5

Process Management - Tasks and Threads

5.1 Introduction

Significant changes in the use of multiprocessor systems require new support from operating system schedulers. Earlier, multiprocessor systems have increased throughput by running several applications simultaneously, but no individual application ran faster. Such usage has given way to parallel programming, which reduces the runtime of individual applications.

Parallel-programming models and languages often anticipate dedicated use of processors or an entire multiprocessor, but few machines are used in this fashion. Most modern general purpose multiprocessors run a time sharing operating system such as Unix. The shared use model of these systems conflicts with the dedicated-use model of many programs, but the conflict is seldom resolved by restricting multiprocessor use to one application at a time.

Another impact on scheduler comes from the increased popularity of concurrency. The *application parallelism* for a multiprocessor application is the actual degree of parallel execution achieved, while *application concurrency* is the maximum degree of parallel execution that could be achieved with unlimited processors. For instance, an application consists of 10 independent processes/tasks running on a six CPU multiprocessor system has an application parallelism of six, based on the six CPUs, and an application concurrency is 10, because it could use up to 10 processors. Application concurrency beyond hardware parallelism can improve hardware use; if one portion of the application blocks (for example, a disk or network operation), other portions can still proceed. The use of concurrency can also simplify programming of concurrent applications by capturing the state of ongoing interactions in local variables of executing entities instead of in a state table.

Application parallelism and concurrency complicate two areas of scheduling: making effective use of the processing resources available to individual applications and dividing processing resources among competing applications. The problem of scheduling within applications seldom arises for serial applications because they can often be scheduled independently with little impact on overall performance. In contrast, the medium- to fine-grain interactions of parallel and concurrent programs together to achieve acceptable performance. Parallel applications that require a fixed number of processors complicate scheduling across applications. They make division of machine time more difficult and may introduce a situations for which a fair-sharing policy is inappropriate. For instance, an application configured for a fixed number of processors may be unable to cope efficiently with fewer processors.

C-DAC's PARAS operating system introduced some new approaches to scheduling which are similar to Carnegie Mellon University's Mach operating system. PARAS provides flexible memory management and sharing, multiple *threads* within a single address space or task for concurrency and parallelism, and a network transparent communication subsystem. The communication subsystem is

called IPC (interprocess communication) subsystem for historical reasons; all communication in PARAS is actually between tasks.

The PARAS microkernel does not provide the traditional notion of the process. This is due to the following two reasons:

1. Traditional operating system environment has considerable semantics (abstractions supported) associated with a process (such as user-ID, signal state, etc.). It is not the purpose of the PARAS microkernel to recognize or provide these extended semantics since it aims at providing only fundamental services.
2. Many systems (BSD, for instance) equate a process with an execution point of control. Some systems (POSIX 1003.4a—threads interface, for instance) do not. PARAS kernel as a microkernel decides to support multiple points of control in a way separate from any given operating environment's notion of process.

Instead, PARAS provides two notions: the *task* and the *thread*. A thread is PARAS's notion of the point of control. A task exist to provide resources for its containing threads. This split is made to provide for parallelism and resource sharing. Hence, UNIX equivalent of a process is a task with a single thread of control.

A thread:

- Is a point of control flow in a task.
- Has access to all of the elements of the task's resources.
- Potentially executes in parallel with other threads, even threads within the same task.
- Has minimal state for low overhead; thread appear like a mini-program.

A task:

- Is a collection of system resources. These resources, with the exception of the address space, are referenced by ports. These resources may be shared with other tasks if rights to the ports are so distributed.
- Provides a large, potentially sparse address space, referenced by machine address. Portions of this space may be shared through external memory management.
- Contains some number of threads.

PARAS Microkernel Interface

The PARAS microkenel provides the following interface calls for both tasks and threads:

- *create*: create a new task/thread.
- *terminate*: destroy an existing task/thread.
- *suspend*: suspend a task/thread.
- *resume*: resume a task/thread.
- *info*: obtain useful information about a task/thread.

E.g: *thread_create* creates a new thread, *task_creaste* creates a new task.

A thread is defined as a piece of code that can execute in concurrent with other threads. Thread consists of registers, status word, program counter, thread procedure with all the resources allocated to the task. Thread object is shown in the Figure 5.1. Threads contains statically ordered sequence of instructions. Multiple threads may operate concurrently within a task or process, each with its own program counter and local state, but with some state shared by all the threads in the process.

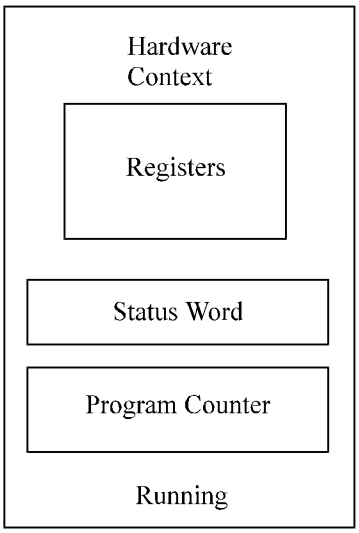

Figure 5.1: Thread Object

Multithreading is defined as a technique of parallelization of programs for execution on parallel computers. The program having multiple threads of execution is called as multithreaded program.

