Solaris Containers: Resource Management and Solaris Zones Developer's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–1975–13
April 2008

080401@19860

Contents

Preface 7

1 **Resource Management in the Solaris Operating System** 11
Understanding Resource Management in the Solaris OS 11
Workload Organization 11
Resource Organization 12
Resource Controls 13
Extended Accounting Facility 14
Writing Resource Management Applications 14

2 **Projects and Tasks** 15
Overview of Projects and Tasks 15
`/etc/project` File 16
Project and Task API Functions 17
Code Examples for Accessing `project` Database Entries 18
Programming Issues Associated With Projects and Tasks 19

3 **Using the C Interface to Extended Accounting** 21
Overview of the C Interface to Extended Accounting 21
Extended Accounting API Functions 21
`exacct` System Calls 22
Operations on the `exacct` File 22
Operations on `exacct` Objects 22
Miscellaneous Operations 23
C Code Examples for Accessing `exacct` Files 23

4 Using the Perl Interface to Extended Accounting 29
Extended Accounting Overview 29
Perl Interface to `libexacct` 30
Object Model 30
Benefits of Using the Perl Interface to `libexacct` 30
Perl Double-Typed Scalars 31
Perl Modules 31
`Sun::Solaris::Project` Module 33
`Sun::Solaris::Task` Module 34
`Sun::Solaris::Exacct` Module 35
`Sun::Solaris::Exacct::Catalog` Module 37
`Sun::Solaris::Exacct::File` Module 38
`Sun::Solaris::Exacct::Object` Module 40
`Sun::Solaris::Exacct::Object::Item` Module 41
`Sun::Solaris::Exacct::Object::Group` Module 42
`Sun::Solaris::Exacct::Object::_Array` Module 43
Perl Code Examples 44
Output From `dump` Method 47

5 Resource Controls 51
Overview of Resource Controls 51
Resource Controls Flags and Actions 52
`rlimit`, Resource Limit 52
`rctl`, Resource Control 52
Resource Control Values and Privilege Levels 53
Local Actions and Local Flags 53
Global Actions and Global Flags 54
Resource Control Sets Associated With a Project, Processes and Tasks 55
Signals Used With Resource Controls 60
Resource Controls API Functions 61
Operate on Action-Value Pairs of a Resource Control 61
Operate on Local Modifiable Values 61
Retrieve Local Read-Only Values 62
Retrieve Global Read-Only Actions 62
Resource Control Code Examples 62

Master Observing Process for Resource Controls 62
List all the Value-action Pairs for a Specific Resource Control 64
Set `project.cpu-shares` and Add a New Value 65
Set LWP Limit on Resource Control Blocks 66
Programming Issues Associated With Resource Controls 67

6 Dynamic Resource Pools 69
Overview of Resource Pools 69
Scheduling Class 70
Dynamic Resource Pool Constraints and Objectives 70
System Properties 71
Pools Properties 71
Processor Set Properties 72
Using `libpool` to Manipulate Pool Configurations 73
Manipulate `psets` 73
Resource Pools API Functions 74
Functions for Operating on Resource Pools and Associated Elements 74
Functions for Querying Resource Pools and Associated Elements 76
Resource Pool Code Examples 78
Ascertain the Number of CPUs in the Resource Pool 78
List All Resource Pools 79
Report Pool Statistics for a Given Pool 79
Set `pool.comment` Property and Add New Property 80
Programming Issues Associated With Resource Pools 81

7 Design Considerations for Resource Management Applications in Solaris Zones 83
Zones Overview 83
IP Networking in Zones 84
Design Considerations for Resource Management Applications in Zones 84
General Considerations When Writing Applications for Non-Global Zones 84
Specific Considerations for Shared-IP Non-Global Zones 86

8 Configuration Examples 89
`/etc/project` Project File 89

Define Two Projects ... 89
Configure Resource Controls ... 90
Configure Resource Pools ... 90
Configure FSS `project.cpu-shares` for a Project ... 90
Configure Five Applications with Different Characteristics ... 91

Index ... 95

Preface

The *Solaris Containers: Resource Management and Solaris Zones Developer's Guide* describes how to write applications that partition and manage system resources and discusses which APIs to use. This book provides programming examples and a discussion of programming issues to consider when writing an application.

Who Should Use This Book

This book is for application developers and ISVs who write applications that control or monitor the Solaris Operating System resources.

Before You Read This Book

For a detailed overview of resource management, see the *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

How This Book Is Organized

This guide is organized as follows:

Chapter 1, "Resource Management in the Solaris Operating System," introduces the Solaris 10 Resource Manager product.

Chapter 2, "Projects and Tasks," provides information about the projects and tasks facilities.

Chapter 3, "Using the C Interface to Extended Accounting," describes the C interface to the extended accounting facility.

Chapter 4, "Using the Perl Interface to Extended Accounting," describes the Perl interface to the extended accounting facility.

Chapter 5, "Resource Controls," discusses resource controls and their use.

Chapter 6, "Dynamic Resource Pools," covers dynamic resource pools.

Chapter 7, "Design Considerations for Resource Management Applications in Solaris Zones," describes the precautions that need to be taken for applications to work in Solaris zones.

Chapter 8, "Configuration Examples," provides configuration examples for the `/etc/project` file.

Documentation, Support, and Training

Sun Function	URL	Description
Documentation	`http://www.sun.com/documentation/`	Download PDF and HTML documents, and order printed documents
Support and Training	`http://www.sun.com/supportraining/`	Obtain technical support, download patches, and learn about Sun courses

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. Perform a *patch analysis*. Do *not* save the file. [Note that some emphasized items appear bold online.]

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Resource Management in the Solaris Operating System

The purpose of this manual is to help developers who are writing either utility applications for managing computer resources or self-monitoring applications that can check their own usage and adjust accordingly. This chapter provides an introduction to resource management in the Solaris Operating System (OS). The following topics are included:

- "Understanding Resource Management in the Solaris OS" on page 11
- "Writing Resource Management Applications" on page 14

Understanding Resource Management in the Solaris OS

The main concept behind resource management is that workloads on a server need to be balanced for the system to work efficiently. Without good resource management, faulty runaway workloads can bring progress to a halt, causing unnecessary delays to priority jobs. An additional benefit is that efficient resource management enables organizations to economize by consolidating servers. To enable the management of resources, the Solaris OS provides a structure for organizing workloads and resources, and provides controls for defining the quantity of resources that a particular unit of workload can consume. For an in-depth discussion of resource management from the system administrator's viewpoint, see Chapter 1, "Introduction to Solaris 10 Resource Manager," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Workload Organization

The basic unit of workload is the *process*. Process IDs (PIDs) are numbered sequentially throughout the system. By default, each user is assigned by the system administrator to a *project*, which is a network–wide administrative identifier. Each successful login to a project creates a new *task*, which is a grouping mechanism for processes. A task contains the login process as well as subsequent child processes.

For more information on projects and tasks, see Chapter 2, "Projects and Tasks (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* for the system administrator's perspective or Chapter 2, "Projects and Tasks," for the developer's point of view.

Projects can optionally be grouped into *zones*, which are set up by system administrators for security purposes to isolate groups of users. A zone can be thought of as a box in which one or more applications run isolated from all other applications on the system. Solaris zones are discussed thoroughly in Part II, "Zones," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. To learn more about special precautions for writing resource management applications that run in zones, see Chapter 7, "Design Considerations for Resource Management Applications in Solaris Zones,"

Resource Organization

The system administrator can assign workloads to specific CPUs or defined groups of CPUs in the system. CPUs can be grouped into *processor sets*, otherwise known as *psets*. A `pset` in turn can be coupled with one or more thread scheduling classes, which define CPU priorities, into a *resource pool*. Resource pools provide a convenient mechanism for a system administrator to make system resources available to users. Chapter 12, "Resource Pools (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* covers resource pools for system administrators. Programming considerations are described in Chapter 6, "Dynamic Resource Pools."

The following diagram illustrates how workload and computer resources are organized in the Solaris OS.

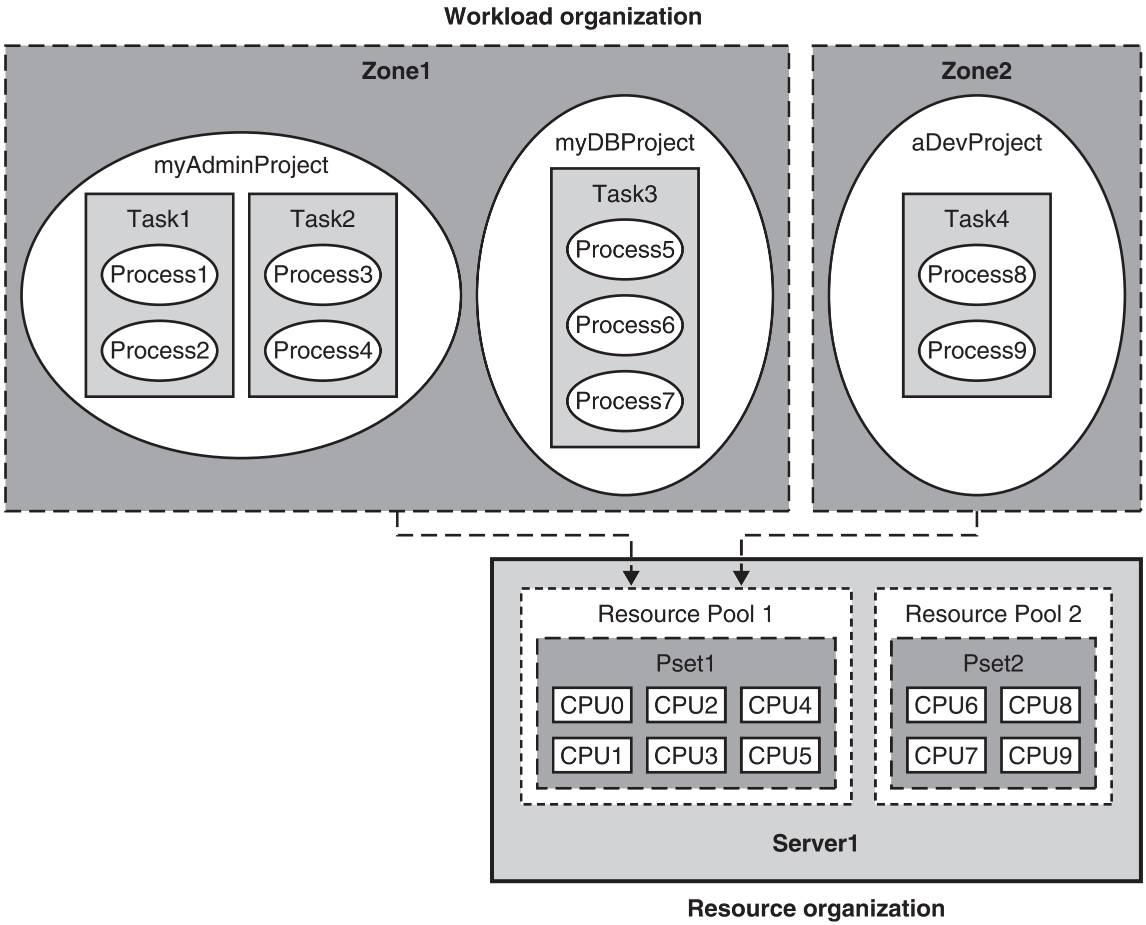


FIGURE 1–1 Workload and Resource Organization in the Solaris Operating System

Resource Controls

Simply assigning a workload unit to a resource unit is insufficient for managing the quantity of resources that users consume. To manage resources, the Solaris OS provides a set of flags, actions, and signals that are referred to collectively as *resource controls* and are stored in the `/etc/project` file. The Fair Share Scheduler (FSS), for example, can allocate shares of CPU resources among workloads based on the specified importance factor for the workloads. With these resource controls, a system administrator can set privilege levels and limit definitions for a specific project, task, or process. To learn how a system administrator uses resource controls, see Chapter 6, "Resource Controls (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. For programming considerations, see Chapter 5, "Resource Controls."

Extended Accounting Facility

In addition to the workload and resource organization, the Solaris OS provides the *extended accounting facility* for monitoring and recording system resource usage. The extended accounting facility provides system administrators with a detailed set of resource consumption statistics on processes and tasks.

The facility is described in depth for system administrators in Chapter 4, "Extended Accounting (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. The Solaris OS provides developers with both a C interface and a PERL interface to the extended accounting facility. Refer to Chapter 3, "Using the C Interface to Extended Accounting," for the C interface or Chapter 4, "Using the Perl Interface to Extended Accounting," for the PERL interface.

Writing Resource Management Applications

This manual focuses on resource management from the developer's point of view and presents information for writing the following kinds of applications:

- Resource administration applications – Utilities to perform such tasks as allocating resources, creating partitions, and scheduling jobs.
- Resource monitoring applications – Applications that check system statistics through `kstats` to determine resource usage by systems, workloads, processes, and users.
- Resource accounting utilities – Applications that provide accounting information for analysis, billing, and capacity planning.
- Self-adjusting applications – Applications that can determine their use of resources and can adjust consumption as necessary.
- Resource advisory applications – Provide hints of resource needs.???

CHAPTER 2

Projects and Tasks

The chapter discusses the workload hierarchy and provides information about projects and tasks. The following topics are covered:

- "Overview of Projects and Tasks" on page 15
- "Project and Task API Functions" on page 17
- "Code Examples for Accessing `project` Database Entries" on page 18
- "Programming Issues Associated With Projects and Tasks" on page 19

Overview of Projects and Tasks

The Solaris Operating System uses the workload hierarchy to organize the work being performed on the system. A *task* is a collection of processes that represents a workload component. A *project* is a collection of tasks that represents an entire workload. At any given time, a process can be a component of only one task and one project. The relationships in the workload hierarchy are illustrated in the following figure.

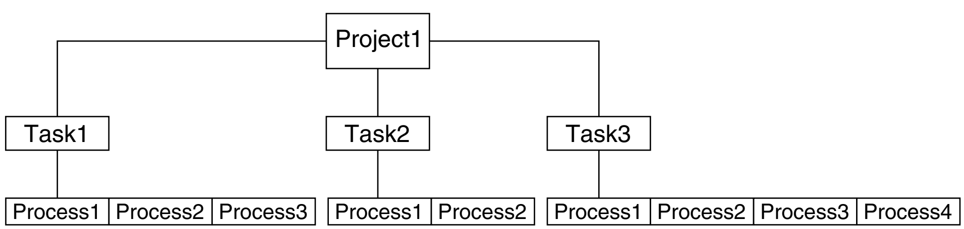


FIGURE 2–1 Workload Hierarchy

A user who is a member of more than one project can run processes in multiple projects at the same time. All processes that are started by a process inherit the project of the parent process. When you switch to a new project in a startup script, all child processes run in the new project.

An executing user process has an associated user identity (`uid`), group identity (`gid`), and project identity (`projid`). Process attributes and abilities are inherited from the user, group, and project identities to form the execution context for a task.

