Mica Working Design Document
Process Structure

Revision 0.6
29–OCT–1987

Issued by:
Mark Lucovsky

digital™

TABLE OF CONTENTS

CHAPTER 1 PROCESS STRUCTURE 1–1

1.1 Introduction 1–1

1.2 Goals/Requirements 1–1

1.3 UJPT Hierarchy 1–1

1.3.1 The User Object 1–1

1.3.1.1 Object Structure 1–1

1.3.1.1.1 Security Profile 1–2

1.3.1.1.2 Resource Control 1–2

1.3.1.1.2.1 Deductable Resource Limits 1–3

1.3.1.1.2.2 Non-Deductable Resource Limits 1–4

1.3.1.1.3 Access Restrictions 1–4

1.3.1.2 Functional Interface 1–4

1.3.1.2.1 User Creation 1–4

1.3.1.2.2 Get/Set User Information 1–6

1.3.1.2.3 User Deletion 1–7

1.3.2 The Job Object 1–8

1.3.2.1 Object Structure 1–8

1.3.2.1.1 Resource Control 1–9

1.3.2.2 Functional Interface 1–9

1.3.2.2.1 Job Creation 1–9

1.3.2.2.2 Job Deletion 1–11

1.3.2.2.3 Get/Set Job Information 1–11

1.3.3 The Process Object 1–13

1.3.3.1 Object Structure 1–13

1.3.3.1.1 Resource Control 1–14

1.3.3.1.2 Process Accounting 1–14

1.3.3.2 Functional Interface 1–15

1.3.3.2.1 Process Creation 1–15

1.3.3.2.2 Process Deletion 1–17

1.3.3.2.3 Get/Set Process Information 1–17

1.3.3.2.4 Process Control Operations 1–19

1.3.3.2.4.1 Process Signaling 1–19

1.3.3.2.4.2 Process Hibernate/Wake 1–20

1.3.3.2.4.3 Process Suspend/Resume 1–21

1.3.4 The Thread Object 1–22

1.3.4.1 Object Structure 1–22

1.3.4.2 Functional Interface 1–24

1.3.4.2.1 Thread Creation 1–24

1.3.4.2.2 Thread Deletion 1–25

1.3.4.2.3 Get/Set Thread Information 1–26

1.3.4.2.4 Thread Control Operations 1–27
1.3.4.2.4.1 Thread Signaling 1–28
1.3.4.2.4.2 Thread Hibernate/Wake 1–28
1.3.4.2.4.3 Thread Suspend/Resume 1–29
1.3.4.2.4.4 Hibernate and Suspend Comparison 1–30
1.4 UJPT Object Linkages 1–30
1.4.1 Linkage Structure 1–31
1.4.2 Hierarchy Creation 1–31
1.4.3 Hierarchy Collapse/Deletion 1–32
1.4.3.1 Force-Exit Routines 1–33
1.4.3.1.1 User-Object Force-Exit Routine 1–33
1.4.3.1.2 Job-Object Force-Exit Routine 1–33
1.4.3.1.3 Process-Object Force-Exit Routine 1–33
1.4.3.1.4 Thread Object Force Exit Routine 1–33
1.4.3.1.4.1 Thread Context Entry 1–33
1.4.3.1.4.2 Thread Exit 1–34
1.4.3.2 Object Remove Routines 1–34
1.4.3.2.1 User-Object Remove Routine 1–35
1.4.3.2.2 Job Object Remove Routine 1–35
1.4.3.2.3 Process Object Remove Routine 1–35
1.4.3.2.4 Thread Object Remove Routine 1–35
1.4.3.3 Object Delete Routines 1–35
1.4.3.3.1 User-Object Delete Routine 1–36
1.4.3.3.2 Job-Object Delete Routine 1–36
1.4.3.3.3 Process-Object Delete Routine 1–36
1.4.3.3.4 Thread-Object Delete Routine 1–36
1.5 Address Space and Execution Threads 1–36
1.5.1 Creation 1–36
1.5.1.1 Initial Thread Creation 1–37
1.5.1.1.1 Address Space Creation 1–37
1.5.1.1.2 Execution Thread Creation 1–38
1.5.1.1.2.1 Address Space Initialization 1–38
1.5.1.1.2.2 Control Region Initialization 1–39
1.5.1.1.2.3 Program Image Mapping 1–39
1.5.1.2 Subsequent Thread Creation 1–39
1.5.1.2.1 Thread Stack Creation 1–39
1.5.1.2.2 Control Region Initialization 1–39
1.5.1.2.3 Transition to new Thread 1–40
1.5.2 Deletion 1–40
1.5.2.1 Execution Thread Deletion 1–40
1.5.2.1.1 In-Context Thread Deletion 1–40
1.5.2.1.2 Out of Context Thread Deletion 1–40
1.5.2.2 Address Space Deletion 1–41
1.6 Exit Status 1–41
1.6.1 Object Structure 1–41

1.6.2 Functional Interface 1–41
1.6.2.1 Exit Status Object Creation 1–42
1.6.2.2 Get Exit Status Information 1–42
1.6.3 Usage 1–42
1.6.3.1 Thread Exit Status Object Usage 1–43
1.6.3.2 Process Exit Status Object Usage 1–43
1.7 Process/Thread Startup/Rundown Summary 1–43
1.7.1 Startup Summary 1–43
1.7.1.1 Additional Thread Startup Summary 1–45
1.7.2 Rundown Summary 1–45
1.8 System Threads 1–48
1.8.1 System Thread Creation 1–48
1.8.2 System Thread Restrictions 1–48

INDEX

EXAMPLES

1–1 User Object Structure 1–2
1–2 Resource Control Structures 1–3
1–3 Access Restriction Data Structures 1–4
1–4 User Object Creation System Interface 1–5
1–5 User Record Structure 1–6
1–6 Get/Set User Information System Interface 1–6
1–7 User Object Deletion System Interface 1–8
1–8 Job Object Structure 1–8
1–9 Job Object Creation System Interface 1–10
1–10 Job Record Structure 1–11
1–11 Job Object Deletion System Interface 1–11
1–12 Get/Set Job Information System Interface 1–12
1–13 Process Object Structure 1–13
1–14 Process Accounting Structure 1–15
1–15 Process Object Creation System Interface 1–16
1–16 Process Record Structure 1–17
1–17 Process Object Deletion System Interface 1–17
1–18 Get/Set Process Information System Interface 1–18
1–19 Signal Process System Interface 1–20
1–20 Hibernate/Wake Process System Interface 1–20
1–21 Suspend/Resume Process System Interface 1–21
1–22 Thread Object Structure 1–23
1–23 Thread Object Creation System Interface 1–24
1–24 Thread Record Structure 1–25
1–25 Thread Object Deletion System Interfaces 1–26
1–26 Get/Set Thread Information System Interface 1–27
1–27 Signal Thread System Interface 1–28
1–28 Hibernate/Wake Thread System Interface 1–29
1–29 Suspend/Resume Thread System Interface 1–30

1–30 Address Space Creation 1–37
1–31 Initial Thread Entry Point 1–38
1–32 Address Space Initialization 1–38
1–33 Exit Status Object Structure 1–41
1–34 Exit Status Object Creation System Interface 1–42
1–35 Get Exit Status Information System Interface 1–42
1–36 System Thread Creation Executive Interface 1–48

FIGURES

1–1 Complex UJPT Hierarchy 1–31

TABLES

1–1 Get/Set User Information Item Codes 1–7
1–2 Get/Set Job Information Item Codes 1–12
1–3 Get/Set Process Information Item Codes 1–19
1–4 Get/Set Thread Information Item Codes 1–27

Revision History

Date	Revision Number	Author	Summary of Changes
10–Jun–86	0.0	Tom Miller	Initial entry
29–Jun–86	0.1	Tom Miller	Incorporating review comments
27–Aug–86	0.2	Tom Miller	Multiple environment support
06–Apr–87	0.3	Tom Miller	Rewrite for second WDD
27–AUG–1987	x.1	Mark Lucovsky	First Draft for third WDD
04–SEP–1987	x.2	Mark Lucovsky	Incorporate comments from first draft. Most notable change was the addition of IO accounting, process and thread exit status, and section on thread /process startup/rundown summary
08–OCT–1987	x.3	Mark Lucovsky	Incorporate comments from Second draft. Most notable change was the section on system threads, and the thread parameter passing scheme
09–OCT–1987	0.4	Mark Lucovsky	Added exec$create_user( ) and description of thread_ record
16–OCT–1987	0.5	Mark Lucovsky	Proofreading corections, moved security profile from process object to the thread object, added cancel io by thread support to the thread object.
29–OCT–1987	0.6	Mark Lucovsky	Added access restrictions to user object, revised hierarchy collapse description

CHAPTER 1

PROCESS STRUCTURE

1.1 Introduction

This chapter describes the external interfaces and data structures of the Mica process structure, the architecture of which is based on the User, Job, Process, Thread (UJPT) hierarchy. This chapter also describes the UJPT implementation in terms of its algorithms and dependencies on other portions of the Mica system (e.g. the kernel and object architecture).

1.2 Goals/Requirements

The goal of the UJPT architecture is to provide a vehicle for controlling multiple threads of execution in a single address space. The architecture provides facilities for resource usage control, security profile management, address space and image management, and object container directory services.

1.3 UJPT Hierarchy

The UJPT architecture consists of a hierarchy of objects. The objects provide a logical grouping of functionality and control.

1.3.1 The User Object

The User object appears at the highest level of the UJPT hierarchy. Its primary function is to provide a focal point for acquiring security profiles and resource quotas/limits for its underlying objects.

The User object is implemented as a system level object in the "USER$OBJECT_CONTAINER" object container.

1.3.1.1 Object Structure

Each user of the Mica system is assigned a unique username, a security profile, and a set of resource limits or quotas. The Mica system keeps track of this information in a system-wide authorization file. If the user has at least one active job, the information is also kept in his user object. As we shall see later in this chapter, information from the user object is propagated down the UJPT hierarchy on an as-needed basis.

NOTE

The intent of the Mica executive is to remain independent of the system-wide authorization file. Therefore, all Mica user attributes are stored in the user object. In addition, the Mica executive places no restrictions on the source of information stored in the user object. It does, however, place a Digital-reserved identifier in the ACL for the user object OTD which limits who can create user objects.

The user object is split into a user object body and a user control block. The user object body contains the information necessary to support the UJPT hierarchy. The user control block contains the vital information of the user object. Example 1–1 illustrates the data structures used to represent the user object.

Example 1–1: User Object Structure

```
e$t_user_object_body: RECORD
    u_obj_id: e$t_object_id;                        ! Object ID of the user object
    u_user_flags: e$t_user_flags;                   ! User object flags
    u_job_queue_mutex: k$dispatcher_object(mutex)   ! Mutex for job management
    u_job_count: integer;                           ! Number of Jobs owned by the user
    u_job_queue_hd: e$t_linked_list;                ! List head of job objects
    u_ucb: e$t_user_control_block;                  ! User Control Block
END RECORD;

e$t_user_control_block: RECORD
    ucb_username: string(e$c_max_user_name);            ! User Name
    ucb_security_profile: e$t_security_profile;         ! User Security Profile
    ucb_quotas: e$t_quotas;                             ! Resource usage control information
    ucb_thread_priority: k$combined_priority;           ! Default thread priority
    ucb_access_restrictions: e$t_access_restrictions;   ! Access Restrictions
    ucb_user_allocation_list: e$t_allocation_list;      ! objects allocated to the user object
END RECORD;
```

1.3.1.1.1 Security Profile

The security profile maintained in the user object contains the list of identifiers assigned to the Mica user. The identifier list gives access rights to the user object as described in Chapter 11, Security and Privileges.

1.3.1.1.2 Resource Control

The goals of the Mica system resource control and quota architecture are:

- Prevent a single user from abusing the system by over running system resources.
- Be simple, predictable and easy to understand.
- Provide repeatable consistent behavior.

The Mica system achieves these goals through data structures maintained in the user object and through policies implemented in the object architecture, memory management system, and the kernel. Example 1–2 illustrates the resource-control data structures maintained in the user object.

