2 Suspending, Resuming, and Terminating of Actors

Introduction

This chapter is divided into two parts. The first considers language-defined constructs for asynchronous suspending and resuming of actors; that means, the ability of an actor to hold and continue other actors. The second part deals with the termination of actors in the three languages. The Ada section starts with a consideration of task dependence and difficulties that might occur when nesting of tasks is allowed. Throughout this part, we will see, that there are several kinds of termination that may be safe or unsafe. (We call the termination of an actor A safe if there is the possibility for A to perform some cleanup action before A is terminated. Otherwise — if no cleanup action can be performed — we regard the termination to be unsafe.)

For a better illustration, some useful examples together with diagrams showing the execution of the examples are given.

2.1 Suspending and Resuming of Actors

2.1.1 Suspending and Resuming of Ada Tasks

2.1.1.1 The delay Statement

The delay_statement is used to block further execution of a task until a specified expiration time is reached. The delay_statement is always applied to the task executing it; it is not possible to block any other task by using a delay_statement. The expiration time can be specified either as a particular point in time (delay_until_statement) or in seconds from the current time (delay_relative_statement). The syntax of the delay_statement is as follows:

```
delay_statement ::= delay_until_statement | delay_relative_statement
delay_until_statement ::= DELAY UNTIL delay_expression;
delay_relative_statement ::= DELAY delay_expression;
```

The expected type for the delay_expression in a delay_relative_statement is the predefined type `Duration`. The delay_expression in a delay_until_statement is expected to be of a time type — either the type `Time` declared in the language-defined package `Calendar` or some other implementation-defined time type.

The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes running again. That means, resuming is done automatically. If the expiration time has already passed, the task is not blocked.

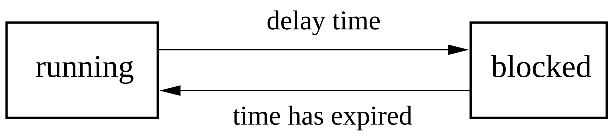

Figure 2.1: Using the delay_statement

2.1.1.2 Asynchronous Task Control

To enable asynchronous suspending and resuming of tasks, Ada provides the language-defined package `Ada.Asynchronous_Task_Control`, the specification of which reads:

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
   procedure Hold (T : in Ada.Task_Identification.Task_ID);
   procedure Continue (T : in Ada.Task_Identification.Task_ID);
   function Is_Held (T : in Ada.Task_Identification.Task_ID)
      return Boolean;
end Ada.Asynchronous_Task_Control;
```

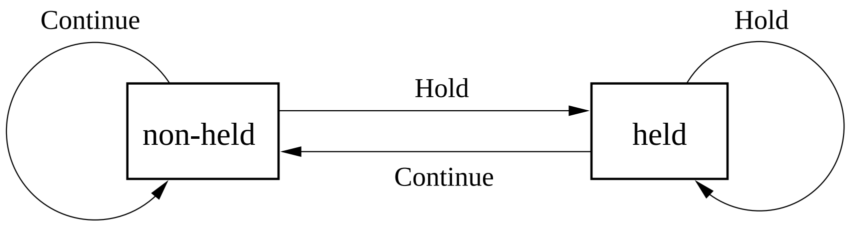

Figure 2.2: Using `Asynchronous_Task_Control` to change the state of an Ada task

After the operation `Hold` has been applied to a task, the state of that task is set to "held". To continue the task, the `Continue` operation resets the state of that task to "non-held". The function `Is_Held` returns true if and only if the task is in the held state. Calling `Hold` on a held task or `Continue` on a non-held task has no effect. The ALRM does not specify whether the procedures `Hold` and `Continue` are atomic to each other.

The procedures `Hold` and `Continue` are defined as follows: For each processor there is a conceptual idle task, which is always ready. The base priority of the idle task is below `System.Any_Priority'First`. If `Hold` is performed on a task T, then T's active priority is set to a value below the base priority of the idle task. As a consequence of the priority rules (see Chapter 6), the held task cannot be dispatched on any processor since its active priority is below the priority of any idle task. The `Continue` operation simply changes the active priority of T back to the former value.

It should be mentioned that a call to `Hold` or `Continue` is not equivalent to a call to `Ada.Dynamic_Priorities.Set_Priority` with the corresponding priority as argument. The difference is that `Ada.Dynamic_Priorities.Set_Priority` changes the base priority and not the active priority of a given task. As mentioned in Chapter 1, only the active priority of a given task T is used when T competes for processing resources.

An Ada implementation need not support `Asynchronous_Task_Control` if it is infeasible to support it in the target environment.

The combination of the `delay_statement` and the procedures of `Ada.Asynchronous_Task_Control` leads to the following state diagram:

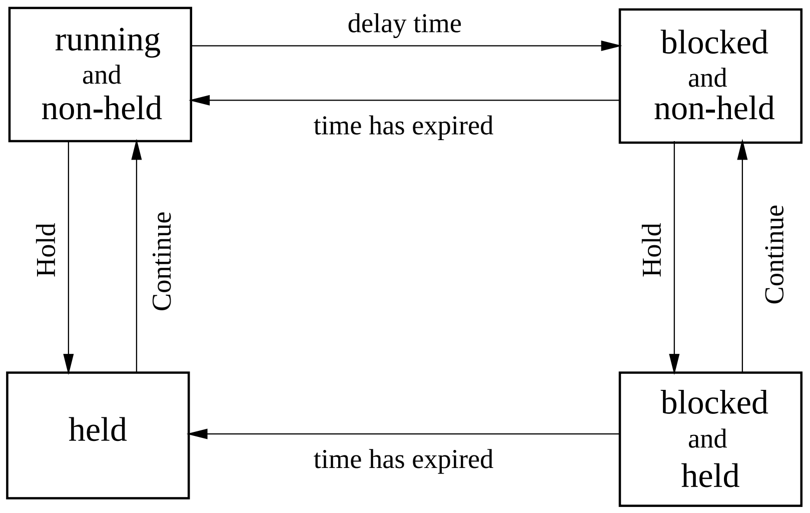

Figure 2.3: Combination of the `delay_statement` and `Asynchronous_Task_Control`

2.1.2 Suspending and Resuming of CHILL Processes and Tasks

CHILL has no possibilities for explicit suspending and resuming of tasks or process instances.

There are, in fact, the operations DELAY, DELAY CASE and CONTINUE. But these are only used together with events for the synchronization of actors. Chapter 5 will discuss that.

2.1.3 Suspending and Resuming of Java Threads

To realize suspending and resuming of threads, Java has the following methods defined in the class `java.lang.Thread`:

- `suspend()`
  A thread may invoke its own `suspend()` method and it may invoke the `suspend()` method of any other thread.
- `resume()`
  A thread may invoke its own `resume()` method and it may invoke the `resume()` method of any other thread.
- `sleep()`
  A thread may only invoke the `sleep()` method of class `java.lang.Thread`; this method is always applied to the thread calling it.
- `join()`
  A thread may invoke its own `join()` method and it may invoke the `join()` method of any other thread.

Figure 2.4 illustrates how these methods are applied to change the state of a thread from runnable to blocked and vice versa.

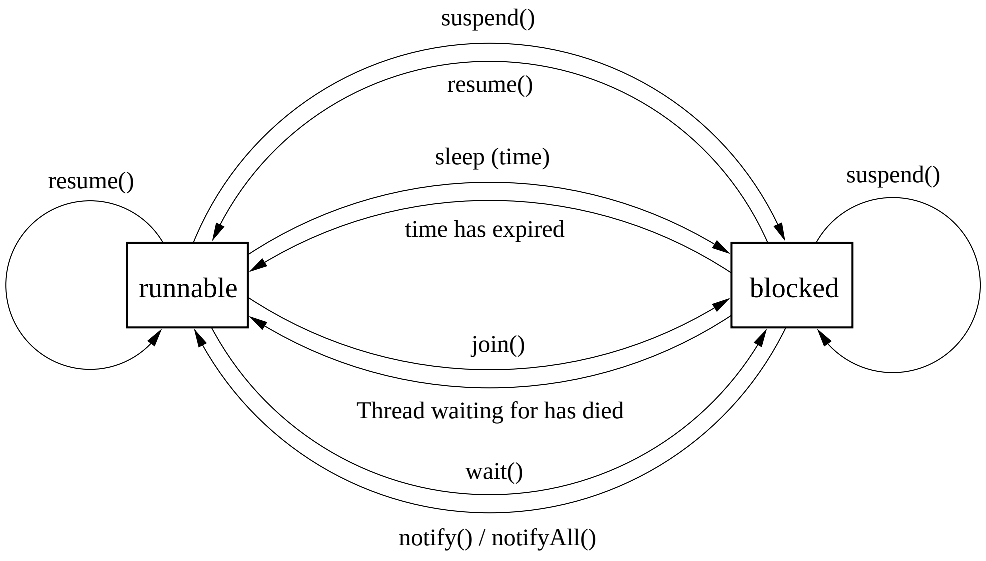

Figure 2.4: Changing the state of a Java thread

Now these methods will be described in more detail.