For an in-depth discussion of projects and tasks, see Chapter 2, "Projects and Tasks (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*. For the administration commands for managing projects and tasks, see Chapter 3, "Administering Projects and Tasks," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

`/etc/project` File

The `project` file is the heart of workload hierarchy. The `project` database is maintained on a system through the `/etc/project` file or over the network through a naming service, such as NIS or LDAP.

The `/etc/project` file contains five standard projects.

`system` This project is used for all system processes and daemons.

`user.root` All root processes run in the `user.root` project.

`noproject` This special project is for IPQoS.

`default` A default project is assigned to every user.

`group.staff` This project is used for all users in the group `staff`.

To access the project file programmatically, use the following structure:

```
struct project {
  char      *pj_name;       /* name of the project */
  projid_t   pj_projid;     /* numerical project ID */
  char      *pj_comment;    /* project comment */
  char     **pj_users;      /* vector of pointers to project user names */
  char     **pj_groups;     /* vector of pointers to project group names */
  char      *pj_attr;       /* project attributes */
};
```

The `project` structure members include the following:

`*pj_name`
Name of the project.

`pj_projid`
Project ID.

`*pj_comment`
User-supplied project description.

**pj_users
: Pointers to project user members.

**pj_groups
: Pointers to project group members.

*pj_attr
: Project attributes. Use these attributes to set values for resource controls and project pools.

Through project attributes, the resource usage can be controlled. Four prefixes are used to group the types of resource control attributes:

- `project.*` – This prefix denotes attributes that are used to control projects. For example, `project.max-device-locked-memory` indicates the total amount of locked memory allowed, expressed as a number of bytes. The project.pool attributes binds a project to a resource pool. See Chapter 6, "Dynamic Resource Pools."
- `task.*` – This prefix is used for attributes that are applied to tasks. For example, the `task.max-cpu-time` attribute sets the maximum CPU time that is available to this task's processes, expressed as a number of seconds.
- `process.*` – This prefix is used for process controls. For example, the `process.max-file-size` control sets the maximum file offset that is available for writing by this process, expressed as a number of bytes.
- `zone.*` – The `zone.*` prefix is applied to projects, tasks, and processes in a zone. For example, `zone.max-lwps` prevents too many LWPs in one zone from affecting other zones. A zone's total LWPs can be further subdivided among projects within the zone within the zone by using `project.max-lwps` entries.

For the complete list of resource controls, see `resource_controls`(5).

Project and Task API Functions

The following functions are provided to assist developers in working with projects. The functions use entries that describe user projects in the `project` database.

`endprojent`(3PROJECT)	Close the project database and deallocate resources when processing is complete.
`fgetprojent`(3PROJECT)	Returns a pointer to a structure containing an entry in the project database. Rather than using `nsswitch.conf`, `fgetprojent()` reads a line from a stream.
`getdefaultproj`(3PROJECT)	Check the validity of the project keyword, look up the project, and return a pointer to the project structure if found.
`getprojbyid`(3PROJECT)	Search the `project` database for an entry with the number that specifies the project ID.

getprojbyname(3PROJECT)	Search the project database for an entry with the string that specifies project name.
getprojent(3PROJECT)	Returns a pointer to a structure containing an entry in the project database.
inproj(3PROJECT)	Check whether the specified user is permitted to use the specified project.
setproject(3PROJECT)	Add a user process to a project.
setprojent(3PROJECT)	Rewind the project database to allow repeated searches.

Code Examples for Accessing project Database Entries

EXAMPLE 2–1 Printing the First Three Fields of Each Entry in the project Database

The key points for this example include the following:

- setprojent() rewinds the project database to start at the beginning.
- getprojent() is called with a conservative maximum buffer size that is defined in project.h.
- endprojent() closes the project database and frees resources.

```
#include <project.h>

struct project projent;
char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */
    ...
struct project *pp;

setprojent();  /* Rewind the project database to start at the beginning */

while (1) {
   pp = getprojent(&projent, buffer, PROJECT_BUFSZ);
      if (pp == NULL)
          break;
    printf("%s:%d:%s\n", pp->pj_name, pp->pj_projid, pp->pj_comment);
        ...
};

endprojent();   /* Close the database and free project resources */
```

EXAMPLE 2–2 Getting a `project` Database Entry That Matches the Caller's Project ID

The following example calls `getprojbyid()` to get a project database entry that matches the caller's project ID. The example then prints the project name and the project ID.

```
#include <project.h>

struct project *pj;
char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */

main()
{
   projid_t pjid;
   pjid = getprojid();
   pj = getprojbyid(pjid, &projent, buffer, PROJECT_BUFSZ);
   if (pj == NULL) {
       /* fail; */
   }
   printf("My project (name, id) is (%s, %d)\n", pp->pj_name, pp->pj_projid);
}
```

Programming Issues Associated With Projects and Tasks

Consider the following issues when writing your application:

- No function exists to explicitly create a new project.
- A user cannot log in if no default project for the user exists in the `project` database.
- A new task in the user's default project is created when the user logs in.
- Process association with a new project applies the new project's resource controls and pools membership to the process.
- `setproject()` requires privilege. The `newtask` command does not require privilege if you own the process. Either can be used to create a task, but only `newtask` can change the project of a running process.
- No parent/child relationship exists between tasks.
- Finalized tasks can be created by using `newtask -F` or by using `setproject()` to associate the caller with a new project. Finalized tasks are useful when trying to accurately estimate aggregate resource accounting.
- The reentrant functions, `getprojent()`, `getprojbyname()`, `getprojbyid()`, `getdefaultproj()`, and `inproj()`, use buffers supplied by the caller to store returned results. These functions are safe for use in both single-threaded applications and multithreaded applications.

- Reentrant functions require these additional arguments: proj, buffer, and bufsize. The proj argument must be a pointer to a project structure allocated by the caller. On successful completion, these functions return the project entry in this structure. Storage referenced by the project structure is allocated from the memory specified by the buffer argument. bufsize specifies the size in number of bytes.
- If an incorrect buffer size is used, getprojent() returns NULL and sets errno to ERANGE.

CHAPTER 3

Using the C Interface to Extended Accounting

This chapter describes the C interface to extended accounting and covers the following topics:

- "Overview of the C Interface to Extended Accounting" on page 21
- "Extended Accounting API Functions" on page 21
- "C Code Examples for Accessing `exacct` Files" on page 23

Overview of the C Interface to Extended Accounting

Projects and tasks are used to label and separate workloads. The extended accounting subsystem is used to monitor resource consumption by workloads that are running on the system. Extended accounting produces accounting records for the workload tasks and processes.

For an overview of extended accounting and example procedures for administering extended accounting, see Chapter 4, "Extended Accounting (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* and Chapter 5, "Administering Extended Accounting (Tasks)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Extended Accounting API Functions

The extended accounting API contains functions that perform the following:

- `exacct` system calls
- Operations on the `exacct` file
- Operations on `exacct` objects
- Miscellaneous Operations

exacct System Calls

The following table lists the system calls that interact with the extended accounting subsystem.

TABLE 3–1 Extended Accounting System Calls

Function	Description
putacct(2)	Provides privileged processes with the ability to tag accounting records with additional data that is specific to the process
getacct(2)	Enables privileged processes to request extended accounting buffers from the kernel for currently executing tasks and processes
wracct(2)	Requests the kernel to write resource usage data for a specified task or process

Operations on the exacct File

These functions provide access to the exacct files:

TABLE 3–2 exacct File Functions

Function	Description
ea_open(3EXACCT)	Opens an exacct file.
ea_close(3EXACCT)	Closes an exacct file.
ea_get_object(3EXACCT)	First time use on a group of objects reads data into an ea_object_t structure. Subsequent use on the group cycles through the objects in the group.
ea_write_object(3EXACCT)	Appends the specified object to the open exacct file.
ea_next_object(3EXACCT)	Reads the basic fields (eo_catalog and eo_type) into an ea_object_t structure and rewinds to the head of the record.
ea_previous_object(3EXACCT)	Skips back one object in the exacct file and reads the basic fields (eo_catalog and eo_type) into an ea_object_t.
ea_get_hostname(3EXACCT)	Gets the name of the host on which the exacct file was created.
ea_get_creator(3EXACCT)	Determines the creator of the exacct file.

Operations on exacct Objects

These functions are used to access exacct objects:

TABLE 3–3 exacct Object Functions

Function	Description
ea_set_item(3EXACCT)	Assigns an exacct object and sets the value(s).
ea_set_group(3EXACCT)	Sets the values of a group of exacct objects.
ea_match_object_catalog(3EXACCT)	Checks an exacct object's mask to see if that object has a specific catalog tag.
ea_attach_to_object(3EXACCT)	Attaches an exacct object to a specified exacct object.
ea_attach_to_group(3EXACCT)	Attaches a chain of exacct objects as member items of a specified group.
ea_free_item(3EXACCT)	Frees the value fields in the specified exacct object.
ea_free_object(3EXACCT)	Frees the specified exacct object and any attached hierarchies of objects.

Miscellaneous Operations

These functions are associated with miscellaneous operations:

ea_error(3EXACCT)
ea_match_object_catalog(3EXACCT)

C Code Examples for Accessing exacct Files

This section provides code examples for accessing exacct files.

EXAMPLE 3–1 Displaying exacct Data for a Designated pid

This example displays a specific pid's exacct data snapshot from the kernel.

```
...
  ea_object_t *scratch;
  int unpk_flag = EUP_ALLOC;  /* use the same allocation flag */
                              /* for unpack and free */

  /* Omit return value checking, to keep code samples short */

  bsize = getacct(P_PID, pid, NULL, 0);
  buf = malloc(bsize);

  /* Retrieve exacct object and unpack */
```

EXAMPLE 3–1 Displaying exacct Data for a Designated pid *(Continued)*

```
  getacct(P_PID, pid, buf, bsize);
  ea_unpack_object(&scratch, unpk_flag, buf, bsize);

  /* Display the exacct record */
  disp_obj(scratch);
  if (scratch->eo_type == EO_GROUP) {
         disp_group(scratch);
  }
  ea_free_object(scratch, unpk_flag);
        ...
```

EXAMPLE 3–2 Identifying Individual Tasks During a Kernel Build

This example evaluates kernel builds and displays a string that describes the portion of the source tree being built by this task make. Display the portion of the source being built to aid in the per-source-directory analysis.

The key points for this example include the following:

- To aggregate the time for a make, which could include many processes, each make is initiated as a task. Child make processes are created as different tasks. To aggregate across the makefile tree, the parent-child task relationship must be identified.
- Add a tag with this information to the task's exacct file. Add a current working directory string that describes the portion of the source tree being built by this task make.

  ```
  ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD,
                                  cwdbuf, strlen(cwdbuf));
  ```

```
  ...
/* Omit return value checking and error processing */
/* to keep code sample short */
ptid = gettaskid();    /* Save "parent" task-id */
tid = settaskid(getprojid(), TASK_NORMAL);    /* Create new task */

/* Set data for item objects ptskid and cwd */
ea_set_item(&ptskid, EXT_UINT32 | EXC_LOCAL | MY_PTID, &ptid, 0);
ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD, cwdbuf, strlen(cwdbuf));

/* Set grp object and attach ptskid and cwd to grp */
ea_set_group(&grp, EXT_GROUP | EXC_LOCAL |  EXD_GROUP_HEADER);
ea_attach_to_group(&grp, &ptskid);
ea_attach_to_group(&grp, &cwd);

/* Pack the object and put it back into the accounting stream */
ea_buflen = ea_pack_object(&grp, ea_buf, sizeof(ea_buf));
```

EXAMPLE 3–2 Identifying Individual Tasks During a Kernel Build *(Continued)*

```
putacct(P_TASKID, tid, ea_buf, ea_buflen, EP_EXACCT_OBJECT);

/* Memory management: free memory allocate in ea_set_item */
ea_free_item(&cwd, EUP_ALLOC);
  ...
```

EXAMPLE 3–3 Reading and Displaying the Contents of a System `exacct` File

This example shows how to read and display a system `exacct` file for a process or a task.

The key points for this example include the following:

- Call `ea_get_object()` to get the next object in the file. Call `ea_get_object()` in a loop until EOF enables a complete traversal of the `exacct` file.
- `catalog_name()` uses the `catalog_item` structure to convert a Solaris catalog's type ID to a meaningful string that describes the content of the object's data. The type ID is obtained by masking the lowest 24 bits, or 3 bytes.

  ```
  switch(o->eo_catalog & EXT_TYPE_MASK) {
    case EXT_UINT8:
        printf(" 8: %u", o->eo_item.ei_uint8);
        break;
    case EXT_UINT16:
    ...
  }
  ```
- The upper 4 bits of `TYPE_MASK` are used to find out the data type to print the object's actual data.
- `disp_group()` takes a pointer to a group object and the number of objects in the group. For each object in the group, `disp_group()` calls `disp_obj()` and recursively calls `disp_group()` if the object is a group object.

```
/* Omit return value checking and error processing */
/* to keep code sample short */
main(int argc, char *argv)
{
  ea_file_t ef;
  ea_object_t scratch;
  char *fname;

  fname = argv[1];
  ea_open(&ef, fname, NULL,  EO_NO_VALID_HDR, O_RDONLY, 0);
  bzero(&scratch, sizeof (ea_object_t));
  while (ea_get_object(&ef, &scratch)  != -1) {
         disp_obj(&scratch);
```

EXAMPLE 3–3 Reading and Displaying the Contents of a System exacct File *(Continued)*

```
        if (scratch.eo_type == EO_GROUP)
            disp_group(&ef, scratch.eo_group.eg_nobjs);
        bzero(&scratch, sizeof (ea_object_t));
  }
  ea_close(&ef);
}

struct catalog_item {   /* convert Solaris catalog's type ID */
                        /* to a meaningful string */
     int    type;
     char *name;
 } catalog[] = {
     { EXD_VERSION,     "version\t" },
     ...
     { EXD_PROC_PID,    "  pid\t" },
     ...
   };

 static char *
 catalog_name(int type)
 {
     int i = 0;
     while (catalog[i].type != EXD_NONE) {
         if (catalog[i].type == type)
             return (catalog[i].name);
         else
             i++;
     }
     return ("unknown\t");
 }

 static void disp_obj(ea_object_t *o)
 {
     printf("%s\t", catalog_name(o->eo_catalog & 0xffffff));
     switch(o->eo_catalog & EXT_TYPE_MASK) {
     case EXT_UINT8:
         printf(" 8: %u", o->eo_item.ei_uint8);
         break;
     case EXT_UINT16:
     ...
 }
 static void disp_group(ea_file_t *ef, uint_t nobjs)
 {
     for (i = 0; i < nobjs; i++) {
         ea_get_object(ef, &scratch));
                  disp_obj(&scratch);
```

EXAMPLE 3–3 Reading and Displaying the Contents of a System exacct File *(Continued)*

```
                if (scratch.eo_type == EO_GROUP)
                        disp_group(ef, scratch.eo_group.eg_nobjs);
        }
}
```

CHAPTER 4

Using the Perl Interface to Extended Accounting

The Perl interface provides a Perl binding to the extended accounting tasks and projects. The interface allows the accounting files produced by the `exacct` framework to be read by Perl scripts. The interface also allows the writing of `exacct` files by Perl scripts.