Example 1–2: Resource Control Structures

```
!
! User Object Resource Control
!
e$t_quotas: RECORD
    q_usage_and_limits: e$t_quota_usage_and_limits;  ! Currently Used Quotas and Quota Limits
    q_per_job_limits: e$t_quota_limits;              ! Per Job Limits
    q_per_process_limits: e$t_quota_limits;          ! Per Process Limits
END RECORD;

!
! Quota Limits
!
e$t_quota_limits: RECORD
    ql_deductable_limits: e$t_quota_deductable_limits;       ! Deductable Resource Limits
    ql_nondeductable_limits: e$t_quota_nondeductable_limits; ! Non-Deductable Resource Limits
END RECORD;

!
! Quota Usage and Limits
!
e$t_quota_usage_and_limits: RECORD
    qual_mutex: k$dispatcher_object(mutex);             ! Used for block quota allocations
    qual_limits: e$t_quota_limits;                      ! Resource limits for this object
    qual_usage: e$t_quota_usage;                        ! Resources used by this object
END RECORD;

!
! Deductable Limits
!
e$t_quota_deductable_limits: RECORD
    qdl_paging_file_limit: e$t_resource_counter;     ! Max blocks of paging file usable by object
    qdl_paged_pool_limit: e$t_resource_counter;      ! Max number bytes paged pool usable by object
    qdl_non_paged_pool_limit: e$t_resource_counter;  ! Max number bytes non paged pool usable by object
    qdl_cpu_time_limit: e$t_time_value;              ! Max cpu time used by object
END RECORD;

!
! Non Deductable Limits
!
e$t_quota_nondeductable_limits: RECORD
    qnl_working_set_limit: e$t_resource_counter;     ! Max pages in working set
    qnl_working_set_extent: e$t_resource_counter;    ! Largest Possible Working Set
END RECORD;

!
! Quota Usage
!
e$t_quota_usage: RECORD
    qu_paging_file_in_use: e$t_resource_counter;     ! Number blocks of paging file in use by object
    qu_paged_pool_in_use: e$t_resource_counter;      ! Number bytes paged pool in use by object
    qu_non_paged_pool_in_use: e$t_resource_counter;  ! Number bytes non paged pool in use by object
    qu_working_set_in_use: e$t_resource_counter;     ! Pages in working set for this object
    qu_cpu_time_used: e$t_time_value;                ! Cpu time used by object
END RECORD;
```

During user-object creation, the *ucb_quotas* field of the user control block is initialized. The values are obtained from the *user_record* parameter to the *exec$create_user()* system service.

Once established, the *ucb_quotas* field of the user control block becomes the focal point for resource allocation limitation. The Mica system organizes resource limits as deductable and non-deductable resources. All operations on *e$t_quota_limits* are performed in terms of the attributes of deductable and non-deductable resource limits.

1.3.1.1.2.1 Deductable Resource Limits

Deductable resource limits are charged to the next highest object in the UJPT hierarchy at object creation time. An example of this property can be seen in the creation of a process object. Assume a job object had 100 units of paged pool available in *qdl_paged_pool_limit*, and the user object specified that the per process limit for *qdl_paged_pool_limit* was 50 units. After the process object was created, the job object would be charged with 50 units of paged pool in *qu_paged_pool_in_use*. The process object would have 50 units of paged pool available in *qdl_paged_pool_limit*, and would be charged with 0 units of paged pool in *qu_paged_pool_in_use*.

1.3.1.1.2.2 Non-Deductable Resource Limits

Non-deductable resource limits are limits enforced by policies of the Mica system, but are not charged for against the higher level objects pool of available resources. For example, assume that in the creation of a process the user object specified a working set limit of 50 units. As a consequence, all job objects and process objects would contain the 50 units of resource in their *qnl_working_set_limit* fields.

1.3.1.1.3 Access Restrictions

The user object maintains the current system access restrictions for the Mica user that it represents. The access restrictions are not enforced by the UJPT architecture. External processes may inspect the access restrictions in the current set of user objects and determine what type of enforcement actions are necessary. Example 1–3 illustrates the data structures used to maintain the access restrictions placed in the user object.

Example 1–3: Access Restriction Data Structures

```
!
! Access Restrictions
!
e$t_access_restrictions: RECORD
    ar_restriction_vector: ARRAY[e$t_job_class] OF e$t_class_access_restrictions;
    ar_expiration_date: e$t_date;       ! The last day that user can access the system
END RECORD;

!
! Per Job Class Access Restrictions
!
e$t_class_access_restrictions: RECORD
    car_prime_days: e$t_day_set;        ! The prime days user can access system
    car_non_prime_days: e$t_day_set;    ! The non-prime days user can access system
    car_prime_hours: e$t_hour_set;      ! The hours on prime days user can access system
    car_non_prime_hours: e$t_hour_set;  ! The hours on non prime days user can access system
END RECORD;
```

1.3.1.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting user objects, and setting and extracting various attributes of a User object.

1.3.1.2.1 User Creation

Creating a user object also causes a UJPT hierarchy to be created. The system service *exec$create_user()* creates a user object, job object, process object, and thread object. If there is a name collision between the new user object and an existing user object for the same user, then the new user object is discarded, and the job, process, and thread objects are attached to the existing user object. Example 1–4 illustrates the interface to *exec$create_user()*.

Example 1–4: User Object Creation System Interface

```
PROCEDURE exec$create_user (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN user_record: exec$t_user_record;
            IN user_allocation_list: exec$t_allocation_list = DEFAULT;

            IN job_record: exec$t_job_record = DEFAULT;
            IN job_initial_container: exec$t_object_id = DEFAULT;
            IN job_allocation_list: exec$t_allocation_list = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a user, job, process, and thread object as specified by the parameters.
!   If the user object collides with an existing user object, then use the existing
!   user object
!
! Arguments:
!
!   object_id                   Object ID of the resulting user object
!   container                   Object container for user object (ignored)
!   name                        Name of user object
!   acl                         ACL to place on user object
!   user_record                 Attributes of new user (from authorization file ?)
!   user_allocation_list        Objects to be allocated to the user object. If not present then
!                               no objects are allocated to the user
!   job_record                  Attributes of the job being created. If not present, then
!                               values are obtained from current user object
!   job_initial_container       Job level object container to be transfered into the job
!                               level container directory for this job. If not present then
!                               container directory comes up empty
!   job_allocation_list         Objects to be allocated to the job object. If not present then
!                               no objects are allocated to the job
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_user()*, it is clear that the *user_record* can have an impact on the structure of the user being created. Example 1–5 illustrates the layout of the *user_record*.

Example 1–5: User Record Structure

```
    !
    ! The User Record
    !
    exec$t_user_record: RECORD
        !
        ! User Fields
        !
        ! The User fields are only used to initialize a user object if no user
        ! object exists. The intent is for the contents of these fields come from
        ! the system wide authorization file
        !
        user_username: string(e$c_max_user_name);          ! User Name
        user_security_profile: e$t_security_profile;       ! User Security Profile from Authorization File
        user_per_user_limits: e$t_quota_limits;            ! Per User Resource Limits
        user_per_job_limits: e$t_quota_limits;             ! Per Job Resource Limits
        user_per_process_limits: e$t_quota_limits;         ! Per Process Resource Limits
        user_thread_priority: k$combined_priority;         ! Default thread priority
        user_access_restrictions: e$t_access_restrictions  ! Users Access Restrictions
    END RECORD;
```

1.3.1.2.2 Get/Set User Information

The *exec$get_user_information* and *exec$set_user_information* system services provide a mechanism to obtain and to modify attributes of the specified user object. Example 1–6 illustrates the interfaces to the user object get/set system services.

Example 1–6: Get/Set User Information System Interface

```
PROCEDURE exec$get_user_information (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN user_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the user object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   user_object_id      if present, the object ID of user object that is to be inspected
!                       otherwise, the user object of the calling thread is assumed
!   user_get_items      item list identifying user object information to be extracted
!
! Return value:
!
!   TBS
!
!--
```

Example 1–6 Cont'd. on next page

Example 1–6 (Cont.): Get/Set User Information System Interface

```
PROCEDURE exec$set_user_information (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN user_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the user object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   user_object_id      if present, the object ID of user object that is to be modified
!                       otherwise, the user object of the calling thread is assumed
!   user_get_items      item list identifying user object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the user object may be inspected or modified. Table 1–1 illustrates the possible item codes and the information read or written when using the item code.

Table 1–1: Get/Set User Information Item Codes

Item Code	Set Action	Get Action
e$i_job_count	error	return u_job_count
e$i_job_ids	error	return object ID's of jobs owned by user
e$i_username	error	return username of user
e$i_security_profile	replace ucb_security_profile	return ucb_security_profile
e$i_quotas	error	return ucb_quotas
e$i_user_limits	replace qual_limits	return qual_limits
e$i_job_limits	replace q_per_job_limits	return q_per_job_limits
e$i_process_limits	replace q_per_process_limits	return q_per_process_limits
e$i_thread_priority	replace ucb_thread_priority	return ucb_thread_priority
e$i_access_restrictions	replace ucb_access_restrictions	return ucb_access_restrictions
e$i_allocation_list	error	return ucb_user_allocation_list

1.3.1.2.3 User Deletion

The *exec$force_exit_user()* system service provides a mechanism for removing an active Mica user from the system. The service effectively causes an entire UJPT hierarchy to be removed, including all jobs, processes, and threads that are directly beneath the user object. Example 1–7 illustrates the interface used to remove a user object from the Mica system.

Example 1–7: User Object Deletion System Interface

```
PROCEDURE exec$force_exit_user (
            IN user_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the UJPT hierarchy whose user object head is user_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   user_object_id      the user object to be removed. If not specified,
!                       then the current user is assumed
!   exit_status         the reason that the user is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.2 The Job Object

The job object appears at the second level of the UJPT hierarchy. Its sole function is to provide a set of resource limits for a collection of processes running as a job. The job object also provides a job level container directory.

The job object is implemented as a system level object in the "JOB$OBJECT_CONTAINER" object container.

1.3.2.1 Object Structure

Each job in the Mica system represents a set of active processes and is responsible for controlling the resources used by those processes.

The job object is split into a job object body and a job control block. The job object body contains the information necessary to maintain its position in a UJPT hierarchy. The job control block contains the information necessary to provide resource management for the job's processes. Example 1–8 illustrates the job object.

Example 1–8: Job Object Structure

```
    !
    ! Job Object Body
    !
    e$t_job_object_body: RECORD
        j_obj_id: e$t_object_id;                            ! Object ID of The job object
        j_user_pointer: POINTER e$t_user_object_body;       ! Referenced Pointer to owning User
        j_job_flags: e$t_job_flags;                         ! Job Flags
        j_job_queue: e$t_linked_list;                       ! List of users jobs
        j_process_queue_mutex: k$dispatcher_object(mutex);  ! Mutex for process management
        j_process_count: integer;                           ! Number Of processes of the job
        j_process_queue_hd: e$t_linked_list;                ! List head of jobs processes
        j_jcb: e$t_job_control_block;                       ! Job Control Block
    END RECORD;
```

Example 1–8 Cont'd. on next page

Example 1–8 (Cont.): Job Object Structure

```
!
! Job Control Block
!
e$t_job_control_block: RECORD
    jcb_job_class: e$t_job_class;                          ! The jobs class
    jcb_usage_and_limits: e$t_quota_usage_and_limits;      ! Current resources used/resource limits
    jcb_job_condir_mutex: k$dispatcher_object(mutex);      ! Job Level Condir mutex
    jcb_job_condir_id: e$t_object_id;                      ! Job Level Container directory ID
                                                           !   visible in jobs context
    jcb_job_alt_condir_id: e$t_object_id;                  ! Job Level Container directory ID
                                                           !   visible in an arbitrary context
    jcb_job_condir_pointer: POINTER e$t_object_header;     ! Pointer to Job Level Condir
    jcb_job_allocation_list: e$t_allocation_list;          ! Objects allocated to the job objects
END RECORD;
```

1.3.2.1.1 Resource Control

The job object maintains resource usage information for itself, in addition to providing a pool of resources to its processes on an as-needed basis. During job-object creation, the *jcb_usage_and_limits.qual_limits* field of the job control-block is set to the value of *q_per_job_limits* from the user control block. The *jcb_usage_and_limits.qual_usage* field of the job control-block is then set to *zero()*, and the *q_usage_and_limits.qual_usage* field of the user control block is incremented by *q_per_job_limits* to reflect the resources allocated to the job. Once this resource shuffling operation has completed, the value of *jcb_usage_and_limits.qual_limits* represents the amount of system resources available to the job object and to all its process.

While the above resource allocation scheme is the normal case, during job creation a parameter specifying the per-job limits for the job can be specified, altering the algorithm. This value simply overrides the value from *q_per_job_limits* in the above example and applies to the newly created job.

1.3.2.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting job objects, and setting and extracting various attributes of a job object.

As part of job object creation, all of the necessary support data structures are created, including a job level container directory and associated kernel mutex dispatcher object.

1.3.2.2.1 Job Creation

The system service *exec$create_job()* causes the creation of a job object, a process object, and a thread object. These objects appear beneath the user object of the calling thread. Example 1–9 illustrates the interface to *exec$create_job()*.

Example 1–9: Job Object Creation System Interface

```
PROCEDURE exec$create_job (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN job_record: exec$t_job_record = DEFAULT;
            IN job_initial_container: exec$t_object_id = DEFAULT;
            IN job_allocation_list: exec$t_allocation_list = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;

            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a job, process, and thread object as specified by the parameters.
!   If no user object exists, then also create a user object.
!
! Arguments:
!
!   object_id                   Object ID of the resulting job object
!   container                   Object container for job object (ignored)
!   name                        Name of job object
!   acl                         ACL to place on job object
!   job_record                  Attributes of the job being created. If not present, then
!                               values are obtained from current user object
!   job_initial_container       Job level object container to be transfered into the job
!                               level container directory for this job. If not present then
!                               container directory comes up empty
!   job_allocation_list         Objects to be allocated to the job object. If not present then
!                               no objects are allocated to the job
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_job()*, it is clear that the *job_record* can have an impact on the structure of the job being created. Example 1–10 illustrates the layout of the *job_record*.

