Chapter 13
Tolerating timing faults

Throughout this book it has been assumed that real-time systems have high reliability requirements. One method of achieving this reliability is to incorporate fault tolerance into the software. The inclusion of timing constraints introduces the possibility of these constraints being broken at run-time and failures occurring in the time domain. With soft systems, a task may need to know if a timing constraint has been missed, even though it can accommodate this under normal execution. More importantly, in a hard system (or subsystem), where deadlines are critical, a missed deadline needs to trigger some error recovery routine.

If the system has been shown to be schedulable under worst-case execution times then it is arguable that deadlines cannot be missed. However, the discussions of reliability in Chapter 2 indicated strongly the need for a multifaceted approach to reliability; that is, prove that nothing can go wrong and include routines for adequately dealing with the problems that arise when they do.

This chapter considers the causes of timing faults and how they can be tolerated within the context of the dynamic redundancy approach to fault tolerance introduced in Chapter 2.

13.1 Dynamic redundancy and timing faults

In Chapter 2, the four phases of dynamic software fault tolerance were introduced. These are now reviewed in the context of timing faults.

(1) **Error detection** – Most timing faults will eventually manifest themselves in the form of missed deadlines.

(2) **Damage confinement and assessment** – When deadlines have been missed, it must be decided which tasks in the system are at fault. Ideally confinement techniques should ensure that only faulty tasks miss their deadlines.

(3) **Error recovery** – The response to deadline misses requires that the application undertakes some recovery, perhaps providing a degraded service.

(4) **Fault treatment and continued service** – Timing errors often result from transient overloads. Hence they can often be ignored. However, persistent deadline misses may indicate more serious problems and require some form of maintenance to be undertaken.

In a system that has been 'proved' correct in the timing domain via the schedulability analysis techniques presented in Chapter 11, deadlines could still be missed if:

- worst-case execution time (WCET) calculations were inaccurate (optimistic rather than pessimistic);
- blocking times were underestimated;
- assumptions made in the schedulability checker were not valid;
- the schedulability checker itself had an error;
- the scheduling algorithm could not cope with a load even though it is theoretically schedulable;
- the system is working outside its design parameters, for example sporadic events occurring more frequently than was assumed in the schedulability analysis.

In this latter case (for instance, an information overflow manifesting itself as an unacceptable rate of interrupts), the system designers may still wish for fail-soft or fail-safe behaviour.

Chapter 2 introduced the *fault* → *error* → *failure* chain. Assuming the schedulability analysis is correct, the following chains are possible in the context of priority-based systems (dos Santos and Wellings, 2008).

(1) *Fault* (in task τ_i's WCET calculation or assumptions) → *error* (overrun of τ_i's WCET) → *error propagation* (deadline miss of τ_i) → *failure* (to deliver service in a timely manner).

(2) *Fault* (in task τ_i's WCET calculation or assumptions) → *error* (overrun of τ_i's WCET) → *error propagation* (greater interference on lower-priority tasks) → *error propagation* (deadline miss of lower priority tasks) → *failure* (to deliver service in a timely manner).

(3) *Fault* (in task τ_i's minimum inter-arrival time assumptions) → *error* (greater computation requirement for τ_i) → *error propagation* (deadline miss of τ_i) → *failure* (to deliver service in a timely manner).

(4) *Fault* (in task τ_i's minimum inter-arrival time assumptions) → *error* (greater interference on lower priority tasks) → *error propagation* (deadline miss of lower-priority tasks) → *failure* (to deliver service in a timely manner).

(5) *Fault* (in task τ_i's WCET calculation or assumptions when using a shared resource) → *error* (overrun of τ_i's resource usage) → *error propagation* (greater blocking time of higher-priority tasks sharing the resource) → *error propagation* (deadline miss of higher-priority tasks) → *failure* (to deliver service in a timely manner).

Similar chains will exist for other scheduling approaches and faults, where instead of the term 'lower/higher priority' the corresponding eligibility criterion can be substituted (e.g. 'later/earlier absolute deadline').

To be tolerant of timing faults, it is necessary to be able to detect:

- miss of a deadline – the final error in all the above error propagation chains;
- overrun of a worst-case execution time – potentially causing the task and/or lower eligibility tasks to miss their deadlines (error chains 1 and 2);
- a sporadic event occurring more often than predicted – potentially causing the task and/or lower eligibility tasks to miss their deadlines (error chains 3 and 4);
- overrun in the usage of a resource – potentially causing higher eligibility tasks to miss their deadlines (error chain 5).

Of course the last three error conditions do not necessarily indicate that deadlines will be missed; for example, an overrun of WCET in one task might be compensated by a sporadic event occurring less often than the maximum allowed. Hence, the damage confinement and assessment phase of providing fault tolerance (introduced in Section 2.5) must determine what actions to take. Both forward and backward recovery are then possible.

If timing faults are to be handled, their associated error conditions have to be detected first. If the run-time environment or operating system is aware of the salient characteristics of a task (as it is in the Real-Time Java approach, for example), it will be able to detect problems and bring them to the attention of the application. Alternatively, it is necessary to provide primitive facilities that will allow the application to detect its own timing errors. With all error detection mechanisms, the earlier the problem can be detected the more chance there is to pinpoint the problem and the more time there is to recover.

The following sections will discuss error detection mechanisms for the above timing faults, consider possible error confinement approaches, and identify strategies for recovery. Fault treatment typically involves maintenance, which is a topic outside the scope of this book. However, it is noted that for non-stop real-time applications some form of dynamic change management or mode change is required. The impact of this can have serious real-time implications.

13.2 Deadline miss detection

Deadline miss detection is the minimum that is required if a real-time system is to tolerate timing failures. It is a 'catch all' mechanism that will detect even problems outside the failure hypothesis. For example, it will detect problems resulting from errors in the schedulability analysis. Of course, with all 'catch all' mechanisms, it may be difficult to pinpoint the cause of the problem and it leaves little time for recovery.

This section discusses how deadline miss detection is facilitated in Ada, Real-Time Java and C/Real-Time POSIX. Strategies for dealing with deadline misses are considered in Section 13.7.

13.2.1 Ada

Ada 2005 allows the deadline of a task to be specified and this parameter can be used to influence scheduling (see Section 12.8). However, the Ada run-time support system

does not use this information to detect deadline misses. For this, the language provides primitive mechanisms that the programmer has to use to detect the missed deadline itself. One way of achieving this is to use the asynchronous transfer of control facility discussed in Section 7.6.1. Hence to detect a deadline overrun of the periodic task given in Section 10.2 requires the main functionality of the task to be embedded in a 'select then abort' statement.

```
task body Periodic_T is
  Next_Release : Time;
  Next_Deadline : Time;
  Release_Interval : constant Time_Span := Milliseconds(...);
  Deadline : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release) and
  -- next deadline (Next_Deadline)
  loop
    select
      delay until Next_Deadline;
      -- deadline miss detected here
      -- perform recovery
    then abort
      -- sample data (for example) or
      -- calculate and send a control signal
    end select;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
    Next_Deadline := Next_Release + Deadline;
  end loop;
end Periodic_T;
```

A similar approach can be used to detect a deadline miss in a sporadic task.

One of the problems with this approach is that it combines detection with a particular recovery strategy, that of stopping the task from what it is doing. This is, clearly, one option. Another is to use a different task to handle the deadline miss. The possible recovery strategies could include extending the deadline, lowering the errant task's priority, or some other action short of terminating it (see Section 13.7).

To simply detect a deadline miss, the Ada timing event facility that was discussed in Section 10.4 can be used (see Program 10.6). In Section 2.5.1, the watchdog timer approach to fault detection was presented. This can be easily programmed using Ada's timing events. The essential idea is that on initializing the watchdog with a first deadline and a period, the watchdog sets up a timing event for the first deadline. The task must now call the watchdog to reset the timing event before its deadline has expired. If it does so, the watchdog sets the event to expire at the next deadline, and so on. If the task does not call to reset the event, the event will be triggered.