As whole task can be viewed as the parent and threads can be viewed as children of the task in which they are created. Hence, a task can have any number of threads running within its address space. Note that a task has no life of its own; only threads execute instructions. When it is said that "a task A does B" what is really meant is that "a thread contained within task A does B".

A task is a fairly expensive entity (see Figure 5.2). It exists to be a collection of resources. All of the threads in a task share everything. Two tasks share nothing without explicit action (although the active is simple) and some resources cannot be shared between tasks at all (such as port receive rights).

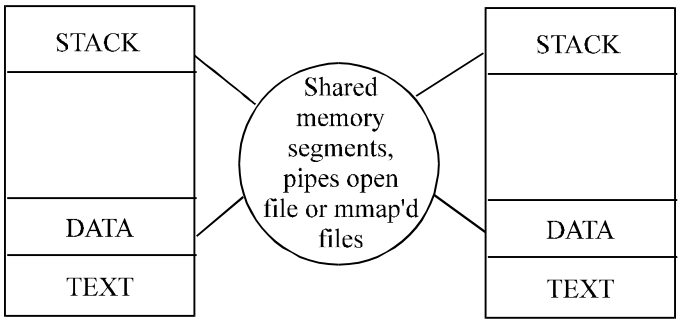

Figure 5.2: Basic Process Model

A thread is a fairly lightweight entity (see Figure 5.3). It is fairly cheap to create and has low overhead to operate. This is true because a thread has little state (mostly its register state); its owning task bears the burden of resource management. On a multiprocessor it is possible for multiple threads in a task to

execute in parallel. Even when parallelism is not the goal, multiple threads have an advantage in that each thread can use a synchronous programming style, instead of asynchronous programming with a single thread attempting to provide multiple services. That is if task is exhibiting any nature, it can be exploited to achieve better utilization of the CPU and speedups program execution (by overlapped computation and communication).

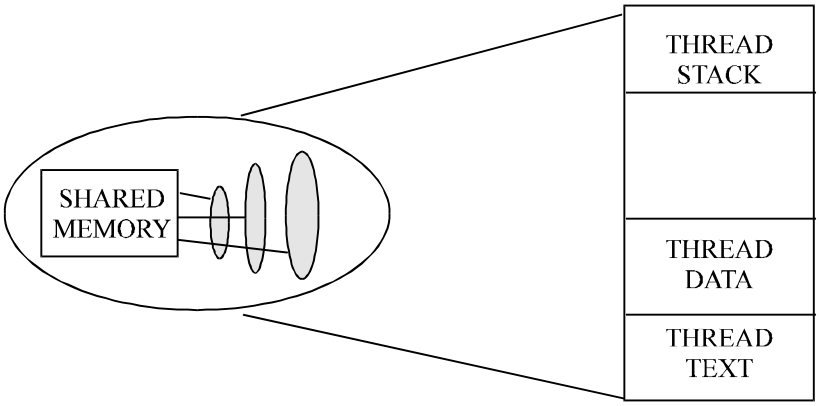

Figure 5.3: Threaded Process Model

The process management is one of the core component of operating systems. The microkernel process manager is responsible for handling issues related to the process/thread creation, scheduling, dispatching, control, resource handling, etc. To support these, the kernel essentially has deal with trap handling, context switching, scheduler logic, schedulable entities, scheduling schemes, interface call handlers, and IEEE 754 exception handlers.

In order to execute a task (also called process), it must reside in the main memory. The issues related to task loading and managing its memory will be discussed in Virtual Memory Management chapter. It also covers how a process' address space is mapped or how physical memory is managed.

For parallel programs to be executed on distributed memory MIMD machines (PARAM), the Interprocess Communication (IPC) plays a major role. The issues related to how messages are received (either from within the node or from outside) or sent out (local/outside) will be discussed in the Interprocess Communication chapter.

5.2 PARAS Tasks and Threads

A *task* in PARAS consists of an address space and a collection of threads that execute in that address space. Tasks are passive (they perform no computation) and are mainly used to collect all the resources needed for its threads. Tasks are a framework for running threads.

Threads are the basic schedulable entity with the Microkernel. Each thread is associated a kernel data structure maintained by the Microkernel. At any point of time, this data structure reflects the "run state" of the thread, which may contain values for the processor's special registers, windows, fp registers, etc. The Microkernel ensures that every thread starts up with a kernel-defined startup state of registers.

Threads are the basic units of execution and hence they contain the processing state associated with a computation. (e.g. A program counter, a stack pointer and machine registers). All threads within a task share the task's resources and resource limits, and are not protected from one another. Threads are the active units which are scheduled.

A thread may be in a *suspended* state (prevented from running) or in a "runnable" state (may be running or scheduled to run). There is a non-negative "suspend count" associated with each thread. This count may be modified by suspend and resume calls. The suspend count is zero for runnable threads and is positive for suspended threads.

Tasks may also be suspended or resumed as a whole. Like the thread's suspend count, a counter, the *task's suspend count*, may be modified by suspend and resume calls. A thread may only execute if it and its task are runnable. Resuming a task causes all threads which are not suspended to be scheduled and run.