Example 1–10: Job Record Structure

```
    !
    ! The Job Record
    !
    exec$t_job_record: RECORD
        !
        ! Job Fields
        !
        job_class: e$t_job_class;              ! The class of the job being created (i.e. network,batch...)
        !
        ! Per Job Resource limits. This value is used as the
        ! qual_limits value for the job object, and is deducted
        ! from the qual_usage field of the owning user object.
        ! A value of zero() in any one of fields means to use the
        ! corresponding value of the q_per_job_limit from the
        ! user structure
        !
        job_per_job_limits: e$t_quota_limits;
    END RECORD;
```

1.3.2.2.2 Job Deletion

The *exec$force_exit_job()* system service provides a mechanism for removing job objects from the system. The removal of a job has the following system-wide effects:

- All processes beneath the job are removed from the system.
- The amount of resources available to the job (*qual_limits–qual_usage*) is returned to the job's user object by decrementing *qual_usage* in the user object.
- If the job object is the last job owned by its user object, then the user object is removed from the system.

Example 1–11 illustrates the interface to *exec$force_exit_job()*.

Example 1–11: Job Object Deletion System Interface

```
PROCEDURE exec$force_exit_job (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the job object specified by job_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   job_object_id       the job object to be removed. If not specified,
!                       then the current job is assumed
!   exit_status         the reason that the job is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.2.2.3 Get/Set Job Information

The *exec$get_job_information()* and *exec$set_job_information()* system services provide a mechanism to obtain and to modify attributes of the specified job object. Example 1–12 illustrates the interfaces to the job object get/set system services.

Example 1–12: Get/Set Job Information System Interface

```
PROCEDURE exec$get_job_information (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN job_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the job object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   job_object_id       if present, the object ID of job object that is to be inspected
!                       otherwise, the job object of the calling thread is assumed
!   job_get_items       item list identifying job object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_job_information (
            IN job_object_id: exec$t_object_id = DEFAULT;
            IN job_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the job object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   job_object_id       if present, the object ID of job object that is to be modified
!                       otherwise, the job object of the calling thread is assumed
!   job_get_items       item list identifying job object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the job object may be inspected or modified. Table 1–2 illustrates the possible item codes and the information read or written when using the item code.

Table 1–2: Get/Set Job Information Item Codes

Item Code	Set Action	Get Action
e$i_user_id	error	return object ID of jobs user object
e$i_process_count	error	return j_process_count
e$i_process_ids	error	return object ID's of processes owned by job
e$i_usage_and_limits	error	return jcb_usage_and_limits
e$i_job_limits	replace qual_limits	return qual_limits
e$i_job_condir_id	error	return jcb_job_condir_id
e$i_allocation_list	error	return jcb_job_allocation_list
e$i_job_class	error	return jcb_job_class

1.3.3 The Process Object

The Process object appears at the third level of the UJPT hierarchy. Its primary function is to provide address space support and program image support for a set of execution threads, and to manage the set of process-level objects. The process object is the target of all accounting information. The process object can also act as a focal point for control operations.

There can be multiple processes in a job. Processes created as a result of job creation are *top level* processes. Once established, a process may cause the creation of other processes. These new processes are *sub–processes*, or *child processes*. Their creating processes are referred to as *parent processes*.

The Process object is implemented as a system level object in the "PROCESS$OBJECT_CONTAINER" object container.

1.3.3.1 Object Structure

Each process in the Mica system represents a set of execution threads and in some cases a set of sub-processes. The process object is responsible for managing the address spaces of its execution threads and for controlling the resource allocation limits of its execution threads.

The process object is split into a process object body and a process control block. The process object body contains the information necessary to maintain its position in the UJPT hierarchy, a task which includes coordinating its sub-process objects. The process control block contains the information necessary to manage the address space, to control the resource usage, and to pool accounting information of all of its execution threads. Example 1–13 illustrates the process object.

Example 1–13: Process Object Structure

```
!
! Process Object Body
!
e$t_process_object_body: RECORD
    p_obj_id: e$t_object_id;                                ! Obje9ct ID of process object
    p_job_pointer: POINTER e$t_job;                         ! Referenced pointer to owning job
    p_parent_process: POINTER e$t_process;                  ! Referenced pointer to owning process, or NIL
    p_process_flags: e$t_process_flags;                     ! Process Flags
    p_process_queue: e$t_linked_list;                       ! List of jobs processes
    p_sub_process_queue: e$t_linked_list;                   ! List of parents sub-processes
    p_thread_queue_mutex: k$dispatcher_object(mutex);       ! Mutex for thread management
    p_thread_count: integer;                                ! Number of threads of the process
    p_thread_queue_hd: e$t_linked_list;                     ! List head of processes threads
    p_sub_process_queue_mutex: k$dispatcher_object(mutex);  ! Mutex for sub-process management
    p_sub_process_count: integer;                           ! Number of sub-processes of the process
    p_sub_process_queue_hd: e$t_linked_list;                ! List head of processes sub-processes
    p_pcb: e$t_process_control_block;                       ! Process Control Block
END RECORD;

!
! Process Control Block
!
e$t_process_control_block: RECORD
    pcb_usage_and_limits: e$t_quota_usage_and_limits;       ! Current resources used/resource limits
    pcb_process_condir_id: e$t_object_id;                   ! Process Level Container directory ID
                                                            !   visible in an processes context
    pcb_process_alt_condir_id: e$t_object_id;               ! Process  Level Container directory ID
                                                            !   visible in an arbitrary context
    pcb_accounting: e$t_accounting_summary;                 ! Process accounting summary
    pcb_pcr_base: POINTER e$t_process_control_region;       ! User Readable Process Control Region
    pcb_process_control_pte: mm$pte;                        ! Prototype PTE for seg 1 page table page
    pcb_ptbr: POINTER k$page_table;                         ! Pointer to page table
    pcb_kernel_process_block: k$process;                    ! Kernel Process Block
    pcb_exit_status_id: e$t_object_id;                      ! Exit Status Object ID for process
    pcb_exit_status_ptr: POINTER e$t_exit_status_body;      ! Exit Status for Process
    pcb_process_allocation_list: e$t_allocation_list;       ! objects allocated to the process object
    !
    ! Object Architecture Defined Container Directory Vector
    !
    pcb_condir_mutex: ARRAY [e$t_level_type] OF POINTER k$dispatcher_object(mutex);
    pcb_condir_address: ARRAY [e$t_level_type] OF POINTER e$t_object_header;
END RECORD;
```

Example 1–13 Cont'd. on next page

Example 1–13 (Cont.): Process Object Structure

```
!
! Process Control Region
!
! The process control region appears in the processes address space as user read only/ system
! read write.
!
e$t_process_control_region: RECORD
    pcr_image_name: string(e$c_max_image_name);          ! process image name
    pcr_number_running_threads: e$t_resource_counter;    ! number of running threads for this process
    pcr_object_id: e$t_object_id;                        ! process object id - duplicate of p_obj_id
    pcr_exit_handlers: e$t_exit_handlers;                ! Process exit Handlers
    pcr_exec_dispatch_table: e$t_dispatch_table;         ! Executive routines dispatch table
END RECORD;
```

1.3.3.1.1 Resource Control

The process object maintains resource usage information for all of its threads. Unlike the job object, the process object's *qual_usage* values represent resources actively in use by its threads. Each time one of the process objects threads consume paged pool, the *qu_paged_pool_in_use* field is incremented by the amount of pool actually used. This action is called *pooling* the resource usage from the thread level to the process level.

During process object creation, the *pcb_usage_and_limits.qual_limits* field of the process control block is set to the value of *q_per_process_limits* from the user control block. The *pcb_usage_and_limits.qual_usage* field of the process control block is then set to *zero()*, and the *q_usage_and_limits.qual_usage* field of the job control block is incremented by *q_per_process_limits* to reflect the resources allocated to the process. Once this resource shuffling operation has completed, the value of *pcb_usage_and_limits.qual_limits* represents the amount of system resources available to the process object which can be consumed by all its thread objects.

While the above resource allocation scheme is the normal case, during process creation a parameter specifying the per-process limits for the process can be specified, altering the algorithm. This value simply overrides the value from *q_per_process_limits* in the above example and applies to the newly created process.

1.3.3.1.2 Process Accounting

The process object maintains accounting information for all of its threads. Process accounting information is pooled from the thread level to the process level. Example 1–14 illustrates the types of information accounted for at the process level in the Mica system.

NOTE

Process accounting information is recorded with interlocked instructions, such that the information is always maintained in an up-to-date state.

Example 1–14: Process Accounting Structure

```
!
! Process Accounting Summary
!
! The final accounting record contains this information in TLV format
! in addition to fields identifying the process, image name, user ...
!
e$t_accounting_summary: RECORD
    acct_cpu_cycles: e$t_counter;                    ! Number of cycles used by the process
    acct_total_page_faults: e$t_counter;             ! Total number of page faults
    acct_hard_page_faults: e$t_counter;              ! Number of page faults for non resident pages
    acct_soft_page_faults: e$t_counter;              ! Number of page faults fixed from reclaim list
    acct_dzro_page_faults: e$t_counter;              ! Number of demand zero page faults
    acct_com_page_faults: e$t_counter;               ! Number of copy on modify page faults
    acct_peak_virtual_memory: e$t_counter;           ! Peak virtual memory size
    acct_peak_working_set_size: e$t_counter;         ! Peak working set size
    acct_start_time: e$t_time_value;                 ! Start time of process
    acct_end_time: e$t_time_value;                   ! End time of process
    acct_page_file_usage: e$t_counter;               ! Peak page file usage
    acct_paged_pool_usage: e$t_counter;              ! Peak paged pool usage
    acct_non_paged_pool_usage: e$t_counter;          ! Peak non paged pool usage
    !
    ! IO Accounting
    ! Request IO's are counted once.
    ! Each FPU that passes on an IRP (execute_io's) must also record the transfer
    ! by incrementing the counter for its class of FPU
    !
    acct_request_io_count: e$t_counter;              ! Number of request_io's
    acct_execute_io_count: ARRAY[e$fpu_class]        ! Number of execute_io's per fpu class
                           OF e$t_counter;
END RECORD;
```

1.3.3.2 Functional Interface

The Mica executive provides entry points capable of creating and deleting process objects, setting and extracting various attributes of a Process object, and performing control operations on all threads of a process. Control operations are Suspend/Resume Process, Hibernate/Wake Process, and Signal Process.

As part of process-object creation, all of the necessary support data structures are created, including a read only process control region (PCR), and a process-level object-container directory. The PCR is part of the process's user-mode read-only address space. The Mica executive places information in the PCR so that the process can read it without entering the system.

1.3.3.2.1 Process Creation

The *exec$create_process()* system service extends an existing UJPT hierarchy by causing the creation of a process object and a thread object. The newly created process object becomes a sub-process of the process above the calling thread. Example 1–15 illustrates the interface to *exec$create_process()*.

Example 1–15: Process Object Creation System Interface

```
PROCEDURE exec$create_process (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN process_record: exec$t_process_record;
            IN process_public_container: exec$t_object_id = DEFAULT;
            IN process_private_container: exec$t_object_id = DEFAULT;
            IN process_allocation_list: exec$t_allocation_list = DEFAULT;

            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;

            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a Process and thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                   Object ID of the resulting process object
!   container                   Object container for process object (ignored)
!   name                        Name of process object
!   acl                         ACL to place on process object
!   process_record              Attributes of the process being created
!   process_public_container    Process level public container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_private_container   Process level private container to be transfered into the process
!                               level container directory for the process. If not present then
!                               container comes up empty.
!   process_allocation_list     Objects to be allocated to the process object. If not present then
!                               no objects are allocated to the process
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_data_block           Arbitrary data block passed to initial thread. Pointer in TCR, if
!                               pointer is NIL, then no data block was passed
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_status               Exit status object to be bound to the initial thread. If not present
!                               then the thread is created without an exit status object
!   process_status              TBS
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_process()*, it is clear that the *process_record* has an impact on the structure of the process being created. Example 1–16 illustrates the layout of the *process_record*.

Example 1–16: Process Record Structure

```
    !
    !The Process Record
    !
    exec$t_process_record: RECORD
        process_status_object: e$t_object_id;        ! Object ID of processes status object
        process_image_name: string(e$c_max_image_name); ! Image name for process being created
        !
        ! Per Process Resource limits. This value is used as the
        ! qual_limits value for the process object, and is deducted
        ! from the qual_usage field of the owning job object.
        ! A value of zero() in any one of fields means to use the
        ! corresponding value of the q_per_process_limit from the
        ! user structure
        !
        process_per_process_limits: e$t_quota_limits; ! Resource limits for this process
    END RECORD;
```

1.3.3.2.2 Process Deletion

The *exec$force_exit_process()* system service provides a mechanism for removing process objects from the system. The removal of a process has the following system-wide effects:

- All threads of the process are removed from the system.
- The amount of resources available to the process (*qual_limits–qual_usage*) is returned to the processes job object by decrementing *qual_usage* in the job object.
- If the process object is the last process owned by its job object, then the job object is removed from the system.

Example 1–17 illustrates the interface to *exec$force_exit_process()*.

Example 1–17: Process Object Deletion System Interface

```
PROCEDURE exec$force_exit_process (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Causes the Process object specified by process_object_id to
!   be removed from the Mica system
!
! Arguments:
!
!   process_object_id   the process object to be removed. If not specified,
!                       then the current process is assumed
!   exit_status         the reason that the process is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.3 Get/Set Process Information

The *exec$get_process_information()* and *exec$set_process_information()* system services provide a mechanism to obtain and modify attributes of the specified process object. Example 1–18 illustrates the interfaces to the process object get/set system services.