This chapter includes the following topics:

- "Extended Accounting Overview" on page 29
- "Perl Code Examples" on page 44
- "Output From `dump` Method" on page 47

Extended Accounting Overview

The `exacct` is a new accounting framework for the Solaris operating system provides additional functionality to that provided by the traditional SVR4 accounting mechanism. Traditional SVR4 accounting has these drawbacks:

- The data collected by SVR4 accounting cannot be modified.

  The type or quantity of statistics SVR4 accounting gathers cannot be customized for each application. Changes to the data SVR4 accounting collects would not work with all of the existing applications that use the accounting files.
- The SVR4 accounting mechanism is not open.

  Applications cannot embed their own data in the system accounting data stream.
- The SVR4 accounting mechanism has no aggregation facilities.

  The Solaris Operating system writes an individual record for each process that exists. No facilities are provided for grouping sets of accounting records into higher-level aggregates.

The `exacct` framework addresses the limitations of SVR4 accounting and provides a configurable, open, and extensible framework for the collection of accounting data.

- The data that is collected can be configured using the `exacct` API.

- Applications can either embed their own data inside the system accounting files, or create and manipulate their own custom accounting files.
- The lack of data aggregation facilities in the traditional accounting mechanism are addressed by *tasks* and *projects*. Tasks identify a set of processes that are to be considered as a unit of work. Projects allow the processes executed by a set of users to be aggregated into a higher-level entity. See the`project`(4) man page for more details about tasks and projects.

For a more extensive overview of extended accounting, see Chapter 4, "Extended Accounting (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Perl Interface to `libexacct`

Object Model

The `Sun::Solaris::Exacct` module is the parent of all the classes provided by `libexacct`(3LIB) library. `libexacct`(3LIB) provides operations on types of entities: `exacct` format files, `catalog` tags and `exacct` objects. `exacct` objects are subdivided into two types.

- Items

  Single data values
- Groups

  Lists of items

Benefits of Using the Perl Interface to `libexacct`

The Perl extensions to extended accounting provide a Perl interface to the underlying `libexacct`(3LIB) API and offer the following enhancements.

- Full equivalence to C API provide a Perl interface that is functionally equivalent to the underlying C API.

  The interface provides a mechanism for accessing `exacct` files that does not require C coding. All the functionality that is available from C is also available by using the Perl interface.
- Ease of use.

  Data obtained from the underlying C API is presented as Perl data types. Perl data types ease access to the data and remove the need for buffer pack and unpack operations.
- Automated memory management.

The C API requires that the programmer take responsibility for managing memory when accessing exacct files. Memory management takes the form of passing the appropriate flags to functions, such as ea_unpack_object(3EXACCT), and explicitly allocating buffers to pass to the API. The Perl API removes these requirements, as all memory management is performed by the Perl library.

- Prevent incorrect use of API.

  The ea_object_t structure provides the in-memory representation of exacct records. The ea_object_t structure is a union type that is used for manipulating both Group and Item records. As a result, an incorrectly typed structure can be passed to some of the API functions. The addition of a class hierarchy prevents this type of programming error.

Perl Double-Typed Scalars

The modules described in this document make extensive use of the Perl double-typed scalar facility. The *double-typed scalar* facility allows a scalar value to behave either as an integer or as a string, depending upon the context. This behavior is the same as exhibited by the $! Perl variable (errno). The double-typed scalar facility avoids the need to map from an integer value into the corresponding string in order to display a value. The following example illustrates the use of double-typed scalars.

```
# Assume $obj is a Sun::Solaris::Item
my $type = $obj->type();

# prints out "2 EO_ITEM"
printf("%d %s\n", $type, $type);

# Behaves as an integer, $i == 2
my $i = 0 + $type;

# Behaves as a string, $s = "abc EO_ITEM xyx"
my $s = "abc $type xyz";
```

Perl Modules

The various project, task and exacct-related functions have been separated into groups, and each group is placed in a separate Perl module. Each function has the Sun Microsystems standard Sun::Solaris:: Perl package prefix.

TABLE 4–1 Perl Modules

Module	Description
"Sun::Solaris::Project Module" on page 33	Provides functions to access the project manipulation functions: getprojid(2), setproject(3PROJECT), project_walk(3PROJECT), getprojent(3PROJECT), getprojbyname(3PROJECT), getprojbyid(3PROJECT), getdefaultproj(3PROJECT), inproj(3PROJECT), getprojidbyname(3PROJECT), setprojent(3PROJECT), endprojent(3PROJECT), fgetprojent(3PROJECT).
"Sun::Solaris::Task Module" on page 34	Provides functions to access the task manipulation functions settaskid(2) and gettaskid(2).
"Sun::Solaris::Exacct Module" on page 35	Top-level exacct module. Functions in this module access both the exacct-related system calls getacct(2), putacct(2), and wracct(2) as well as the libexacct(3LIB) library function ea_error(3EXACCT). This module contains constants for all the various exacct EO_*, EW_*, EXR_*, P_* and TASK_* macros.
"Sun::Solaris::Exacct::Catalog Module" on page 37	Provides object-oriented methods to access the bitfields within an exacct catalog tag as well as the EXC_*, EXD_* and EXD_* macros.
"Sun::Solaris::Exacct::File Module" on page 38	Provides object-oriented methods to access the libexacct(3LIB) accounting file functions: ea_open(3EXACCT), ea_close(3EXACCT), ea_get_creator(3EXACCT), ea_get_hostname(3EXACCT), ea_next_object(3XACCT), ea_previous_object(3EXACCT), ea_write_object(3EXACCT).

TABLE 4–1 Perl Modules *(Continued)*

Module	Description
"Sun::Solaris::Exacct::Object Module" on page 40	Provides object-oriented methods to access the individual exacct accounting file object. An exacct object is represented as an opaque reference that is blessed into the appropriate Sun::Solaris::Exacct::Object subclass. This module is further subdivided into the two types of possible object: Item and Group. Methods are also provided to access the ea_match_object_catalog(3EXACCT), ea_attach_to_object(3EXACCT) functions.
"Sun::Solaris::Exacct::Object::Item Module" on page 41	Provides object-oriented methods to access an individual exacct accounting file Item. Objects of this type inherit from Sun::Solaris::Exacct::Object.
"Sun::Solaris::Exacct::Object::Group Module" on page 42	Provides object-oriented methods to access an individual exacct accounting file Group. Objects of this type inherit from Sun::Solaris::Exacct::Object, and provide access to the ea_attach_to_group(3EXACCT) function. The Items contained within the Group are presented as a perl array.
"Sun::Solaris::Exacct::Object::_Array Module" on page 43	Private array type, used as the type of the array within a Sun::Solaris::Exacct::Object::Group.

Sun::Solaris::Project **Module**

The Sun::Solaris::Project module provides wrappers for the project-related system calls and the libproject(3LIB) library.

Sun::Solaris::Project **Constants**

The Sun::Solaris::Project module uses constants from the project-related header files.

```
MAXPROJID
PROJNAME_MAX
PROJF_PATH
PROJECT_BUFSZ
SETPROJ_ERR_TASK
SETPROJ_ERR_POOL
```

Sun::Solaris::Project Functions, Class Methods, and Object Methods

The perl extensions to the libexacct(3LIB) API provide the following functions for projects.

setproject(3PROJECT)
setprojent(3PROJECT)
getdefaultproj(3PROJECT)
inproj(3PROJECT)
getprojent(3PROJECT)
fgetprojent(3PROJECT)
getprojbyname(3PROJECT)
getprojbyid(3PROJECT)
getprojbyname(3PROJECT)
endprojent(3PROJECT)

The Sun::Solaris::Project module has no class methods.

The Sun::Solaris::Project module has no object methods.

Sun::Solaris::Project Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module.

Tag	Constant or Function
:SYSCALLS	getprojid()
:LIBCALLS	setproject(), activeprojects(), getprojent(), setprojent(), endprojent(), getprojbyname(), getprojbyid(), getdefaultproj(), fgetprojent(), inproj(), getprojidbyname()
:CONSTANTS	MAXPROJID_TASK, PROJNAME_MAX, PROJF_PATH, PROJECT_BUFSZ, SETPROJ_ERR, SETPROJ_ERR_POOL
:ALL	:SYSCALLS, :LIBCALLS, :CONSTANTS

Sun::Solaris::Task Module

The Sun::Solaris::Task module provides wrappers for the settaskid(2) and gettaskid(2) system calls.

Sun::Solaris::Task Constants

The Sun::Solaris::Task module uses the following constants.

```
TASK_NORMAL
TASK_FINAL
```

Sun::Solaris::Task Functions, Class Methods, and Object Methods

The perl extensions to the libexacct(3LIB) API provides the following functions for tasks.

```
settaskid(2)
gettaskid(2)
```

The Sun::Solaris::Task module has no class methods.

The Sun::Solaris::Task module has no object methods.

Sun::Solaris::Task Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module.

Tag	Constant or Function
:SYSCALLS	settaskid(), gettaskid()
:CONSTANTS	TASK_NORMAL and TASK_FINAL
:ALL	:SYSCALLS and :CONSTANTS

Sun::Solaris::Exacct Module

The Sun::Solaris::Exacct module provides wrappers for the ea_error(3EXACCT) function, and for all the exacct system calls.

Sun::Solaris::Exacct Constants

The Sun::Solaris::Exacct module provides constants from the various exacct header files. The P_PID, P_TASKID, P_PROJID and all the EW_*, EP_*, EXR_* macros are extracted during the module build process. The macros are extracted from the exacct header files under /usr/include and provided as Perl constants. Constants passed to the Sun::Solaris::Exacct functions can either be an integer value such as. EW_FINAL or a string representation of the same variable such as. "EW_FINAL".

Sun::Solaris::Exacct Functions, Class Methods, and Object Methods

The perl extensions to the libexacct(3LIB) API provide the following functions for the Sun::Solaris::Exacct module.

getacct(2)
putacct(2)
wracct(2)
ea_error(3EXACCT)
ea_error_str
ea_register_catalog
ea_new_file
ea_new_item
ea_new_group
ea_dump_object

Note – ea_error_str() is provided as a convenience, so that repeated blocks of code like the following can be avoided:

```
if (ea_error() == EXR_SYSCALL_FAIL) {
        print("error: $!\n");
} else {
        print("error: ", ea_error(), "\n");
}
```

The Sun::Solaris::Exacct module has no class methods.

The Sun::Solaris::Exacct module has no object methods.

Sun::Solaris::Exacct Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module.

Tag	Constant or Function
:SYSCALLS	getacct(), putacct(), wracct()
:LIBCALLS	ea_error(), ea_error_str()
:CONSTANTS	P_PID, P_TASKID, P_PROJID
	, EW_*, EP_*, EXR_*

Tag	Constant or Function
:SHORTAND	ea_register_catalog(), ea_new_catalog(), ea_new_file(), ea_new_item(), ea_new_group(), ea_dump_object()
:ALL	:SYSCALLS, :LIBCALLS, :CONSTANTS and :SHORTHAND
:EXACCT_CONSTANTS	:CONSTANTS, plus the :CONSTANTS tags for Sun::Solaris::Catalog, Sun::Solaris::File, Sun::Solaris::Object
:EXACCT_ALL	:ALL, plus the :ALL tags for Sun::Solaris::Catalog, Sun::Solaris::File, Sun::Solaris::Object

Sun::Solaris::Exacct::Catalog Module

The Sun::Solaris::Exacct::Catalog module provides a wrapper around the 32-bit integer used as a catalog tag. The catalog tag is represented as a Perl object blessed into the Sun::Solaris::Exacct::Catalog class. Methods can be used to manipulate fields in a catalog tag.

Sun::Solaris::Exacct::Catalog Constants

All the EXT_*, EXC_* and EXD_* macros are extracted during the module build process from the /usr/include/sys/exact_catalog.h file and are provided as constants. Constants passed to the Sun::Solaris::Exacct::Catalog methods can either be an integer value, such as EXT_UINT8, or the string representation of the same variable, such as "EXT_UINT8".

Sun::Solaris::Exacct::Catalog Functions, Class Methods, and Object Methods

The Perl extensions to the libexacct(3LIB) API provide the following class methods forSun::Solaris::Exacct::Catalog. Exacct(3PERL) andExacct::Catalog(3PERL)

```
register
new
```

The Perl extensions to the libexacct(3LIB) API provide the following object methods for Sun::Solaris::Exacct::Catalog.

```
value
type
catalog
id
type_str
catalog_str
```

id_str

Sun::Solaris::Exacct::Catalog Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module.

Tag	Constant or Function
:CONSTANTS	EXT_*, EXC_* and EXD_*.
:ALL	:CONSTANTS

Additionally, any constants defined with the register() function can optionally be exported into the caller's package.

Sun::Solaris::Exacct::File Module

The Sun::Solaris::Exacct::File module provides wrappers for the exacct functions that manipulate accounting files. The interface is object-oriented, and allows the creation and reading of exacct files. The C library calls that are wrapped by this module are:

ea_open(3EXACCT)
ea_close(3EXACCT)
ea_next_object(3EXACCT)
ea_previous_object(3EXACCT)
ea_write_object(3EXACCT)
ea_get_object(3EXACCT)
ea_get_creator(3EXACCT)
ea_get_hostname(3EXACCT)

The file read and write methods operate on Sun::Solaris::Exacct::Object objects. These methods perform all the necessary memory management, packing, unpacking and structure conversions that are required.

Sun::Solaris::Exacct::File Constants

Sun::Solaris::Exacct::File provides the EO_HEAD, EO_TAIL, EO_NO_VALID_HDR, EO_POSN_MSK and EO_VALIDATE_MSK constants. Other constants that are needed by the new() method are in the standard Perl Fcntl module. Table 4–2 describes the action of new() for various values of $oflags and $aflags.

Sun::Solaris::Exacct::File Functions, Class Methods, and Object Methods

The Sun::Solaris::Exacct::File module has no functions.

The Perl extensions to the libexacct(3LIB) API provide the following class method forSun::Solaris::Exacct::File.

```
new
```

The following table describes the new() action for combinations of the $oflags and $aflags parameters.

TABLE 4–2 $oflags and $aflags Parameters

$oflags	$aflags	Action
O_RDONLY	Absent or EO_HEAD	Open for reading at the start of the file.
O_RDONLY	EO_TAIL	Open for reading at the end of the file.
O_WRONLY	Ignored	File must exist, open for writing at the end of the file.
O_WRONLY | O_CREAT	Ignored	Create file if the file does not exist. Otherwise, truncate, and open for writing.
O_RDWR	Ignored	File must exist, open for reading or writing, at the end of the file.
O_RDWR | O_CREAT	Ignored	Create file if the file does not exist. Otherwise, truncate, and open for reading or writing.

Note – The only valid values for $oflags are the combinations of O_RDONLY, O_WRONLY, O_RDWR or O_CREAT. $aflags describes the required positioning in the file for O_RDONLY. Either EO_HEAD or EO_TAIL are allowed. If absent, EO_HEAD is assumed.

