Comment                              _FIFO  _CLASS
schedAdmin      Administer scheduling classes               +
threadScheduler Get/set thread scheduling information   +     +
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

RPC – RPC compatible I/O system calls

FEATURE SUMMARY

The RPC library compatible with Sun RPC.

API

The RPC library calls are summarized in the following table. These library calls are available with the POSIX_SOCKETS feature. None of these calls support multi-threaded applications. Note that this feature is simply a library which may or may not be linked with an application. It is not a feature which may be turned on or off when configuring a system.

Function	**Comment**
`bindresvport`	Bind a socket to a privileged IP port.
`getrpcport`	Get RPC port number.
`getrpcent` `getrpcbyname` `getrpcbynumber`	Get RPC entry.
`rpc`	Library routines for remote procedure calls:

`auth_destroy`, `authnone_create`, `authunix_create`, `authunix_create_default`, `callrpc`, `clnt_broadcast`, `clnt_call`, `clnt_destroy`, `clnt_create`, `clnt_control`, `clnt_freeres`, `clnt_getres`, `clnt_pcreateerror`, `clnt_perrno`, `clnt_perror`, `clnt_spcreateerror`, `clnt_sperrno`, `clnt_sperror`, `clnt_raw_create`, `clnttcp_create`, `clntudp_create`, `clntudp_bufcreate`, `get_myaddress`, `pmap_getmaps`, `pmap_getport`, `pmap_rmtcall`, `pmap_set`, `pmap_unset`, `registerrpc`, `svc_destroy`, `svc_freeargs`, `svc_getargs`, `svc_getcaller`, `svc_getreqset`, `svc_getreq`, `svc_register`, `svc_sendreply`, `svc_unregister`, `svcerr_auth`, `svcerr_decode`, `svcerr_noproc`, `svcerr_noprog`, `svcerr_progvers`, `svcerr_systemerr`, `svcerr_wakeauth`, `svcraw_create`, `svctcp_create`, `svcfd_create`, `svcudp_bufcreate`, `xdr_accepted_reply`, `xdr_authunix_parms`, `xdr_callmsg`, `xdr_opaque_auth`, `xdr_pmap`, `xdr_pmaplist`, `xdr_rejected_reply`, `xdr_replymsg`, `xprt_register`, `xprt_unregister`

`xdr`	Library routines for external data representation:

`xdr_array`, `xdr_bool`, `xdr_bytes`, `xdr_char`, `xdr_destroy`, `xdr_double`, `sdr_enum`, `xdr_float`, `xdr_free`, `xdr_getpos`, `xdr_inline`, `xdr_int`, `xdr_long`, `xdrmem_create`, `xdr_opaque`, `xdr_pointer`, `xdrrec_create`, `xdrrec_endofrecord`, `xdrrec_eof`, `xdrrec_skiprecord`, `xdr_reference`, `xdr_setpos`, `xdr_short`, `xdrstdio_create`, `xdr_string`, `xdr_u_char`, `xdr_u_int`, `xdr_u_long`, `xdr_u_short`, `xdr_union`, `xdr_vector`, `xdr_void`, `xdr_wrapstring`

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

RSH – rsh command interpetor

FEATURE SUMMARY

The RSH feature affects the configuration of the C_INIT actor, see `C_INIT`(1M). When configured, it starts running the C_INIT command interpreter within a rsh daemon thread on the target system forever. All C_INIT commands described within the `C_INIT`(1M) man page are available. This allows a ChorusOS system to be administered from a host without needing to access the local console of the target system. This feature is not exclusive to the C_INIT `LOCAL_CONSOLE`(5FEA) feature. Both can be set, enabling the C_INIT command interpreter to be accessed either locally or remotely through the rsh protocol at the same time.