The specification of the watchdog is given first:

```
protected type Watchdog(Event : access Timing_Event) is
  procedure Initialize(First_Deadline : Time;
                       Required_Period : Time_Span);
  entry Alarm_Control(T: out Task_Identity);
     -- Called by alarm handling task
```

```
  procedure Call_In;
     -- Called by application code when it completes.

  pragma Interrupt_Priority (Interrupt_Priority'Last);
private
  procedure Timer(Event : in out Timing_Event);
     -- Timer event code, ie the handler.
  Alarm : Boolean := False;
  Tid : Task_Identifier;
  Next_Deadline : Time;
  Period : Time_Span;
end Watchdog;
```

This watchdog object has a common structure. An entry with an initially closed barrier holds back a monitoring task that will be released by the handler if the handler executes. In this example the handler is actually never executed unless there is a missed deadline. Each time the monitored task calls `Call_In`, the timing event is reset to a point in the future. Only if another call does not occur before its next deadline will the handler be executed and the barrier opened releasing the monitoring task.

```
protected body Watchdog is
  procedure Initialize(First_Deadline : Time;
                       Required_Period : Time_Span) is
  begin
    Next_Deadline := First_Deadline;
    Period := Required_Period;
    Set_Handler(Event.all, Next_Deadline, Timer'Access)
    Tid = Current_Task;
  end Initialize;

  entry Alarm_Control(T: out Task_Identity) when Alarm is
  begin
    T := Tid;
    Alarm := False;
  end Alarm_Control;

  procedure Timer(Event : in out Timing_Event) is
  begin
    Alarm := True;
    -- Note no use is made of the parameter in this example
  end Timer;

  procedure Call_in is
  begin
    Next_Deadline := Next_Deadline + Period;
    Set_Handler(Event.all, Next_Deadline, Timer'Access);
    -- Note this call to Set_Handler cancels the previous call
  end Call_in;
end Watchdog;
```

The revised structure of the periodic task is shown below:

```
with Watchdogs; use Watchdogs;
...
Watch : Watchdog;
```

```
Deadline_Miss_Event : aliased Timing_Event;
Set_Handler(Deadline_Miss_Event, Next_Deadline, Timer'Access);

task body Periodic_T is
  Next_Release : Time;
  Release_Interval : constant Duration := ...; -- or
  Release_Interval : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  -- and first deadline (First_Deadline)
  Watch.Initialize(First_Deadline, Release_Interval);
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    Watch.Call_In;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

13.2.2 Real-Time Java

Unlike the Ada run-time system, the Real-Time Java virtual machine does monitor the deadlines of real-time threads and will release asynchronous event handlers when periodic or sporadic tasks are still executing when their deadlines have passed, the handlers (`missHandler`) being identified with the release parameters associated with the real-time threads (see Program 10.3).

The full semantics of Real-Time Java's deadline miss detection are somewhat complex. Program 13.1 shows the associated methods. Recall the structure of a period real-time thread from Chapter 10:

```
public class Periodic extends RealtimeThread {
  public Periodic(PeriodicParameters P)
  { ... };

  public void run() {
    boolean deadlineMet = true;
    while(deadlineMet) {
     // code to be run each period
     ...
     deadlineMet = waitForNextPeriod();
    }
  }
}
```

An aperiodic or sporadic thread has a similar structure only with a call to `waitForNextRelease` instead of `waitForNextPeriod`.

The full semantics can be summarized by the following points.

- If the real-time thread misses its deadline, and it has an associated deadline miss handler, this is released at the point the deadline expires. The real-time thread is

Program 13.1 An extract of the `RealtimeThread` class showing methods used for deadline miss detection.

```
package javax.realtime;
public class RealtimeThread extends Thread
       implements Schedulable {
  ...

  // the following methods are used with periodic execution
  public boolean waitForNextPeriod();
  public void deschedulePeriodic();
    // deschedules the periodic thread at the end
    // of its current release
  public void schedulePeriodic();
    // reschedules the periodic thread at its next release event

  // used for aperiodic and sporadic execution
  public boolean waitForNextRelease();
  public void deschedule();
    // deschedules the aperiodic thread at the end
    // of its current release
  public void schedule();
    // reschedules the aperiodic thread at its next release event
  ...
}
```

automatically de-scheduled – this means that at the end of its current release (when it calls `waitForNextPeriod`/`waitForNextRelease`), the scheduler will no longer consider the thread for possible execution until it has been explicitly rescheduled by the application via a call to `schedulePeriodic`/`schedule`. At this point the thread becomes eligible for execution at its next release event.

- If there is no associated handler, when the deadline miss occurs a count (called `deadlineMiss`) of the number of missed deadlines is incremented.
- The `waitForNextPeriod` method (wFNP) has the following semantics (`waitForNextRelease` has similar semantics):
  - If no deadlines have been missed, wFNP returns true when its next release event occurs.
  - When the `deadlineMiss` count is greater than zero and the previous call to wFNP returned true, wFNP decrements the `deadlineMiss` count and returns false immediately. This situation indicates that the current release has missed its deadline. At this point, the current release is still active.
  - When the `deadlineMiss` count is greater than zero and the previous call to wFNP returned false, wFNP decrements the `deadlineMiss` count and returns false immediately. This situation indicates that the next release time has already passed and the next deadline has already been missed. At this point, the current release has completed and the next release is active.

- When a deadline miss handler has been released and the `deadlineMiss` count equals zero and no call to the `schedulePeriodic` method has occurred since the deadline miss handler was released, wFNP de-schedules the real-time thread until an explicit call to the `schedulePeriodic` method occurs (probably by the released handler); wfNP then returns true at the point of the next release after the call to `schedulePeriodic`. At this point, the next release is active.

An example of using the Real-Time Java facilities will be given in Section 13.7 where strategies for recovery are considered.

13.2.3 C/Real-Time POSIX

C/Real-Time POSIX allows timers to be created and set which will generate user-defined signals (`SIGALRM` by default) when they expire. Hence this will allow the process to decide what is the correct course of action to pursue. Program 10.7 shows a typical C interface.

The watchdog timer approach to fault detection can be easily programmed using POSIX signals. For example, consider the case where one thread (`monitor`) creates another thread (`server`) and wishes to monitor its progress to see if it meets a deadline. The deadline of the `server` is given by `struct timespec deadline`. The monitor thread creates a per process timer indicating a signal handler to be executed if the timer expires. It then creates the `server` thread and passes a pointer to the timer. The `server` thread performs its action and then deletes the timer. If the alarm goes off, the `server` is late.

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer; /* timer shared between monitor and server */

struct timespec deadline = ...;
struct timespec zero = ...;

struct itimerspec alarm_time, old_alarm;

struct sigevent s;

void server(timer_t *watchdog) {
  /* perform required service */
  TIMER_DELETE(*watchdog);
}

void watchdog_handler(int signum, siginfo_t *data,
                      void *extra)
{
  /* SIGALRM handler */

  /* server is late: undertake recovery */
}
```

```
void monitor() {
  pthread_attr_t attributes;
  pthread_t serve;


  sigset_t mask, omask;
  struct sigaction sa, osa;
  int local_mode;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, SIGALRM);

  sa.sa_flags = SA_SIGINFO;
  sa.sa_mask = mask;
  sa.sa_sigaction = &watchdog_handler;

  SIGACTION(SIGALRM, &sa, &osa); /* assign handler */


  alarm_time.it_value = deadline;
  alarm_time.it_interval = zero; /* one shot timer */

  s.sigev_notify = SIGEV_SIGNAL;
  s.sigev_signo = SIGALRM;

  TIMER_CREATE(CLOCK_REALTIME, &s, &timer);

  TIMER_SETTIME(timer, TIMER_ABSTIME, &alarm_time, &old_alarm);

  PTHREAD_ATTR_INIT(&attributes);
  PTHREAD_CREATE(&serve, &attributes, (void *)server, &timer);

}
```

However, as noted in Section 7.5.1, if a process is multithreaded, the signal is sent to the whole process, not an individual thread. In general, therefore, to generalize the above approach it is necessary to pass a value with the signal being generated to indicate the associated thread (via the `sigev_value` component of the `sigevent` structure).

13.2.4 Timing error detection at the block level

Task-level deadlines are the most common deadlines. However, as noted in Section 9.6, deadlines can occur at a finer granularity, for example at the block level. The watchdog approach given in the previous sections allows this block-level detection to be achieved. However, neither Ada, Real-Time Java or C/Real-Time POSIX provides direct support for block-level deadlines. The research language DPS considered in Section 10.6.3 illustrates the type of support that could be given.

In DPS, timing errors are associated with exceptions:

```
start <timing constraints> do
  -- statements
exception
```

```
  -- handlers
end
```

In addition to the necessary computations required for damage limitation, error recovery and so on, the handler may wish to extend the deadline period and continue execution of the original block. Thus a *resumption* rather than *termination* model may be more appropriate (see Chapter 3).