Tasks and threads can have a number of *special ports* associated with them. In general, these are ports that the kernel must know about in order to communicate with the task or thread in a structured manner. Currently, there is one special port, the "task exception port", associated with each task and one special port, the "thread exception port" associated with each thread. Details of exception handling are described later section in this chapter.

The Microkernel manages the creation, suspension, resumption and termination of both user and kernel threads apart from other query routines to manage user threads. Most of the process management primitives have been derived from the Mach Microkernel of Carnegie Mellon University.

Basic internal primitives of the PARAS microkernel are the following:

[illegible]: assert that thread is above sleep
[illegible]: give up CPU if others can execute
[illegible]: clear waiting condition
[illegible]: cause thread to stop if not already stopped.
[illegible]: wait for thread to stop.
[illegible]: release one hold on thread.

The last three primitives are used to implement both thread and task operations.

5.4 PARAS Scheduler

The PARAS operating system splits the usual process notion into task and thread abstractions, but the PARAS time-sharing scheduler only schedules threads. Each task is represented in the PARAS microkernel by its own Task Control Block (TCB; also called a Process Control Block). A TCB is a data block or record containing many pieces of information associated with a specific task, including task state, program counter, CPU registers and scheduling information, memory management information, accounting information, I/O status information, task's exception ports, etc. The data structures declarations related to the task can be found in the header file task.h. The members of task's data structure are given below:

```
[illegible]
```

The task structure mainly keeps status of various resources associated with it. Such as memory usage, ports, threads owned by it. The information related to memory resources owned by a task has to be maintained since microkernel allows multiple tasks to be executed by time sharing on a single node. But, on PARAM it is advisable to execute one task per node instead on multiple tasks. Each task can have any threads execute simultaneously on a single node and they are executed by time sharing based on priorities using round-robin scheduling algorithm. A thread during its lifespan, goes through many states depending on a operation it wants to perform and external conditions such as sharing CPU when multiple threads are to be executed. Thread's states are explained in the following section using the thread state machine.

Thread State Machine

As a thread executes, it changes state. The state of a thread is defined in part by that thread's current activity. Each thread may be in one of the following states and they can be combinations of these as shown in Figure 5.4:

- R running — Instructions are being executed
- W waiting (or on wait queue) — A thread is waiting for some even to occur
- S suspended (or will suspend) — suspended by the user
- N non-interruptible — during which thread cannot be preempted. Threads manipulating kernel data structure can move to this state before they can modify.

State is based on 3 bits: WAIT (W), SUSP (S), RUN (R). They have the following meaning:

RUN: running normally (processor or run queue)

WAIT: waiting for event (on wait queue). Interruptable and non-interruptable waits are distinguesged.

SUSP: suspended (not on any queue). Driven by suspend count.

RUN+WAIT: after □□□□□□□□□□□□□□. Will wait at □□□□□□□□□□□□□□□.

RUN+SUSP: will suspend at next □□□□□□□□□□□□□□□.

WAIT+SUSP: waiting and suspended if wait is interruptable. Waiting and will suspend otherwise.

RUN+WAIT+SUSP: will wait at next □□□□□□□□□□□□□□□ if wait is non-interruptable. Will suspend at next □□□□□□□□□□□□□□□ otherwise.