Example 1–18: Get/Set Process Information System Interface

```
PROCEDURE exec$get_process_information (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN process_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the process object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   process_object_id   if present, the object ID of process object that is to be inspected
!                       otherwise, the process object of the calling thread is assumed
!   process_get_items   item list identifying process object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_process_information (
            IN process_object_id: exec$t_object_id = DEFAULT;
            IN process_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the process object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   process_object_id   if present, the object ID of process object that is to be modified
!                       otherwise, the process object of the calling thread is assumed
!   process_get_items   item list identifying process object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the process object may be inspected or modified. Table 1–3 illustrates the possible item codes and the information read or written by using the item code.

Table 1–3: Get/Set Process Information Item Codes

Item Code	Set Action	Get Action
e$i_job_id	error	return object ID of processes job object
e$i_parent_id	error	return object ID of processes parent process object
e$i_sub_process_count	error	return p_sub_process_count
e$i_sub_process_ids	error	return object ID's of sub_processes owned by process
e$i_thread_count	error	return p_thread_count
e$i_thread_ids	error	return object ID's of threads owned by process
e$i_usage_and_limits	error	return pcb_usage_and_limits
e$i_process_limits	replace qual_limits	return qual_limits
e$i_process_condir_id	error	return pcb_process_condir_id
e$i_accounting	error	return pcb_accounting
e$i_pcr_base	error	return pcb_pcr_base
e$i_allocation_list	error	return pcb_process_allocation_list

1.3.3.2.4 Process Control Operations

Two process control operations exist in the Mica system to coordinate the execution of all threads of a process. The first provides a primitive which can alter the execution flow of another process by causing a condition to be raised in the target process. The second provides primitives to *block* and *unblock* the execution of the target process. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the process to continue via user-mode AST routines, while the other class disables user-mode activity.

1.3.3.2.4.1 Process Signaling

The *exec$signal_process()* system service provides a mechanism to alter the execution flow of all threads of the process by causing a *condition* to be raised in the threads context.

NOTE

Process signalling is implemented through user-mode ASTs; therefore, if ASTs are disabled then so are signals.

Example 1–19 illustrates the interface to *exec$signal_process()*.

Example 1–19: Signal Process System Interface

```
PROCEDURE exec$signal_process (
            IN object_id: exec$t_object_id;
            IN condition_value: exec$t_condition_value;
            IN argument: longword CONFORM = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause a condition of type condition_value to be raised in all threads owned by the process
!   specified by object_id. The condition handler is passed argument.
!
! Arguments:
!
!   object_id           the object_id of the process to be signaled
!   condition_value     A descriptor for the condition to be raised in all threads
!                       of the target process
!   argument            If present, the value that is passed to the condition handler
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.4.2 Process Hibernate/Wake

The *exec$hibernate_process()* and *exec$wake_process()* provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing hibernate-event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines; kernel-mode ASTs remain enabled. The unblock of the process is implemented by setting the auto-clearing hibernate event object within the thread control block of all threads of the target process. Example 1–20 illustrates the interfaces to *exec$hibernate_process()* and *exec$wake_process()*.

Example 1–20: Hibernate/Wake Process System Interface

```
!
! Hibernate Process
!
PROCEDURE exec$hibernate_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to issue a wait on the
!   auto-clearing hibernate event object in their TCB. User mode AST's remain enabled
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--

!
! Wake Process
!
PROCEDURE exec$wake_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;
```

Example 1–20 Cont'd. on next page

Example 1–20 (Cont.): Hibernate/Wake Process System Interface

```
!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to have their waits on the
!   auto-clearing hibernate event object in their TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--
```

1.3.3.2.4.3 Process Suspend/Resume

The *exec$suspend_process()* and *exec$resume_process()* provide a mechanism to block and unblock the execution flow of all threads within the target process. The block is implemented by causing all threads within the target process to issue a wait on the auto-clearing suspend event object within the thread control block. During the block, no user-mode activity is possible; only kernel-mode normal and special AST routines may be executed. The unblock of the process is implemented by setting the auto-clearing suspend event object within the thread control block of all threads of the target process. Example 1–21 illustrates the interfaces to *exec$suspend_process()* and *exec$resume_process()*.

Example 1–21: Suspend/Resume Process System Interface

```
!
! Suspend Process
!
PROCEDURE exec$suspend_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to issue a wait on the
!   auto-clearing suspend event object in their TCB. User mode AST's are disabled.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--

!
! Resume Process
!
PROCEDURE exec$resume_process (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;
```

Example 1–21 Cont'd. on next page

Example 1–21 (Cont.): Suspend/Resume Process System Interface

```
!++
!
! Routine description:
!
!   Cause all threads owned by the process specified by object_id to have their waits on the
!   auto-clearing suspend event object in their TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target process
!
! Return value:
!
!   TBS
!
!--
```

1.3.4 The Thread Object

The thread object appears at the lowest level of the UJPT hierarchy. Its primary function is to provide a thread of execution.

In addition, the thread object has the following functions:

- It is the schedulable entity in the Mica system.
- It maintains the processor state as it executes the program steps of an image.
- It is the consumer of resources. All accounting and resource limitation data structures reside in the thread's process object, with the thread's activity pooled to the process level.
- It can act as a focal point for synchronization.

The thread object is implemented as a system level object in the "THREAD$OBJECT_CONTAINER" object container.

1.3.4.1 Object Structure

The thread object maintains the state of the processor as it moves through the program steps of the program image mapped into its processes address space.

The thread object is split into a thread object body and a thread control block. The thread object body contains information necessary to maintain the thread's position within the UJPT hierarchy. The thread control block contains the information necessary to move the execution thread through the steps of the program image. Example 1–22 illustrates the thread object.

Example 1–22: Thread Object Structure

```
!
! Thread Object Body
!
e$t_thread: RECORD
    t_obj_id: e$t_object_id;                                ! Object ID of thread object
    t_process_pointer: POINTER e$t_process;                 ! Referenced pointer to owning process
    t_thread_flags: e$t_thread_flags;                       ! Thread Flags
    t_thread_queue: e$t_linked_list;                        ! List of processes threads
    t_tcb: e$t_thread_control_block;                        ! Thread Control Block
END RECORD;

!
! Thread Control Block
!
e$t_thread_control_block: RECORD
    tcb_previous_mode: e$t_processor_status;                ! saved processor status
    tcb_thread_context: e$t_thread_context;                 ! Processor State of Thread
    tcb_kernel_thread_block: k$dispatcher_object(thread);   ! Kernel Thread Block
    tcb_hibernate_event: k$dispatcher_object(event);        ! auto-clearing hibernate event
    tcb_suspend_event: k$dispatcher_object(event);          ! auto-clearing suspend event
    tcb_pcb_pointer: POINTER e$t_process_control_block;     ! Pointer to PCB
    tcb_tcr_base: POINTER e$t_thread_control_region;        ! Pointer to TCR
    tcb_exit_status_id: e$t_object_id;                      ! Exit Status Object ID for Thread
    tcb_exit_status_ptr: POINTER e$t_exit_status_body;      ! Exit Status for Thread
    tcb_exit_status_value: e$t_exit_status;                 ! Exit Status
    tcb_security_profile: e$t_security_profile;             ! The threads security profile
    tcb_thread_allocation_list: e$t_allocation_list;        ! Objects allocated to the thread object
    !
    ! Memory Management Events
    !
    tcb_initial_page_event: k$dispatcher_object(event);     ! Memory Management
    tcb_secondary_page_event: k$dispatcher_object(event);   ! Memory Management
    tcb_current_page_event: integer;                        ! Memory Management
    !
    ! I/O
    !
    tcb_io_synchronization_event: k$dispatcher_object(event);   ! I/O synchronization event
    tcb_irp_list_head: e$t_linked_list;                         ! I/O Request Packet List Head
    tcb_cancel_io: boolean;                                     ! Cancel io by thread in progress
    tcb_cancel_event: k$dispatcher_object(event);               ! Cancel io synchronization

END RECORD;

!
! Thread Context
!
e$t_thread_context: RECORD
    tc_priviledged_context_block: k$hwpcb;                              ! Hardware Privileged Context Block
    tc_general_purpose_registers: POINTER e$t_general_purpose_registers;! Scalar Register Set
    tc_vector_registers: POINTER e$t_vector_registers;                  ! Vector Register Set
END RECORD;

!
! Thread Control Region
!
! The thread control region appears in the processes address space as user read only/ system
! read write
!
e$t_thread_control_region: RECORD
    tcr_object_id: e$t_object_id;                           ! Object ID of this thread
    tcr_pcr_pointer: POINTER e$t_process_control_region;    ! Pointer to process control region
    tcr_start_address: e$t_thread_entry_point;              ! initial start address of thread
    tcr_initial_sp: e$t_scalar_register;                    ! Initial Value of Stack Pointer
    tcr_stack_limit: e$t_scalar_register;                   ! Primary Stack Limit
    tcr_stack_base: e$t_scalar_register;                    ! Primary Stack Base
    tcr_condition_initial_sp: e$t_scalar_register;          ! Initial Value of Condition Stack Ptr
    tcr_condition_stack_limit: e$t_scalar_register;         ! Condition Stack Limit
    tcr_condition_stack_base: e$t_scalar_register;          ! Condition Stack Base
    tcr_exit_handlers: e$t_exit_handlers;                   ! Thread exit handlers
    tcr_vectored_handlers: e$t_vectored_handlers;           ! Entry descriptors for vectored
                                                            ! condition handlers
    !
    ! Initial Thread Parameters
    !
    tcr_block_data: POINTER anytype;                        ! Initial thread data block or NIL
    tcr_block_data_length: integer;                         ! Byte length of data block rounded to quadword
    tcr_parameter1: e$t_thread_parameter;                   ! Immediate parameter / or zero()
```

Example 1–22 Cont'd. on next page

Example 1–22 (Cont.): Thread Object Structure

```
        tcr_parameter2: e$t_thread_parameter;                    ! Immediate parameter / or zero()
    END RECORD;

    !
    ! Immediate Parameter
    !
    e$t_thread_parameter: e$t_register;      ! Same size as a machine register

    !
    ! Thread Entry Point
    !
    e$t_thread_entry_point: PROCEDURE();
```

1.3.4.2 Functional Interface

The Mica executive provides entry points capable of creating, deleting, and controlling thread objects, in addition to setting and extracting various attributes of a thread object.

Thread object control services are Suspend/Resume thread, Hibernate/Wake thread, and Signal thread.

As part of Thread object creation, all of the necessary support data structures are created including the read-only thread control region (TCR), the read/write thread environment block (TEB), and user and kernel stacks. The TCR is part of the process's user-mode read-only address space. The Mica executive places information in the TCR so that the thread can read it without entering the system. The TEB is part of the user-mode thread architecture. The MICA executive initializes the TEB to point to the TCR.

1.3.4.2.1 Thread Creation

The *exec$create_thread()* system service extends an existing UJPT hierarchy by causing the creation of a thread object. The newly created thread object begins execution within the address space of its process at a start address passed to the system interface. Example 1–23 illustrates the interface to *exec$create_thread()*.