In a time-dependent system, it may also be necessary to give the deadline constraints of the handlers. Usually the execution time for the handler is taken from the temporal scope itself; for example, in the following, the statement sequence will be prematurely terminated after 19 time units:

```
start elapse 22 do
  -- statements
exception
  when elapse_error within 3 do
    -- handler
end
```

As with all exception models, if the handler itself gives rise to an exception this can only be dealt with at a higher level within the program hierarchy. If a timing error occurs within a handler at the task level then the task must be terminated (or at least the current iteration of the task). There might then be some system-level handlers to deal with failed tasks or it may be left to the application software to recognize and cope with such events.

If exception handlers are added to the coffee-making example given with DPS in Section 10.6.3, the code would have the following form (exceptions for logic errors such as 'no cups available' are not included). It is assumed that only `boil_water` and `drink_coffee` have any significant temporal properties; timing errors are, therefore, due to overrun on these activities.

```
from 9:00 to 16:15 every 45 do
  start elapse 11 do
    get_cup
    boil_water
    put_coffee_in_cup
    put_water_in_cup
  exception
    when elapse_error within 1 do
      turn_off_kettle  -- for safety
      report_fault
      get_new_cup
      put_orange_in_cup
      put_water_in_cup
    end
  end

  start after 3 elapse 26 do
    drink
  exception
    when elapse_error within 1 do
```

```
      empty_cup
    end
  end
  replace_cup
exception
  when any_exception do
    null   -- go on to next iteration
  end
end
```

13.3 Overrun of worst-case execution time

Good fault tolerance practices attempt to confine the consequences of an error to a well-defined region of the program. Facilities such as modules, packages and atomic actions help with this goal. However, if a task consumes more of the CPU resource than has been anticipated, then it may not be that task that misses its deadline. For example, in the case of a high-priority task with a fair amount of slack time, the tasks that will miss their deadlines may be lower-priority tasks with less slack available. Ideally, it should be possible to catch the timing error in the task that caused it. This implies that it is necessary to be able to detect when a task overruns the worst-case execution time that the implementer has allowed for it. Of course, if a task is non-preemptively scheduled (and does not block waiting for resources), its CPU execution time is equal to its elapse time and the same mechanisms that were used to detect deadline overrun can be used. However, tasks are usually preemptively scheduled, and this makes measuring CPU time usage more difficult. It usually has to be supported explicitly in the host operating system.

13.3.1 Execution-time clocks in Real-Time POSIX

POSIX supports execution-time monitoring using its clock and timer facilities. Two clocks are defined: `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID`. These can be used in the same way as `CLOCK_REALTIME`. Each process/thread has an associated execution-time clock; calls to:

```
clock_settime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);

clock_getres(CLOCK_PROCESS_CPUTIME_ID, &some_timespec_value);
```

will set/get the execution-time or get the resolution of the execution-time clock associated with the calling process (similarly for threads).

Two functions allow a process/thread to obtain and then access the clock of another process/thread.

```
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

The timers defined in Program 10.7 can be used to create timers which, in turn, can be used to generate process-signals when the execution time set has expired. It is implementation-defined what happens if a `timer_create` is used with a `clock_id` different from that of the calling process/thread. As the signal generated by the expiry of the timer is directed at the process, it is application-dependent which thread will get the signal if a thread's execution-time timer expires. An application can disallow the use of the timer for a thread (because of the overhead in supporting the facility).

As with all execution-time monitoring, it is difficult to guarantee the accuracy of the execution-time clock in the presence of context switches and interrupts.

13.3.2 Execution-time clocks in Ada

Ada 2005 directly supports execution-time clocks for tasks, and supports timers that can be fired when tasks have used a defined amount of execution time. Indeed it has added a clock per task that measures the task's execution time. A package, `Ada.Execution_Time` – see Program 13.2, is defined that is similar in structure to `Ada.Calendar` and `Ada.Real_Time`.

Program 13.2 An abridged version of the `Ada.Execution_Time` package.

```
with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;

package Ada.Execution_Time is
   type CPU_Time is private;
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant :=
         <implementation-defined-real-number>;
   CPU_Tick : constant Time_Span;

   function Clock
     (T : Ada.Task_Identification.Task_ID
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;

   function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
   -- similarly for "-", "<", "<=", ">" and ">="

   procedure Split
      (T : CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count; TS : Time_Span)
            return CPU_Time;
   ...
private
   -- Not specified by the language.
end Ada.Execution_Time;
```

When a task is created so is an execution-time clock. This clock registers zero at creation and starts recording the task execution time from the point at which the task starts its *activation* (see Section 4.4). To read the value of any task's clock a `Clock` function is defined. So in the following a loop is allocated 7 ms of execution time before exiting at the end of its current iteration:

```
Start : CPU_Time;
Interval : Time_Span := Milliseconds(7);
...

Start := Ada.Execution_Time.Clock;
while Ada.Execution_Time.Clock - Start < Interval loop
  --code
end loop;
```

Note the '-' operator returns a value of type `Time_Span` which can then be compared with `Interval`.

To monitor the execution time of each invocation of a periodic task, for example, is simple:

```
Last : CPU_Time;
Exe_Time : Time_Span;
Last := Execution_Time.Clock;
loop
  -- code of task
  Exe_Time := Ada.Execution_Time.Clock - Last;
  Last := Ada.Execution_Time.Clock;
  -- print out or store Exe_Time
 delay until ...
end loop;
...
```

As well as monitoring a task's execution time profile, it is also possible to trigger an event if its execution-time clock gets to some specified value. A child package of `Ada.Execution_Time` provides support for this type of event – see Program 13.3.

Each `Timer` event is strongly linked to the task that will trigger it. This static linkage is ensured by the access discriminant for the type that is required to be **constant** and **not null**.

The handler type is standard, but there is now a need to specify the minimum ceiling priority the associated protected object must have if ceiling violation is to be avoided. This priority will be set by the supporting implementation.

The `Set_Handler` procedures and the other routines all have the same properties as those defined with timing events. However, in recognition that an implementation may have a limited capacity for timers, or that only one timer per task is possible, the exception `Timer_Resource_Error` may be raised when a `Timer` is defined or when a `Set_Handler` procedure is called for the first time with a new `Timer` parameter.

An example of using the Ada facilities will be given in Section 13.7 where strategies for recovery are considered.

Program 13.3 The `Ada.Execution_Time.Timers` package.

```
package Ada.Execution_Time.Timers is

   type Timer(T : not null access constant
      Ada.Task_Identification.Task_ID) is tagged
      limited private;

   type Timer_Handler is access protected
      procedure(TM : in out Timer);

   Min_Handler_Ceiling : constant System.Any_Priority :=
    <Implementation Defined>;

   procedure Set_Handler(TM: in out Timer;
          In_Time : Time_Span; Handler : Timer_Handler);
   procedure Set_Handler(TM: in out Timer;
          At_Time : CPU_Time;  Handler : Timer_Handler);

   procedure Cancel_Handler(TM : in out Timer;
                            Cancelled : in out Boolean);

   function Current_Handler(TM : Timer) return Timer_Handler;

   function Time_Remaining(TM : Timer) return Time_Span;

   Timer_Resource_Error : exception;

private
   -- Not specified by the language.
end Ada.Execution_Time.Timers;
```

13.3.3 Execution-time monitoring in Real-Time Java

Unlike, Ada and Real-Time POSIX, Real-Time Java does not support explicitly CPU-time clocks. Instead it allows a 'cost' value to be associated with the execution of a schedulable object as one of the attributes in the `ReleaseParameters` class (see Program 10.3). This is some measure of how much of the processor's time is required to execute the computation associated with the real-time thread's release (that is, after it has been released and until it has completed). An optional facility can be supported by the real-time virtual machine that keeps track of the CPU time consumed by the thread on each release. An asynchronous event can be fired if the real-time thread consumes more than this CPU time.

Real-Time Java (as of RTSJ Version 1.1) also allows information on the amount of execution time consumed by a real-time thread during its releases to be obtained via the following methods in the `RealtimeThread` class:

```
RelativeTime getMaxConsumption()
RelativeTime getMaxConsumption(RelativeTime dest);
// Returns the maximum amount of CPU time this
```

```
// schedulable object has used in a single release
// in a newly-allocated RelativeTime object or
// in an instance supplied by the caller.