The perl extensions to the libexacct(3LIB) API provide the following object methods forSun::Solaris::Exacct::File.

```
creator
hostname
next
previous
get
write
```

Note – Close a `Sun::Solaris::Exacct::File`. There is no explicit `close()` method for a `Sun::Solaris::Exacct::File`. The file is closed when the filehandle object is undefined or reassigned.

`Sun::Solaris::Exacct::File` Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants that are defined in this module.

Tag	Constant or Function
`:CONSTANTS`	`EO_HEAD`, `EO_TAIL`, `EO_NO_VALID_HDR`, `EO_POSN_MSK`, `EO_VALIDATE_MSK`.
`:ALL`	`:CONSTANTS` and `Fcntl(:DEFAULT)`.

`Sun::Solaris::Exacct::Object` Module

The `Sun::Solaris::Exacct::Object` module serves as a parent of the two possible types of `exacct` objects: Items and Groups. An `exacct Item` is a single data value, an embedded `exacct` object, or a block of raw data. An example of a single data value is the number of seconds of user CPU time consumed by a process. An `exacct Group` is an ordered collection of `exacct` Items such as all of the resource usage values for a particular process or task. If Groups need to be nested within each other, the inner Groups can be stored as embedded `exacct` objects inside the enclosing Group.

The `Sun::Solaris::Exacct::Object` module contains methods that are common to both `exacct` Items and Groups. Note that the attributes of `Sun::Solaris::Exacct::Object` and all classes derived from it are read-only after initial creation via `new()`. The attributes made read-only prevents the inadvertent modification of the attributes which could give rise to inconsistent catalog tags and data values. The only exception to the read-only attributes is the array used to store the Items inside a Group object. This array can be modified using the normal perl array operators.

`Sun::Solaris::Exacct::Object` Constants

`Sun::Solaris::Exacct::Object` provides the `EO_ERROR`, `EO_NONE`, `EO_ITEM` and `EO_GROUP` constants.

`Sun::Solaris::Exacct::Object` Functions, Class Methods, and Object Methods

The `Sun::Solaris::Exacct::Object` module has no functions.

The Perl extensions to the libexacct(3LIB) API provide the following class method forSun::Solaris::Exacct::Object.

dump

The Perl extensions to the libexacct(3LIB) API provide the following object methods forSun::Solaris::Exacct::Object.

type
catalog
match_catalog
value

Sun::Solaris::Exacct::Object Exports

By default, nothing is exported from this module. The following tags can be used to selectively import constants and functions defined in this module.

Tag	Constant or Function
:CONSTANTS	EO_ERROR, EO_NONE, EO_ITEM and EO_GROUP
:ALL	:CONSTANTS

Sun::Solaris::Exacct::Object::Item Module

The Sun::Solaris::Exacct::Object::Item module is used for exacct data Items. An exacct data Item is represented as an opaque reference, blessed into the Sun::Solaris::Exacct::Object::Item class, which is a subclass of the Sun::Solaris::Exacct::Object class. The underlying exacct data types are mapped onto Perl types as follows.

TABLE 4–3 exacct Data Types Mapped to Perl Data Types

exacct type	Perl internal type
EXT_UINT8	IV (integer)
EXT_UINT16	IV (integer)
EXT_UINT32	IV (integer)
EXT_UINT64	IV (integer)

TABLE 4–3 `exacct` Data Types Mapped to Perl Data Types *(Continued)*

`exacct` type	Perl internal type
`EXT_DOUBLE`	NV (double)
`EXT_STRING`	PV (string)
`EXT_EXACCT_OBJECT`	`Sun::Solaris::Exacct::Object` subclass
`EXT_RAW`	PV (string)

`Sun::Solaris::Exacct::Object::Item` Constants

`Sun::Solaris::Exacct::Object::Item` has no constants.

`Sun::Solaris::Exacct::Object::Item` Functions, Class Methods, and Object Methods

`Sun::Solaris::Exacct::Object::Item` has no functions.

`Sun::Solaris::Exacct::Object::Item` inherits all class methods from the `Sun::Solaris::Exacct::Object` base class, plus the `new()` class method.

`new`

`Sun::Solaris::Exacct::Object::Item` inherits all object methods from the `Sun::Solaris::Exacct::Object` base class.

`Sun::Solaris::Exacct::Object::Item` Exports

`Sun::Solaris::Exacct::Object::Item` has no exports.

`Sun::Solaris::Exacct::Object::Group` Module

The `Sun::Solaris::Exacct::Object::Group` module is used for `exacct` Group objects. An `exacct` Group object is represented as an opaque reference, blessed into the `Sun::Solaris::Exacct::Object::Group` class, which is a subclass of the `Sun::Solaris::Exacct::Object` class. The Items within a Group are stored inside a Perl array, and a reference to the array can be accessed via the inherited `value()` method. This means that the individual Items within a Group can be manipulated with the normal Perl array syntax and operators. All data elements of the array must be derived from the `Sun::Solaris::Exacct::Object` class. Group objects can also be nested inside each other merely by adding an existing Group as a data Item.

`Sun::Solaris::Exacct::Object::Group` Constants

`Sun::Solaris::Exacct::Object::Group` has no constants.

Sun::Solaris::Exacct::Object::Group Functions, Class Methods, and Object Methods

Sun::Solaris::Exacct::Object::Group has no functions.

Sun::Solaris::Exacct::Object::Group inherits all class methods from the Sun::Solaris::Exacct::Object base class, plus the new() class method.

new

Sun::Solaris::Exacct::Object::Group inherits all object methods from the Sun::Solaris::Exacct::Object base class, plus the new() class method.

as_hash
as_hashlist

Sun::Solaris::Exacct::Object::Group Exports

Sun::Solaris::Exacct::Object::Group has no exports.

Sun::Solaris::Exacct::Object::_Array Module

The Sun::Solaris::Exacct::Object::_Array class is used internally for enforcing type checking of the data Items that are placed in an exacct Group. Sun::Solaris::Exacct::Object::_Array should not be created directly by the user.

Sun::Solaris::Exacct::Object::_Array Constants

Sun::Solaris::Exacct::Object::_Array has no constants.

Sun::Solaris::Exacct::Object::_Array Functions, Class Methods, and Object Methods

Sun::Solaris::Exacct::Object::_Array has no functions.

Sun::Solaris::Exacct::Object::_Array has internal-use class methods.

Sun::Solaris::Exacct::Object::_Array uses perl TIEARRAY methods.

Sun::Solaris::Exacct::Object::_Array Exports

Sun::Solaris::Exacct::Object::_Array has no exports.

Perl Code Examples

This section shows perl code examples for accessing exacct files.

EXAMPLE 4–1 Using the Pseudocode Prototype

In typical use the Perl exacct library reads existing exacct files. Use pseudocode to show the relationships of the various Perl exacct classes. Illustrate in pseudocode the process of opening and scanning an exacct file, and processing objects of interest. In the following pseudocode, the 'convenience' functions are used in the interest of clarity.

```
-- Open the exacct file ($f is a Sun::Solaris::Exacct::File)
my $f = ea_new_file(...)

-- While not EOF ($o is a Sun::Solaris::Exacct::Object)
while (my $o = $f->get())

        -- Check to see if object is of interest
        if ($o->type() == &EO_ITEM)
             ...

        -- Retrieve the catalog ($c is a Sun::Solaris::Exacct::Catalog)
        $c = $o->catalog()

        -- Retrieve the value
        $v = $o->value();

        -- $v is a reference to a Sun::Solaris::Exacct::Group for a Group
        if (ref($v))
              ....

        -- $v is perl scalar for Items
        else
```

EXAMPLE 4–2 Recursively dumping an exacct Object

```
sub dump_object
{
    my ($obj, $indent) = @_;
    my $istr = '  ' x $indent;

    #
    # Retrieve the catalog tag.  Because we are doing this in an array
    # context, the catalog tag will be returned as a (type, catalog, id)
    # triplet, where each member of the triplet will behave as an integer
    # or a string, depending on context.  If instead this next line provided
    # a scalar context, e.g.
```

EXAMPLE 4–2 Recursively dumping an exacct Object *(Continued)*

```
#    my $cat  = $obj->catalog()->value();
# then $cat would be set to the integer value of the catalog tag.
#
my @cat = $obj->catalog()->value();

#
# If the object is a plain item
#
if ($obj->type() == &EO_ITEM) {
      #
      # Note:  The '%s' formats provide s string context, so the
      # components of the catalog tag will be displayed as the
      # symbolic values.  If we changed the '%s' formats to '%d',
      # the numeric value of the components would be displayed.
      #
      printf("%sITEM\n%s  Catalog = %s|%s|%s\n",
          $istr, $istr, @cat);
      $indent++;

      #
      # Retrieve the value of the item.  If the item contains in
      # turn a nested exacct object (i.e. a item or group), then
      # the value method will return a reference to the appropriate
      # sort of perl object (Exacct::Object::Item or
      # Exacct::Object::Group). We could of course figure out that
      # the item contained a nested item or group by examining
      # the catalog tag in @cat and looking for a type of
      # EXT_EXACCT_OBJECT or EXT_GROUP.
      my $val = $obj->value();
      if (ref($val)) {
         # If it is a nested object, recurse to dump it.
         dump_object($val, $indent);
      } else {
         # Otherwise it is just a 'plain' value, so display it.
         printf("%s  Value = %s\n", $istr, $val);
      }

    #
    # Otherwise we know we are dealing with a group.  Groups represent
    # contents as a perl list or array (depending on context), so we
    # can process the contents of the group with a 'foreach' loop, which
    # provides a list context.  In a list context the value method
    # returns the content of the group as a perl list, which is the
    # quickest mechanism, but doesn't allow the group to be modified.
    # If we wanted to modify the contents of the group we could do so
    # like this:
```

EXAMPLE 4–2 Recursively dumping an exacct Object *(Continued)*

```
        #    my $grp = $obj->value();   # Returns an array reference
        #    $grp->[0] = $newitem;
        # but accessing the group elements this way is much slower.
        #
        } else {
                printf("%sGROUP\n%s  Catalog = %s|%s|%s\n",
                    $istr, $istr, @cat);
                $indent++;
                # 'foreach' provides a list context.
                foreach my $val ($obj->value()) {
                        dump_object($val, $indent);
                }
                printf("%sENDGROUP\n", $istr);
        }
}
```

EXAMPLE 4–3 Creating a New Group Record and Writing to a File

```
# Prototype list of catalog tags and values.
my @items = (
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_CREATOR      => "me"         ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_PID     => $$           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_UID     => $<           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_GID     => $(           ],
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_PROC_COMMAND => "/bin/stuff" ],
);

# Create a new group catalog object.
my $cat = new_catalog(&EXT_GROUP | &EXC_DEFAULT | &EXD_NONE);

# Create a new Group object and retrieve its data array.
my $group = new_group($cat);
my $ary = $group->value();

# Push the new Items onto the Group array.
foreach my $v (@items) {
        push(@$ary, new_item(new_catalog($v->[0]), $v->[1]));
}

# Nest the group within itself (performs a deep copy).
push(@$ary, $group);

# Dump out the group.
dump_object($group);
```

EXAMPLE 4–4 Dumping an exacct File

```
#!/usr/bin/perl

use strict;
use warnings;
use blib;
use Sun::Solaris::Exacct qw(:EXACCT_ALL);

die("Usage is dumpexacct

# Open the exact file and display the header information.
my $ef = ea_new_file($ARGV[0], &O_RDONLY) || die(error_str());
printf("Creator:  %s\n", $ef->creator());
printf("Hostname: %s\n\n", $ef->hostname());

# Dump the file contents
while (my $obj = $ef->get()) {
        ea_dump_object($obj);
}

# Report any errors
if (ea_error() != EXR_OK && ea_error() != EXR_EOF)  {
       printf("\nERROR: %s\n", ea_error_str());
       exit(1);
}
exit(0);
```

Output From dump Method

This example shows the formatted output of the `Sun::Solaris::Exacct::Object->dump()` method.

```
GROUP
  Catalog = EXT_GROUP|EXC_DEFAULT|EXD_GROUP_PROC_PARTIAL
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PID
    Value = 3
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_UID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_GID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PROJID
```

```
  Value = 0
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TASKID
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_SEC
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_NSEC
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_SEC
  Value = 890
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_NSEC
  Value = 760000000
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_SEC
  Value = 1011869897
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_NSEC
  Value = 380771911
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_SEC
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_NSEC
  Value = 0
ITEM
  Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_COMMAND
  Value = fsflush
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MAJOR
  Value = 4294967295
ITEM
  Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MINOR
  Value = 4294967295
ITEM
  Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_HOSTNAME
  Value = mower
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MAJOR
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MINOR
  Value = 0
ITEM
  Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_SND
```

```
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_RCV
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_IN
    Value = 19
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_OUT
    Value = 40833
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CHARS_RDWR
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_VOL
    Value = 129747
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_INV
    Value = 79
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SIGNALS
    Value = 0
    ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SYSCALLS
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ACCT_FLAGS
    Value = 1
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ANCPID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_WAIT_STATUS
    Value = 0
ENDGROUP
```

◆ ◆ ◆ CHAPTER 5

Resource Controls

This chapter describes resource controls and their properties.

- "Overview of Resource Controls" on page 51
- "Resource Controls Flags and Actions" on page 52
- "Resource Controls API Functions" on page 61
- "Resource Control Code Examples" on page 62
- "Programming Issues Associated With Resource Controls" on page 67

Overview of Resource Controls

Use the extended accounting facility to determine the resource consumption of workloads on your system. After the resource consumption has been determined, use the resource control facility to place bounds on resource usage. Bounds that are placed on resources prevent workloads from over-consuming resources.

For an overview of resource controls and example commands for administering resource controls, see Chapter 6, "Resource Controls (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* and Chapter 7, "Administering Resource Controls (Tasks)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

The resource control facility adds the following benefits.

- **`Dynamically set`**

  Resource controls can be adjusted while the system is running.
- **`Containment level granularity`**

  Resource controls are arranged in a containment level of project, task, or process. The containment level simplifies the configuration and aligns the collected values closer to the particular project, task, or process.
- **`Threshold preservation`**

If an attempt is made to set the maximum value less than the actual resource consumption, no change in to the maximum value is made.

Resource Controls Flags and Actions

This section describes flags, actions, and signals associated with resource controls.

`rlimit`, Resource Limit

`rlimit` is process-based. `rlimit` establishes a restricting boundary on the consumption of a variety of system resources by a process. Each process that the process creates inherits from the original process. A resource limit is defined by a pair of values. The values specify the current (soft) limit and the maximum (hard) limit.

A process might irreversibly lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with superuser ID can raise the hard limit. See `setrlimit()` and `getrlimit()`.

The `rlimit` structure contains two members that define the soft limit and hard limit.

```
rlim_t     rlim_cur;       /* current (soft) limit */
rlim_t     rlim_max        /* hard limit */
```

`rctl`, Resource Control

`rctl` extends the process-based limits of `rlimit` by controlling resource consumption by processes, tasks, and projects defined in the project database.