Example 1–23: Thread Object Creation System Interface

```
PROCEDURE exec$create_thread (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;

            IN thread_procedure: exec$t_thread_entry_point;
            IN thread_record: exec$t_thread_record = DEFAULT;
            IN thread_allocation_list: exec$t_allocation_list = DEFAULT;
            IN thread_data_block: quadword_data(*) CONFORM OPTIONAL;
            IN thread_immediate_parameter1: exec$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: exec$t_thread_parameter = DEFAULT;
            IN thread_status: exec$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                 Object ID of the resulting process object
!   container                 Object container for thread object (ignored)
!   name                      Name of thread object
!   acl                       ACL to place on thread object
!   thread_record             Attributes of the thread being created
!   thread_allocation_list    Objects to be allocated to the thread object. If not present then
!                             no objects are allocated to the thread
!   thread_data_block         Arbitrary data block passed to initial thread. Pointer in TCR, if
!                             pointer is NIL, then no data block was passed
```

Example 1–23 Cont'd. on next page

Example 1–23 (Cont.): Thread Object Creation System Interface

```
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_procedure            pointer to thread entry point entry descriptor
!   thread_status               Exit status object to be bound to the thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

From the interface to *exec$create_thread()*, it is clear that the *thread_record* can have an impact on the structure of the thread being created. Example 1–24 illustrates the layout of the *thread_record*.

Example 1–24: Thread Record Structure

```
    !
    ! The thread record
    !
    e$type_thread_record: RECORD
        thread_stack_size: integer;                 ! If 0 then system wide default
        thread_priority: k$combined_priority;       ! initial thread priority if all 0 then default
        thread_affinity: k$affinity;                ! processor affinity If all 0 then all processors
    END RECORD;
```

1.3.4.2.2 Thread Deletion

Thread deletion is the action which causes the removal of a thread object. The Mica system provides two mechanisms for deleting thread objects. The first mechanism, *simple exit*, will in some cases not cause the thread object to be removed; however, it is the normal path for thread exit when a thread wants to exit. The second mechanism, *forced exit*, will cause the thread object to be removed unconditionally. The forced exit path occurs when any thread wants the specified thread to exit.

The deletion of a thread object causes the thread's exit handlers to execute. In the simple exit case, exit handlers may run indefinitely, possibly never completing; thus, thread object may not occur. In the forced exit case, the thread's exit handlers are executed with a CPU time limit. If a time limit is exceeded, the next handler is executed. This technique guarantees that all exit handlers will be invoked and that afterwards thread object deletion will proceed. The *exec$exit_thread()* system interface provides the simple exit functionality. The *exec$force_exit_thread()* system service provides the forced-exit functionality.

When the last thread of a process is deleted, the process object is removed from the system.

Example 1–25 illustrates the interfaces to *exec$exit_thread()*, and *exec$force_exit_thread()*.

Example 1–25: Thread Object Deletion System Interfaces

```
!
! Thread Exit System Service
!
PROCEDURE exec$exit_thread (
            IN exit_status: exec$t_exit_status;
            );

!++
!
! Routine description:
!
!   Cause the deletion of the calling thread object. Place
!   thread_status in the threads tcb at tcb_exit_status_value
!
! Arguments:
!
!   thread_status       the exit status of the thread
!
! Return value:
!
!   none
!
!--

!
! Thread Force Exit System Service
!
PROCEDURE exec$force_exit_thread (
            IN object_id: exec$t_object_id = DEFAULT;
            IN exit_status: exec$t_exit_status;
            ) RETURNS status;

!++
!
! Routine description:
!
!   Cause the deletion of the thread object specified by object_id
!
! Arguments:
!
!   object_id       the object ID of the thread object being deleted. If not specified,
!                   then the calling thread is assumed
!   exit_status     the reason that the thread is force-exiting
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.3 Get/Set Thread Information

The *exec$get_thread_information()* and *exec$set_thread_information()* system services provide a mechanism to obtain and modify attributes of the specified thread object. Example 1–26 illustrates the interfaces to the thread object get/set system services.

Example 1–26: Get/Set Thread Information System Interface

```
PROCEDURE exec$get_thread_information (
            IN thread_object_id: exec$t_object_id = DEFAULT;
            IN thread_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Return information about the thread object to the caller. The
!   information returned is item list driven
!
! Arguments:
!
!   thread_object_id   if present, the object id of thread object that is to be inspected
!                      otherwise, the calling thread is assumed
!   thread_get_items   item list identifying thread object information to be extracted
!
! Return value:
!
!   TBS
!
!--

PROCEDURE exec$set_thread_information (
            IN thread_object_id: exec$t_object_id = DEFAULT;
            IN thread_get_items: exec$t_item_list;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Modify information in the thread object. The
!   information to be modified is item list driven
!
! Arguments:
!
!   thread_object_id   if present, the object ID of thread object that is to be modified
!                      otherwise, the calling thread is assumed
!   thread_get_items   item list identifying thread object information to be modified
!
! Return value:
!
!   TBS
!
!--
```

Only certain pieces of the thread object may be inspected or modified. Table 1–4 illustrates the possible item codes and the information read or written by using the item code.

Table 1–4: Get/Set Thread Information Item Codes

Item Code	Set Action	Get Action
e$i_process_id	error	return object ID of threads process object
e$i_tcr_base	error	return tcb_tcr_base
e$i_tcr_start_address	set tcr_start_address	error
e$i_allocation_list	error	return tcb_thread_allocation_list

1.3.4.2.4 Thread Control Operations

Two thread control operations exist in the Mica system to coordinate the execution of threads. The first provides a primitive which can alter the execution flow of another thread by causing a condition to be raised in the target thread. The second provides primitives to block and unblock the execution of the target thread. In this latter technique, there are two classes of control operations. One class allows user-mode activity within the thread to continue via user-mode AST routines, while the other class disables user-mode activity.

1.3.4.2.4.1 Thread Signaling

The *exec$signal_thread()* system service provides a mechanism to alter the execution flow of a thread by causing a condition to be raised in the context of the target thread.

NOTE

The thread signalling mechanism is implemented through user-mode ASTs; therefore, if ASTs are disabled, then so are signals.

Example 1–27 illustrates the interface to *exec$signal_thread()*.

Example 1–27: Signal Thread System Interface

```
PROCEDURE exec$signal_thread (
            IN object_id: exec$t_object_id;
            IN condition_value: exec$t_condition_value;
            IN argument: longword CONFORM = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause a condition of type condition_value to be raised in the thread
!   specified by object_id. The condition handler is passed argument.
!
! Arguments:
!
!   object_id           the object_id of the thread to be signaled
!   condition_value     A descriptor for the condition to be raised in the target thread
!   argument            If present, the value that is passed to the condition handler
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.2 Thread Hibernate/Wake

The *exec$hibernate_thread()* and *exec$wake_thread()* provide a mechanism to block and unblock the execution flow of a thread. The block is implemented by causing the thread to issue a wait on the auto-clearing hibernate event object within the thread control block. During the block, the only user-mode activity that is allowed is execution within user-mode AST routines. The unblock of the thread is implemented by setting the auto-clearing hibernate event object within the thread control block. Example 1–28 illustrates the interfaces to *exec$hibernate_thread()* and *exec$wake_thread()*.

Example 1–28: Hibernate/Wake Thread System Interface

```
!
! Hibernate Thread
!
PROCEDURE exec$hibernate_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to issue a wait on the
!   auto-clearing hibernate event object in the TCB. User mode AST's remain enabled
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--

!
! Wake Thread
!
PROCEDURE exec$wake_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to have the wait on the
!   auto-clearing hibernate event object in the TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.3 Thread Suspend/Resume

The *exec$suspend_thread()* and *exec$resume_thread()* provide a mechanism to block and unblock the execution flow of the target thread. The block is implemented by causing the thread to issue a wait on the auto-clearing suspend-event object within the thread control block. During the block, no user-mode activity is possible. Only kernel-mode normal and special AST routines may be executed. The unblock of the thread is implemented by setting the auto-clearing suspend-event object within the thread control block. Example 1–29 illustrates the interfaces to *exec$suspend_thread()* and *exec$resume_thread()*.

Example 1–29: Suspend/Resume Thread System Interface

```
!
! Suspend Thread
!
PROCEDURE exec$suspend_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to issue a wait on the
!   auto-clearing suspend event object in the TCB. User mode AST's are disabled.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--

!
! Resume Thread
!
PROCEDURE exec$resume_thread (
            IN object_id: exec$t_object_id;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Cause the thread specified by object_id to have the wait on the
!   auto-clearing suspend event object in the TCB to be satisfied by setting the event.
!
! Arguments:
!
!   object_id       object ID of target thread
!
! Return value:
!
!   TBS
!
!--
```

1.3.4.2.4.4 Hibernate and Suspend Comparison

Both the *exec$hibernate_thread()* system service, and the *exec$suspend_thread()* system service block the execution of the specified thread. The difference between these two types of blocked states is the ability of the blocked thread to receive and execute in the context of user-mode ASTs. Threads that are blocked due to the *exec$hibernate_thread()* system service are able to receive and execute in the context of user-mode ASTs; threads that are blocked due to the *exec$suspend_thread()* system service are not.

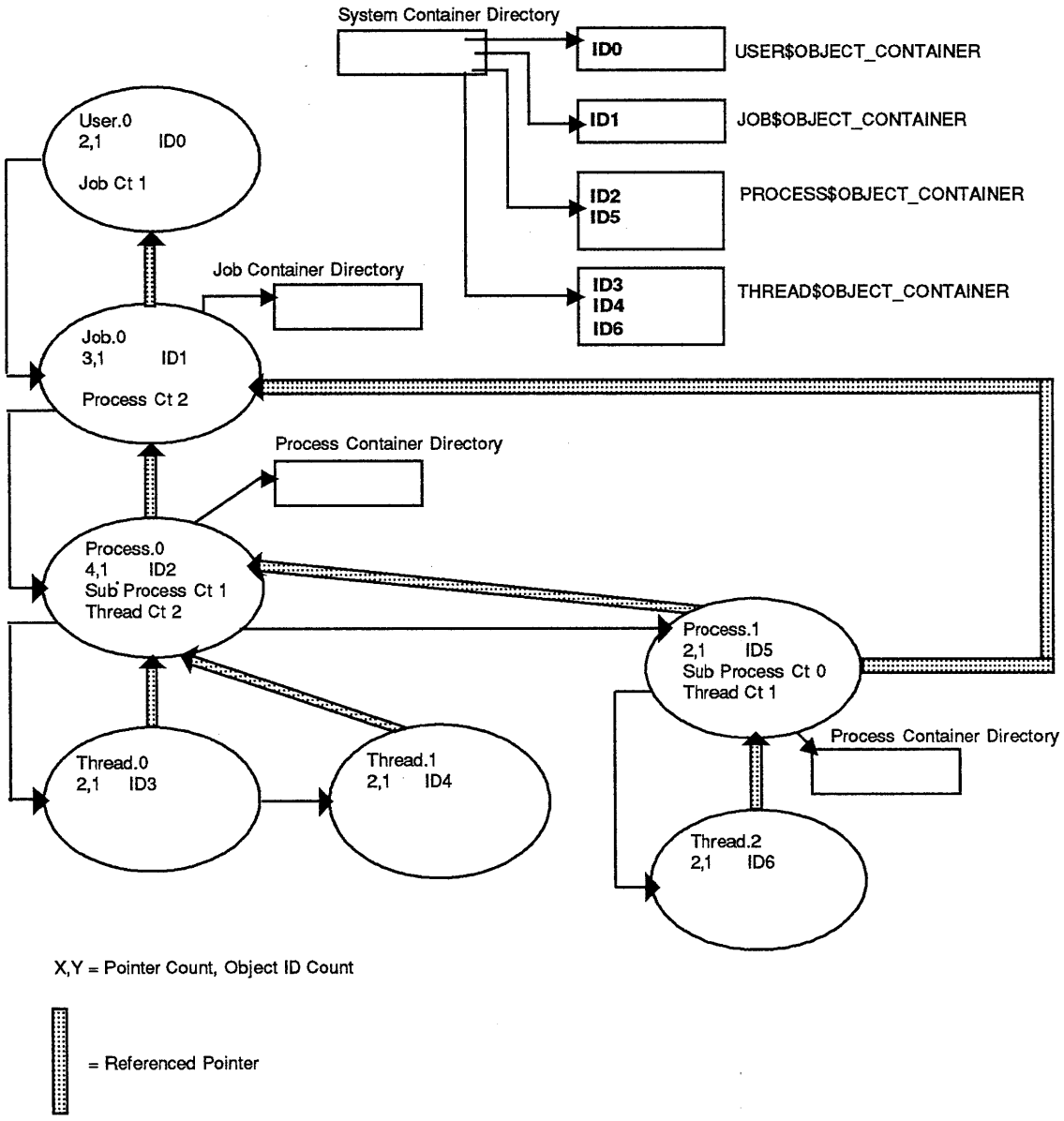
1.4 UJPT Object Linkages

The UJPT hierarchy is bound together through the existence of object IDs and referenced pointers. The following section describes the implementation of the object linkages, the steps of hierarchy creation, and the actions which lead to the collapse of a UJPT hierarchy. This section does not describe process or thread creation in terms of address space creation or the intricate details of kernel, memory management, or object architecture interactions.

1.4.1 Linkage Structure

The UJPT object linkage structure requires that objects in lower levels of the hierarchy point to the object immediately above them using a referenced pointer. The reference pointer guarantees the existence of the higher-level object for the life of the lower-level object. Figure 1–1 illustrates a complex UJPT hierarchy consisting of a user object, a job object, and a process object consisting of two immediate threads and a sub-process object with a single thread.

Figure 1–1: Complex UJPT Hierarchy



ZS-24347-87

1.4.2 Hierarchy Creation

The creation of a UJPT hierarchy is triggered by the *exec$create_user()* system service. At this time, a hierarchy is either created or extended, depending on the existence of a user object representing the Mica user specified in the *user_record.user_username* field of the *user_record* parameter.

The following steps occur during the creation of a UJPT hierarchy.