```
public final void suspend() throws SecurityException
```

If the thread whose `suspend()` was invoked is alive (i.e., it is runnable or blocked), it is suspended and makes no further progress until it is resumed. It is permitted to suspend a thread that is already suspended; it remains suspended. If a thread is suspended several times, only one call to `resume()` is required to resume it.

```
public final void resume() throws SecurityException
```

If the thread whose `resume()` was invoked is alive but suspended, then it is resumed. It is permitted to resume a thread that has never been suspended or has already been resumed. If a thread is resumed several times, only one call to `suspend()` is required to suspend it.

```
public static void sleep(long millis) throws
   InterruptedException
```

This method causes the thread calling `sleep()` to yield and not to be scheduled for further execution until a certain amount of time has elapsed. This amount of real time measured in milliseconds is given by `millis`.
There is another version of `sleep` that measures the amount of time in nanoseconds.

```
public final void join() throws InterruptedException
```

This method causes the thread containing the invocation of `join()` to wait until the thread whose `join()` was invoked is no longer alive. If a thread invokes its own `join()` method, then this thread will be deadlocked.

There are 3 other methods that can be used to suspend and resume threads: `wait()`, `notify()` and `notifyAll()`. Since these methods are used together with synchronization of objects, they will be described later.

We finish this section with an example showing the use of the above methods.