API

The RSH feature does not have its own APIS. All commands defined by C_INIT may be typed in on the target console. It is accessed from the host using the standard rsh protocol.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

RTC – realtime clock

DESCRIPTION

This feature indicates whether a realtime clock (RTC) device is present on the target machine. When configured ’in’, it specifies that an RTC device is present and requires that the DATE feature use this RTC device. When configured ’out’, it specifies that no RTC device is present on the target and requires that the DATE feature emulate the RTC function purely in software.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`DATE`(5FEA)

NAME

RTMUTEX – real-time mutex

DESCRIPTION

The `RTMUTEX` feature implements a priority inheritance protocol in order to avoid priority inversion problems.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SCHED, ROUND_ROBIN – scheduler features

FEATURE SUMMARY

A scheduler is a feature which provides *scheduling policies* . A scheduling policy is a set of rules, procedures, or criteria used in making processor scheduling decisions. Each scheduler feature implements one or more scheduling policies (also known as scheduling classes), interacting with the Core Executive according to a defined kernel internal interface. A scheduler is mandatory in all kernel instances.

Schedulers may base scheduling decisions on thread priorities, task deadlines, quality of service objectives, time sharing response, fairness among groups of threads, or any other paradigm. The Core Executive places no constraints on the type of policy, nor on the data structures used to implement a policy. Run queue structure, for example, is left to the scheduler.

All schedulers manage a certain number of per-thread and per-system parameters and attributes, and export an API for manipulation of this information or for other operations. Several system calls may be involved. `threadScheduler` is implemented by all schedulers, for manipulation of thread-specific scheduling attributes. Scheduling parameter descriptors defined for `threadScheduler` are also used in the *schedparam* argument of the `threadCreate` system call (see the description of the CORE feature).

`schedAdmin` is supported in some schedulers for site-wide administration of scheduling parameters.

Any scheduler may export additional system calls for specific operations.

The default basic scheduler implements the CLASS_FIFO scheduling class, which provides simple preemptive scheduling based on thread priorities. The optional ROUND_ROBIN scheduler feature enables THE additional CLASS_RR scheduling class which is similar to CLASS_FIFO but adds round-robin time slicing based on a configurable time quantum.

More detailed information about these scheduling classes is found in the `threadScheduler` description.

API

The SCHED features API is summarized in the following table:

```
Function        Comment                              _FIFO  _CLASS
schedAdmin      Administer scheduling classes                  +
threadScheduler Get/set thread scheduling information   +     +
```

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SCSI_DISK – disk device SCSI bus

FEATURE SUMMARY

The SCSI_DISK feature provides an interface to access SCSI disks. These “disks” may then be initialized and used as regular file systems. The SCSI_DISK feature relies on the SCSI bus support provided by the BSP to access the disks connected on that bus.

API

The SCSI_DISK feature does not itself export an API.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SEM – semaphores

FEATURE SUMMARY

The SEM feature provides *semaphore* synchronization objects. A semaphore is an integer counter and an associated thread wait queue. When initialized, the semaphore counter receives a user-defined positive or null value.

Two main atomic operations are available on semaphores: P (or “wait”) and V (or “signal”).

The counter is decremented when a thread performs a P on a semaphore. If the counter reaches a negative value, the thread is blocked and put in the semaphore’s queue, otherwise, the thread continues its execution normally.

The counter is incremented when a thread performs a V on a semaphore. If the counter is still lower than or equal to zero, one of the threads queued in the semaphore queue is picked up and awakened.

Semaphores are data structures allocated in the client actors’ address spaces. No kernel data structure is allocated for these objects, they are simply designated by the address of the structures. The number of these types of objects that threads may use is thus unlimited.

API

The SEM API is summarized in the following table:

Function	**Comment**
`semInit`	Initialize a semaphore.
`semP`	Wait on a semaphore.
`semV`	Signal a semaphore.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SLIP – point to point protocol network interface

FEATURE SUMMARY

The SLIP feature allows serial lines to be used as network interfaces using the point-to-point protocol. This feature needs to be configured in order to support the ifnet interface.

API

The SLIP feature does not itself export any APIs.It simply adds the support of the SLIP ifnet within the IOM.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

SYSTIME – system time

FEATURE SUMMARY

The SYSTIME feature maintains the current system time, that is, the time since boot. In the default implementation, the resolution of system time is determined by the standard clock tick.