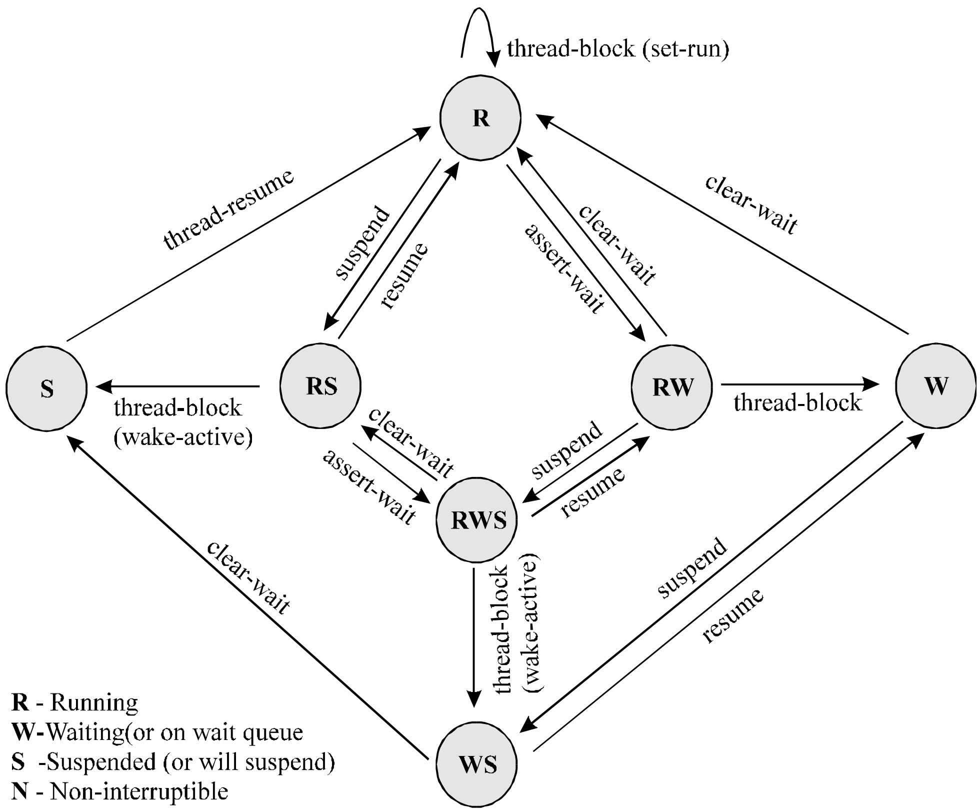

Figure 5.4: Thread State Diagram

The PARAS microkernel as a multitasking and multithreaded kernel, not only maintains Task Control Block, but also Thread Control Block. Thread control block maintains information such as thread registers (CPU registers), its state (ready, run, etc.), scheduling factor and policy, cpu usage, priority. When thread is created, it will have the same priority as its creater. The PARAS microkernel, allows the user to change priority explicitly. The structure of thread control block is shown below:

The PARAS microkernel maintains two main queues for threads management. First, Run Queue and Second, Wait Queue.

Run Queue

The primary data structure used by the PARAS scheduler is the *run queue*, a priority queue of runnable threads implemented by an array of doubly linked queues. PARAS uses 32 queues, so four priorities from the Unix range 0 to 127 map to each queue. Current implementation use a priority range of 0 to 31 so that queues and priorities correspond. Lower priorities correspond to higher numbers and vice versa.

All those threads which are in run state are maintained in a run-state table. The number of entries in this table can accommodate is depends on maximum priority any thread can assume (or user may set). Each entry in this table holds a pointer to queue of threads whose priority is the same. The maximum priority value in the PARAS Microkernel is NRQS (current implementation's NRQS value is 32). The organization of run-queue in shown in Figure 5.5.

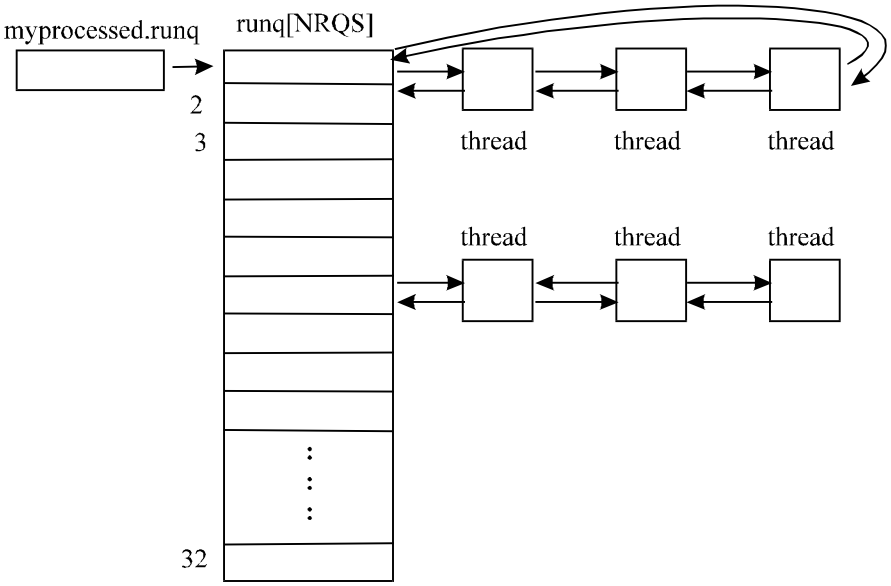

Figure 5.5: Thread Run Queue

Wait Queue

Wait queue is organized in the form of hash table. The waiting protocols and implementation details are given below:

Each thread may be waiting for exactly one event; this event is set using assert_wait(). That thread may be awakened either by performing a thread_wakeup_prim() on its event, or by directly waking that thread up with clear_wait().

The implementation of wait events uses a hash table. Each bucket is queue of threads having the same hash function value; the chain for the queue (linked list) is the run queue field. (It is not possible to be waiting and runnable at the same time.)

Scheduling operations may also occur at interrupt level.

The wait event hash table declarations are as follows:

There are totally NUMQUEUES queues are maintained in wait queue (see Figure 5.6). Depending on even type, hash table (wait-queue table) entry is made. The hash table entry point is found using the following relation:

Interruptible and Non-interruptible Threads

If a thread is not interruptible, it may not be suspended until it becomes interruptible. In this case, we wait for the thread to stop itself, and indicate that we are waiting for it to stop so that it can wake us up when it does stop.

If the thread is interruptible, we may be able to suspend it immediately. There are several cases:

1) The thread is already stopped (trivial).
2) The thread is runnable (marked RUN and on a run queue). We pull it off the run queue and mark it stopped. This assumes the thread was interrupted in user code.
3) The thread is running. We wait for it to stop.