1. Determine if a user object exists for *user_record.user_username*. If the object exists, then obtain a referenced pointer to the user object. Otherwise, create the user object in the system container directory, initialize the user object with the information from the *user_record* and then obtain a referenced pointer to the user object.
2. Create the job object in the system container directory and obtain a referenced pointer to the job object. Initialize the job object according to the following tasks.
   - Set *j_obj_id* equal to the object ID of the job being created.
   - Set *j_user_pointer* to the referenced pointer of the proper user object.
   - Link the job object to the user object's *u_job_queue_hd*, and initialize the *j_process...* fields of the job object.
   - Create the job-level container directory, and populate it with the *job_initial_container* parameter.
3. Create the process object in the system container directory and obtain a referenced pointer to the process object. Initialize the process object according to the following tasks.
   - Set *p_obj_id* equal to the object ID of the process being created.
   - Set *p_job_pointer* to the referenced pointer of the proper job object.
   - Link the process object to the job object's *j_process_queue_hd*.
   - Initialize the *p_thread...* fields and *p_sub_process...* fields of the job object.
   - Create the process level container directory, and populate it with the *process_public_container* parameter and the *process_private_container* parameter.
4. Create the thread object in the system container directory.
5. Obtain a referenced pointer to the thread object.
6. Initialize the thread object such that *t_obj_id* contains the object ID of the thread, and *t_process_pointer* contains the referenced pointer to the proper process object.
7. Link the thread object to the process object's p_thread_queue_hd.

1.4.3 Hierarchy Collapse/Deletion

The collapse of a UJPT hierarchy can be triggered by force-exiting any component of a hierarchy. The ultimate collapse is always the result of a thread's exit, whether it be a forced exit or a voluntary exit.

The forced exit of a component in the UJPT hierarchy eventually causes all threads beneath that object to exit. The following actions occur during a thread exit.

- If the exiting thread is the last thread in its process, then cause the process to exit by removing its object ID.
- If the exiting process has any sub-processes, then cause its sub-processes to exit by force-exiting them.
- If the exiting process is the last process in its job, then cause the job to exit by removing its object ID.
- If the exiting job is the last job in its user, then cause the user to exit by removing its object ID.

1.4.3.1 Force-Exit Routines

Each component of the hierarchy provides a force-exit interface as part of its primitive object service routines. The basic action performed in these routines is the forced exit of the object's sub-objects.

1.4.3.1.1 User-Object Force-Exit Routine

The user-object force-exit routine is responsible for causing the forced exit of all of its job objects. This is implemented by setting its force-exit in-progress flag, and looping over the linked list of its job objects headed by *u_job_queue_hd* and a force-exit of that job via *e$force_exit_job()*.

1.4.3.1.2 Job-Object Force-Exit Routine

The job object force-exit routine is responsible for causing the forced exit of all of its process objects. This is implemented by setting its force-exit in progress flag, looping over the linked list of its process objects headed by *j_process_queue_hd*, and causing a forced exit of that process via *e$force_exit_process()*.

1.4.3.1.3 Process-Object Force-Exit Routine

The process object force-exit routine is responsible for causing the removal of all of its thread objects and sub-processes represented as process objects. This is implemented by setting its force-exit in progress flag, and looping over the linked list of its thread objects headed by *p_thread_queue_hd* and causing a force-exit of that thread via *e$force_exit_thread()*. Then the routine loops over the linked list of sub-processes headed by *p_sub_process_queue_hd* and causes a forced exit of that process via *e$force_exit_process()*.

1.4.3.1.4 Thread Object Force Exit Routine

The routine occurs in two phases. The first phase is to cleanly enter the exiting thread's context to begin the thread exit. The second phase is to complete the exit of the thread by calling *exec$exit_thread()*, an action which starts the second phase of hierarchy collapse and finally brings the "exiting" thread out of the system. Before starting the forced-exit processing, the force-exit in-progress flag is set in the thread object.

During a thread forced-exit, there is a moment when control is returned to the original caller of *exec$exit_thread()* even though the thread to be exited is still part of the system. The exit is considered complete with respect to the caller after the system has delivered an AST to the exiting thread that will cause the thread itself to exit. The exit is complete with respect to the exiting thread once the thread has issued its call to *k$terminate_thread()*;

1.4.3.1.4.1 Thread Context Entry

To force-exit a thread, that thread's context must be entered in a controlled manner in a "trusted" user-mode routine. This is achieved by delivering a user-mode AST to the thread. The target procedure of the AST is a routine that is part of the Mica executive but is executed in user mode. The AST target procedure is the function *e$in_context_force_exit()*. The purpose of this function is to bring the thread into a "clean" state so that it can complete its exit. The following steps occur in *e$in_context_force_exit()*:

- The thread issues an *e$unwind()* specifying an exit unwind.
- Once the unwind has completed, the thread issues a call to *exec$exit_thread()*.

1.4.3.1.4.2 Thread Exit

The second phase of thread exit processing begins at the entry point *exec$exit_thread()*. The purpose of this function is to execute all of the exit handlers for the thread and, when completed, to bring the thread object out of the system. The following steps occur in *exec$exit_thread()*:

- Dequeue the first exit handler from the thread control region.
- If the thread is in the force-exit in progress state, then establish a CPU time quota for the thread.

NOTE

The thread-exit CPU time quota is on accumulated user-mode CPU time. It is not an elapsed time limit.

- If the CPU time quota expires, then deliver a user-mode AST to the thread. The target procedure of the AST is executive code that runs in "trusted" user-mode at the *e$exit_handler_quota_expire()*. This entry point causes the termination of the current exit handler and begins the next by calling *exec$exit_thread()*.
- Vector to the exit handler in user-mode.
- If no more exit handlers for the thread exist, then remove the object ID of the thread by calling *e$remove_object_id()*, passing it the object ID of the thread stored in *t_obj_id* in the thread object body. This action begins the second phase of hierarchy collapse by causing the execution of the affected object's remove routines. If there are more exit handlers, then repeat the above steps.
- After completion of *e$remove_object_id()*, the thread removes itself from the system by calling *k$terminate_thread()*. This action begins the third phase of hierarchy collapse by causing the execution of the affected object's delete routines.

1.4.3.2 Object Remove Routines

The object remove routines are called when the *objhdr$object_id_count* within the object header decrements to zero. This occurs during the second phase of hierarchy collapse as a result of a call to *e$remove_object_id()* for the "exiting" object. Object remove routines are always executed in the context of the object being removed.

NOTE

In order to ensure the above context restrictions, objects within the UJPT hierarchy may not have alias object IDs, and their ACLs are such that only the function *exec$exit_thread()* is capable of removing their object IDs.

Assuming the UJPT hierarchy from Figure 1–1, the following legal contexts exist to execute the remove routines for the hierarchy.

- Thread.0 will execute its remove routine in the context of thread.0.
- Thread.1 will execute its remove routine in the context of thread.1.
- Thread.2 will execute its remove routine in the context of thread.2.
- Process.0 could execute its remove routine in either the context of thread.0, or thread.1. The context would be determined by the context of the last thread to begin the second phase of exit.
- Process.1 will execute its remove routine in the context of thread.2.
- Job.0 will execute its remove routine in the context that was used to execute process.0's remove routine.
- User.0 will execute its remove routine in the context that was used to execute job.0's remove routine.

1.4.3.2.1 User-Object Remove Routine

The user-object remove routine performs no actions related to hierarchy collapse.

1.4.3.2.2 Job Object Remove Routine

The job-object remove routine is responsible for breaking the link between itself and its user object. If the job object is the last object of its user object then it must guarantee the removal of the user object. This occurs as follows:

- The job object is de-linked from the *u_job_queue_hd* in the user object pointed to by *j_user_pointer*.
- If the *u_job_count* field is decremented to zero by this action, then the user object is removed by calling *e$remove_object_id()* specifying the object ID of the user object (*u_obj_id*) stored in the user object body.

1.4.3.2.3 Process Object Remove Routine

The process object remove routine is responsible for breaking the link between itself and its job object, and if the process is a sub-process, it must break the link between itself and its *parent process* i.e. the process above it. Two different paths are followed during the process remove routine. The following occurs in the remove routine for a process without a parent.

- The process object is de-linked from the *j_process_queue_hd* in the job object pointed to by *p_job_pointer*.
- If the *j_process_count* field is decremented to zero by this action, then the job object is removed by calling *e$remove_object_id()* specifying the object ID of the job object (*j_obj_id*) stored in the job object body.

The remove routine for a sub-process i.e. a process with a parent simply de-links itself from the *p_sub_process_queue_hd* in the process object pointed to by *p_parent_pointer*.

1.4.3.2.4 Thread Object Remove Routine

The thread object remove routine is responsible for breaking the link between itself, and its process object. If the thread object is the last object of its process object then it must guarantee the removal of the process object. This occurs as follows:

- The thread object is de-linked from the *p_thread_queue_hd* in the process object pointed to by *t_process_pointer*.
- If the *p_thread_count* field is decremented to zero by this action, then the process object is removed by calling *e$remove_object_id()* specifying the object ID of the process object (*p_obj_id*) stored in the process object body.

1.4.3.3 Object Delete Routines

The object delete routines are called as a result of the *objhdr$pointer_count* field decrementing to zero. This occurs during the third phase of hierarchy collapse as a result of the call to *k$terminate_thread()* in *exec$exit_thread()*.

The function of *k$terminate_thread()* is to remove the thread from the system. This is accomplished by queuing a pointer to the thread object to a queue served by a system thread running *e$terminate_thread()*. This thread is responsible for dereferencing the thread object which begins the third phase of hierarchy collapse.

Object delete routines always execute in the context of the system thread running *e$terminate_thread()*.

NOTE

At the time that *k$terminate_thread()* is called, the thread object's *objhdr$pointer_count* is 1, and the *objhdr$object_id_count* is 0.

1.4.3.3.1 User-Object Delete Routine

The user-object delete routine performs no actions related to hierarchy collapse.

1.4.3.3.2 Job-Object Delete Routine

The job-object delete routine simply dereferences its user object by calling *e$dereference_object()* passing it the referenced pointer to the user object stored in *j_user_pointer*.

1.4.3.3.3 Process-Object Delete Routine

The process-object delete routine performs the following actions:

- If the process has a parent process, its parent process object is dereferenced by calling *e$dereference_object()*, passing it the referenced pointer to the parent process object stored in *p_parent_pointer*.
- The job object is dereferenced by calling *e$dereference_object()*, passing it the referenced pointer to the job object stored in *p_job_pointer*.

1.4.3.3.4 Thread-Object Delete Routine

The thread-object delete routine simply dereferences its process object by calling *e$dereference_object()*, passing it the referenced pointer to the process object stored in *t_process_pointer*.

1.5 Address Space and Execution Threads

Execution threads exist within a context which includes an address space and processor state. The creation and deletion of execution threads involves heavy interactions with the Mica kernel and memory management subsystems. This section describes execution thread creation and deletion in terms of its interactions with the Mica kernel, executive, and memory management subsystems. Interactions with the object architecture are not discussed.

1.5.1 Creation

The creation of an execution thread has two distinct paths.

The first path occurs when an execution thread is being created, an action which requires the creation of both an address space and a processor state. This path is a result of an *exec$create_user()*, an *exec$create_job()*, or *exec$create_process()* system service. This path is known as *initial thread creation*.

The second path occurs when an execution thread is being created within an existing address space. The only context that needs to be established is the processor state. This path occurs as a result of an *exec$create_thread()* system service and is known as *subsequent thread creation*.

1.5.1.1 Initial Thread Creation

During initial thread creation, the following actions occur.

- An address space must be created and initialized.
- A transition to the new thread's partial context must occur.
- Both thread- and process-control region address space must be created and initialized.
- The program image for the new process must be mapped into the process address space.
- The thread must begin execution at the program image starting address

1.5.1.1.1 Address Space Creation

The creation of a Mica address space occurs as a result of a call to *e$create_process_address_space()*. Example 1–30 illustrates the interface to this function.

Example 1–30: Address Space Creation

```
PROCEDURE e$create_process_address_space (
            IN process_control_pte : POINTER mm$pte;
            OUT ptbr : integer;                                  !page table base register
            OUT kernel_stack_pointer : POINTER anytype;
            );
            EXTERNAL;
!++
!
! Routine description:
!
!   This routine creates the foundation of a process address space.
!   Pages are allocated for the segment 1 page table, the segment 2
!   page table for the control region, the kernel stack and the
!   working set list.
!
!   NO ADDRESSES WITHIN THE ADDRESS SPACE ARE VALID, THIS INCLUDES THE
!   KERNEL STACK POINTER WHICH IS RETURNED.
!
!   Once an address space foundation has been created, k$initialize_thread
!   and k$ready_thread are invoked to create the initial thread running
!   within this new address space.
!
! Arguments:
!
!   IN  process_control_pte - pointer to the process_control_pte in the process
!                             control block.  Upon return the prototype PTE
!                             referred to by process_control_pte will contain
!                             the prototype PTE for the segment 1 page table
!                             page.  The PFN database PTP element will contain
!                             this address (process_control_pte) so it must
!                             be in non paged system space.
!
!   OUT ptbr - the value to be used for the page table base register
!
!   OUT kernel_stack_pointer - the value to be used for the kernel stack pointer
!
! Return value:
!
!   none.
!
!--
```

The created address space is only valid in the context of the new thread. The next phase of address space creation occurs in the context of the new thread.