API

The SYSTIME API is summarized in the following table:

Function	**Comment**
`sysTime`	Get system time
`sysTime`	Resolution

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

TIMEOUT – interrupt-level timing

FEATURE SUMMARY

The TIMEOUT feature provides the traditional one-shot timeout service. At timeout expiration, a caller-provided handler is executed directly at interrupt level (generally on the interrupt stack, if any, and with thread scheduling disabled), and the handler execution environment is restricted correspondingly. This feature is restricted to supervisor threads.

In the current implementation, timeouts are based on a regular system-wide clock tick, and timeout granularity is determined by the clock tick. Alternative versions may implement timeouts directly on top of a hardware interval timer, at a much finer resolution.

API

The TIMEOUT API is summarized in the following table:

Function	**Comment**
`svSysTimeoutCancel`	Cancel a timeout
`svSysTimeoutSet`	Request a timeout
`svTimeoutGetRes`	Get timeout resolution

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

TIMER – general interval timing service

FEATURE SUMMARY

The TIMER feature implements a high-level timer service for both user and supervisor actors. One-shot and periodic timers are provided, with timeout notification by execution of a user-provided upcall function by a handler thread in the application actor. Handler threads may invoke any kernel or subsystem system call. This service is implemented using the TIMEOUT feature.

The extended timer facility uses the concept of a *timer* object within the actor environment. Timers are created and deleted dynamically. Once created, they are addressed by a local identifier in the context of their owning actor, and are deleted automatically when that actor terminates. Timer objects provide a naming mechanism and a locus of control for timing activities. All high-level timer operations, for example, setting, modifying, querying, or cancelling pending timeouts, refer to timer objects. Timer objects are also involved in coordination with the threads used to execute application-level notification handlers.

Applications will typically use extended timer functions via a standard application-level library (see the POSIX-TIMERS feature). Timer handler threads are created and managed by this library. The library is expected to pre-allocate stack area for the notification functions, create the thread, and initialize the thread's priority, per-thread data, and all other aspects of its execution context, using standard system calls. The thread then declares itself available for execution of the timer notification (`timerThreadPoolWait`) system call to wait for the first or next relevant timeout event. Event arrival will cause the thread to return from the system call, at which point the library code can call the application's handler. The "thread pool" interface is designed to allow one or a small number of handler threads to service an arbitrary number of timers. An application can thus create a large number of handlers without the expense of a dedicated handler thread for each one.

At most, a single notification will be active for a given timer at any point in time. If no handler thread is available when the timer interval expires, either because the notification function is still executing from a previous expiration or because the handler thread(s) is (are) occupied executing notifications for other timers, an overrun occurs. When a handler thread becomes available (that is, re-executes `timerThreadPoolWait`), it will return immediately and the notification function may be invoked immediately. At return from `timerThreadPoolWait`, an *overrun count* is delivered to the thread. An overrun count value pertains to a particular execution of the notification function. The overrun count is defined as the number of timer expirations that occurred since the previous invocation of the notify function without a handler thread available. Thus for a periodic timer, an overrun count equal to one indicates that the current invocation was delayed, but by less than the period interval.

API

The TIMER API is summarized in the following table:

Function	**Comment**
`timerThreadPoolInit`	Initialize a thread pool
`timerThreadPoolWait`	Wait for timer events
`timerCreate`	Create a timer
`timerDelete`	Delete a timer
`timerGetRes`	Get timer resolution
`timerSet`	Set a timer

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

UFS – UNIX File System

FEATURE SUMMARY

The `UFS` feature provides POSIX-compatible file I / O system calls on top of the UFS file system on a local disk. Thus, it requires a local disk to be configured and accessible on the target system. At least one of the `RAM_DISK`, `IDE_DISK` or `SCSI_DISK` features must be configured. `UFS` must be embedded in any configuration which exports local files through NFS.