Scheduler Operations:

Initializes the data structures required for scheduling; hz is the number of clock ticks per second; min_quantum is the time quantum given for each thread. Preempt during quantum requires higher priority than preempt after quantum. The clock handler decrements quantum and reschedules when quantum expires.

5.5 Trap Handling

User tasks invoke microkernel services using traps; traps allows to provide location independent services. The Microkernel has a two-level trap handling: low level and high level trap handlers.

The first level in assembly is where the control gets transferred on any kind of software or hardware trap and from where control returns to the user once the trap is serviced. The second level trap handler is a sandwich between the entry and exit portions of the first level handler that inspects of the cause of trap in detail and takes appropriate action.

The Microkernel trap handler saves all the windows of the executing thread on a trap instead of selective window-saving, since selective window-saving would make the trap handler too complicated. Consequently, the first-level trap handler's entry code goes through the following steps:

first_level_trap_handler_entry:

1. save special registers;
2. save floating-point registers and fp queue if required;
3. save all windows of the currently executing thread;
4. transfer control to the second-level trap handler;

The trap handling model used in the PARAS microkernel is shown in Figure 5.7.

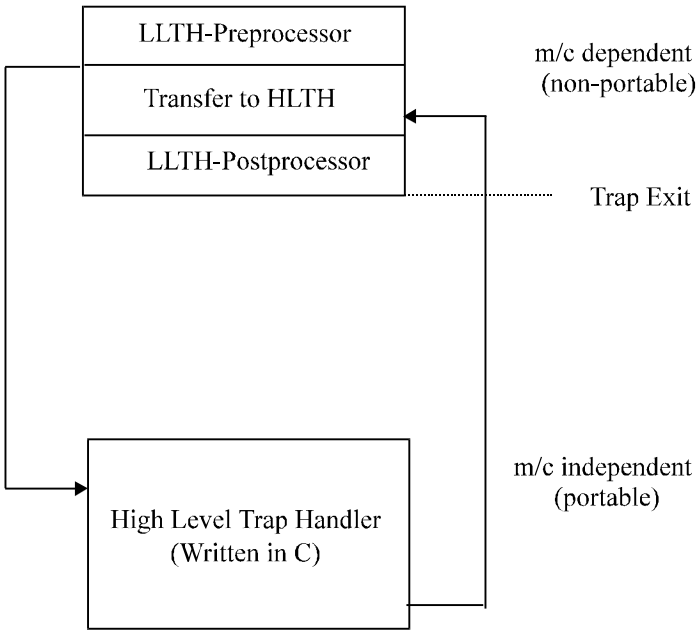

Figure 5.7: Trap handler model of Microkernel

The second-level C trap handler decides what trap has occurred and calls appropriate routines to service the trap. Broadly, the algorithm followed by this C trap handler is as follows:

```
second_level_trap_handler:
  check for the cause of the trap;
  if (trap due to microkernel system call)
     call kernel function to service system call;
  else if (trap due to UNIX system call)
     format a message detailing the trap reason;
     send message to MKFS;
    block the executing thread until a reply from MKFS;
  else if (trap due to hardware interrupt)
  begin
    case TIMER_INTERRUPT:
    call routines that periodically update
     priorities of threads and set Asynchronous
    System Traps (ASTs);
            case CCP_INTERRUPT:
                    call ccp handler routines that handle
                    incoming/outgoing messages to/from the CCP;
  end;
  else if (trap due to exception)
    dump core by sending message to PS (Process Server)
  return;
```

On return from this second-level trap handler, the control is transferred to the first level handler, which then inspects data structures to see if there are any pending ASTs (probably as a result of a change in the AST (Asynchronous System Traps) status of the current thread by a timer interrupt) for the current thread before returning control to the user. It uses the following algorithm:

```
first_level_trap_handler_exit:
if (ASTs pending for thread)
  if (thread to be halted)
    terminate current thread
  else if (thread's quantum over)
    switch to next eligible thread
    restore windows;
     restore floating point registers and fp queue
if required;
  restore special registers;
return control to user;
```

5.6 Context Switching

When trap occurs, execution context switches from user to kernel mode and one servicing these requests, again context is switches from kernel to user mode. The issues related to various operations to be performed when context is switched is handled by context-switching routines. They are similar to the trap handling entry and exit routines. A context_save saves all the windows and special registers of the currently running thread and a context_restore restores all the windows and special registers of a previously stored thread and transfers control to it.

The Microkernel performs a context switch from the current thread on two occasions :

a. the current blocks

b. the current has run out of its quantum and there is another eligible thread waiting to run

5.7 Predefined Threads in PARAS Microkernel

For the sake of its own operation, the microkernel creates kernel threads that execute purely within kernel context to provide various support function. For example, scheduler thread handles the issues related to threads's scheduling. Threads are the basic schedulable entity with the Microkernel. Each thread is associated a kernel data structure maintained by the Microkernel. At any point of time, this data structure reflects the "run state" of the thread, which may contain values for the processor's special registers, windows, fp registers, etc. The Microkernel ensures that every thread starts up with a kernel-defined startup state of registers.

The Microkernel manages the creation, suspension, resumption and termination of both user and kernel threads apart from other query routines to manage user threads.