```
class actor extends Thread {
   private String s;
      public actor (String s) {
      this.s = s;
   }
   public void run() {
      for (int i=0; i<10000; i++) {
      System.out.println (s);
      }
   }
}
public class resuming_and_suspending_threads {
   public static void main (String args[]) throws
         InterruptedException {
      actor a1 = new actor("Actor 1");
      actor a2 = new actor("Actor 2");
      a1.start();
      a2.start();
      Thread.sleep(200);
      a1.suspend();
      a2.suspend();
      Thread.sleep(2000)
      a1.resume();
      a2.resume();
      a1.join();
      a2.join();
   }
}
```

Example 2.1: Suspending and resuming of threads

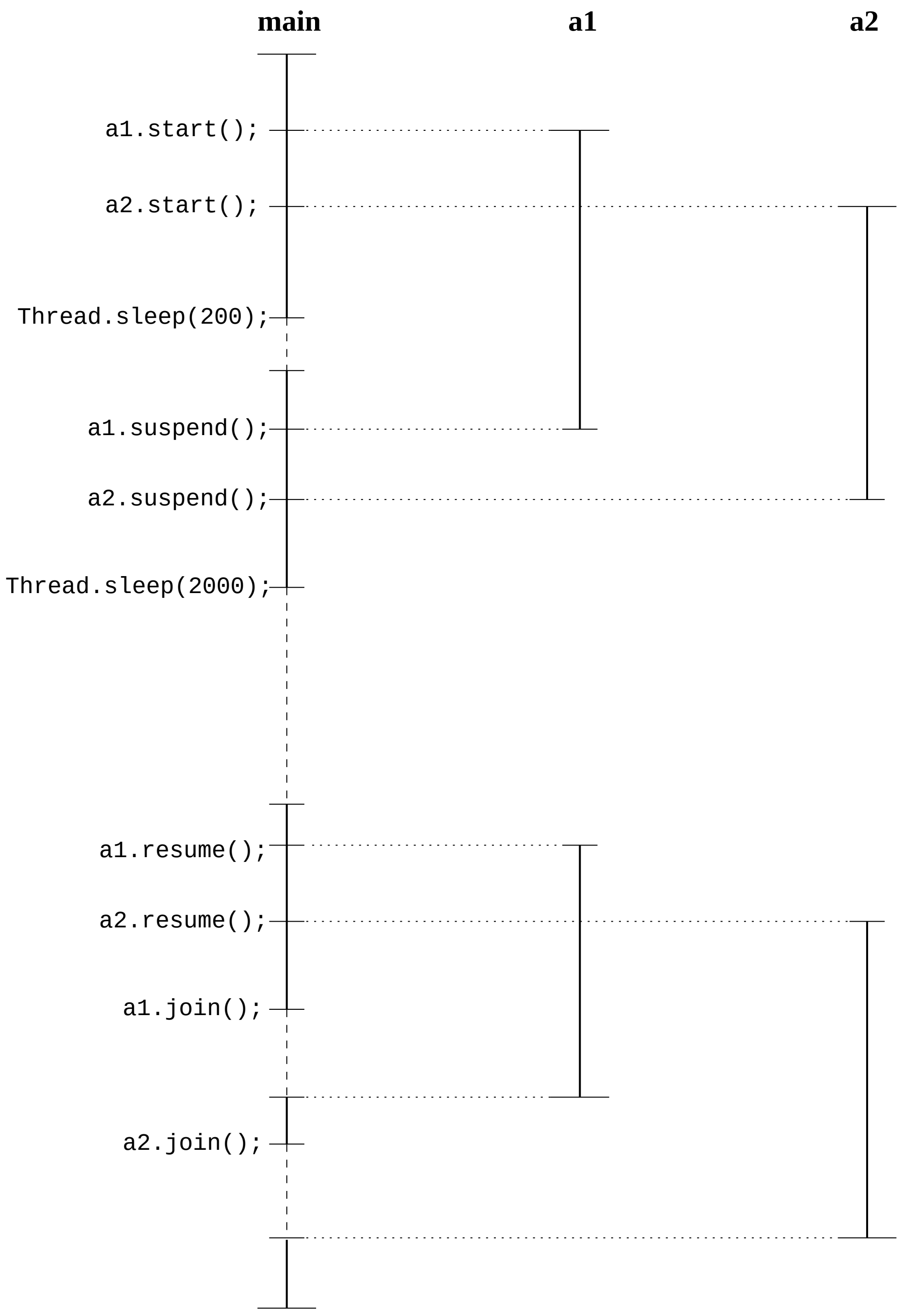

Figure 2.5: A possible execution for Example 2.1

2.2 Termination of Actors

2.2.1 Termination of Ada Tasks

2.2.1.1 Task Dependence and Termination

For better understanding of task termination in Ada, we first have to talk about dependency of Ada tasks.

Since an Ada task is not a compilation unit, it is always nested within other program units. That is why it always depends on one or more masters. A master is the execution of a construct that includes the finalization of local objects after it is complete (and after waiting for any local tasks), but before leaving. There are 2 possibilities how a task can depend on its master:

1. If the task is created by the evaluation of an allocator for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

2. If the task is created by the elaboration of an `object_declaration`, it depends on each master that includes this elaboration.

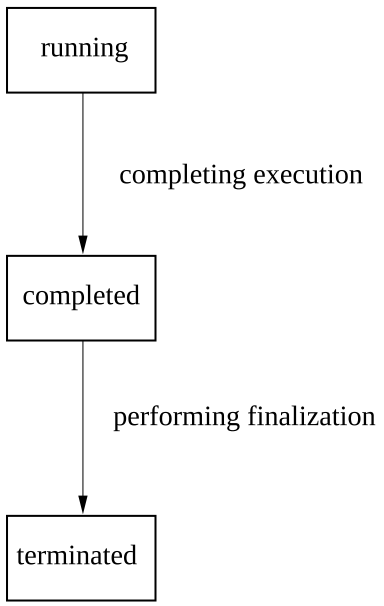

Figure 2.6: Termination of an Ada task

As illustrated in Figure 2.6, termination of an Ada task occurs in 2 stages. The first stage is completion of the task, the second stage is termination. A task is said to be completed when the execution of its corresponding `task_body` is completed; that is, when it reaches its final end. A task is said to be terminated when any finalization of the `task_body` has been performed. The first step of finalizing a master that includes a task body is to wait for the termination of any tasks dependent on the master. The task executing the master is blocked until all the dependents have terminated. Any remaining finalization is then performed and the master continues.

This formal description reflects the important rule that a unit cannot be left until all tasks that depend on it are terminated. For a better understanding why such a rule is useful, consider Example 2.2:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure P is
   count : Integer := 1;
   task type Read_Count;
   task body Read_Count is
   begin
      delay 10.0;
      Put(count);
   end;
   T1, T2 : Read_Count;
begin
   null;
end P;
```