API

The UFS feature API is summarized in the following table. It is identical to the API exported by the `NFS_CLIENT` feature. However, some system calls within this API will return with error codes since the underlying file system layout does not allow support all of these operations. For general information on the API provided by this feature, see `intro`(2POSIX), and the POSIX standard (IEEE Std 1003.1b-1993). Note that some of the calls listed are also included in other features.

Function	**Comment**
`access`	Check access permissions.
`chdir`, `fchdir`	Change current directory.
`chmod`, `fchmod`	Change access mode.
`chown`, `fchown`	Change owner.
`chroot`	Change root directory.
`close`	Close a file descriptor.
`dup`, `dup2`	Duplicate an open file descriptor.
`fcntl`	File control.
`flock`	Apply or remove an advisory lock on an open file.
`fpathconf`	Get configurable pathname variables.
`fsync`	Synchronize a file’s in-core statistics with that on disk.
`getdirentries`	Get directory entries in a filesystem independent format.
`getfsstat`	Get list of all mounted filesystems.
`ioctl`	Device control.
`link`	Make a hard file link.
`lseek`	Move read / write file pointer.
`mkdir`	Make a directory file.

`mkfifo`	Make fifos.
`mknod`	Create a special file.
`mount`, `umount`	Mount or unmount a filesystem.
`open`	Open for reading or writing.
`read`, `readv`	Read from file.
`readlink`	Read a value of a symbolic link.
`rename`	Change the name of a file.
`rmdir`	Remove a directory file.
`stat`, `fstat`, `lstat`	Get file status.
`statfs`, `fstatfs`	Get file system statistics.
`symlink`	Make a symbolic link to a file.
`sync`	Synchronize disk block in-core status with that on disk.
`truncate`, `ftruncate`	Truncate a file.
`umask`	Set file creation mode mask.
`unlink`	Remove a directory entry.
`utimes`	Set file access and modification times.
`write`, `writev`	Write to a file.

The following library calls do not support multithreaded applications:

`opendir`	Open a directory.
`closedir`	Close a directory.
`readdir`	Read directory entry.
`rewinddir`	Reset directory stream.
`scandir`	Scan a directory for matching entries.
`seekdir`	Set the position of the next `readdir()` call in the directory stream.
`telldir`	Return current location in directory stream.

ATTRIBUTES See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

USER_MODE – support for user actors

FEATURE SUMMARY

The USER_MODE feature enables support for user mode actors that require direct access to kernel API features.

DESCRIPTION

USER_MODE is used in all memory models.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

MEM, MEM_FLAT, MEM_PROTECTED, MEM_VIRTUAL, VIRTUAL_ADDRESS_SPACE, ON_DEMAND_PAGING – memory management features

MODULE SUMMARY

There are three distinct memory management modules, providing different levels of functionality. Note that any of the modules can be used in conjunction with the persistent memory services provided by the hot restart feature (see `HOT_RESTART` (5fea)).

MEM_FLAT: Flat memory module

The kernel and all applications run in one unique unprotected address space. This module provides simple memory allocation services.

MEM_PROTECTED: Protected memory module

This module is targeted at real-time applications able to benefit from the flexibility and protection offered by memory management units, address translation and separate address spaces. No swap or demand paging is provided. No mapper interface is provided. Accesses to programs and data stored on secondary devices must be done by application-specific file servers.

MEM_VIRTUAL: Virtual memory module

This module supports full virtual memory with swapping in and out on secondary devices. This module has been specifically designed to implement distributed UNIX; subsystems on top of the kernel.

The module exports a generic interface to implement shared memory functionalities over a network. One “coherency” mapper can be present on each site to implement the application specific memory sharing strategy.

In addition to kernel code, at least one mapper must be available on each site where secondary devices are managed.

No such mapper is needed when no secondary devices are used and all the memory needed is allocated from the physical memory. In this case, the MEM_PROTECTED module is preferable to the full MEM_VIRTUAL module.

Whenever needed by the hardware, the memory module permits access to (with system specific protections) special memory such as video RAM or memory mapped I/O.