Note – The `rctl` mechanism is preferred to the use of `rlimit` to set resource limits. The only reason to use the `rlimit` facility is when portability is required across UNIX platforms.

Applications fall into the following broad categories depending on how an application deals with resource controls. Based on the action that is taken, resource controls can be further classified. Most report an error and terminate operation. Other resource controls allow applications to resume operation and adapt to the reduced resource usage. A progressive chain of actions at increasing values can be specified for each resource control.

The list of attributes for a resource control consists of a privilege level, a threshold value, and an action that is taken when the threshold is exceeded.

Resource Control Values and Privilege Levels

Each threshold value on a resource control must be associated with one of the following privilege levels:

RCPRIV_BASIC
: Privilege level can be modified by the owner of the calling process. RCPRIV_BASIC is associated with a resource's soft limit.

RCPRIV_PRIVILEGED
: Privilege level can be modified only by privileged (superuser) callers. RCPRIV_PRIVILEGED is associated with a resource's hard limit.

RCPRIV_SYSTEM
: Privilege level remains fixed for the duration of the operating system instance.

Figure 5–2 shows the timeline for setting privilege levels for signals that are defined by the `/etc/project` file `process.max-cpu-time` resource control.

Local Actions and Local Flags

The local action and local flags are applied to the current resource control value represented by this resource control block. Local actions and local flags are value-specific. For each threshold value that is placed on a resource control, the following local actions and local flags are available:

RCTL_LOCAL_NOACTION
: No local action is taken when this resource control value is exceeded.

RCTL_LOCAL_SIGNAL
: The specified signal, set by `rctlblk_set_local_action()`, is sent to the process that placed this resource control value in the value sequence.

RCTL_LOCAL_DENY
: When this resource control value is encountered, the request for the resource is denied. Set on all values if RCTL_GLOBAL_DENY_ALWAYS is set for this control. Cleared on all values if RCTL_GLOBAL_DENY_NEVER is set for this control.

RCTL_LOCAL_MAXIMAL
: This resource control value represents a request for the maximum amount of resource for this control. If RCTL_GLOBAL_INFINITE is set for this resource control, RCTL_LOCAL_MAXIMAL indicates an unlimited resource control value that is never exceeded.

Global Actions and Global Flags

Global flags apply to all current resource control values represented by this resource control block. Global actions and global flags are set by `rctladm`(1M). Global actions and global flags cannot be set with `setrctl()`. Global flags apply to all resource controls. For each threshold value that is placed on a resource control, the following global actions and global flags are available:

RCTL_GLOBAL_NOACTION
: No global action is taken when a resource control value is exceeded on this control.

RCTL_GLOBAL_SYSLOG
: A standard message is logged by the `syslog()` facility when any resource control value on a sequence associated with this control is exceeded.

RCTL_GLOBAL_NOBASIC
: No values with the RCPRIV_BASIC privilege are permitted on this control.

RCTL_GLOBAL_LOWERABLE
: Non-privileged callers are able to lower the value of privileged resource control values on this control.

RCTL_GLOBAL_DENY_ALWAYS
: The action that is taken when a control value is exceeded on this control always includes denial of the resource.

RCTL_GLOBAL_DENY_NEVER
: The action that is taken when a control value is exceeded on this control always excludes denial of the resource. The resource is always granted, although other actions can also be taken.

RCTL_GLOBAL_FILE_SIZE
: The valid signals for local actions include the SIGXFSZ signal.

RCTL_GLOBAL_CPU_TIME
: The valid signals for local actions include the SIGXCPU signal.

RCTL_GLOBAL_SIGNAL_NEVER
: No local actions are permitted on this control. The resource is always granted.

RCTL_GLOBAL_INFINITE
: This resource control supports the concept of an unlimited value. Generally, an unlimited value applies only to accumulation-oriented resources, such as CPU time.

RCTL_GLOBAL_UNOBSERVABLE
: Generally, a task or project related resource control does not support observational control values. An RCPRIV_BASIC privileged control value placed on a task or process generates an action only if the value is exceeded by that process.

In the Solaris 10 5/08 release, the level `n/a` was added for resource controls on which no global action can be configured.

Resource Control Sets Associated With a Project, Processes and Tasks

The following figure shows the resource control sets associated with tasks, processes and a project.

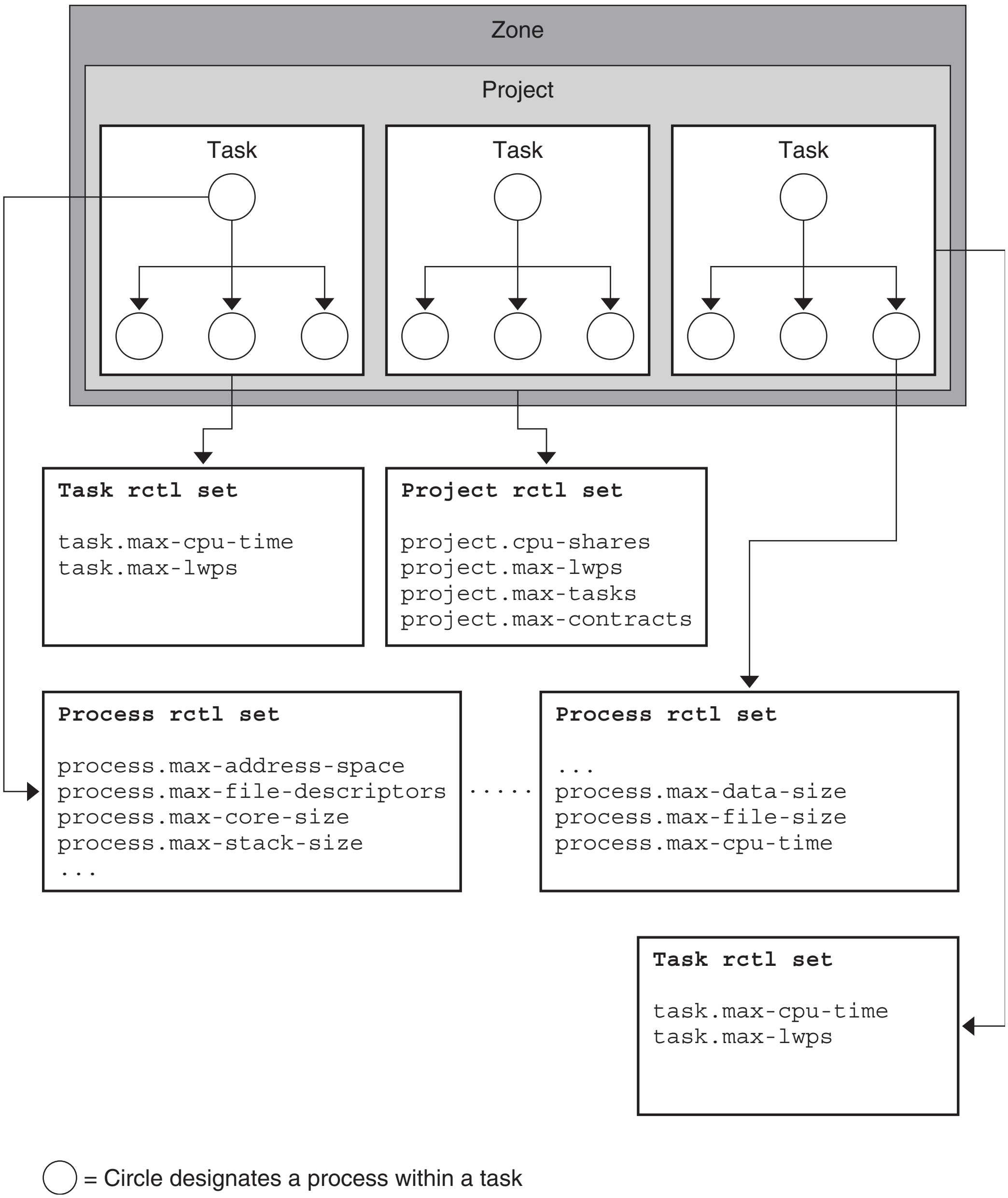


FIGURE 5–1 Resource Control Sets for Task, Project, and Process

More than one resource control can exist on a resource, each resource control at a containment level in the process model. Resource controls can be active on the same resource for both a process and collective task or collective project. In this case, the action for the process takes precedence. For example, action is taken on process.max-cpu-time before task.max-cpu-time if both controls are encountered simultaneously.

Resource Controls Associated With a Project

Resource controls associated with a project include the following:

project.cpu-cap
: In Solaris 10 8/07, the absolute limit on the amount of CPU resources that can be consumed by a project. A value of 100 means 100% of one CPU as the project.cpu-cap setting. A value of 125 is 125%, because 100% corresponds to one full CPU on the system when using CPU caps.

project.cpu-shares
: The number of CPU shares that are granted to this project for use with the fair share scheduler, FSS(7).

project.max-crypto-memory
: Total amount of kernel memory that can be used by libpkcs11 for hardware crypto acceleration. Allocations for kernel buffers and session-related structures are charged against this resource control.

project.max-locked-memory
: Total amount of physical locked memory allowed.

  Note that this resource control replaced project.max-device-locked-memory, which has been removed.

project.max-msg-ids
: Maximum number of System V message queues allowed for a project.

project.max-port-ids
: Maximum allowable number of event ports.

project.max-sem-ids
: Maximum number of System V semaphores allowed for a project.

project.max-shm-ids
: Maximum number of shared memory IDs allowed for this project.

project.max-msg-ids
: Maximum number of message queue IDs allowed for this project.

project.max-shm-memory
: Total amount of System V shared memory allowed for this project.

project.max-lwps
: Maximum number of LWPs simultaneously available to this project.

`project.max-tasks`
Maximum number of tasks allowable in this project.

`project.max-contracts`
Maximum number of contracts allowed in this project.

Resource Controls Associated With Tasks

Resource controls associated with tasks include the following:

`task.max-cpu-time`
Maximum CPU time (seconds) available to this task's processes.

`task.max-lwps`
Maximum number of LWPs simultaneously available to this task's processes.

Resource Controls Associated With Processes

Resource controls associated with processes include the following:

`process.max-address-space`
Maximum amount of address space (bytes), as summed over segment sizes, available to this process.

`process.max-core-size`
Maximum size (bytes) of a core file that is created by this process.

`process.max-cpu-time`
Maximum CPU time (seconds) available to this process.

`process.max-file-descriptor`
Maximum file descriptor index that is available to this process.

`process.max-file-size`
Maximum file offset (bytes) available for writing by this process.

`process.max-msg-messages`
Maximum number of messages on a message queue. This value is copied from the resource control at `msgget()` time.

`process.max-msg-qbytes`
Maximum number (bytes) of messages on a message queue. This value is copied from the resource control at `msgget()` time.When you set a new `project.max-msg-qbytes` value, initialization occurs only on the subsequently created values. The new `project.max-msg-qbytes` value does not effect existing values.

`process.max-sem-nsems`
Maximum number of semaphores allowed for a semaphore set.

`process.max-sem-ops`
Maximum number of semaphore operations that are allowed for a `semop()` call. This value is copied from the resource control at `msgget()` time.A new `project.max-sem-ops` value only affects the initialization of subsequently created values and has no effect on existing values.

`process.max-port-events`
Maximum number of events that are allowed per event port.

Zone-Wide Resource Controls

Zone-wide resource controls are available on a system with zones installed. Zone-wide resource controls limit the total resource usage of all process entities within a zone.

`zone.cpu-cap`	In the Solaris 10 5/08 release, sets an absolute limit on the amount of CPU resources that can be consumed by a zone. A value of `100` means 100 percent of one CPU as the `project.cpu-cap` setting. A value of `125` is 125 percent, because 100 percent corresponds to one full CPU on the system when using CPU caps.
`zone.cpu-shares`	Limit on the number of fair share scheduler (FSS) CPU shares for a zone. The scheduling class must be FSS. CPU shares are first allocated to the zone, and then further subdivided among projects within the zone as specified in the `project.cpu-shares` entries. A zone with a higher number of `zone.cpu-shares` is allowed to use more CPU than a zone with a low number of shares.
`zone.max-locked-memory`	Total amount of physical locked memory available to a zone.
`zone.max-lwps`	Maximum number of LWPs simultaneously available to this zone
`zone.max-msg-ids`	Maximum number of message queue IDs allowed for this zone
`zone.max-sem-ids`	Maximum number of semaphore IDs allowed for this zone
`zone.max-shm-ids`	Maximum number of shared memory IDs allowed for this zone
`zone.max-shm-memory`	Total amount of shared memory allowed for this zone
`zone.max-swap`	Total amount of swap that can be consumed by user process address space mappings and `tmpfs` mounts for this zone.

For information on configuring zone-wide resource controls, see Chapter 17, "Non-Global Zone Configuration (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* and Chapter 18, "Planning and Configuring Non-Global Zones (Tasks)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris*

Zones. Note that it is possible to use the `zonecfg` command to apply a zone-wide resource control to the global zone on a system with non-global zones installed.

Signals Used With Resource Controls

For each threshold value that is placed on a resource control, the following restricted set of signals is available:

SIGBART
: Terminate the process.

SIGXRES
: Signal generated by resource control facility when the resource control limit is exceeded.

SIGHUP
: When carrier drops on an open line, the process group that controls the terminal is sent a hangup signal, SIGHUP.

SIGSTOP
: Job control signal. Stop the process. Stop signal not from terminal.

SIGTERM
: Terminate the process. Termination signal sent by software.

SIGKILL
: Terminate the process. Kill the program.

SIGXFSX
: Terminate the process. File size limit exceeded. Available only to resource controls with the RCTL_GLOBAL_FILE_SIZE property.

SIGXCPU
: Terminate the process. CPU time limit exceeded. Available only to resource controls with the RCTL_GLOBAL_CPUTIME property. See `rctlblk_set_value`(3C) for more information.

Other signals might be permitted due to global properties of a specific control.

Note – Calls to `setrctl()` with illegal signals fail.

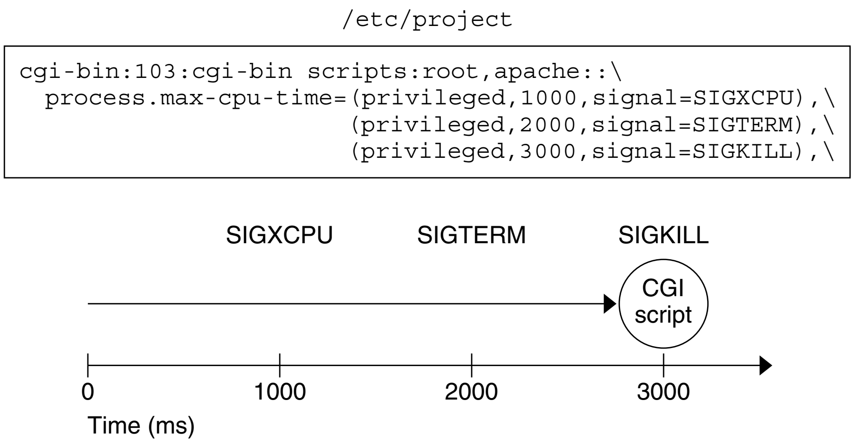


FIGURE 5–2 Setting Privilege Levels for Signals

Resource Controls API Functions

The resource controls API contains functions that:

- “Operate on Action-Value Pairs of a Resource Control” on page 61
- “Operate on Local Modifiable Values” on page 61
- “Retrieve Local Read-Only Values” on page 62
- “Retrieve Global Read-Only Actions” on page 62

Operate on Action-Value Pairs of a Resource Control

The following list contains the functions that set or get the resource control block.