1.5.1.1.2 Execution Thread Creation

Once the address space for the initial thread is created, the thread must be started in its context. This occurs by calls to the the kernel interfaces *k$initialize_thread()*, and *k$ready_thread()*. After the completion of *k$ready_thread()*, the new thread is eligible to run in its own context, and the calling thread considers the thread creation complete.

The new thread begins execution at *e$initial_thread_startup()*. Example 1–31 illustrates the entry point for all initial threads.

Example 1–31: Initial Thread Entry Point

```
PROCEDURE e$initial_thread_startup ();
            EXTERNAL;
!++
!
! Routine description:
!
!   The entry point for all initial threads. This routine is responsible for completing an
!   execution thread which involves
!
!       o Initializing the threads address space
!       o creating and initializing the control region memory pool
!       o initializing the pcr and tcr
!       o mapping the program image into the new address space
!       o starting the thread at the image entry point
!
! Arguments:
!
!   none
!
! Return value:
!
!   none
!
!--
```

1.5.1.1.2.1 Address Space Initialization

The first action performed by *e$initial_thread_startup()* is the initialization of the process address space. This action makes it possible for the thread to begin taking page faults within its address space. Address space initialization is accomplished by calling *e$initialize_address_space()*. Example 1–32 illustrates the interface to *e$initialize_address_space()*.

Example 1–32: Address Space Initialization

```
PROCEDURE e$initialize_address_space (
            IN working_set_extent: e$t_resource_counter;
            IN working_set_quota: e$t_resource_counter;
            );
            EXTERNAL;
!++
!
! Routine description:
!
!   This routine initializes an address space which was previously
!   created by e$create_process_address_space.
!
!   It must now be running in the non paged portion of the exec
!   with the newly created address space mapped.  No page faults
!   may be taken until this routine has been invoked.
!
!   This routine will create the working set list, mark the control
!   region, kernel stack, and working set list as locked in the
!   working set.
!
!   The arguments are derived from the process control block
!   qnl_working_set_limit and qnl_working_set_extent fields of
!   pcb_usage_and_limits structure.
!
! Arguments:
!
!   IN working_set_extent - maximum size of the working set.
!   IN working_set_quota - current size of the working set.
```

Example 1–32 Cont'd. on next page

Example 1–32 (Cont.): Address Space Initialization

```
!
! Return value:
!
!   none - it had better work.
!
!--
```

1.5.1.1.2.2 Control Region Initialization

Once the process address space has been initialized, the control region memory pool must be initialized. The control region is at a fixed virtual address within the process's address space and is user read-only, kernel read/write. The standard Mica pool header for pool type *e$k_pool_control* is initialized and fed by calling *e$initialize_control_region()*.

Once the control region pool has been created, a process control region and thread control region are allocated from the control region pool. The control regions are then initialized by copying dummy control regions allocated from non-paged pool to the real control regions. Finally, the thread control region is linked to its thread control block, and the process control region is linked to the process control block and the thread control region.

1.5.1.1.2.3 Program Image Mapping

The program image to be executed must be mapped into the newly created process address space. This occurs by transitioning into user-mode at the entry point *e$program_image_startup()*.

The function of *e$program_image_startup()* is to map the program image and cause it to begin execution at the image start address. To map the image, the function *exec$map_image()* is called passing it the image name stored in its process control region. Once mapped, the thread startup address stored in the thread control region is set using *exec$set_thread_information()*. The image is then called. The initial thread parameters may be found in the thread control region.

1.5.1.2 Subsequent Thread Creation

During subsequent thread creation the following must occur.

- Creation of a kernel mode and user mode stack for the thread.
- Creation and initialization of the thread control region.
- Transition to the new thread's context at the proper start address.

1.5.1.2.1 Thread Stack Creation

The creation of a kernel and user mode stack for the new thread occurs as a result of calling *e$create_thread_stacks()*.

1.5.1.2.2 Control Region Initialization

A thread control region is allocated for the new thread from the control region pool of the calling thread's process. The thread control region is then initialized with the values obtained from the *exec$create_thread()* parameters. The thread control region is then linked to the thread control block and is set to point to the proper process control region.

1.5.1.2.3 Transition to new Thread

The final steps in subsequent thread creation require that the thread be started in its context. This is achieved by making calls to *k$initialize_thread()*, and *k$ready_thread()*. After the completion of the call to *k$ready_thread()*, the new thread is eligible to be run in its own context, and the calling thread assumes that the thread creation has completed.

The new thread begins execution at *e$subsequent_thread_startup()*. This entry point simply forces a transition to user-mode at the address specified by the thread control blocks *tcr_start_address_field*.

1.5.2 Deletion

Address space and execution thread deletion happen as part of the process object and thread object delete routines.

1.5.2.1 Execution Thread Deletion

Execution thread deletion happens in two phases. The first phase is executed within the context of the terminating thread and is responsible for thread resource cleanup. The second phase occurs outside the context of the calling thread and is responsible for the deletion of the kernel stack of the terminating thread.

NOTE

The context restrictions are enforced by the lack of alias object IDs on components of the UJPT hierarchy, and through restrictions on the removal of objects within the hierarchy.

1.5.2.1.1 In-Context Thread Deletion

In-context thread deletion involves returning to the system all resources owned by the thread. This may include AST control blocks, IO request packets, and other outstanding system resources. All mutexes owned by the thread must be dealt with, and the thread control region must be returned to the control region pool of its process. These actions occur as part of the thread object's remove routine.

The second phase of execution thread deletion is then started by calling the kernel primitive *k$terminate_thread()*.

1.5.2.1.2 Out of Context Thread Deletion

The call to *k$terminate_thread()* is responsible for queuing a terminate-thread descriptor on a queue served by the system thread responsible for out-of-context thread deletion. The server causes the thread object's delete routine to be executed by dereferencing the pointer to the thread object.

The thread object delete routine deletes the kernel stack of the terminating thread by calling *e$delete_thread_stack()*.

At the end of out-of-context thread deletion, all data structures that represent the thread are returned to the system. This includes the entire thread object and thread control block.

NOTE

The thread control region is deallocated during in context thread deletion because it must refer to the thread's process address space.

1.5.2.2 Address Space Deletion

If the terminating execution thread is the last thread of its process, then the address space of the process must also be deleted. This occurs in the process delete routine.

NOTE

Address space, as used above, means address-space management data structures such as page tables, working set lists, and the last thread's kernel stack.

The user-mode address space is deleted mostly as a result of removing the process level container directory, since user-mode address space is represented as section objects.

The process delete routine calls *e$delete_process_address_space()*, specifying the page table base register value from process object body.

1.6 Exit Status

The exit status mechanism in the Mica system supports the ability to obtain the exit status from a process and, in some cases, from an individual thread within a process.

The exit status mechanism is coordinated through the exit status object.

1.6.1 Object Structure

The exit status object contains information describing the termination state of the object it is bound to. Example 1–33 illustrates the layout of the exit status object.

Example 1–33: Exit Status Object Structure

```
!
! Exit Status Object Body
!
e$t_exit_status_body: RECORD
    es_exit_status_summary: e$t_exit_status_summary;     ! Exit Status Summary
    es_exit_status_event: k$dispatcher_object(event);      ! Signaled on status summary valid
END RECORD;

!
! Exit Status Summary
!
e$t_exit_status_summary: RECORD
    status_valid: boolean;                                 ! True if status summary valid
    status_bound_object_type: e$t_status_object_types;   ! Process or Thread
    status_bound_object_id: e$t_object_id;               ! Object ID of object reporting status
    status_value: e$t_exit_status;                       ! Exit Status
END RECORD;
```

1.6.2 Functional Interface

The Mica executive provides interfaces to create and obtain information from exit status objects.

1.6.2.1 Exit Status Object Creation

Exit status objects are created by the *exec$create_exit_status()* system service. Exit status objects are created in a "invalid" state and are not bound to either a process or a thread object. The object binding occurs during thread and process object creation. The "validation" of exit status objects occurs during process and thread deletion. Example 1–34 illustrates the interface to *exec$create_exit_status()*.

Example 1–34: Exit Status Object Creation System Interface

```
PROCEDURE exec$create_exit_status (
            OUT object_id: exec$t_object_id;
            IN  container: exec$t_object_id = DEFAULT;
            IN  name: exec$t_object_name = DEFAULT;
            IN  acl: exec$t_acl = DEFAULT;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Create an invalid exit status object
!
! Arguments:
!
!   object_id           The object ID of the created exit status object
!   container           Object container for exit status object
!   name                Name of exit status object
!   acl                 ACL to place on exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.6.2.2 Get Exit Status Information

The *exec$get_exit_status_information()* system service provides a mechanism for obtaining the information stored in an exit status object. Example 1–35 illustrates the interface to *exec$get_exit_status_information()*.

Example 1–35: Get Exit Status Information System Interface

```
PROCEDURE exec$get_exit_status_information (
            IN object_id: exec$t_object_id;
            OUT status_summary: e$t_status_summary;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Return the es_exit_status_summary field from the exit status object
!   specified by object_id
!
! Arguments:
!
!   object_id       object ID of exit status object
!   status_summary  es_exit_status_summary field from specified exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.6.3 Usage

Exit status objects are used to report the exit status of exiting processes and exiting threads.

Each thread in the Mica system may optionally be bound to an exit status object. The binding occurs during the creation of the thread.

Each process in the Mica system is bound to an exit status object. The binding occurs during the creation of the process.

Exit status objects are "invalid" at object creation time and remain invalid until the object that they are bound to is removed from the system.

1.6.3.1 Thread Exit Status Object Usage

If the *thread_status* parameter is specified during the direct or indirect creation of a thread, then the thread is bound to the specified exit status object. The exit status object is made valid during the object remove routine for an exiting thread. This occurs as follows:

- Set the *tcb_exit_status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the existing status object bound to the exiting thread.
- Set the *status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the exit status object bound to the exiting threads process.
- Signal the *es_exit_status_event* in the exit status object bound to the exiting thread.

If the exiting thread is not bound to an exit status object, then the following occurs.

- Set the *status_value* field to the value stored in the thread control block *tcb_exit_status_value* field.
- Set to true the *status_valid* field in the exit status object bound to the exiting threads process.

1.6.3.2 Process Exit Status Object Usage

Each process in the Mica system is bound to an exit status object. During the object remove routine for a process object, the process exit status object is signaled by setting the *es_exit_status_event* in the exit status object bound to the exiting process. The *status_valid* and *status_value* fields were previously set during the individual thread exits for all of the processes threads.

1.7 Process/Thread Startup/Rundown Summary

This section is an attempt to summarize the steps that occur during the creation, execution, and termination of a thread in the Mica system. A very simple hierarchy will be studied in this description. The hierarchy consists of user.0, job.0, process.0, and thread.0 from Figure 1–1.

1.7.1 Startup Summary

The sample hierarchy is created as a result of the *job controller* calling *exec$create_user()*. The following steps occur as a result of this call.

1. A user object named user.0 is created. The user control block is initialized from the *user_record* parameter. The user object body is initialized to contain an empty job list and a job count of zero.
2. A job object named job.0 is created. The job control block is initialized by allocating *q_per_job_limits* quota from user.0, and assigning it to *jcb_usage_and_limits*. A job level container directory is created and optionally populated based on the existence of the *job_initial_container* parameter. The job object body is initialized to contain an empty process list and a process count of zero. The *j_user_pointer* is set to be a referenced pointer to user.0, and job.0 is linked to user.0's job list. User.0's job count is incremented to 1.

3. A process object named process.0 is created. The process control block is initialized by allocating *q_per_process_limits* quota from user.0 and assigning it to *pcb_usage_and_limits*. A security profile for the process is obtained from user.0. The accounting structure in the process control block is initialized to *zero()*. A process level container directory is created and optionally populated based on the existence of the *process_public_container* and *process_private_container* parameters. The *pcb_condir_address* and *pcb_condir_mutex* vectors are initialized. The process object body is initialized to contain an empty thread and sub-process list. The thread count and sub-process count is set to zero. The *p_job_pointer* is set to be a referenced pointer to job.0, and process.0 is linked to job.0's process list. Job.0's process count is incremented to 1.

4. A thread object named thread.0 is created. The thread control block is initialized by clearing all events and setting the *tcb_irp_list_head* to empty. The *tcb_pcb_pointer* field is initialized to point to process.0's process control block. If specified in the *thread_status* parameter, the exit status object for the thread is referenced and stored in *tcb_exit_status_ptr*. The *tcb_exit_status_value* is cleared. The thread object body is initialized by setting the *t_process_pointer* to be a referenced pointer to process.0, and thread.0 is linked to process.0's thread list. Process.0's thread count is incremented to 1.

5. An address space is created for process.0 by calling *e$create_process_address_space()*. This call initializes portions of the *tcb_thread_context*, and the *pcb_ptbr*.

6. The kernel context for the thread is initialized by calling *k$initialize_thread()*.

7. The thread is made eligible to run in kernel mode at the *e$initial_thread_startup()* entry point by calling *k$ready_thread()*. At this point, the original caller of *exec$create_user()* is returned to with a "successful" user creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.