The three kernel threads supported for managing kernel resources, cleanup and user thread priorities. They are idle thread, reaper thread, and scheduler thread. These threads with their responsibilities are discussed in the following sections:

idle_thread

This thread has the lowest priority among all threads, whether user or kernel, and runs only when no other thread is eligible for running. This thread just continuously checks to see if any other thread may be scheduled using the following algorithm:

```
while true
    if( any eligible thread to be run )
        block and give up execution;
```

reaper_thread

When a thread wishes to halt itself, the reaper thread's services are required. The thread wishing to get itself terminated queues itself up with the a queue that the reaper thread periodically checks.

Normally the reaper thread gets scheduled immediately after a thread_halt_self code is run by any thread. It uses the following algorithm:

```
while( more threads in the reaper queue )
   get thread from head of queue;
   clean up thread;
```

scheduler_thread

The scheduler thread is executes periodically (every 2 seconds) and updates the priorities of all threads, whether in run queue or in wait queue. On updating the thread priority, the scheduler based on the scheduling policy, decides the next user thread to be scheduled for execution. It follows the algorithm listed below:

```
update priorities of all threads;
check to see which is the most eligible thread to be run;
if( current thread can continue )
  return;
else
  put current thread out of action;
  schedule new thread;
```

5.8 Tasks

The tasks are the entities that "contain" threads. Each task has at least a single thread of control. The microkernel views these tasks as passive entities. Each task is associated with an address space with a unique mapping of virtual pages to physical pages. All threads associated with a particular task share and execute "within" this address space.

Threads belonging to different tasks have distinct page mappings. Hence, any context switch involving two threads from two different tasks incur more overhead for the microkernel than a context switch within threads of the same task.

Just as in the case of threads, the Microkernel maintains a unique data structure associated with each task. Unlike multiple kernel threads, however, there is only one *kernel* task.

The Microkernel manages the creation, suspension, resumption, and termination of user tasks.

5.9 Priority Mechanism

The Microkernel follows a priority based timesharing policy for scheduling threads. A thread's priority to be scheduled keeps changing depending on the amount of CPU time it is using. As it uses more and more CPU time, its priority becomes lower and lower; this avoids starvation of low priority threads. Conversely, the more time a thread is on the wait queue, the more its priority increases.

Priority of threads is updated when any of the following event occurs:

a. when a thread gets to run

b. when scheduler does its periodic priority update of "stuck" threads

The need for the scheduler to do periodic priority updation stems from the fact that a thread's priority is updated only when it is scheduled to run (rule a) but a thread cannot be scheduled unless it runs and raises its priority. In order to remove such deadlock situations and wake up the "stuck" threads, the scheduler periodically inspects and updates the priorities of all threads.

5.10 Asynchronous System Traps

Asynchronous System Traps (AST) are the clean way to handle context switches, current thread termination and other similar complex situations.

When an AST situation arises as a result of servicing a trap, the current thread that was trapped has to give up execution or has to be halted. The thread may have to give up execution either because it has to block for some information that is not available at the time or because its time quantum has expired and there is another eligible thread waiting to be run. In any case, the situation has arisen while the trap handler is servicing a specific trap. And as long as the trap is not serviced completely, the Microkernel's data structures are in inconsistent state and therefore no new servicing could be taken.

To resolve this difficulty, the Microkernel marks the thread data structure with a "need for AST" and then returns to servicing the trap. Once the trap is serviced and the control is about to be returned to the user, the Microkernel checks for pending ASTs and acts accordingly.

5.11 Loading and Initialization

The Microkernel on gaining control immediately after loading, resets and initializes all special registers, relocates itself to link address, creates and sets up new trap table, sets up memory, initializes data structures, switches to new trap table, creates kernel threads and then jumps to idle thread. The Microkernel does the following operations loading and initialization:

```
start:
  reset and initialize special registers;
  relocate self;
  create new trap table;
  if(window overflow/underflow)
    install fast handlers in new trap table;
  else if(level 14 ticker)
    retain OBP handler in new trap table;
    else
      install generic handler in new trap table;
```

```
clean up and organize memory map;
initialize all data structures (threads, tasks, page maps, ports, etc.);
switch to new trap table;
create load thread to load servers (first kernel thread to be scheduled when idle thread blocks
below);
create ipc threads;
create scheduler thread, reaper thread and idle thread;
jump to idle thread;
```

The Load thread then hand creates the System Servers (NS, PM, PS and MKFS) by communicating with the SLD through a well-known port.

5.12 Exception Handling

Tasks and threads can have an *exception port* associated with them. The thread's exception port is the port to which the kernel sends messages signaling an exception in the thread. "Exceptions" are synchronous interruptions to the normal flow of program control caused by the program itself. They include illegal memory accesses, protection violations, arithmetic exceptions, etc.

The kernel sends exception messages to the task's exception port if the thread causing the exception has no exception port registered. If neither the task nor the thread have exception ports registered, the thread encountering the exception is terminated.

The occurrence of an exception invokes a four step process involving the thread that caused the exception (the 'victim') and the process that handles the exception (the 'handler').