RelativeTime getMinConsumption()
RelativeTime getMinConsumption(RelativeTime dest)
// Returns the minimum amount of CPU time this
// schedulable object has used in a single release
// in a newly-allocated RelativeTime object or
// in an instance supplied by the caller.
```

Real-Time Java does not require that an implementation monitor the processing time consumed by schedulable objects, as this requires support from the underlying operating system. Many operating systems do not currently provide this support.

If cost monitoring is supported, then Version 1.02 of Real-Time Java requires that the priority scheduler gives a real-time thread a CPU budget of no more than its cost value on each release. Hence, if a real-time thread overruns its cost budget, it is automatically de-scheduled (made not eligible for execution) immediately. It will not be rescheduled until either its next release occurs (in which case its budget is replenished) or its associated cost value is increased. In Version 1.1 of Real-Time Java, this will be the default policy. However, there will also be a mechanism that allows just notification rather than enforcement.

13.4 Overrun of sporadic events

A sporadic event firing more frequently than anticipated can have an enormous impact on a system attempting to meet hard deadlines. Here the term **sporadic overrun** is used to denote this fault. Where the event is the result of an interrupt, the consequences can be potentially devastating. For example, during the first landing on the moon, the guidance computer was reset after a CPU on the Lunar Landing Module was flooded with radar data interrupts. The landing was nearly aborted as a result (Regehr, 2007).

There are a variety of techniques that have been developed over the years to deal with this situation. The two basic approaches are either to stop the early firing from occurring, or to bound the total amount of the CPU time allocated to all events from the same source. The latter use 'server' technology which will be discussed in detail in Section 13.6.1 in the context of damage confinement. The features provided by C/Real-Time POSIX will also be considered there.

Here the focus is on what support can be provided to prohibit the firings or to detect them when they occur and take some corrective action. Two types of events are considered: those resulting from hardware interrupts and those resulting from the software firing of an event.

13.4.1 Handling sporadic event overruns in Ada

In keeping with the overall Ada philosophy, low-level mechanisms can be used to handle event overruns.

Where the event is triggered by a hardware interrupt, on most occasions the interrupt can be inhibited from occurring by manipulating the associated device control registers (see Chapter 14). A simple approach is illustrated below. Here, the assumption is that there is a required minimum inter-arrival time (MIT) between interrupt occurrences.

```
protected Sporadic_Interrupt_Controller is
  procedure Interrupt; -- mapped onto interrupt
  entry Wait_For_Next_Interrupt;
private
  procedure Timer(Event : in out Timing_Event);
  Call_Outstanding : Boolean := False;
  MIT : Time_Span := Milliseconds(...);
end Sporadic_Interrupt_Controller;

Event: Timing_Event;

protected body Sporadic_Interrupt_Controller is
  procedure Interrupt is
  begin
    -- turn off interrupts
    Set_Handler(Event, MIT, Timer'Access);
    Call_Outstanding := True;
  end Interrupt;

  entry Wait_For_Next_Interrupt when Call_Outstanding is
  begin
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;

  procedure Timer(Event : in out Timing_Event) is
  begin
    -- Turn interrupts back on
  end Timer;
end Sporadic_Interrupt_Controller;
```

Once an interrupt from the device occurs, interrupts are disabled and a timing event is set to expire at the required minimum inter-arrival time. Once this occur, the device's interrupts are enabled.

The sporadic task has a familiar structure:

```
task body Sporadic_T is
begin
  loop
    Sporadic_Interrupt_Controller.Wait_For_Next_Interrupt;
    -- action
  end loop;
end Sporadic_T;
```

Of course, it is device dependent what happens if the device wants to interrupt and is unable to. Usually, the device overruns and any data is lost.

If the event is fired from a software task, then the above approach can be modified so that (for example) an exception is raised.

```
protected Sporadic_Interrupt_Controller is
  procedure Release; -- mapped onto interrupt
  entry Wait_For_Next_Release;
private
  Call_Outstanding : Boolean := False;
  MIT : Time_Span := Milliseconds(...); -- Minimum inter-arrive time
  Last_Release : Time;
end Sporadic_Interrupt_Controller;

MIT_VIOLATION : exception;

protected body Sporadic_Interrupt_Controller is
  procedure Release is
    Now : Time := Clock;
  begin
    if Now - Last_Release < MIT then
      raise MIT_VIOLATION;
    else
      Last_Release := Now;
    end if;
    Call_Outstanding := True;
  end Interrupt;

  entry Wait_For_Next_Release when Call_Outstanding is
  begin
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;
end Sporadic_Interrupt_Controller;
```

Raising an exception is only one possible approach.

The usual constraint on a sporadic task is that there is a minimum separation between any two releases. A generalization of this constraint, which allows for bursts of release events, is to set a limit of M release in any length of time L. Although these constraints are a little more complicated to program, the above approaches can be extended to this *M in L* case.

13.4.2 Real-Time Java and minimum inter-arrival time violations

The previous subsection illustrated how to handle various MIT violation conditions using Ada. In keeping with Ada's philosophy, it is up to the program to detect these conditions and act accordingly.

The Real-Time Java philosophy is the opposite. It provides the mechanisms that allow the real-time JVM to detect MIT violation. The `SporadicParameters` class defines a subclass of release parameters that allow the programmer to specify that a real-time thread is a sporadic thread. The class is shown in Program 13.4.

The following policies are available.

- `mitViolationExcept` – an exception is thrown in the releasing real-time thread. If the violation is on an asynchronous event handler released by the firing of an interrupt, then the policy defaults to the `mitViolationIgnore` policy.

Program 13.4 An abridged version of the `SporadicParameters` class.

```
package javax.realtime;
public class SporadicParameters
        extends AperiodicParameters {
  // fields
  public static final String mitViolationExcept;
  public static final String mitViolationIgnore;
  public static final String mitViolationReplace;
  public static final String mitViolationSave;

  public SporadicParameters(RelativeTime minInterarrival);
  public SporadicParameters(RelativeTime minInterarrival,
    RelativeTime cost, RelativeTime deadline,
    AsyncEventHandler overrunHandler,
    AsyncEventHandler missHandler);

  // methods
  public String getMitViolationBehavior();
  public void setMitViolationBehavior(String behavior);
  public RelativeTime getMinimumInterarrival();
  public void setMinimumInterarrival(
    RelativeTime interarrival);