FEATURES

To select one of the memory management modules for your system configuration use the `configurator` command or the *ews* GUI tool.

If you set the VIRTUAL_ADDRESS_SPACE feature to `false` the MEM_FLAT module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE feature to `true` and the ON_DEMAND_PAGING feature to `false` the MEM_PROTECTED module will be selected.

If you set the VIRTUAL_ADDRESS_SPACE and ON_DEMAND_PAGING features to `true` the MEM_VIRTUAL module will be selected.

The combination of VIRTUAL_ADDRESS_SPACE is `false` and ON_DEMAND_PAGING is `true` , is illegal.

Note - For some target platforms, ChorusOS does not implement all memory management modules. For example, ChorusOS for PowerPC 60x / 750 based platforms does not implement the MEM_PROTECTED module whereas ChorusOS for UltraSPARC-IIi based platforms does not implement the MEM_VIRTUAL module.

BASIC CONCEPTS

In this section, the basic memory management concepts are described. Each memory management module provides semantics for a subset or variants of these concepts, presented here as a general introduction.

Address Spaces

The address space of a processor is split into two subsets: the *supervisor address space* and the *user address space* . A separate user address space is associated with each user actor. The address space of an actor is also called the *memory context* of the actor.

A memory management module may support several different user address spaces, and perform memory context switches when required in thread scheduling.

The supervisor address space is shared by every actor, but only accessible to threads running with the SUPERVISOR privilege level. The kernel code and data are located in the supervisor address space.

In addition, some privileged actors, the *supervisor actors* , also use the supervisor address space. No user address space is allocated for these supervisor actors.

Regions

The address space is divided into non overlapping *regions* . A region is a contiguous range of *logical* memory addresses, to which certain *attributes* are associated (such as access rights, for example).

Regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance” in an actor other than the thread’s home actor.

Protections

Regions may be created with a set of access rights or protections.

The virtual pages that constitute a memory region can be protected against certain types of accesses. Protection modes are machine-dependent, but most architectures provide at least read-write and read-only. Any attempt to violate the protections triggers a *page fault* . The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are merged into one

region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

MEM_FLAT

This module is suited for systems which do not have a memory management unit, or when use of the memory management unit is required for efficiency only.

Virtual addresses match physical addresses directly. Applications may not allocate more memory than physically available.

Address Spaces

A unique supervisor address space, matching the physical address space, is featured. Any actor can access any part of the physically mapped memory, such as the ROM, the memory mapped I/O, and anywhere in the RAM.

Regions

The context of an actor is a collection of non overlapping regions. The kernel associates a linear buffer of physical memory to each region, consisting of a *memory object* . The memory object and the region have the same address and size.

On a given site, memory objects can be shared by several actors. Sharing of fractions of one memory object is not available.

At region creation time, the memory object is either allocated from free physical RAM memory, or shared from the memory object of another region.

The concept of “sharing of memory objects” is provided to control the freeing of physical memory. The memory object associated with a region is returned to the pool of free memory when the associated region is removed from its last context. This concept of sharing does not prevent an actor from accessing any part of the physically mapped memory.

It is not possible to wait for memory at region creation time. The memory object must be immediately obtainable, either by sharing or by allocating free physical memory.

Protections

There is no default protection mechanism available.

MEM_PROTECTED

The Protected Memory module (MEM_PROTECTED) is suited to systems where memory management units are available, and where the application programs are able to benefit from the flexibility and protection offered by separate address spaces. Unlike the full virtual memory management module (MEM_VIRTUAL), it is not directly possible to use secondary storage to emulate more memory than physically available. This module is primarily targeted at critical real-time applications, where memory protection is mandatory, and where low priority access to secondary storage is kept very simple.

No external segments are defined, no mapper is used and no swap to external device is available.

Regions

The kernel associates a set of physical pages with each region, consisting of a *memory object* .

At region creation time, the memory object is either allocated from free physical memory, or shared from the memory object of another region.

Sharing has a semantic of “physical sharing”.