Example 2.2: Termination of local tasks

Example 2.2 shows a procedure `P` that includes the definition of a local variable (`count`) and the definition of two local tasks `T1` and `T2` accessing `count`. If `P` did not wait for `T1` and `T2` to terminate, then the local variable `count` would disappear and the execution of the two tasks would be erroneous. This situation is shown in Figure 2.7.

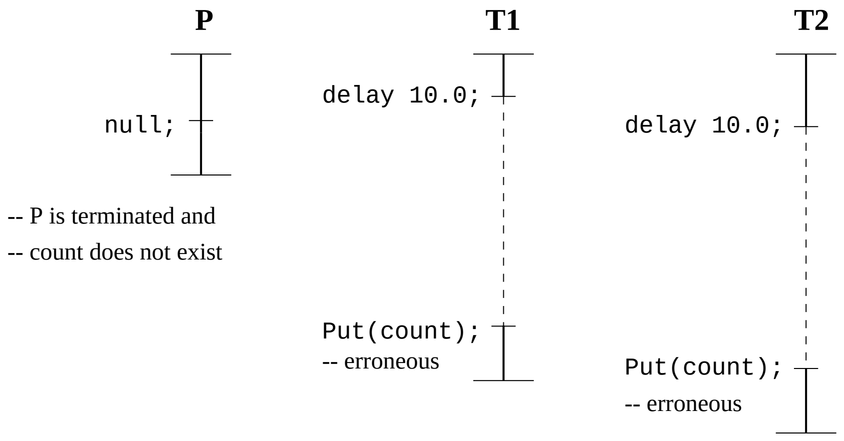

Figure 2.7: An erroneous execution of Example 2.2

The correct execution is given in Figure 2.8. Although `P` is completed, it must not terminate. `P` has to wait until `T1` and `T2` are terminated. After termination of `T1` and `T2`, `P` may terminate, too.

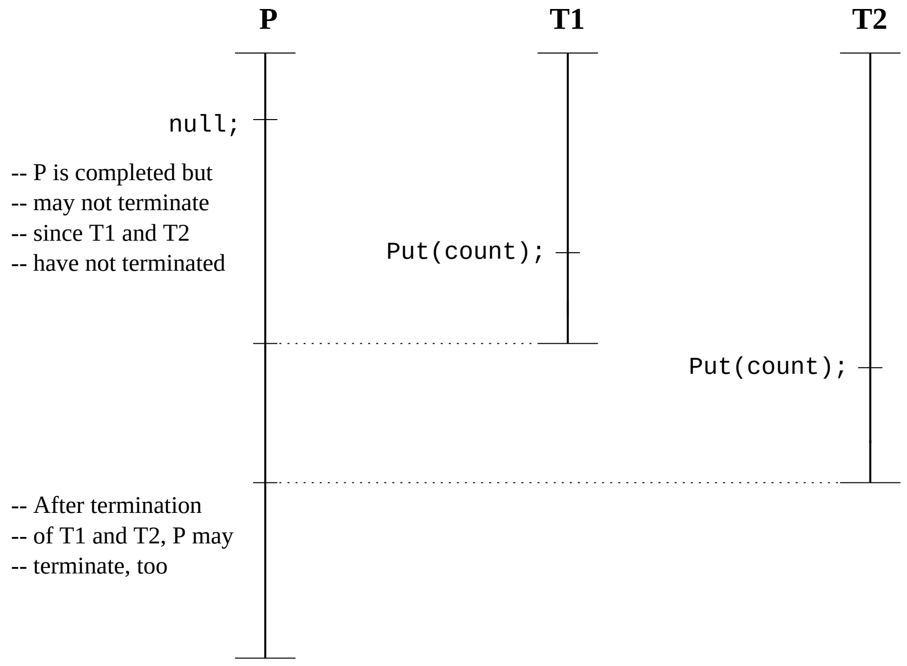

Figure 2.8: A correct execution; P has to wait for its dependent tasks to terminate