`setrctl`(2)
`getrctl`(2)

Operate on Local Modifiable Values

The following list contains the functions associated with the local, modifiable resource control block.

`rctlblk_set_privilege`(3C)
`rctlblk_get_privilege`(3C)
`rctlblk_set_value`(3C)
`rctlblk_get_value`(3C)

```
rctlblk_set_local_action(3C)
rctlblk_get_local_action(3C)
rctlblk_set_local_flags(3C)
rctlblk_get_local_flags(3C)
```

Retrieve Local Read-Only Values

The following list contains the functions associated with the local, read-only resource control block.

```
rctlblk_get_recipient_pid(3C)
rctlblk_get_firing_time(3C)
rctlblk_get_enforced_value(3C)
```

Retrieve Global Read-Only Actions

The following list contains the functions associated with the global, read-only resource control block.

```
rctlblk_get_global_action(3C)
rctlblk_get_global_flags(3C)
```

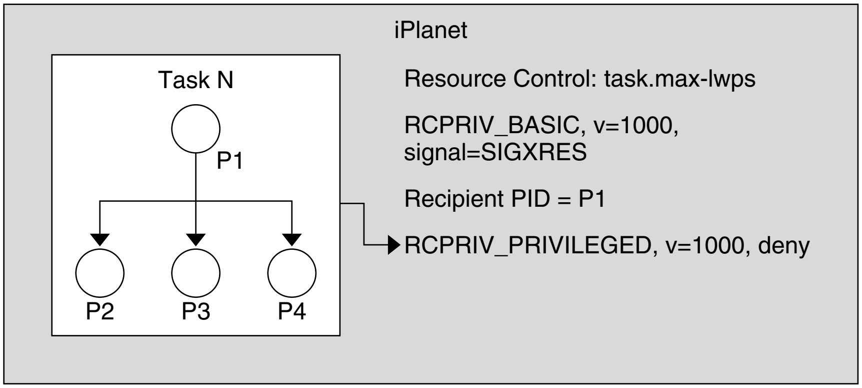
Resource Control Code Examples

Master Observing Process for Resource Controls

The following example is the master observer process. Figure 5–3 shows the resource controls for the master observing process.

Note – The line break is not valid in an `/etc/project` file. The line break is shown here only to allow the example to display on a printed or displayed page. Each entry in the `/etc/project` file must be on a separate line.

The key points for the example include the following:



/etc/project

```
...
iPlanet:200:iPlanet Application Server:root::\
            task.max=lwps=(PRIVILEGED,1000,deny)
```

FIGURE 5–3 Master Observing Process

- Because the task's limit is privileged, the application cannot change the limit, or specify an action, such as a signal. A master process solves this problem by establishing the same resource control as a basic resource control on the task. The master process uses the same value or a little less on the resource, but with a different action, signal = XRES. The master process creates a thread to wait for this signal.
- The `rctlblk` is opaque. The struct needs to be dynamically allocated.
- Note the blocking of all signals before creating the thread, as required by sigwait(2).
- The thread calls `sigwait`(2) to block for the signal. If `sigwait()` returns the SIGXRES signal, the thread notifies the master process' children, which adapts to reduce the number of LWPs being used. Each child should also be modelled similarly, with a thread in each child, waiting for this signal, and adapting its process' LWP usage appropriately.

```
rctlblk_t *mlwprcb;
sigset_t smask;

/* Omit return value checking/error processing to keep code sample short */
/* First, install a RCPRIV_BASIC, v=1000, signal=SIGXRES rctl */
mlwprcb = calloc(1, rctlblk_size());     /* rctl blocks are opaque: */
       rctlblk_set_value(mlwprcb, 1000);
       rctlblk_set_privilege(mlwprcb, RCPRIV_BASIC);
       rctlblk_set_local_action(mlwprcb, RCTL_LOCAL_SIGNAL, SIGXRES);
       if (setrctl("task.max-lwps", NULL, mlwprcb, RCTL_INSERT) == -1) {
           perror("setrctl");
           exit (1);
```

```
        }

/* Now, create the thread which waits for the signal */
        sigemptyset(&smask);
        sigaddset(&smask, SIGXRES);
        thr_sigsetmask(SIG_BLOCK, &smask, NULL);
thr_create(NULL, 0, sigthread, (void *)SIGXRES, THR_DETACHED, NULL));

/* Omit return value checking/error processing to keep code sample short */

void *sigthread(void *a)
{
        int sig = (int)a;
        int rsig;
        sigset_t sset;

        sigemptyset(&sset);
        sigaddset(&sset, sig);

        while (1) {
                 rsig = sigwait(&sset);
          if (rsig == SIGXRES) {
              notify_all_children();
              /* e.g. sigsend(P_PID, child_pid, SIGXRES); */
             }
        }
}
```

List all the Value-action Pairs for a Specific Resource Control

The following example lists all the value-action pairs for a specific resource control, `task.max-lwps`. The key point for the example is that `getrctl`(2) takes two resource control blocks, and returns the resource control block for the RCTL_NEXT flag. To iterate through all resource control blocks, repeatedly swap the resource control block values, as shown here using the `rcb_tmp` `rctl` block.

```
rctlblk_t *rcb1, *rcb2, *rcb_tmp;
    ...
/* Omit return value checking/error processing to keep code sample short */
rcb1 = calloc(1, rctlblk_size()); /* rctl blocks are opaque: */
                               /* "rctlblk_t rcb" does not work */
rcb2 = calloc(1, rctlblk_size());
getrctl("task.max-lwps", NULL, rcb1, RCTL_FIRST);
while (1) {
     print_rctl(rcb1);
```

```
            rcb_tmp = rcb2;
            rcb2 = rcb1;
            rcb1 = rcb_tmp;        /* swap rcb1 with rcb2 */
            if (getrctl("task.max-lwps", rcb2,  rcb1, RCTL_NEXT) == -1) {
                 if (errno == ENOENT) {
                      break;
            } else {
                 perror("getrctl");
                 exit (1);
            }
            }
}
```

Set `project.cpu-shares` and Add a New Value

The key points of the example include the following:

- This example is similar to the example shown in "Set `pool.comment` Property and Add New Property" on page 80.
- Use `bcopy()`, rather than buffer swapping as in "List all the Value-action Pairs for a Specific Resource Control" on page 64.
- To change the resource control value, call `setrctl()` with the RCTL_REPLACE flag. The new resource control block is identical to the old resource control block except for the new control value.

```
rctlblk_set_value(blk1, nshares);
if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0)
```

The example gets the project's CPU share allocation, `project.cpu-shares`, and changes its value to *nshares*.

```
/* Omit return value checking/error processing to keep code sample short */
blk1 = malloc(rctlblk_size());
getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST);
my_shares = rctlblk_get_value(blk1);
printout_my_shares(my_shares);
/* if privileged, do the following to */
/* change project.cpu-shares to "nshares" */
blk1 = malloc(rctlblk_size());
blk2 = malloc(rctlblk_size());
if (getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST) != 0) {
     perror("getrctl failed");
     exit(1);
}
bcopy(blk1, blk2, rctlblk_size());
rctlblk_set_value(blk1, nshares);
```

```
if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0) {
     perror("setrctl failed");
     exit(1);
}
```

Set LWP Limit on Resource Control Blocks

In the following example, our application has set a privileged limit of 3000 LWPs that may not be exceeded. In addition, our application has set a basic limit of 2000 LWPs. When this limit is exceeded, a SIGXRES is sent to the application. Upon receiving a SIGXRES, our application might send notification to its child processes that might in turn reduce the number of LWPs the processes use or need.

```
/* Omit return value and error checking */

#include <rctl.h>

rctlblk_t *rcb1, *rcb2;

/*
         * Resource control blocks are opaque
         * and must be explicitly allocated.
         */
rcb1 = calloc(rctlblk_size());

rcb2 = calloc(rctlblk_size());


/* Install an RCPRIV_PRIVILEGED, v=3000: do not allow more than 3000 LWPs */
rctlblk_set_value(rcb1, 3000);
rctlblk_set_privilege(rcb1, RCPRIV_PRIVILEGED);
rctlblk_set_local_action(rcb1, RCTL_LOCAL_DENY);
setrctl("task.max-lwps", NULL, rcb1, RCTL_INSERT);


/* Install an RCPRIV_BASIC, v=2000 to send SIGXRES when LWPs exceeds 2000 */
rctlblk_set_value(rcb2, 2000);
rctlblk_set_privilege(rcb2, RCPRIV_BASIC);
rctlblk_set_local_action(rcb2, RCTL_LOCAL_SIGNAL, SIGXRES);
setrctl("task.max-lwps", NULL, rcb2, RCTL_INSERT);
```

Programming Issues Associated With Resource Controls

Consider the following issues when writing your application:

- The resource control block is opaque. The control block needs to be dynamically allocated.
- If a basic resource control is established on a task or project, the process that establishes this resource control becomes an observer. The action for this resource control block is applied to the observer. However, some resources cannot be observed in this manner.
- If a privileged resource control is set on a task or project, no observer process exists. However, any process that violates the limit becomes the subject of the resource control action.
- Only one action is permitted for each type: global and local.
- Only one basic `rctl` is allowed per process per resource control.

CHAPTER 6

Dynamic Resource Pools

This chapter describes resource pools and their properties.

- "Overview of Resource Pools" on page 69
- "Dynamic Resource Pool Constraints and Objectives" on page 70
- "Resource Pools API Functions" on page 74
- "Resource Pool Code Examples" on page 78
- "Programming Issues Associated With Resource Pools" on page 81

Overview of Resource Pools

Resource pools provide a framework for managing processor sets and thread scheduling classes. Resource pools are used for partitioning machine resources. Resource pools enable you to separate workloads so that workload consumption of certain resources does not overlap. The resource reservation helps to achieve predictable performance on systems with mixed workloads.

For an overview of resource pools and example commands for administering resource pools, see Chapter 12, "Resource Pools (Overview)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* and Chapter 13, "Creating and Administering Resource Pools (Tasks)," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

A processor set groups the CPUs on a system into a bounded entity, on which a process or processes can run exclusively. Processes cannot extend beyond the processor set, nor can other processes extend into the processor set. A processor set enables tasks of similar characteristics to be grouped together and a hard upper boundary for CPU use to be set.

The resource pool framework allows the definition of a soft processor set with a maximum and minimum CPU count requirement. Additionally, the framework provides a hard-defined scheduling class for that processor set.

A resource pool defines

- Processor set groups
- Scheduling class

Scheduling Class

Scheduling classes provide different CPU access characteristics to threads that are based on algorithmic logic. The scheduling classes include:

- Realtime scheduling class
- Interactive scheduling class
- Fixed priority scheduling class
- Timesharing scheduling class
- Fair share scheduling class

For an overview of fair share scheduler and example commands for administering the fair share scheduler, see Chapter 8, “Fair Share Scheduler (Overview),” in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* and Chapter 9, “Administering the Fair Share Scheduler (Tasks),” in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Do not mix scheduling classes in a set of CPUs. If scheduling classes are mixed in a CPU set, system performance might become erratic and unpredictable. Use processor sets to segregate applications by their characteristics. Assign scheduling classes under which the application best performs. For more information about the characteristics of an individual scheduling class, see `priocntl`(1).

For an overview of resource pools and a discussion of when to use pools, see Chapter 6, “Dynamic Resource Pools.”

Dynamic Resource Pool Constraints and Objectives

The `libpool` library defines properties that are available to the various entities that are managed within the pools facility. Each property falls into the following categories:

`Configuration constraints`
A constraint defines boundaries of a property. Typical constraints are the maximum and minimum allocations specified in the `libpool` configuration.

`Objective`
An objective changes the resource assignments of the current configuration to generate new candidate configurations that observe the established constraints. (See `poold`(1M).) An objective has the following categories:

`Workload dependent`	A workload-dependent objective varies according to the conditions imposed by the workload. An example of the workload dependent objective is the `utilization` objective.

Workload independent A workload-independent objective does not vary according to the conditions imposed by the workload. An example of the workload independent objective is the `cpu locality` objective.

An objective can take an optional prefix to indicate the importance of the objective. The objective is multiplied by this prefix, which is an integer from 0 to INT64_MAX,, to determine the significance of the objective.

System Properties

`system.bind-default` (**writable boolean**)
If the specified pool is not found in `<filename>/etc/project</filename>`, bind to pool with the `pool.default` property set to TRUE.

`system.comment` (**writable string**)
User description of system. `system.comment` is not used by the default pools commands, except when a configuration is initiated by the `poolcfg` utility. In this case, the system puts an informative message in the `system.comment` property for that configuration.

`system.name` (**writable string**)
User name for the configuration.

`system.version` (read-only integer)
`libpool` version required to manipulate this configuration.

Pools Properties

All pool properties are writable.

`pool.active` (**writable boolean**)
If TRUE, mark this pool as active.

`pool.comment` (**writable string**)
User description of pool.

`pool.default` (**writable boolean**)
If TRUE, mark this pool as the default pool. See the `system.bind-default` property.

`pool.importance` (**writable integer**)
Relative importance of this pool. Used for possible resource dispute resolution.

`pool.name` (**writable string**)
User name for pool. `setproject`(3PROJECT) uses `pool.name` as the value for the `project.pool` project attribute in the `project`(4) database.

`pool.scheduler` (**writable string**)
Scheduler class to which consumers of this pool are bound. This property is optional and if not specified, the scheduler bindings for consumers of this pool are not affected. For more information about the characteristics of an individual scheduling class, see `priocntl`(1). Scheduler classes include:

- RT for realtime scheduler
- TS for timesharing scheduler
- IA for interactive scheduler
- FSS for fair share scheduler
- FX for fixed priority scheduler

Processor Set Properties

`pset.comment` (**writable string**)
User description of resource.

`pset.default` (read-only boolean)
Identifies the default processor set.

`pset.escapable` (**writable boolean**)
Represents whether PSET_NOESCAPE is set for this pset. See the `pset_setattr`(2) man page.

`pset.load` (read-only unsigned integer)
The load for this processor set. The lowest value is 0. The value increases in a linear fashion with the load on the set, as measured by the number of jobs in the system run queue.

`pset.max` (**writable unsigned integer**)
Maximum number of CPUs that are permitted in this processor set.

`pset.min` (**writable unsigned integer**)
Minimum number of CPUs that are permitted in this processor set.

`pset.name` (**writable string**)
User name for the resource.

`pset.size` (read-only unsigned integer)
Current number of CPUs in this processor set.