At region creation time, it is possible to initialize a region from another region. This initialization has a semantic of physical allocation and copy at region creation time. In order to keep the MEM_PROTECTED module small, no deferred *on demand paging* technique is used.

A region of an actor maps a memory object at a given virtual address with associated access rights.

The size of a memory object is equal to the size of the associated region(s).

It is not possible to wait for memory at region creation time, the memory object must be obtainable immediately, either by sharing or by allocating free physical memory.

Protections

Violations of memory protection trigger memory fault exceptions that can be handled at the application level by supervisor actors.

For typical real-time applications, memory faults denote a catastrophic software failure that should be properly logged for off~line analysis. It should also trigger an application-designed fault recovery procedure.

MEM_VIRTUAL

This module is suitable for systems with page-based memory management units, and where the application programs need a high-level virtual memory management system.

The main functionalities are:

- This module supports multiple protected address spaces.
- On systems with secondary storage (the usual case), it is possible to allow applications to use much more virtual memory than the memory physically available.
- Pages are automatically swapped in and out when appropriate.
- This implementation also supports the mapping of *segments* into the address spaces. In the distributed case, a set of functionalities is optionally available, which allows a *distributed shared memory system* to be built. In

this type of system, several “readers” and “writers” over the network can access the same data in a controlled manner: a *coherency mapper* is needed in order to do this..

Segments

The unit of representation of information in the system is the segment.

Segments are usually located in secondary storage. The segment can be persistent (for example, files), or temporary with a lifetime tied to that of an actor or a thread, for example, swap objects.

The segments are managed by independent system servers, called *mappers* . There can be several independent mappers coexisting in the system at a given time.

The mode of representation of the objects, the identification of the objects and the access rules (protection and sharing, for instance) are defined by these mappers. The kernel defines a uniform interface to the mappers.

The kernel itself implements special forms of segment: the *memory objects* that are allocated along with the regions. Optionally, the kernel can request the default mapper to create temporary external objects in order to swap these objects.

Designation of segments

Like actors, segments are designated by capabilities.

Regions

A region of an actor maps a portion of a segment at a given virtual address with associated access rights.

The memory management provides the mapping between regions within an actor and segments (for example, files, swap objects, and shared memory).

The segments and the regions can be created and destroyed dynamically by threads. Within the limits of the protection rules, a region can be created “at a distance”: in an actor other than the requesting actor.

Many regions can define overlapping (or not) portions of segments. The segment can be shared by different actors. Segments can then be shared across the network.

The kernel also implements optimized region copying (*copy on write*).

Protections

Regions may be created with a set of access rights or *protections* .

The virtual pages that constitute a memory region can be protected against certain types of access.

An attempt to violate the protections triggers a page fault. The application can provide its own page fault handler.

Protections can be independently set for “subregions” within a source region. In this case, the source region is split into several new regions. Similarly, when two contiguous regions get the same protections back, they are combined into one region. The programmer is warned that abusing this module could result in consuming too much of the kernel resources associated with regions.

Segment representation: local caches

For each mapped segment on its site, the kernel encapsulates the physical memory holding portions of the object data in a *local cache* .

Local caches are the memory objects for this module.

Page faults generated during access to portions of a segment which are not accessible are handled by the kernel. In order to resolve these exceptions, the kernel may invoke the object’s mapper and fill the local cache with the data received from that mapper. No access to the mapper is required when the physical page is already present in the local cache. Typically, this occurs when another actor has already mapped the page on the same site.

The consistency of an object shared among regions belonging to several actors at the same site is guaranteed by the uniqueness of the segment local cache in physical memory.

When an object is shared among actors of different sites, there is one local cache per site, the *coherency mappers* maintain the consistency of these distributed caches.

Explicit access to a segment

The memory management also allows explicit access to (that is, copy of) segments without mapping them into an address space.

This kind of access to a segment uses the same local cache mechanism as described above.

Object consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read / write instructions.

Explicit access to a local cache

When complex operations are applied to segments (such as distributed cache consistency or segment truncation), explicit access to the local cache data is necessary.