  ...
}
```

- `mitViolationIgnore` – the release event is ignored.
- `mitViolationReplace` – the last release event is overwritten with the current one.
- `mitViolationSave` – the release event is delayed until the MIT has passed.

It perhaps should be noted that in Real-Time Java interrupt handers are second level interrupt handlers. The program cannot rely on the above approach to prohibit the interrupt from occurring. To do this, it must adopt the approach illustrated in Ada, and turn off interrupts. It is also not possible for the programmer to specify other more general constraints, such as *M in L* discussed above.

13.5 Overrun of resource usage

Problems caused by errors in accessing resources are notoriously difficult to handle. At a functional level, they can corrupt shared state and potentially lead to deadlock. From a timing perspective, the whole *raison d'être* for priority (or more generally, eligibility) inheritance protocols such as those discussed in Section 11.8 is to avoid timing problems. However, even approaches involving inheritance and ceiling protocols can cause problems if the blocking time assumed by the schedulability analysis is incorrect. There are two main potential causes for this:

- a task may overrun it allotted access time for the resource; or
- unanticipated resource contention that has not been taken into account in the blocking-time analysis.

The latter is possible in large systems with the use of prewritten library code. In Section 9.4 timeouts were introduced as a mechanisms for detecting the absence of some expected communication or event. There are several reasons why these are inadequate in the context of blocking time.

(1) With inheritance protocols, blocking is cumulative. Timeout could be used on entry to critical sections; however, the programmer would have to keep a running track of the total blocking time in the current release.
(2) With immediate ceiling protocols, the blocking occurs before execution of the task. Hence a timeout has no use, as there is no contention when accessing the critical section.
(3) Support for critical sections does not always have a timeout mechanisms. For example, both Java's synchronized methods and Ada protected types have no associated timeout mechanisms. Although C/Real-Time POSIX provides a timed version of `mutex_lock`, no indication of how long the calling task was blocked for is returned.

Hence, whilst timeouts have a role to play, their use is limited in this context.

For finer control, it is possible to use detection of WCET overruns at the block level. Hence all resource accesses would have to be policed to ensure that the calling task did not overrun its allotted usage. Of course, as already pointed out in Section 13.1, an overrun in one resource may be compensated by underuse of another. Furthermore, detecting overruns on every resource access may be prohibitively expensive.

As a last resort, overruns in blocking times will, if significant, cause tasks to miss their deadlines, which will be detected. Good programming practice dictates that synchronized code is short and of a simple form. Errors are therefore far less likely.

13.6 Damage confinement

The role of damage confinement of time-related faults is to prevent propagation of the resulting errors to other components in the system. There are two aspects of this that can be identified:

- protecting the system from the impact of sporadic task overruns and unbounded aperiodic activities;
- supporting composability and temporal isolations.

The problem of overruns in sporadic objects has already been mentioned in Section 13.4. Aperiodic events also present a similar problem. As they have no well-defined release characteristics, they can impose an unbounded demand on the processor's time. If not handled properly, they can result in periodic or sporadic tasks missing their deadlines,

even though those tasks have been 'guaranteed'. In Section 11.6.2, **aperiodic servers** were introduced. Aperiodic servers protect the processing resources needed by periodic tasks but otherwise allow aperiodic and sporadic tasks to run as soon as possible. Several types of servers were discussed including the Sporadic Server and Deferrable Server.

When composing systems of multiple applications, whether dynamically or statically, it is often required that each application be isolated from one another. Memory management hardware has provided that isolation in the spatial domain for many years. However, the facility to support temporal isolation, where the applications share the same computing resource, has only recently become available. This has been brought about by **hierarchy schedulers** and **reservation-based systems**. Usually, two levels of scheduling are used; a global (top-level) scheduler and multiple application-level (second-level) schedulers. Typically the application-level scheduler is also called a **server** or **execution-time server** or **group server**. The latter term will be used in this book.

Although the above confinement techniques are similar, they have slightly different emphases. For temporal isolation, the key requirement is that the group server be *guaranteed its budget each period*; that is, it must be possible for the tasks contained within the group to consume all the group's budget on each release (and not be allowed to consume any more). To support aperiodic execution, it is sufficient that the aperiodic server *consumes no more than its budget each period*. Hence schedulability analysis can be undertaken on tasks scheduled within a group. Whereas, the analysis of the impact of an aperiodic server can be bounded, typically no analysis of the tasks contained within the server need be done. If a group contains only sporadic tasks, the budget must be guaranteed. The goal here is to ensure that the sporadic task does not violated the CPU time that has been allocated to it, for example by being released more often than its minimum inter-arrival time and consuming more than its maximum budget on each release.

With group servers, the schedulability analysis is simpler if the associated tasks are 'bound'. The term 'bound' in this context refers to the relationship between the tasks' periods and the period of the group server. A bound relationship is where the periods of the tasks are exact multiples of the period of the group and have arrival times that coincide with its replenishment.

13.6.1 Programming servers with C/Real-Time POSIX

C/Real-Time POSIX supports the Sporadic Server approach to damage confinement as one of the scheduling policies (see Program 12.2). The policy can be applied at both the thread and process levels. As discussed in Section 11.6.2, a Sporadic Server assigns a limited amount of CPU capacity to handle events, and has a replenishment period, a budget and two priorities. The server runs at a high priority when it has some budget left and a low one when its budget is exhausted. When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget. The amount of budget consumed is replenished at the time the server was activated plus the replenishment period. When its budget reaches zero, the server's priority is set to the low value.

At the thread level, the Sporadic Server provides confinement for sporadic and aperiodic activities. Each sporadic thread is assigned a Sporadic Server scheduling policy, and is given appropriate parameters. Note that these say nothing about the minimum

Program 13.5 A typical abridged C interface to support the POSIX sporadic server facilities.

```
#define SCHED_SPORADIC ... /* sporadic server */
#define PTHREAD_SCOPE_SYSTEM ... /* system-wide contention */
#define PTHREAD_SCOPE_PROCESS ...  /* local contention */

typedef ... pid_t;
struct sched_param {
  ...
  timespec sched_ss_repl_period
  timespec sched_ss_init_budget
  int  sched_ss_max_repl
  ...
  };

int sched_setparam(pid_t pid, const struct sched_param *param);
/*  set the scheduling parameters of process pid */

int sched_get_priority_max(int policy);
/* returns the maximum priority for the policy specified */

int sched_get_priority_min(int policy);
/* returns the minimum priority for the policy specified */

int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope);
/* set the contention scope for a thread attribute object */


int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
/* set the scheduling policy for a thread attribute object */

/* All the above integer functions return 0 if successful */
/* There are similar setter methods */
```

inter-arrival time; instead they bound the impact that the thread can have to be equivalent to a periodic activity whose characteristics are that of the server's parameters (shown in Program 13.5). Handling aperiodic activities with this approach is more problematic as it not possible to assign the budget to a group of threads. For them it is necessary to use a Sporadic Server *process*.

A Sporadic Server process is a process that is scheduled according to the sporadic server policy. Hence *all* the threads contained within it share the allocated budget. Hence, timing errors are confined to those threads. C/Real-Time POSIX also supports shared memory objects between processes; hence it is possible to partition a single application between processes and still communicate via shared memory (there is an attribute to POSIX mutexes that caters for this option). If the scheduling has system-wide contention, then the threads will compete at their own priority. If the scheduling has local contention, then the process's priority will be the dominating factor.

13.6.2 Programming servers with Real-Time Java

Real-Time Java provides support for temporal isolation via:

- an optional cost monitoring and enforcement model – see Section 13.3.3;
- sporadic release parameters – see Section 13.4.2;
- processing group parameters – considered in this subsection.

Real-Time Java provides support for group servers via **processing groups**. A processing group is implicitly created when an instance of the `ProcessingGroupParameter` class is created. When processing group parameters are assigned to one or more aperiodic schedulable objects, a server is effectively created. The server's start time, cost (capacity) and period are defined by the particular instance of the parameters. These collectively define the points in time when the server's capacity is replenished. Any aperiodic schedulable object that belongs to a processing group is executed at its own defined priority. However, it only executes if the server still has capacity. As it executes, each unit of CPU time consumed is subtracted from the server's capacity. When the capacity is exhausted, the aperiodic schedulable objects are not allowed to execute until the start of the next replenishment period. If the application only assigns aperiodic schedulable objects of the same priority level to a single `ProcessingGroupParameters` object, then the functionality of a Deferrable Server can be obtained.

Real-Time Java is, however, a little more general. It allows schedulable objects of different priorities to be assigned to the same group, the inclusion of sporadic and periodic schedulable objects, the 'servers' to be given a deadline, and cost overrun and deadline miss handlers to be set. This represents an extensive set of facilities.

If used within the context of an aperiodic server, a cost overrun potentially indicates a transient overload where the aperiodic load cannot be handled effectively. Of course, the tasks will be executed across one of more of the following aperiodic server's periods, but this will impact on their response times. For a Deferrable Server, the deadline would equal the period and deadline misses are not relevant at the server level.

In the context of a group server (execution-time server), a cost overrun is potentially more severe. It indicates that the workload assigned to the group is an underestimate, and consequently some deadlines of the threads may be missed.[1] If the group itself has missed its deadline then the guaranteed capacity given to the server has been violated. The `ProcessingGroupParameters` class is given below in Program 13.6.

Although Real-Time Java does not define any feasibility analysis, the `setIfFeasible` method allows the Processing Group to be guaranteed.

13.6.3 Programming servers in Ada

Although, Ada does not directly support servers, it does provide the primitives from which servers can be programmed.

[1] Real-Time Java supports deadline overrun detection on an individual real-time thread/asynchronous event handler basis.

Program 13.6 The `ProcessingGroupParameter` class.

```
package javax.realtime;
public class ProcessingGroupParameters
          implements Cloneable {

  public ProcessingGroupParameters(
       HighResolutionTime start, RelativeTime period,
       RelativeTime cost, RelativeTime deadline,
       AsyncEventHandler overrunHandler,
       AsyncEventHandler missHandler);

  // methods
  public Object clone();
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();

  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(
            AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(
             AsyncEventHandler handler);
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);

  public boolean setIfFeasible(RelativeTime period,
        RelativeTime cost, RelativeTime deadline);
}
```

Group budgets allow different group servers to be implemented; consequently the language itself does not have to provide a small number of predefined server types. Note, servers can be used with fixed-priority or EDF scheduling.

A typical server has a budget and a replenishment period. At the start of each period, the available budget is restored to its maximum amount. Unused budget at this time is discarded. To program a server requires timing events to trigger replenishment and a means of grouping tasks together and allocating them an amount of CPU resource. A standard package (a child of `Ada.Execution_Time` – see Program 13.7) is defined to accomplish this. The type `Group_Budget` represents a CPU budget to be used by a group of tasks.

There are a number of routines defined in this package. Consider first those concerned with the grouping of tasks. Each `Group_Budget` has a set of tasks associated with it. Tasks are added to the set by calls of `Add_Task`, and removed using `Remove_Task`. Functions are defined to test if a task is a member of any group budget,

Program 13.7 The `Ada.Execution_Time.Group_Budgets` package.