`pset.sys_id` (read-only integer)
System-assigned processor set ID.

`pset.type` (read-only string)
Names the resource type. Value for all processor sets is `pset`.

`pset.units` (read-only string)
Identifies the meaning of size-related properties. The value for all processor sets is `population`.

cpu.comment (**writable string**)
User description of CPU

Using libpool to Manipulate Pool Configurations

The libpool(3LIB) pool configuration library defines the interface for reading and writing pools configuration files. The library also defines the interface for committing an existing configuration to becoming the running operating system configuration. The <pool.h> header provides type and function declarations for all library services.

The resource pools facility brings together process-bindable resources into a common abstraction that is called a pool. Processor sets and other entities can be configured, grouped, and labelled in a persistent fashion. Workload components can be associated with a subset of a system's total resources. The libpool(3LIB) library provides a C language API for accessing the resource pools facility. The pooladm(1M), poolbind(1M), and poolcfg(1M) make the resource pools facility available through command invocations from a shell.

Manipulate psets

The following list contains the functions associated with creating or destroying psets and manipulating psets.

processor_bind(2)	Bind an LWP (lightweight process) or set of LWPs to a specified processor.
pset_assign(2)	Assign a processor to a processor set.
pset_bind(2)	Bind one or more LWPs (lightweight processes) to a processor set.
pset_create(2)	Create an empty processor set that contains no processors.
pset_destroy(2)	Destroy a processor set and release the associated constituent processors and processes.
pset_setattr(2), pset_getattr(2)	Set or get processor set attributes.

Resource Pools API Functions

This section lists all of the resource pool functions. Each function has a link to the man page and a short description of the function's purpose. The functions are divided into two groups, depending on whether the function performs an action or a query:

- "Functions for Operating on Resource Pools and Associated Elements" on page 74
- "Functions for Querying Resource Pools and Associated Elements" on page 76

The imported interfaces for libpool for swap sets is identical to the ones defined in this document.

Functions for Operating on Resource Pools and Associated Elements

The interfaces listed in this section are for performing actions related to pools and the associated elements.

`pool_associate`(3POOL)	Associate a resource with a specified pool.
`pool_component_to_elem`(3POOL)	Convert specified component to the pool element type.
`pool_conf_alloc`(3POOL)	Create a pool configuration.
`pool_conf_close`(3POOL)	Close the specified pool configuration and release the associated resources.
`pool_conf_commit`(3POOL)	Commit changes made to the specified pool configuration to permanent storage.
`pool_conf_export`(3POOL)	Save the given configuration to the specified location.
`pool_conf_free`(3POOL)	Release a pool configuration.
`pool_conf_open`(3POOL)	Create a pool configuration at the specified location.
`pool_conf_remove`(3POOL)	Removes the permanent storage for the configuration.
`pool_conf_rollback`(3POOL)	Restore the configuration state to the state that is held in the pool configuration's permanent storage.
`pool_conf_to_elem`(3POOL)	Convert specified pool configuration to the pool element type.
`pool_conf_update`(3POOL)	Update the library snapshot of kernel state.

`pool_create`(3POOL)	Create a new pool with the default properties and with default resources for each type.
`pool_destroy`(3POOL)	Destroy the specified pool. The associated resources are not modified.
`pool_dissociate`(3POOL)	Remove the association between the given resource and pool.
`pool_put_property`(3POOL)	Set the named property on the element to the specified value.
`pool_resource_create`(3POOL)	Create a new resource with the specified name and type for the provided configuration.
`pool_resource_destroy`(3POOL)	Remove the specified resource from the configuration file.
`pool_resource_to_elem`(3POOL)	Convert specified pool resource to the pool element type.
`pool_resource_transfer`(3POOL)	Transfer basic units from the source resource to the target resource.
`pool_resource_xtransfer`(3POOL)	Transfer the specified components from the source resource to the target resource.
`pool_rm_property`(3POOL)	Remove the named property from the element.
`pool_set_binding`(3POOL)	Bind the specified processes to the resources that are associated with pool on the running system.
`pool_set_status`(3POOL)	Modify the current state of the pools facility.
`pool_to_elem`(3POOL)	Convert specified pool to the pool element type.
`pool_value_alloc`(3POOL)	Allocate and return an opaque container for a pool property value.
`pool_value_free`(3POOL)	Release an allocated property values.
`pool_value_set_bool`(3POOL)	Set a property value of type `boolean`.
`pool_value_set_double`(3POOL)	Set a property value of type `double`.
`pool_value_set_int64`(3POOL)	Set a property value of type `int64`.
`pool_value_set_name`(3POOL)	Set a `name=value` pair for a pool property.
`pool_value_set_string`(3POOL)	Copy the string that was passed in.
`pool_value_set_uint64`(3POOL)	Set a property value of type `uint64`.

Functions for Querying Resource Pools and Associated Elements

The interfaces listed in this section are for performing queries related to pools and the associated elements.

`pool_component_info`(3POOL)
Return a string that describes the given component.

`pool_conf_info`(3POOL)
Return a string describing the entire configuration.

`pool_conf_location`(3POOL)
Return the location string that was provided to `pool_conf_open()` for the given specified configuration.

`pool_conf_status`(3POOL)
Return the validity status for a pool configuration.

`pool_conf_validate`(3POOL)
Check the validity of the contents of the given configuration.

`pool_dynamic_location`(3POOL)
Return the location that was used by the pools framework to store the dynamic configuration.

`pool_error`(3POOL)
Return the error value of the last failure that was recorded by calling a resource pool configuration library function.

`pool_get_binding`(3POOL)
Return the name of the pool on the running system that contains the set of resources to which the specified process is bound.

`pool_get_owning_resource`(3POOL)
Return the resource that currently contains the specified component.

`pool_get_pool`(3POOL)
Return the pool with the specified name from the provided configuration.

`pool_get_property`(3POOL)
Retrieve the value of the named property from the element.

`pool_get_resource`(3POOL)
Return the resource with the given name and type from the provided configuration.

`pool_get_resource_binding`(3POOL)
Return the name of the pool on the running system that contains the set of resources to which the given process is bound.

`pool_get_status`(3POOL)
Retrieve the current state of the pools facility.

`pool_info`(3POOL)
Return a description of the specified pool.

`pool_query_components`(3POOL)
Retrieve all resource components that match the specified list of properties.

`pool_query_pool_resources`(3POOL)
Return a null-terminated array of resources currently associated with the pool.

`pool_query_pools`(3POOL)
Return the list of pools that match the specified list of properties.

`pool_query_resource_components`(3POOL)
Return a null-terminated array of the components that make up the specified resource.

`pool_query_resources`(3POOL)
Return the list of resources that match the specified list of properties.

`pool_resource_info`(3POOL)
Return a description of the specified resource.

`pool_resource_type_list`(3POOL)
Enumerate the resource types that are supported by the pools framework on this platform.

`pool_static_location`(3POOL)
Return the location that was used by the pools framework to store the default configuration for pools framework instantiation.

`pool_strerror`(3POOL)
Return a description of each valid pool error code.

`pool_value_get_bool`(3POOL)
Get a property value of type `boolean`.

`pool_value_get_double`(3POOL)
Get a property value of type `double`.

`pool_value_get_int64`(3POOL)
Get a property value of type `int64`.

`pool_value_get_name`(3POOL)
Return the name that was assigned to the specified pool property.

`pool_value_get_string`(3POOL)
Get a property value of type `string`.

`pool_value_get_type`(3POOL)
Return the type of the data that is contained by the specified pool value.

`pool_value_get_uint64`(3POOL)
Get a property value of type `uint64`.

`pool_version`(3POOL)
Get the version number of the pool library.

`pool_walk_components`(3POOL)
Invoke callback on all components that are contained in the resource.

`pool_walk_pools`(3POOL)
Invoke callback on all pools that are defined in the configuration.

`pool_walk_properties`(3POOL)
Invoke callback on all properties defined for the given element.

`pool_walk_resources`(3POOL)
Invoke callback on all resources that are associated with the pool.

Resource Pool Code Examples

This section contains code examples of the resource pools interface.

Ascertain the Number of CPUs in the Resource Pool

`sysconf`(3C) provides information about the number of CPUs on an entire system. The following example provides the granularity of ascertaining the number of CPUs that are defined in a particular application's pools `pset`.

The key points for this example include the following:

- `pvals[]` should be a NULL terminated array.
- `pool_query_pool_resources()` returns a list of all resources that match the `pvals` array type `pset` from the application's pool `my_pool`. Because a pool can have only one instance of the `pset` resource, each instance is always returned in `nelem`. `reslist[]` contains only one element, the `pset` resource.

```
pool_value_t *pvals[2] = {NULL};  /* pvals[] should be NULL terminated */

/* NOTE: Return value checking/error processing omitted */
/* in all examples for brevity */

conf_loc = pool_dynamic_location();
conf = pool_conf_alloc();
pool_conf_open(conf, conf_loc, PO_RDONLY);
my_pool_name = pool_get_binding(getpid());
my_pool = pool_get_pool(conf, my_pool_name);
```

```
pvals[0] = pool_value_alloc();
pvals2[2] = { NULL, NULL };
pool_value_set_name(pvals[0], "type");
pool_value_set_string(pvals[0], "pset");

reslist = pool_query_pool_resources(conf, my_pool, &nelem, pvals);
pool_value_free(pvals[0]);
pool_query_resource_components(conf, reslist[0], &nelem, NULL);
printf("pool %s: %u cpu", my_pool_ name, nelem);
pool_conf_close(conf);
```

List All Resource Pools

The following example lists all resource pools defined in an application's pools `pset`.

The key points of the example include the following:

- Open the dynamic `conf` file read-only, PO_RDONLY. `pool_query_pools()` returns the list of pools in `pl` and the number of pools in `nelem`. For each pool, call `pool_get_property()` to get the `pool.name` property from the element into the `pval` value.
- `pool_get_property()` calls `pool_to_elem()` to convert the `libpool` entity to an opaque value. `pool_value_get_string()` gets the string from the opaque pool value.

```
conf    = pool_conf_alloc();
pool_conf_open(conf, pool_dynamic_location(), PO_RDONLY);
pl = pool_query_pools(conf, &nelem, NULL);
pval = pool_value_alloc();
for (i = 0; i < nelem; i++) {
    pool_get_property(conf, pool_to_elem(conf, pl[i]), "pool.name", pval);
    pool_value_get_string(pval, &fname);
    printf("%s\n", name);
}
pool_value_free(pval);
free(pl);
pool_conf_close(conf);
```

Report Pool Statistics for a Given Pool

The following example reports statistics for the designated pool.

The key points for the example include the following:

- `pool_query_pool_resources()` gets a list of all resources in `rl`. Because the last argument to `pool_query_pool_resources()` is NULL, all resources are returned. For each resource, the `name`, `load` and `size` properties are read, and printed.

- The call to `strdup()` allocates local memory and copies the string returned by `get_string()`. The call to `get_string()` returns a pointer that is freed by the next call to `get_property()`. If the call to `strdup()` is not included, subsequent references to the string(s) could cause the application to fail with a segmentation fault.

```
printf("pool %s\n:" pool_name);
pool = pool_get_pool(conf, pool_name);
rl = pool_query_pool_resources(conf, pool, &nelem, NULL);
for (i = 0; i < nelem; i++) {
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]), "type", pval);
  pool_value_get_string(pval, &type);
  type = strdup(type);
  snprintf(prop_name, 32, "%s.%s", type, "name");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_string(val, &res_name);
  res_name = strdup(res_name);
  snprintf(prop_name, 32, "%s.%s", type, "load");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &load);
  snprintf(prop_name, 32, "%s.%s", type, "size");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &size);
  printf("resource %s: size %llu load %llu\n", res_name, size, load);
  free(type);
  free(res_name);
}
free(rl);
```

Set `pool.comment` Property and Add New Property

The following example sets the `pool.comment` property for the `pset`. The example also creates a new property in `pool.newprop`.

The key point for the example includes the following:

- In the call to `pool_conf_open()`, using PO_RDWR on a static configuration file, requires the caller to be root.
- To commit these changes to the `pset` after running this utility, issue a `pooladm -c` command. To have the utility commit the changes, call `pool_conf_commit()` with a nonzero second argument.

```
pool_set_comment(const char *pool_name, const char *comment)
{
  pool_t *pool;
```

```
  pool_elem_t *pool_elem;
  pool_value_t *pval = pool_value_alloc();
  pool_conf_t  *conf = pool_conf_alloc();
  /* NOTE: need to be root to use PO_RDWR on static configuration file */
  pool_conf_open(conf, pool_static_location(), PO_RDWR);
  pool = pool_get_pool(conf,  pool_name);
  pool_value_set_string(pval, comment);
  pool_elem = pool_to_elem(conf, pool);
  pool_put_property(conf, pool_elem, "pool.comment", pval);
  printf("pool %s: pool.comment set to %s\n:" pool_name, comment);
  /* Now, create a new property, customized to installation site */
  pool_value_set_string(pval, "New String Property");
  pool_put_property(conf, pool_elem, "pool.newprop", pval);
  pool_conf_commit(conf, 0); /* NOTE: use 0 to ensure only */
                             /* static file gets updated */
  pool_value_free(pval);
  pool_conf_close(conf);
  pool_conf_free(conf);
  /* NOTE: Use "pooladm -c" later, or pool_conf_commit(conf, 1) */
  /* above for changes to the running system */
}
```

An alternative way of modifying a pool's comment and adding a new pool property is to use `poolcfg`(1M).

```
poolcfg -c 'modify pool pool-name (string pool.comment = "cmt-string")'
poolcfg -c 'modify pool pool-name (string pool.newprop =
                                   "New String Property")'
```

Programming Issues Associated With Resource Pools

Consider the following issues when writing your application.

- Each site can add its own list of properties to the pools configuration.

  Multiple configurations can be maintained in multiple configuration files. The system administrator can commit different files to reflect changes to the resource consumption at different time slots. These time slots can include different times of the day, week, month, or seasons depending on load conditions.
- Resource sets can be shared between pools, but a pool has only one resource set of a given type. So, the `pset_default` can be shared between the default and a particular application's database pools.
- Use `pool_value_*()` interfaces carefully. Keep in mind the memory allocation issues for string pool values. See “Report Pool Statistics for a Given Pool” on page 79.

CHAPTER 7

Design Considerations for Resource Management Applications in Solaris Zones

This chapter provides a brief overview of Solaris zones technology and discusses potential problems that may be encountered by developers who are writing resource management applications. For more information on zones, see Part II, "Zones," in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Zones Overview

A *zone* is a virtualized operating system environment that is created within a single instance of the Solaris Operating System. Zones are a partitioning technology that provides an isolated, secure environment for applications. When you create a zone, you produce an application execution environment in which processes are isolated from the rest of the system. This isolation prevents a process that is running in one zone from monitoring or affecting processes that are running in other zones. Even a process running with superuser credentials cannot view or affect activity in other zones. A zone also provides an abstract layer that separates applications from the physical attributes of the machine on which the zone is deployed. Examples of these attributes include physical device paths and network interface names.