1. The kernel notifies the *handler* with an exception message.
2. The kernel suspends the *victim*, until the reply message from the handler is received.
3. The handler receives the notification.
4. The handler may either

- Handle exception and cause the thread to resume execution.

- Cause termination of the exception causing thread.

Thus, the exception rpc consists of two messages; an initial message to invoke the rpc (sent from the kernel to the handler), and a reply message to complete the rpc (received from the handler by the kernel). The initial message consists of the following fields :

- Reply port for the rpc.
- The identities of the thread that caused the exception and the corresponding task.
- Exception class.

The reply message contains the return code from the handler that handled the exception. The exception causing thread is resumed or terminated depending on the return code ('SUCCESS' or 'FAILURE'). The format of these messages are defined in '<exception.h>'

The Microkernel provides IEEE 754 Exception handling support for the following types of exceptions.

```
divide_by_zero
underflow
```

overflow
invalid operation

On any of these floating-point exceptions, the IEEE 754 exception handler gains control once the trap handler discovers the type of the exception. The handlers then inspect the appropriate floating point source and destination registers (as could be figured out from the faulting instruction), sets the result registers with correct results (as defined by the IEEE 754 standard) and returns.

fp_exception_handler:
if(ieee_754 exception)
read floating-point queue and get faulting instruction and fault address;
determine kind of operation performed;
determine source and destination registers;
compute correct ieee754 result as defined by the standard;
set destination registers with correct results;
return to trap handler;

5.13 OBP Services

The Microkernel utilizes the standard OBP driver handles for display/serial port I/O. Since the drivers for these devices are complex and would make the Microkernel "huge", they are directly being used from OBP. For this reason, the Microkernel cannot use the top 16MB of the address space (which is where OBP is mapped)

The Microkernel uses the OBP handles "putchar" and "getchar" to read and write from/to the serial port (in the absence of a display). The Microkernel uses other OBP services to search and locate a node and get OBP properties (free physical memory, etc.)

5.14 Machine Dependent Interface

The PARAS microkernel has a few machine dependent internal routines. They are

save_context: Saves the context of a thread.
load_context: Loads the context of a thread.
pcb_init: Initialize a thread's PCB.
thread_start: Initialize a thread's PC.
thread_setstatus: Set thread's hardware state.
thread_getstatus: Get thread's hardware state.
initial_context: Initialize context of first thread.
thread_dup: Hack for UNIX compatibility to implement fork easily.

5.14 Support Functions

The Microkernel contains a number of support functions. In particular, the following are of high importance.

a. queue management functions to manage run queues, wait queues, timer queues, etc.
b. fast string management functions
c. special block copy support for fast intra-kernel copies used specifically by the IPC
d. debug print routines

5.3 PARAS Tasks and Threads Interface

The process and task management operations that can be accessed by the microkernel user are discussed below. More detailed discussion can be found in the *PARAS Microkernel Interface Manual*.

TaskCreate - Create a User Task

□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□ □□□□□□□□□□□□□ □□□□□□□□□□

Description

□□□□□□□□□□ creates a new task. The resulting task (‘child_task’) initially contains no threads, has no special ports associated with it, and has no ports registered.

TaskDelete - Delete a User Task

□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□ □□□□□□□□□□□□□

Description

□□□□□□□□□□ destroys the task specified by *target_task* and all its threads. All resources that are used by the task are freed.

TaskSuspend - Suspend a Specified Task

□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□ □□□□□□□□□□□□□

Description

Increments the task’s suspend count and stops all the threads in the task. This call does not return until all the threads in *target_task* are suspended. The suspend count may become greater than one, with the effect that it will take more than one resume call to restart the task.

TaskResume - Resume a Specified Task to Execution State

□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□ □□□□□□□□□□□□□

Description

Decrements the task’s suspend count. If it becomes zero, all threads with zero suspend count in the task are resumed. The suspend count may not become negative.

ThreadCreate - Create a Thread from a Specified Task

□□□□□□
□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□ □□
□□□□□□ □□□□□□□□□□□□□

▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫

Description

▫▫▫▫▫▫▫▫▫▫▫▫ creates a new thread within the task specified by *parent_task*. The new thread has a suspend count of one and has no associated processor state. To get the new thread to run, *ThreadSetRegisters* is called to set the processor state, and then *ThreadResume* is called to get the thread scheduled to execute. The new thread has no exception port set and starts execution with a priority of 16.

ThreadDelete - Remove a Thread from a Specified Task

▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫

Description

▫▫▫▫▫▫▫▫▫▫▫▫ destroys the thread specified by *target_thread.*

ThreadSuspend - Suspend a Currently Executing Thread

▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫

Description

Increments the thread's suspend count and prevents the thread from executing any more user level instructions. The suspend count may become greater than one with the effect that it will take more than one resume call to restart the thread.

ThreadResume - Resume a Specified Thread to Execution State

▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫

Description

Decrements the thread's suspend count. If the count becomes zero, the thread is resumed. If it is still positive, the thread is left suspended. The suspend count may not become negative.

▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫

Description

▫▫▫▫▫▫▫▫ returns the identifier of the calling task in *task_id*.