For this purpose, the MEM_VIRTUAL module provides facilities for controlling the state and contents of local caches.

API

The memory management API is summarized in the following table:

Function	Comment	FLAT	PROT.	VIRT.
rgnAllocate	Allocate a region	+	+	+
rgnDup	Duplicate an address space			+
rgnFree	Free a region	+	+	+
rgnInit	Allocate a region initialized from a segment			+
rgnInitFromActor	Allocate a region initialized from another region		+	+
rgnMap	Create a region and map it to a segment			+
rgnMapFromActor	Allocate a region mapping another region	+	+	+
rgnSetInherit	Set inheritance options for a region			+
rgnSetPaging	Set paging options for a region			+
rgnSetProtect	Set protection options for a region	+	+	+
rgnStat	Get statistics of a region	+	+	+
svCopyIn	Byte copy from user address space	+	+	+
svCopyInString	String copy to user address space	+	+	+
svCopyOut	Byte to user address space	+	+	+
svPagesAllocate	Supervisor address space page allocator	+	+	+
svPagesFree	Free memory allocated by svPagesAllocate	+	+	+
svPhysAlloc	Physical memory page allocator	+	+	+
svPhysFree	Free memory allocated by svPhysAlloc	+	+	+
svPhysMap	Map a physical address to the supervisor space	+	+	+
svPhysUnMap	Destroy a mapping created by svPhysMap	+	+	+
svMemMap	Map a physical address to the supervisor space	+	+	+
svMemUnMap	Destroy a mapping created by svMemUnMap	+	+	+
vmCopy	Copy data between address spaces	+	+	+
vmFree	Free physical memory			+
vmLock	Lock virtual memory in physical memory			+
vmMapToPhys	Map a physical address to a virtual one		+	+
vmPageSize	Get the page or block size	+	+	+
vmPhysAddr	Get a physical address for a virtual one	+	+	+
vmSetPar	Set the memory management parameters			+
vmUnLock	Unlock virtual memory from physical memory			+

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

VTIMER – thread execution timing

FEATURE SUMMARY

The VTIMER feature is responsible for all functions pertaining to measurement and timing of thread execution. It exports a number of functions which are typically used by higher-level OS (for example, UNIX) subsystems.

VTIMER functions include thread accounting (`threadTimes` system call) and virtual timeouts (`svVirtualTimeoutSet` and `svVirtualTimeoutCancel` calls). A virtual timeout handler is entered as soon as the designated thread(s) has consumed the specified amount of execution time. Virtual timeouts may be set either on individual threads, for support of subsystem-level virtual timers (for example, SVR4 `setitimer` VIRTUAL and PROF timers), or on entire actors, for support of process CPU limits.

Execution time accounting may be limited to execution within the thread's home actor (*internal* execution time), or may include cross-actor invocations such as system calls (*total* execution time).

`svThreadVirtualTimeout` and `svThreadActorTimeout` handlers are invoked at thread level, and thus may use any API service, including blocking system calls. Timeout events are delivered to application threads, much like `threadAbort`. That is, a thread executes a virtual time handler on its own behalf.

API

The Time Management API is summarized in the following table:

Function	**Comment**
`svActorVirtualTimeoutCancel`	Cancel an actor's virtual timeout
`svActorVirtualTimeoutSet`	Set an actor's virtual timeout
`svThreadVirtualTimeoutCancel`	Cancel a thread's virtual timeout
`svThreadVirtualTimeoutSet`	Set a thread's virtual timeout
`svVirtualTimeoutCancel`	Cancel a virtual timeout
`svVirtualTimeoutSet`	Set a virtual timeout
`threadTimes`	Get thread execution times
`virtualTimeGetRes`	Get virtual time resolution

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

VTTY – serial line support

FEATURE SUMMARY

The VTTY feature provides support for a serial line on top of the BSP driver for higher level protocols. It is used by the SLIP and PPP features.

API

The VTTY feature does not itself export any APIs.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Index