8. The first action performed by the thread at *e$initial_thread_startup()* is a call to *e$initialize_address_space()*.

9. Once the address space has been initialized, the thread initializes the control region by calling *e$initialize_control_region()*. The control region appears as a user-mode read-only, kernel-mode read/write portion of process.0's address space. A *buddy system* memory pool is created and initialized in the control region as a result of calling *e$initialize_control_region()*.

10. The process control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *pcb_pcr_base* field of process.0's process control block is set to point to the allocated pcr, and the pcr is initialized by portions of the *initial_thread_parameters* parameter, and the object ID of process.0.

11. The thread control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *tcb_tcr_base* field of thread.0's thread control block is set to point to the allocated tcr, and the tcr is initialized by portions of the *initial_thread_parameters* parameter, the object ID of thread.0, the address of process.0's pcr, and various attributes of the thread specific address space.

12. The program image specified by the *process_record* field of the *initial_thread_parameters* parameter is mapped into process.0's address space by transitioning into user-mode at *e$program_image_startup()*.

13. Once at *e$program_image_startup()*, the thread issues a call to *exec$map_image()* and then sets the thread start address in the thread control region to the value returned by *exec$map_image()* by calling *exec$set_thread_information()*.

14. The thread entry point stored in *tcr_start_address* is "called" and is passed the thread parameters stored in *itp_thread_parameter_list*.

1.7.1.1 Additional Thread Startup Summary

This section describes the startup procedures for subsequent threads of a process. Assuming the hierarchy of the previous section, the following occurs when thread.0 makes a call to *exec$create_thread()* creating thread.1.

1. A thread object named thread.1 is created. The thread control block is initialized by clearing all events and setting the *tcb_irp_list_head* to empty. The *tcb_pcb_pointer* field is initialized to point to process.0's process control block. If specified in the *thread_status* parameter, the exit status object for the thread is referenced and stored in *tcb_exit_status_ptr*. The *tcb_exit_status_value* is cleared. The thread object body is initialized by setting the *t_process_pointer* to be a referenced pointer to process.0, and thread.1 is linked to process.0's thread list. Process.0's thread count is incremented to 2.
2. A partial address space is created for thread.1 calling *e$create_thread_stacks()*. This call initializes portions of the *tcb_thread_context*.
3. The thread control region is allocated by calling *e$pool_allocate()* specifying a pool type of *e$k_pool_control_region*. The *tcb_tcr_base* field of thread.1's thread control block is set to point to the allocated tcr, and the TCR is initialized by portions of the *initial_thread_parameters* parameter, the object ID of thread.1, the address of process.0's pcr, and various attributes of the thread specific address space.
4. The thread start address in thread.1's TCR is initialized to the value specified in the *thread_procedure* parameter.
5. The kernel context for the thread is initialized by calling *k$initialize_thread()*.
6. The thread is made eligible to run in kernel mode at the *e$subsequent_thread_startup()* entry point by calling *k$ready_thread()*. At this point, the caller of *exec$create_thread()* is returned to with an "successful" thread creation. Failures in thread startup after this point occur in the context of the created thread and are treated as an abnormal termination status of the thread.
7. Once at *e$subsequent_thread_startup()*, the thread entry point stored in *tcr_start_address* is "called" and is passed the thread parameters stored in *thread_parameter_list*.

1.7.2 Rundown Summary

At some point in the threads lifetime it will either voluntarily exit by calling *exec$exit_thread()* or be forcibly exited by calling *exec$force_exit_thread()* on itself or having some other thread issue an *exec$force_exit_thread()* specifying that thread.

For the following rundown example, it is assumed that thread.99 issues an *exec$force_exit_thread()* specifying the object ID of thread.0. The hierarchy consists of user.0, job.0, process.0, and thread.0.

1. The Mica executive is entered at *e$force_exit_thread()*. The force-exit in progress flag is set in the thread object body of thread.0. The purpose of this flag is to prevent the creation of new exit handlers for the thread and to prohibit the thread from creating new threads, processes, and jobs.
2. The next step is to cause thread.0 to begin execution in "trusted" user-mode at the *e$in_context_force_exit()* executive entry point. At this point the force-exit of thread.0 is complete with respect to thread.99. The following steps are then taken to force thread.0 into taking an active role in its exit. This occurs as follows:
   - An elapsed timer is set to expire in a TBD period. If the timer expires, all of these steps are repeated, in addition to enabling user mode ASTs, setting the ast queue flush flag in the thread object body, and a call to *k$flush_ast_queue()* is issued.
   - A user-mode AST is queued to thread.0. The target procedure of the AST is *e$in_context_force_exit()*.
3. Once at *e$in_context_force_exit()*, thread.0 unwinds its stack by calling *e$unwind()*.

4. Once the stack has been unwound and all unwind handlers have been executed, thread.0 executes a call to *exec$exit_thread()*.

5. The code at *exec$exit_thread()* assigns the parameter *exit_status* to the *tcb_exit_status_value* field of thread.0's thread control block. If thread.0 was created with an exit status object, the *exit_status* is also assigned to *tcb_exit_status_ptr^.es_exit_status_sumary.status_value*. The value *exit_status* is then assigned to *pcb_exit_status_ptr^.es_exit_status_sumary.status_value* in process.0's exit status object. The force-exit in-progress flag for thread.0 is examined. Since the flag was set, the elapsed timer set up in *e$force_exit_thread()* is dismissed.

NOTE

If the timer in the example above had expired, that would indicate that the user-mode AST was not delivered, or that there was an exceptional delay in making progress through the stack unwind. In any case, timer expiration causes a retry which will eventually be successful.

6. Thread.0 is then allowed to execute each one of its exit handlers. Since the thread is being force-exited, its exit handlers are assigned a small CPU time quota. When the quota expires, a user-mode AST is delivered to the thread that causes it to execute an *exec$exit_thread()*. The method for delivering the user-mode AST is similar to the technique used to cause the thread to execute at *e$in_context_force_exit()*, only the AST procedure target is *e$exit_handler_expire()*. The function of *e$exit_handler_expire()* is to simply call *exec$exit_thread()*.

7. Thread.0 issues a call to *e$remove_object_id()* specifying its object id (*t_obj_id*. This action causes thread.0's object remove routine to be called.

8. Thread.0's object remove routine is entered. It performs the following steps.

   - All outstanding resources that require cleanup by the thread are processed. This includes the dismissal of all outstanding I/O by callin *e$cancel_io_by_thread()*, the dismissal of outstanding ASTs, and ...(TBS).
   - The thread control region is returned to the control region pool of process.0 by calling *e$pool_deallocate()*.
   - The thread object is de-linked from the *p_thread_queue_hd* of process.0.
   - Since the above step causes the *p_thread_count* field to decrement to zero, the process.0 object is removed by calling *e$remove_object_id()* specifying the object ID of process.0.
   - If thread.0 was created with an exit status object, then the *es_exit_status_event* in the object is "set". The exit status object is then dereferenced by calling *e$dereference_object()*.

9. The object remove routine for process.0 is entered as a result of thread.0's object remove routine being entered. The following occurs during process.0's object remove routine.

   - The job level container directory whose address is stored in *pcb_condir_array* is dereferenced.
   - The process level container directory is removed from the system by calling *e$remove_object_id()*, and specifying *pcb_process_condir_id*
   - The process control region is returned to its control region pool by calling *e$pool_deallocate()*.
   - The process object is de-linked from the *j_process_queue_hd* of job.0.
   - Since the above step causes the *j_process_count* field to decrement to zero, the job.0 object is removed by calling *e$remove_object_id()* specifying the object ID of job.0.
   - The *es_exit_status_event* in process.0's exit status object is "set". The exit status object is then dereferenced by calling *e$dereference_object()*.

10. The object remove routine for job.0 is entered as a result of process.0's object remove routine being entered. Th=e following occurs during job.0's object remove routine.
    - The job level container directory is removed from the system by calling *e$remove_object_id()*, and specifying *jcb_job_condir_id*.
    - The job object is de-linked from the *u_job_queue_hd* of user.0.
    - Since the above step causes the *u_job_count* field to decrement to zero, the user9.0 object is removed by calling *e$remove_object_id()* specifying the object ID of user.0.
11. The object remove routine for user.0 is entered as a result of job.0's object remove routine being entered. The routine performs no significant actions
12. The original call in *exec$exit_thread()* which removed the object ID of thread.0 returns. The next step is a call to *k$terminate_thread()*. The purpose of *k$terminate_thread()* is to remove the specified thread (thread.0) from execution within the Mica system. Once all of the kernel related activities are complete, a pointer to thread.0 is queued to a special system thread known as the *thread eater*. The thread eater executes the loop at *e$terminate_thread()*.
13. The function of *e$terminate_thread()* is to dequeue the thread's arriving on its queue, and to dereference the thread objects. When the thread eater processes thread.0, it calls *e$dereference_object()* specifying thread.0. The delete routine for thread.0 is entered. It is important to note that the delete routine for thread.0 is entered in the context of the thread eater.
14. The delete routine for thread.0 is entered. It performs the following actions.
    - Thread level accounting information is rolled up to the thread's process.
    - The thread specific address space (user-mode, and kernel-mode stacks) of thread.0 are returned to the address space of process.0 by calling *e$delete_thread_stacks()*.
    - The referenced pointer to process.0 is dereferenced. This causes the delete routine for process.0 to be executed.
15. The delete routine for process.0 is entered. It performs the following actions.
    - An accounting record is written to the TBD message function processor. The information for the accounting record is obtained from the *pcb_accounting* field from process.0's process control block.
    - All resources accounted for in process.0's *pcb_usage_and_limits* are returned to job.0's *jcb_usage_and_limits* using the rules of deductable and non-deductable resource arithmetic.
    - The address space of process.0 is returned to the system by calling *e$delete_process_address_space()*.
    - The referenced pointer to job.0 is dereferenced. This causes the delete routine for job.0 to be executed.
16. The delete routine for job.0 is entered. It performs the following actions.
    - All resources accounted for in job.0's *jcb_usage_and_limits* are returned to user.0's *ucb_quotas.q_usage_and_limits* using the rules of deductable and non-deductable resource arithmetic.
    - The referenced pointer to user.0 is dereferenced. This causes the delete routine for user.0 to be executed.
17. The delete routine for user.0 is entered. It performs no significant actions.
18. Once the call frame has returned from the original call to *e$dereference_object()* issued by the thread eater on thread.0, the UJPT hierarchy consisting of user.0, job, process.0, and thread.0 is removed from the system, and the thread eater goes back to its queue of threads to be processed.

1.8 System Threads

This section describes the interface for creating system threads. It also describes the differences between system threads and normal threads, and the special restrictions placed on system threads.

1.8.1 System Thread Creation

The *e$create_system_thread()* executive interface creates a system thread. The system thread executes within the UJPT hierarchy of the system. The address space of the system thread is that of the initial system process. Example 1–36 illustrates the interface to *e$create_system_thread()*.

Example 1–36: System Thread Creation Executive Interface

```
PROCEDURE exec$create_system_thread (
            OUT object_id: e$t_object_id;
            IN  container: e$t_object_id = DEFAULT;
            IN  name: e$t_object_name = DEFAULT;
            IN  acl: e$t_acl = DEFAULT;

            IN thread_procedure: e$t_thread_entry_point;
            IN thread_record: e$t_thread_record = DEFAULT;
            IN thread_allocation_list: e$t_allocation_list = DEFAULT;
            IN thread_immediate_parameter1: e$t_thread_parameter = DEFAULT;
            IN thread_immediate_parameter2: e$t_thread_parameter = DEFAULT;
            IN thread_status: e$t_object_id = DEFAULT;
            ) RETURNS status;
            EXTERNAL;
!++
!
! Routine description:
!
!   Create a System thread object as specified by the parameters.
!
! Arguments:
!
!   object_id                   Object ID of the resulting process object
!   container                   Object container for thread object (ignored)
!   name                        Name of thread object
!   acl                         ACL to place on thread object
!   thread_record               Attributes of the thread being created
!   thread_allocation_list      Objects to be allocated to the thread object. If not present then
!                               no objects are allocated to the thread
!   thread_immediate_parameter1 Immediate parameter passed to thread through TCR
!   thread_immediate_parameter2 Immediate parameter passed to thread through TCR
!   thread_procedure            pointer to thread entry point entry descriptor
!   thread_status               Exit status object to be bound to the thread. If not present
!                               then the thread is created without an exit status object
!
! Return value:
!
!   TBS
!
!--
```

1.8.2 System Thread Restrictions

The important differences between system threads and normal threads are as follows:

- System threads may not execute in user-mode.
- System threads are incapable of processing or executing in the context of user-mode ASTs. Algorithms such as the one in *exec$signal_thread()* that employ user-mode ASTs either understand system threads and modify their algorithms or don't support the functions on system threads.
- The thread control region for system threads exists in paged pool.
- There is no thread environment block for system threads.
- System threads execute within the address space of the system.
- There are no provisions for passing block data to a system thread through the *tcr_block_data* field in a system thread's TCR.

- The *exec$force_exit_thread()* system service is not supported for system threads.