After describing how an Ada task terminates, we have to consider how a task may reach its final end. There are the following ways of completion:

1. The Ada task reaches the final end of its body (after normal execution)
2. Control is transferred out of the task due to a requeue_statement
3. Control is transferred out of the task due to selection of a terminate_alternative
4. Control is transferred out of the task due to raising of an exception
5. Control is transferred out of the task due to an abort_statement

In the first 3 cases completion is called normal, otherwise completion is called abnormal.

The requeue_statement will be explained in Chapter 4.

If a task is blocked at a select_statement (see Chapter 4) with an open terminate_alternative, the open terminate_alternative is selected if and only if the following conditions are satisfied:

- The task depends on some completed master M.
- Each task that depends on M is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

When both conditions are satisfied, the task considered becomes completed together with any tasks that depend on M and that are not yet completed.

If an exception is propagated by the elaboration of the declarative_part of a task T, the activation of T is defined to have failed and T becomes completed. In such a case the predefined exception `Tasking_Error` is raised in the unit containing the activation of T. This is due to the fact that exceptions raised in a declarative_part of a body are not handled by the exception_handler of that body.

On the other side, if an exception is raised by the execution of a task_body the execution is not propagated further. That means, if an exception is not handled by a task at all the task is abandoned and the exception is lost. To avoid such “silent death” of tasks, all significant tasks should have a general exception handler at the outermost level.

2.2.1.2 Aborting a Task

The abort_statement unconditionally terminates one or more tasks. The syntax is as follows:

abort_statement ::= ABORT task_name {, task_name}

An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. For the execution of an abort_statement, the given task_names are evaluated in an arbitrary order. Then each task is aborted. That consists of making the task abnormal and aborting the execution of the corresponding task_body unless it is already completed. The order in which tasks become abnormal is not specified by the language. If a task is aborted, all dependent tasks are also aborted except for tasks executing an abort-deferred operation. Abort-deferred operations are operations that cannot be aborted; their execution continues to completion.

The following are abort-deferred operations:

- a protected action (see Chapter 5)
- the execution of an accept_statement (see Chapter 4) after calling the corresponding entry
- waiting for the termination of dependent tasks
- the execution of an `Initialize` procedure of a controlled object
- the execution of a `Finalize` procedure as part of the finalization of a controlled type
- an assignment to an object with a controlled part

A task is allowed to abort any task it can name, including itself. If we abort a task T that is currently blocked and that is outside an abort-deferred operation, then T completes immediately. These immediate effects occur before the execution of the abort_statement completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes.

However, the execution of the aborted construct completes no later than its next abort completion point (if any) that occurs outside of an abort-deferred operation. Abort completion points are:

- the point where the execution initiates the activation of another task
- the end of the activation of a task
- the start or end of the execution of an entry call, accept_statement, delay_statement, or abort_statement
- the start of the execution of a select_statement or of the sequence_of_statements of an exception_handler

The use of the abort statement is illustrated by Example 2.3 and Figure 2.9.

```
with Ada.Text_Io;
use Ada.Text_Io;
procedure Aborting_Tasks is
   task type Delayed_Increasing;
   task body Delayed_Increasing is
      a : Integer := 1;
   begin
      for I in 1 .. 10 loop
         a := a + 1;
         delay 0.5;
      end loop;
   end Delayed_Increasing;

   Task1 : Delayed_Increasing;
   Task2 : Delayed_Increasing;
begin
   Put_Line ("Start");
   delay 2.0;
   Put_Line ("Abort");
   abort Task1, Task2;
end Aborting_Tasks;
```