By default, all systems have a *global zone*. The global zone has a global view of the Solaris environment in similar fashion to the superuser model. All other zones are referred to as *non-global zones*. A non-global zone is analogous to an unprivileged user in the superuser model. Processes in non-global zones can control only the processes and files within that zone. Typically, system administration work is mainly performed in the global zone. In rare cases where a system administrator needs to be isolated, privileged applications can be used in a non-global zone. In general, though, resource management activities take place in the global zone.

IP Networking in Zones

IP networking in a zone can be configured in two different ways, depending on whether the non-global zone is given its own exclusive IP instance or shares the IP layer configuration and state with the global zone. The shared-IP type is the default.

Exclusive-IP zones are assigned zero or more network interface names, and for those network interfaces they can send and receive any packets, snoop, and change the IP configuration, including IP addresses and the routing table. Note that those changes do not affect any of the other IP instances on the system.

Design Considerations for Resource Management Applications in Zones

All applications are fully functional in the global zone as they would be in a conventional Solaris environment. Most applications should run without problem in a non-global environment as long as the application does not need any privileges. If an application does require privileges, then the developer needs to take a close look at which privileges are needed and how a particular privilege is used. If a privilege is required, then a system administrator must assign the appropriate privilege to the application.

General Considerations When Writing Applications for Non-Global Zones

The known situations that a developer needs to investigate are as follows:

- System calls that change the system time require the PRIV_SYS_TIME privilege. These system calls include `adjtime`(2), `ntp_adjtime`(2), and `stime`(2).
- System calls that need to operate on files that have the sticky bit set require the PRIV_SYS_CONFIG privilege. These system calls include `chmod`(2), `creat`(2), and `open`(2).
- The `ioctl`(2) system call requires the PRIV_SYS_NET_CONFIG privilege to be able to unlock an anchor on a STREAMS module. .
- The `link`(2) and `unlink`(2) system calls require the PRIV_SYS_LINKDIR privilege to create a link or unlink a directory in a non-global zone. Applications that install or configure software or that create temporary directories could be affected by this limitation.
- The PRIV_PROC_LOCK_MEMORY privilege is required for the `mlock`(3C), `munlock`(3C), `mlockall`(3C), `munlockall`(3C), and `plock`(3C) functions and the MC_LOCK, MC_LOCKAS, MC_UNLOCK, and MC_UNLOCKAS flags for the `memcntl`(2) system. This privilege is a default privilege in a non-global zone. See "Privileges in a Non-Global Zone" in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones* for more information.

- The mknod(2) system call requires the PRIV_SYS_DEVICES privilege to create a block (S_IFBLK) or character (S_IFCHAR) special file. This limitation affects applications that need to create device nodes on the fly.
- The IPC_SET flag in the msgctl(2) system call requires the PRIV_SYS_IPC_CONFIG privilege to increase the number of message queue bytes. This limitation affects any applications that need to resize the message queue dynamically.
- The nice(2) system call requires the PRIV_PROC_PRIOCNTL privilege to change the priority of a process. This privilege is available by default in a non-global zone. Another way to change the priority is to bind the non-global zone in which the application is running to a resource pool, although scheduling processes in that zone is ultimately decided by the Fair Share Scheduler.
- The P_ONLINE, P_OFFLINE, P_NOINTR, P_FAULTED, P_SPARE, and PZ-FORCED flags in the p_online(2) system call require the PRIV_SYS_RES_CONFIG privilege to return or change process operational status. This limitation affects applications that need to enable or disable CPUs.
- The PC_SETPARMS and PC_SETXPARMS flags in the priocntl(2)system call requires the PRIV_PROC_PRIOCNTL privilege to change the scheduling parameters of a lightweight process (LWP).
- System calls that need to manage processor sets (psets), including binding LWPs to psets and setting pset attributes require the PRIV_SYS_RES_CONFIG privilege. This limitation affects the following system calls: pset_assign(2), pset_bind(2), pset_create(2), pset_destroy(2), and pset_setattr(2).
- The SHM_LOCK and SHM_UNLOCK flags in the shmctl(2) system call require the PRIV_PROC_LOCK_MEMORY privilege to share memory control operations. If the application is locking memory for performance purposes, using the intimate shared memory (ISM) feature provides a potential workaround.
- The swapctl(2)system call requires the PRIV_SYS_CONFIG privilege to add or remove swapping resources. This limitation affects installation and configuration software.
- The uadmin(2) system call requires the PRIV_SYS_CONFIG privilege to use the A_REMOUNT, A_FREEZE, A_DUMP, and AD_IBOOT commands. This limitation affects applications that need to force crash dumps under certain circumstances.
- The clock_settime(3RT) function requires the PRIV_SYS_TIME privilege to set the CLOCK_REALTIME and CLOCK_HIRES clocks.
- The cpc_bind_cpu(3CPC) function requires the PRIV_CPC_CPU privilege to bind request sets to hardware counters. As a workaround, the cpc_bind_curlwp(3CPC) function can be used to monitor CPU counters for the LWP in question.
- The pthread_attr_setschedparam(3C) function requires the PRIV_PROC_PRIOCNTL privilege to change the underlying scheduling policy and parameters for a thread.
- The timer_create(3RT) function requires the PRIV_PROC_CLOCK_HIGHRES privilege to create a timer using the high-resolution system clock.

- The APIs that are provided by the following list of libraries are not supported in a non-global zone. The shared objects are present in the zone's `/usr/lib` directory, so no link time errors occur if your code includes references to these libraries. You can inspect your `make` files to determine if your application has explicit bindings to any of these libraries and use `pmap`(1) while the application is executing to verify that none of these libraries are dynamically loaded.
  - `libdevinfo`(3LIB)
  - `libcfgadm`(3LIB)
  - `libpool`(3LIB)
  - `libtnfctl`(3LIB)
  - `libsysevent`(3LIB)
- Zones have a restricted set of devices, consisting primarily of pseudo devices that form part of the Solaris programming API. These pseudo devices include `/dev/null`, `/dev/zero`, `/dev/poll`, `/dev/random`, `/dev/tcp`, and so on. Physical devices are not directly accessible from within a zone unless the device has been configured by a system administrator. Since devices, in general, are shared resources in a system, to make devices available in a zone requires some restrictions so system security will not be compromised, as follows:
  - The `/dev` name space consists of symbolic links, that is, logical paths, to the physical paths in `/devices`. The `/devices` name space, which is available only in the global zone, reflects the current state of attached device instances that have been created by the driver. Only the logical path `/dev` is visible in a non-global zone.
  - Processes within a non-global zone cannot create new device nodes . For example, `mknod`(2) cannot create special files in a non-global zone. The `creat`(2), `link`(2), `mkdir`(2), `rename`(2), `symlink`(2), and `unlink`(2) system calls fail with EACCES if a file in `/dev` is specified. You can create a symbolic link to an entry in `/dev`, but that link cannot be created in `/dev`.
  - Devices that expose system data are only available in the global zone. Examples of such devices include `dtrace`(7D), `kmem`(7D), `kmdb`(7d), `ksyms`(7D), `lockstat`(7D), and `trapstat`(1M).
  - The `/dev` name space consists of device nodes made up of a default, “safe” set of drivers as well as device nodes that have been specified for the zone by the `zonecfg`(1M) command.

Specific Considerations for Shared-IP Non-Global Zones

For non-global zones that are configured to use the shared-IP instance, the following restrictions apply.

- The `socket`(3SOCKET) function requires the PRIV_NET_RAWACCESS privilege to create a raw socket with the protocol set to IPPROTO_RAW or IPPROTO_IGMP. This limitation affects applications that use raw sockets or need to create or inspect TCP/IP headers.

- The t_open(3NSL) function requires the PRIV_NET_RAWACCESS privilege to establish a transport endpoint. This limitation affects applications that use the /dev/rawip device to implement network protocols as wall as applications that operate on TCP/IP headers.
- No NIC devices that support the DLPI programming interface are accessible in a shared-IP non-global zone, for example, hme(7D) and ce(7D).
- Each non-global shared-IP zone has its own logical network and loopback interface. Bindings between upper layer streams and logical interfaces are restricted such that a stream may only establish bindings to logical interfaces in the same zone. Likewise, packets from a logical interface can only be passed to upper layer streams in the same zone as the logical interface. Bindings to the loopback address are kept within a zone with one exception: When a stream in one zone attempts to access the IP address of an interface in another zone. While applications within a zone can bind to privileged network ports, they have no control over the network configuration, including IP addresses and the routing table.

Note that these restrictions do not apply to exclusive-IP zones.

CHAPTER 8

Configuration Examples

This chapter show example configurations for the `/etc/project` file.

- "Configure Resource Controls" on page 90
- "Configure Resource Pools" on page 90
- "Configure FSS `project.cpu-shares` for a Project" on page 90
- "Configure Five Applications with Different Characteristics" on page 91

`/etc/project` Project File

The project file is a local source of project information. The project file can be used in conjunction with other project sources, including the NIS maps `project.byname` and `project.bynumber` and the LDAP database project. Programs use the `getprojent`(3PROJECT) routines to access this information.

Define Two Projects

`/etc/project` defines two projects: `database` and `appserver`. The *user* defaults are `user.database` and `user.appserver`. The *admin* default can switch between `user.database` or `user.appserver`.

```
hostname# cat /etc/project

.
.
.
user.database:2001:Database backend:admin::
user.appserver:2002:Application Server frontend:admin::
.
.
```

Configure Resource Controls

The /etc/project file shows the resource controls for the application.

```
hostname# cat /etc/project

.
.
.
development:2003:Developers:::task.ax-lwps=(privileged,10,deny);
process.max-addressspace=(privileged,209715200,deny)
.
.
```

Configure Resource Pools

The /etc/project file shows the resource pools for the application.

```
hostname# cat /etc/project

.
.
.
batch:2001:Batch project:::project.pool=batch_pool
process:2002:Process control:::project.pool=process_pool
.
.
.
```

Configure FSS project.cpu-shares for a Project

Set up FSS for two projects: *database* and *appserver*. The *database* project has 20 CPU shares. The *appserver* project has 10 CPU shares.

```
hostname# cat /etc/project

.
.
.
user.database:2001:database backend:admin::project.cpu-shares=(privileged,
     20,deny)
user.appserver:2002:Application Server frontend:admin::project.cpu-shares=
     (privileged,10,deny)
.
.
.
```

Note – The line break in the lines that precede "20,deny" and "(privileged," is not valid in an /etc/project file. The line breaks are shown here only to allow the example to display on a printed or displayed page. Each entry in the /etc/project file must be on a single line.

If the FSS is enabled but each user and application is not assigned to a unique project, then the users and applications will all run in the same project. By running in the same project, all compete for the same share, in a timeshare fashion. This occurs because shares are assigned to projects, not to users or processes. To take advantage of the FSS scheduling capabilities, assign each user and application to a unique project.

To configure a project, see "Local /etc/project File Format" in *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Configure Five Applications with Different Characteristics

The following example configures five applications with different characteristics.

TABLE 8–1 Target Applications and Characteristics

Application Type and Name	Characteristics
Application server, app_server.	Negative scalability beyond two CPUs. Assign a two-CPU processor set to app_server. Use TS scheduling class.
Database instance, app_db.	Heavily multithreaded. Use FSS scheduling class.
Test and development, development.	Motif based. Hosts untested code execution. Interactive scheduling class ensures user interface responsiveness. Use process.max-address-space to impose memory limitations and minimize the effects of antisocial processing.
Transaction processing engine, tp_engine.	Response time is paramount. Assign a dedicated set of at least two CPUs to ensure response latency is kept to a minimum. Use timeshare scheduling class.
Standalone database instance, geo_db.	Heavily multithreaded. Serves multiple time zones. Use FSS scheduling class.

Note – Consolidate database applications (app.db and geo_db) onto a single processor set of at least four CPUs. Use FSS scheduling class. Application app_db gets 25% of the project.cpu-shares. Application geo_db gets 75% of the project.cpu-shares.

Edit the /etc/project file. Map users to resource pools for the app_server, app_db, development, tp_engine, and geo_db project entries.

```
hostname# cat /etc/project

.
.
.
user.app_server:2001:Production Application Server::
     project.pool=appserver_pool
user.app_db:2002:App Server DB:::project.pool=db_pool,
     project.cpu-shares=(privileged,1,deny)
development:2003:Test and delopment::staff:project.pool=dev.pool,
     process.max-addressspace=(privileged,536870912,deny)
user.tp_engine:Transaction Engine:::project.pool=tp_pool
user.geo_db:EDI DB:::project.pool=db_pool;
     project.cpu-shares=(privileged,3,deny)
```

Note – The line break in the lines that begin with “project.pool” , “project.cpu-shares=”, “process.max-addressspace”, and “project.cpu-shares=” is not valid in a project file. The line breaks are shown here only to allow the example to display on a printed or displayed page. Each entry must be on one and only one line.

Create the pool.host script and add entries for resource pools.

```
hostname# cat pool.host

create system host
create pset dev_pset (unit pset.max = 2)
create pset tp_pset (unit pset.min = 2)
create pset db_pset (unit pset.min = 4; uint pset.max = 6)
create pset app_pset (unit pset.min = 1; uint pset.max = 2)
create pool dev_pool (string pool.scheduler="IA")
create pool appserver_pool (string pool.scheduler="TS")
create pool db_pool (string pool.scheduler="FSS")
create pool tp_pool (string pool.scheduler="TS")
associate pool pool_default (pset pset_default)
associate pool dev_pool (pset dev_pset)
associate pool pool appserver_pool (pset app_pset)
associate pool db_pool (pset db_pset)
associate pool tp_pool (pset tp_pset)
```

Note – The line break in the line that begins with “boolean” is not valid in a pool.host file. The line break is shown here only to allow the example to display on a printed or displayed page. Each entry must be on one and only one line.

Run the pool.host script and modify the configuration as specified in the pool.host file.

```
hostname# poolcfg -f pool.host
```

Read the pool.host resource pool configuration file and initialize the resource pools on the system.

```
hostname# pooladm -c
```

Index

E

`exacct` file
- display entry, 23
- display string, 24

`exacct` file, display system file, 25

`exacct` file
- dump, 47

`exacct` object
- create record, 46
- dump, 44
- write file, 46

F

fair share scheduler, access resource control block, 65

L

`libexacct`
- perl interface, 30
- perl module, 31

P

programming issues
- project database, 19-20
- resource controls, 67

project database
- get entry, 19
- print entries, 18

R

resource controls
- display value-action pairs, 64
- global action, 54
- global flag, 54
- local action, 53
- local flag, 53
- master observer process, 62
- privilege levels, 53
- process, 58
- project, 57
- signals, 60
- task, 58
- zone, 59

resource pools
- get defined pools, 79
- get number of CPUS, 78
- get pool statistics, 79
- pool properties, 71
- processor sets properties, 72-73
- properties, 70
- set property, 80
- system properties, 71

Z

zone
- application design considerations, 84
- IP networking, 84
- IP type, 84
- overview, 83

zone *(Continued)*
resource controls, 59