ThreadSelf - Get Thread-Id of a Current Thread

▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫▫▫▫ ▫▫▫▫▫▫▫▫▫

Description

□□□□□□□□□□□□ returns the identifier of the calling thread in *thread_id*.

ThreadSetExceptionPort - Set exception port for thread

Syntax

□□□□□□□□ □□□□□□□□□□□

□□□□□□□□□□□□□□ □□

□□□□□□□□ □□□□□□□

□□□□□□ □□□□□□□□□□□□□□□□□

Description

□□□□□□□□□□□□□□□□□□□□□□□□ sets the specified port as the exception port of the thread specified by *thread*. The thread's exception port is the port to which messages are sent by the kernel when an exception occurs. If the thread has no exception port set, these messages are sent to the task's exception port.

TaskSetExceptionPort - Set exception port for task

□□□□□□

□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□ □□

□□□□□□ □□□□□

□□□□□□ □□□□□□□□□□□□□□□□□

Description

□□□□□□□□□□□□□□□□□□□□□□□□ sets the specified port as the exception port of the task specified by *task*. The task's exception port is the port to which messages are sent by the kernel when an exception occurs and the thread causing the exception has no exception port set. If neither the task nor the thread have exception ports registered, the thread encountering the exception is terminated.

ThreadGetExceptionPort - Read exception port of thread

□□□□□□

□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□ □□

□□□□□□ □□□□□□□

□□□□□□ □□□□□□□□□□□□□□□□□□ □□□□□□□□□□

Description

□□□□□□□□□□□□□□□□□□□□□□□□ returns the port identifier of the exception port of the thread specified by *task* in 'exception_port'.

TaskGetExceptionPort - Read exception port of task

□□□□□□

□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□ □□

□□□□□□ □□□□□

□□□□□□ □□□□□□□□□□□□□□□□□□ □□□□□□□□□□

Description

□□□□□□□□□□□□□□□□□□□□□□□□ returns the port identifier of the exception port of the task specified by *task* in *exception_port*.

TaskListThreads - Read all threads belong to task

Description

gets the thread identifiers of all the threads contained in *target_task*. The microkernel creates an array in the calling thread's address space and returns the address in *thread_list*. The caller may deallocate the array using *RegionDeallocate*, when the data is no longer needed.

TaskRegisterPorts - Register a port to a specified task

Description

registers a list of ports with the kernel on behalf of a specific task. Some of the slots in this list are reserved. The number of reserved slots is given by the constant *N_RESERVED_SLOTS*. Currently, the process server, the Unix server and the debugger reserve one slot each.

These slot numbers are given by the constants *PROC_SERVER_SLOT*, *UNIX_SERVER_SLOT* and *DEBUGGER_SLOT* respectively. These ports can be retrieved by *TaskListRegisteredPorts*. The number of ports which may be registered is fixed and given by the constant *MAX_SLOTS* defined.

This call is intended to be used by runtime support modules.

TaskListRegisteredPorts - Read list of ports registered to a task

Description

`TaskListRegisteredPorts` returns the list of ports registered with the kernel for 'target_task'.

ThreadGetRegisters - Read thread's register status

Description

`ThreadGetRegisters` returns the register information for ‘target_thread’. The ‘target_thread’ may not be the currently executing thread. The definition of the state structures can be found in [illegible] header file.

ThreadSetRegisters - Set thread’s register status

[illegible]
[illegible]
[illegible]
[illegible] [illegible]
[illegible] [illegible]
[illegible] [illegible]

Description

[illegible] sets the register state of *target_thread* to the state specified by the contents of *state*. *target_thread* may not be the currently executing thread.

The definition of the state structures can be found in <*machine.h*>.

ThreadSleep - Place thread in sleep state for a specified milliseconds

[illegible]
[illegible]
[illegible] [illegible]
[illegible] [illegible]

Description

`ThreadSleep` forces the current thread to sleep for a duration of *delay* milliseconds. Since the granularity of the system timer is limited to the internal clock frequency (10 ms on PARAM 9000), the actual elapsed time will be rounded to the next higher clock interval.

ThreadInfo - Read thread’s state information

[illegible]
[illegible]
[illegible]
[illegible] [illegible]
[illegible] [illegible]
[illegible] [illegible] [illegible]

[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]

The possible values of the run_state field are the following:

- [illegible], thread is running normally.
- [illegible], thread is suspended
- [illegible], thread is waiting normally
- [illegible], thread is in an uninterruptible wait
- [illegible], thread is halted at a clean point

[illegible]

`ThreadInfo` returns information of the specified thread. The *thread_info* data structure is supplied by the caller and is returned filled with information.

ThreadGetPriority - Read priority of a thread

[illegible]

[illegible] [illegible]

[illegible] [illegible]

[illegible] [illegible]

[illegible] [illegible] [illegible]

Description

[illegible] returns the priority of *target_thread* in ‘priority’.

ThreadSetPriority - Set priority of a thread

[illegible]

[illegible] [illegible]

[illegible] [illegible]

[illegible] [illegible]

[illegible] [illegible]

Description

`ThreadSetPriority` sets the priority of *target_thread* to *priority*. The value of *priority* is restricted to the legal priority range (16-31)