Example 2.3: Using the abort statement

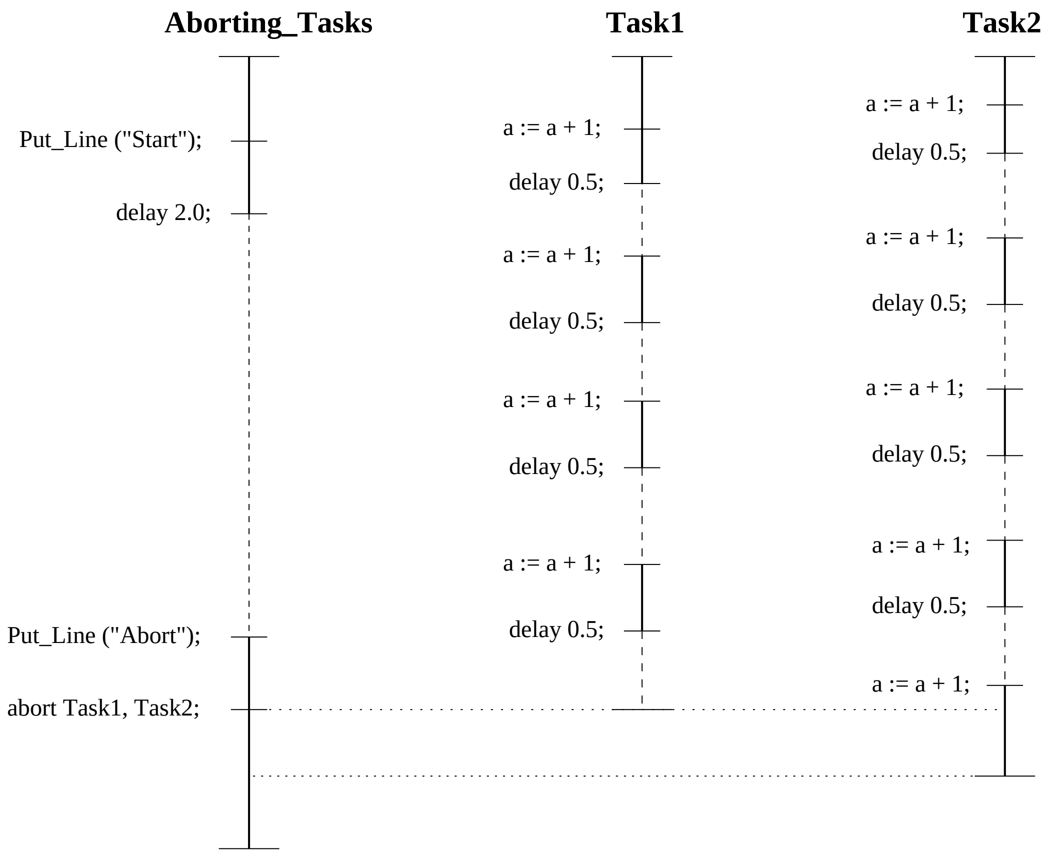

Figure 2.9: A possible execution for Example 2.3

2.2.2 Termination of CHILL Process Instances and Tasks

2.2.2.1 Termination of CHILL Process instances

There are the following possibilities for a CHILL process instance to terminate:

- reaching the end of the body of the process instance
- executing a stop action

The stop action has the following syntax:

```
<stop action> ::=
                    STOP
```

The stop action is always applied to the process instance invoking it; it is not possible to invoke the stop action for another process instance. A stop action terminates the process instance executing it. That means, a process instance can only terminate itself but it is not able to terminate other process instances.

If an exception is caused and the handler specified at the end of the process definition is appropriate for this exception, then the handler is entered. The syntax of a handler is as follows:

```
<handler> ::=
              ON {<on-alternative>}* [ELSE <action statement list>] END
<on-alternative> ::=
                  (<exception list>) : <action statement list>
```

If an exception is mentioned in an exception list in an on-alternative in the handler, the corresponding action statement list is entered; otherwise ELSE is specified and the corresponding action statement list is entered.

When the end of the chosen action statement list is reached, the handler and the construct (e.g. a process) to which the handler is appended are terminated.

Since nesting of processes is not allowed, the earlier mentioned problem of task dependence can only occur at the outermost program level. In such a case — if the imaginary outermost process executes a stop action or reaches the end of its body — the termination will be completed when and only when all other tasks in the program are terminated.

2.2.2.2 Termination of CHILL Tasks

A CHILL task is terminated if its task mode location is destroyed, i.e., if the lifetime of the location ends. The block containing the definition of the task mode location may only be left if the task is not running; otherwise the block must not be left.