```
package Ada.Execution_Time.Group_Budgets is
  type Group_Budget is tagged limited private;

  type Group_Budget_Handler is access
       protected procedure(GB : in out Group_Budget);

  type Task_Array is array(Positive range <>) of
                                  Ada.Task_Identification.Task_ID;

  Min_Handler_Ceiling : constant System.Any_Priority :=
    <Implementation Defined>;

  procedure Add_Task(GB: in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);
  procedure Remove_Task(GB: in out Group_Budget;
                     T : Ada.Task_Identification.Task_ID);
  function Is_Member(GB: Group_Budget;
             T : Ada.Task_Identification.Task_ID) return Boolean;
  function Is_A_Group_Member(
             T : Ada.Task_Identification.Task_ID) return Boolean;
  function Members(GB: Group_Budget) return Task_Array;

  procedure Replenish(GB: in out Group_Budget; To : Time_Span);
  procedure Add(GB: in out Group_Budget; Interval : Time_Span);
  function Budget_Has_Expired(GB: Group_Budget) return Boolean;
  function Budget_Remaining(GB: Group_Budget) return Time_Span;

  procedure Set_Handler(GB: in out Group_Budget;
                        Handler : Group_Budget_Handler);
  function Current_Handler(GB: Group_Budget)
                           return Group_Budget_Handler;
  procedure Cancel_Handler(GB: in out Group_Budget;
               Cancelled : out Boolean);

  Group_Budget_Error : exception;
private
    -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

or one specific group budget. A further function returns the collection of tasks associated with a group budget by returning an unconstrained array type of task IDs.

An important property of these facilities is that a task can be a member of at most one group budget. Attempting to add it to a second group will cause `Group_Budget_Error` to be raised. When a task terminates, if it is still a member of a group budget, it is automatically removed.

The budget decreases whenever a task from the associated set executes. The accuracy of this accounting is again implementation defined. To increase the amount of budget available, two routines are provided. The `Replenish` procedure sets the budget

to the amount of 'real-time' given in the `To` parameter. It replaces the current value of the budget. By comparison, the `Add` procedure increases the budget by the `Interval` amount but, as this parameter can be negative it can also be used to, in effect, reduce the budget.

To inquire about the state of the budget, two functions are provided. Note that when `Budget_Has_Expired` returns `True` then `Budget_Remaining` will return `Time_Span_Zero`.

A handler is associated with a group budget by use of the `Set_Handler` procedure. There is an implicit event associated with a `Group_Budget` that occurs whenever the budget goes to zero. If at that time there is a non-null handler set for the budget, the handler will be executed.

As with timers, an implementation must define the minimum ceiling priority level for the protected object linked to any group budget handler. Also note there are `Current_Handler` and `Cancel_Handler` subprograms defined.

By comparison with timers and timing events, which are triggered when a certain clock value is reached (but will then never be reached again for monotonic clocks), the group budget event can occur many times – whenever the budget goes to zero. So the handler is permanently associated with the group budget, and it is executed every time the budget is exhausted (obviously following replenishment and further usage). The handler can be changed by a further call to `Set_Handler` or removed by using a null parameter to this routine (or by calling `Cancel_Handler`), but for normal execution the same handler is called each time. The better analogy for a group budget event is an interrupt; its handler is called each time the interrupt occurs.

When the budget is zero, the associated tasks *continue* to execute. If action should be taken when there is no budget, this has to be programmed (it must be instigated by the handler). So group budgets are not in themselves a server abstraction – but they allow these abstractions to be constructed.

To give a simple example, consider four aperiodic tasks that should share a budget of 2 ms that is replenished every 10 ms. The tasks first register with a `Controller1` protected object that will manage the budget. They then loop around waiting for the next invocation event. In all of the examples in this section, fixed-priority scheduling on a single processor is assumed.

```
task Aperiodic_Task is
  pragma Priority(Some_Value);
end Aperiodic_Task;

task body Aperiodic_Task is
  ...
begin
  Controller1.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The `Controller1` protected object will use a timer event and a group budget, and hence defines handlers for both.

```
protected Controller1 is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  entry Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  For_All : Boolean := False;
end Controller1;

protected body Controller1 is
  entry Register when Register'Count = 4 or For_All is
  begin
    if not For_All then
      For_All := True;
      G_Budget.Add(Milliseconds(2));
      G_Budget.Add_Task(Register'Caller);
      T_Event.Set_Handler(Milliseconds(10),Timer_Handler'Access);
      G_Budget.Set_Handler(Group_Handler'access);
    else
      G_Budget.Add_Task(Register'Caller);
    end if;
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
  begin
    G_Budget.Replenish(Milliseconds(2));
    for ID in T_Array'Range loop
      Asynchronous_Task_Control.Continue(T_Array(ID));
    end loop;
    E.Set_Handler(Milliseconds(10),Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G.Members;
  begin
    for ID in T_Array'Range loop
      Asynchronous_Task_Control.Hold(T_Array(ID));
    end loop;
  end Group_Handler;
end Controller1;
```

The `Register` entry blocks all calls until each of the four 'clients' has called in. The final task to register (which becomes the first task to enter) sets up the group budget and the timing event, adds itself to the group and alters the boolean flag so that the other three tasks will also complete their registration. For these tasks it is straightforward to add themselves to the group budget. Note the tasks in this example may have different priorities.

The two handlers work together to control the tasks. Whenever the group budget handler executes, it stops the tasks from executing by using the `Hold` routine. It always gets a new list of members in case any have terminated. The `Timer_Handler` releases

all the tasks using `Continue`, it replenishes the budget and then sets up another timing event for the next period (10 ms).

In a less stringent application it may be sufficient to just prevent new invocations of each task if the budget is exhausted. The current execution is allowed to complete and hence tasks are not suspended. The following example implements this simpler scheme, and additionally allows tasks to register dynamically (rather than all together at the beginning). The protected object is made more general-purpose by representing it as a type with discriminants for its main parameters (replenishment in terms of milliseconds and budget measured in microseconds):

```
protected type Controller2(Period, Bud : Positive) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Register;
  entry Proceed;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  First : Boolean := True;
  Allowed : Boolean := False;
  Req_Budget : Time_Span := Microseconds(Bud);
  Req_Period : Time_Span := Milliseconds(Period);
end Controller2;

Con : Controller2(10, 2000);
```

The client task would now have the following structure:

```
task body Aperiodic_Task is
  ...
begin
  Con.Register;
  loop
    -- wait for next invocation
    -- undertake the work of the task
  end loop;
end Aperiodic_Task;
```

The body of the controller is as follows:

```
protected body Controller2 is
  entry Proceed when Allowed is
  begin
    null;
  end Proceed;

  procedure Register is
  begin
    if First then
      First := False;
      Add(G_Budget,Req_Budget);
      T_Event.Set_Handler(Req_Period,Timer_Handler'Access);
```

```
      G_Budget.Set_Handler(Group_Handler'Access);
      Allowed := True;
    end if;
    G_Budget.Add_Task(Current_Task);
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
  begin
    Allowed := True;
    G_Budget.Replenish(Req_Budget);
    E.Set_Handler(Req_Period,Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
  begin
    Allowed := False;
  end Group_Handler;
end Controller2;
```

The next example illustrates a Deferable Server. Here, the server has a fixed priority, and when the budget is exhausted, the tasks are moved to a background priority `Priority'First`. This is closer to the first example, but retains some of the properties of the second approach:

```
protected type Controller3(Period, Bud : Positive;
                           Pri : Priority) is
  pragma Interrupt_Priority (Interrupt_Priority'Last);
  procedure Register;
  procedure Timer_Handler(E : in out Timing_Event);
  procedure Group_Handler(G : in out Group_Budget);
private
  T_Event : Timing_Event;
  G_Budget : Group_Budget;
  First : Boolean := True;
  Req_Budget : Time_Span := Microseconds(Bud);
  Req_Period : Time_Span := Milliseconds(Period);
end Controller3;

Con : Controller3(10, 2000, 12);
-- assume this server has priority 12

protected body Controller3 is
  procedure Register is
  begin
    if First then
      First := False;
      G_Budget.Add(Req_Budget);
      T_Event.Set_Handler(Req_Period,Timer_Handler'Access);
      G_Budget.Set_Handler(Group_Handler'Access);
    end if;
    Add_Task(G_Budget,Current_Task);
    if G_Budget.Budget_Has_Expired then
      Set_Priority(Priority'First);
      -- sets client task to background priority
```

```
    else
      Set_Priority(Pri);
      -- sets client task to servers 'priority'
    end if;
  end Register;

  procedure Timer_Handler(E : in out Timing_Event) is
    T_Array : Task_Array := G_Budget.Members;
  begin
    G_Budget.Replenish(Req_Budget);
    for ID in T_Array'Range loop
      Set_Priority(Pri,T_Array(ID));
    end loop;
    E.Set_Handler(Req_Period,Timer_Handler'Access);
  end Timer_Handler;

  procedure Group_Handler(G : in out Group_Budget) is
    T_Array : Task_Array := G_Budget.Members;
  begin
    for ID in T_Array'Range loop
      Set_Priority(Priority'First,T_Array(ID));
    end loop;
  end Group_Handler;
end Controller3;
```

When a task registers, it is running outside the budget so it is necessary to check if the budget is actually exhausted during registration. If it is then the priority of the task must be set to the low value. Other properties of this algorithm should be clear to the reader from the previous discussions.

13.7 Error recovery

Once timing errors have been detected, strategies for recovery must be developed. Inevitably this is application-dependent; however, there are several techniques that can be utilized. This section first considers recovery at the individual thread/task level, and then considers more system-wide responses.

13.7.1 Task-level recovery

The goal of the damage-confinement techniques outlined in the previous section has been to attempt to isolate timing errors to individual tasks or groups of tasks.

Strategies for handling WCET overrun

Monitoring WCET overrun has been suggested as a mechanism for detecting a common fault before the error propagates outside the errant task. Once detected, the task response will depend on whether it is a hard, soft or firm.

- **WCET overrun in hard real-time tasks** – although the error detection techniques introduced in Chapter 2 have detected functional failures that might cause overruns (such as non-terminating loops), WCET overrun can still occur due to inaccuracies in calculating the WCET values. One possibility is that the WCET values used in the schedulability analysis consist of the addition of two components. The first is the time allocated for the primary algorithm and the second is the time for recovery (assuming a fault hypothesis of a single failure per task per release). The first time is the time that is used by the system when monitoring. When this time passes, forward or backward error recovery occurs and the alternative algorithm is executed. This can either be within the same task and the budget increased (for example, changing the cost in a Real-Time Java thread's release parameters), or by releasing a dedicated recovery task. Typically, these alternative algorithms try to provide a degraded service. Another possibility is simply to do nothing. This assumes that there is enough slack in the system for the task (and other lower-priority tasks) to still meet their deadlines.
- **WCET overruns in soft/firm real-time tasks** – typically overruns in soft and firm real-time tasks can be ignored if the isolation techniques guarantee the capacity needed for the hard real-time tasks. Alternatively, the tasks' priorities can be lowered, or the current releases can be terminated and the tasks re-released when their next release event occurs.

As an example of the latter, consider the use of Ada's execution-time timers and timer events to lower the priority of a task if it overruns its worst-case execution time. In this example, the task has a worst-case execution time of 1.25 ms per invocation. If it executes for more than this value its priority should be lowered from its correct value of 14 to a minimum value of 2. If it is still executing after a further 0.25 ms then that invocation of the task must be terminated; this implies the use of an ATC construct. First, the overrun handler protected type is defined:

```
protected Overrun is
  pragma Priority(Min_Handler_Ceiling);
  entry Stop_Task;
  procedure Handler(TM : in out Timer);
  procedure Reset(C1 : CPU_Time);
private
  Abandon : Boolean := False;
  First_Occurrence : Boolean := True;
  WCET_Overrun : CPU_Time;
end Overrun;

protected body Overrun is
  entry Stop_Task when Abandon is
  begin
    null;
  end Stop_Task;

  procedure Reset(C1 : CPU_Time) is
  begin
    Abandon := False;
    First_Occurrence := True;
```

```
      WCET_Overrun := C1;
    end Reset;

    procedure Handler(TM : in out Timer) is
    begin
      if First_Occurrence then
        Set_Handler(TM,WCET_Overrun,Handler'Access);
        Set_Priority(2, TM.T.all);
        First_Occurrence := False;
      else
        Abandon := True;
      end if;
    end Handler;
  end Overrun;
```

It may not be immediately clear why a `Reset` routine is required, but without it a race condition may lead to incorrect execution. Consider the code of the task:

```
task Hard_Example;
task body Hard_Example is
  ID : aliased Task_ID := Current_Task;
  WCET_Error : Timer(ID'access);
  WCET : CPU_Time := Ada_Execution_Time.Time_Of(0,
                     Microseconds(1250));
  WCET_Overrun : CPU_Time := Time_Of(0,Microseconds(250));
  Bool : Boolean := False;
 ...
begin
  -- initialization
  loop
    Overrun.Reset(WCET_Overrun);
    Set_Handler(WCET_Error,WCET,Overrun.Handler'Access);
    select
      Overrun.Stop_Task;
      -- handler the error if possible at priority level 2
    then abort
      -- code of the application
    end select;
    Cancel_Handler(WCET_Error, Bool);
    Set_Priority(14);
    delay until ...
  end loop;
 ...
end Hard_Example;
```

It is possible for the timer to trigger (or *expire*) after completion of the select statement but before it can be cancelled. This would leave the state of the boolean variable, `Abandon`, with the incorrect value of `True`. Similarly, it is necessary to cancel the timer before changing the priority back to 14 – otherwise the event could trigger just before executing the delay statement and the task would be stuck with the wrong low priority for its next invocation.

Strategies for handling sporadic event overruns

There are several responses to the violation of minimum inter-arrival time of a sporadic task. The mechanism provided by Real-Time Java covers most of them: the release event can be ignored, an exception can be thrown, the last event can be overwritten (if it has not already been acted upon) or the actual release of the thread can be delayed until the MIT has passed. Of course, the thread could ignore the violation and be executed anyway.

Strategies for handling deadline misses

Although the early identification of potential timing problems facilitates damage assessment, many real-time systems just focus on the recovery from missed deadlines. Again, several strategies are possible.

- **Deadline miss of hard real-time tasks** – it is possible to set two deadlines for each task. An early deadline is one whose miss will cause the invocation of forward or backward error recovery. A later deadline is the deadline used by the schedulability test. In both cases, the recovery should again aim to produce a degraded service for the task.
- **Deadline miss of soft real-time task** – typically this can be ignored and treated as a transient overload situation. A count of missed deadlines can be maintained, and when it passes a certain threshold a health monitoring system can be informed (see below).
- **Deadline miss of a firm real-time task** – as a firm task produces no value once it has passed its deadline, its current release can be terminated.

As an example of handling a deadline miss, consider a soft real-time Real-Time Java system where applications will want to monitor any deadline misses, but take no action unless a certain threshold is reached. When it is reached, the tardy thread is de-scheduled.

Here, a health monitor object is assumed with the following interface:

```
import javax.realtime.*;
public class HealthMonitor {
  public void persistentDeadlineMiss(Schedulable s);
}
```

Now consider the following event handler for catching a missed deadline of a periodic real-time thread:

```
import javax.realtime.*;
class DeadlineMissHandler extends AsyncEventHandler {
  public DeadlineMissHandler(HealthMonitor mon,
                             int threshold) {
    super(new PriorityParameters(
              PriorityScheduler.MAX_PRIORITY),
              null, null, null, null, null);
    myHealthMonitor = mon;
```

```
    myThreshold = threshold;
  }

  public void setThread(RealtimeThread rt) {
    myrt = rt;
  }

  public void handleAsyncEvent() {
    if(++missDeadlineCount < myThreshold)
       myrt.schedulePeriodic();
    else
       myHealthMonitor.persistentDeadlineMiss(myrt);
  }
  private RealtimeThread myrt;
  private int missDeadlineCount = 0;
  private HealthMonitor myHealthMonitor;
  private final int myThreshold;
}
```

When the handler is executed, it increments the miss count and reschedules the thread. When the count reaches the threshold, it informs the health monitor and does not reschedule the thread. The following code sets up the real-time periodic thread.

```
{
  PriorityScheduler ps = (PriorityScheduler)Scheduler.
                         getDefaultScheduler();
  HealthMonitor healthMonitor = new HealthMonitor();
  DeadlineMissHandler missHandler = new
     DeadlineMissHandler(healthMonitor, 5);
  PriorityParameters pp1 = new
     PriorityParameters(ps.getMinPriority());
  PeriodicParameters release1 = new PeriodicParameters(
     new RelativeTime( 0,0),     // start,
                       new RelativeTime(1000,0),  // period
                       new RelativeTime( 100,0),  // cost
                       new RelativeTime(500, 0),  // deadline
                       null,                      // no overrun handler
                       missHandler);              // miss handler


  RealtimeThread rtt1 = new RealtimeThread(pp1,release1) {
        public void run() {
           // Code for thread.
        }
     }
  missHandler.setThread(rtt1);
  rtt1.start();
}
```

Of course, the deadline miss detection mechanism can be combined with the Real-Time Java ATC mechanism to stop the task if necessary.

13.7.2 Mode changes and event-based reconfiguration

In the above discussions, it has generally been assumed that a missed deadline and other timing errors can be dealt with by the task that is actually responsible for the problem. This is not always the case. Often the consequences of a timing error are as follows.

- Other tasks must alter their deadlines or even terminate what they are doing.
- New tasks may need to be started.
- Critically important computation may require more processor time than is currently available; to obtain the extra time, other less significant tasks may need to be 'suspended'.
- Tasks may need to be 'interrupted' in order to undertake one of the following (typically):
  - immediately return their best results they have obtained so far;
  - change to quicker (but presumably less accurate) algorithms;
  - forget what they are presently doing and become ready to take new instructions: 'restart without reload'.

These actions are sometimes known as **event-based reconfiguration**.

Some systems may additionally enter anticipated situations in which deadlines are liable to be missed. A good illustration of this is found in systems that experience **mode changes**. This is where some event in the environment occurs which results in certain computations that have already started, no longer being required. If the system were to complete these computations then other deadlines would be missed; it is thus necessary to terminate prematurely the tasks or temporal scopes that contain the computations.

To perform event-based reconfiguration and mode changes requires communication between the tasks concerned. Due to the asynchronous nature of this communication, it is necessary to use the asynchronous notification mechanisms found in languages like Ada, Real-Time Java and C/Real-Time POSIX (see Section 7.4). These mechanisms are low level; arguably what is really required to to be able to tell the scheduler to stop invoking certain threads that are now not required and to begin to invoke other tasks. Real-Time Java goes some way towards this by having methods associated with real-time threads that inform the scheduler that the real-time thread is currently not required. The methods are shown in Program 13.1. De-scheduling and rescheduling real-time threads in Real-Time Java does not alter the phasing of the thread. It simply ignores any release event for the thread. Also note, the real-time thread is allowed to complete its current release.

The research language Real-Time Euclid adopts a slight different approach from Real-Time Java because it ties its asynchronous event-handling mechanism to its real-time release mechanisms. In Real-Time Euclid time constraints are associated with processes (a task is called a process in Real-Time Euclid) and numbered exceptions can be defined. Handlers must be provided in each process. For example, consider the following temperature controller process which defines three exceptions.

```
process TempController : periodic frame 60 first activation
                         atTime 600 or atEvent startMonitoring
  % import list
  handler (except_num)
    exceptions (200,201,304)  % for example
    imports (var consul, ...)
    var message : string(80), ...
    case except_num of
      label 200:  % very low temperature
        message := "reactor is shut down"
        consul := message
      label 201:  % very high temperature
        message := "meltdown has begun - evacuate"
        consul := message
        alarm := true   % activate alarm device
      label 304:  % timeout on sensor
                  % reboot sensor device
    end case
  end handler
  %
  % execution part
  %
end TempController
```

Real-Time Euclid allows a process to raise an exception in another process. Three different kinds of raise statement are supported: *except*, *deactivate* and *kill*; as their names imply they have increasing severity.

The *except* statement is essentially the same as the Ada and Java raise/throw statements, the difference being that once the handler has been executed, control is returned to where it left off (that is, the resumption model). By comparison, the *deactivate* statement causes that iteration of the (periodic) process to be terminated. The victim process still executes the exception handler, but will then only become reactivated when its next period is due. Hence Real-time Euclid allows the current release to be abandoned.

To terminate a process, the *kill* statement is available; this explicitly removes a process (possibly itself) from the set of active processes. It differs from an unconditional abort in that the exception handler is executed before termination. This has the advantage that a process may perform some important 'last rites'. It has the disadvantage that an error in the handler could still cause the process to malfunction.

To illustrate the use of these exceptions, the temperature control process given in Section 10.6.1 will have some detail added to its execution part. Note that, in this example, exceptions are raised and handled synchronously within the same process, and asynchronously in another process. First, the process waits on a condition variable; a timeout is specified and an exception number is given. (If this timeout occurs, the numbered exception is raised using *except*.) A temperature is then read and logged. Tests on the temperature value could lead to other exceptions being raised. A low value will result in an appropriate message and deactivation until the next period; a high value will result in an even more appropriate, if somewhat futile, message, an exception being raised in an alarm process, and the temperature controller terminating. All available processor time can now be dedicated to the alarm process.

```
process TempController : periodic frame 60 first activation
                         atTime 600 or atEvent startMonitoring
  % import list
  handler (except_num)
    exceptions (200,201,304)  % for example
    imports (var consul, ...)
    var message : string(80), ...
    case except_num of
      ... % as before
    end case
  end handler

  wait(temperature_available) noLongerThan 10 : 304
  currentTemperature := ...  % low-level i/o
  log := currentTemperature
  if currentTemperature < 100 then
    deactivate TempController : 200
  elseif currentTemperature > 10000 then
    kill TempController : 201
  end if
  %  other computations
end TempController
```

To perform this form of reconfiguration in Ada, two mechanisms are available:

- abort – similar to *kill*
- ATC – similar to *deactivate*.

Ada allows 'last rites' to be programmed using a *controlled* variable. As indicated in Section 7.6.1, the ATC feature is a general one, and hence it can deal with *deactivate* and most other forms of event-based reconfiguration.

Summary

This chapter has discussed the toleration of timing faults from with the framework of dynamic software fault tolerance. Timing faults manifest themselves in the following conditions:

- overrun of deadline;
- overrun of worst-case execution time;
- sporadic events occurring more often than predicted;
- timeouts on communications.

Ada and Real-Time POSIX provide low-level mechanisms that the programmer can use to detect these conditions, whereas Real-Time Java provides support in the context of real-time threads and release parameters.

Execution time and aperiodic servers provide the main mechanisms in support of damage confinement. Ada and Real-Time Java provide this via the

notion of group budgets. Ada, in particular, has a flexible set of mechanisms that allow various server approaches to be implemented. Real-Time POSIX opts for the support of a single policy, that of a sporadic server.

Error recovery strategies depend on an application's context. Many timing errors can be considered transient and can be ignored. Others require a task to stop what it is doing and undertake an alternative action. On occasions, a task in isolation can not deal with the problem, and reconfiguration and mode changes may need to be performed.

In some high-integrity applications, there is little scope for error handling within the program itself. A deadline miss will lead to an attempt to recover at the system level. This could involve a switch to another version of the software running on another processor, or the cold restarting of the current program.

Further reading

Burns, A. and Wellings, A. J. (2007) *Concurrent and Real-Time Programming in Ada*. Cambridge: Cambridge University Press.

Koptez, H. (1997) *Real-time Systems*. New York: Kluwer Academic.

Wellings, A. J. (2004) *Concurrent and Real-Time Programming in Java*. Chichester: Wiley.

Exercises

13.1 Show how Ada can implement block-level deadline violations. How can the equivalent be done in Real-Time Java or Real-Time POSIX?

13.2 Can the deadline overrun of a periodic thread in Real-Time Java be detected?

13.3 What is the role of deadline in the specification of a Real-Time Java event handler?

13.4 To what extent can the violation of maximum arrival frequency of sporadic events be detected in Ada, Real-Time POSIX and Real-Time Java?

13.5 To what extent can event-based reconfiguration be performed in Real-Time Java?

13.6 Outline (with code) how Ada supports sporadic tasks. How can the task protect itself from executing more often than its minimum inter-arrival time?

13.7 To what extent can sporadic servers be implemented in Ada?

13.8 To what extent can `ProcessingGroupParameters` be used by a Real-Time Java scheduler to support sporadic servers?