If the task mode location was created by a declaration without the attribute STATIC, then its lifetime ends when the block where it was declared is left.

If the task mode location was created by a declaration with the attribute STATIC, then its lifetime ends when the imaginary outermost process terminates.

If the task mode location was created by executing a GETSTACK built-in routine call, its lifetime ends when the directly enclosing block terminates.

If the task mode location was created by an ALLOCATE built-in routine call, its lifetime ends when the location cannot be accessed anymore by any CHILL program. This is always the case if the TERMINATE built-in routine is applied to an allocated reference value that references the location.

2.2.3 Termination of Java Threads

An activated Java thread T can terminate because of one of the following reasons:

1. The initial invocation of the `run()` method of T completes normally through a normal return from the `run()` method.
2. The initial invocation of the `run()` method of T completes abruptly because an exception was thrown.
3. T invokes its own `stop()` method.
4. Some other thread invokes the `stop()` method of T.
5. T invokes its own `destroy()` method.
6. Some other thread invokes the `destroy()` method of T.
7. Some thread invokes the `exit()` method of class `Runtime` or class `System`.

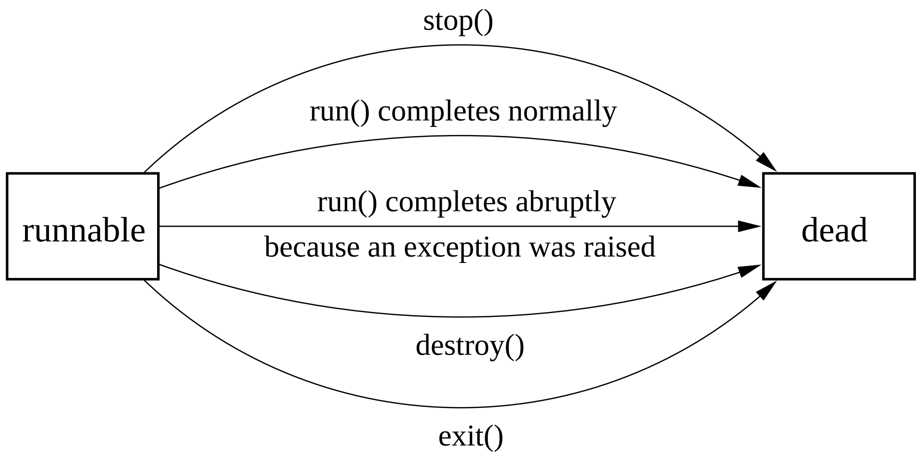

Figure 2.10: Changing the state of a Java thread from runnable to dead

When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing catch clause of the try statement that handles the exception.

If the current thread T contains a catch clause handling the exception, the block of the catch clause is executed. The code that caused the exception is never resumed.

If T contains no catch clause handling the exception, then T is terminated and the exception is propagated further to the dynamically-enclosing block.

The `stop()` method reads as follows:

```
public final void stop() throws SecurityException
```

First, the `checkAccess()` method of the thread whose `stop()` was invoked is called with no arguments. This checks whether the `stop()` operation is permitted.

If the caller is not permitted to execute the `stop()` method, a `SecurityException` is thrown.

If the caller is permitted to execute the `stop()` method, the thread whose `stop()` was invoked is forced to complete abnormally whatever it was doing and to throw a `ThreadDeath` object as an exception. For this purpose, this thread is resumed if it had been suspended, and it is awakened if it had been asleep. If there is no catch clause handling `ThreadDeath` (as in Example 2.4), all `finally` clauses are executed. (In Example 2.4, the message "`Something went wrong !`" is printed in those executions where `a1` is still runnable while `a1.stop()` is executed.). After this the thread is terminated.

This kind of termination has 2 advantages:

1. The execution of the finally clauses allows some cleanup action to be performed.
2. All synchronization locks held by the thread object are released.

Normally, user code should not try to catch `ThreadDeath` unless some extraordinary cleanup operation is necessary. If a catch clause catches a `ThreadDeath` object, then it is important to rethrow the object so that the thread will actually die.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it will immediately terminate.

The destroy method is as follows:

```
public final void destroy() throws SecurityException
```

First, the `checkAccess()` method of the thread whose `destroy()` was invoked object is called. This may result in a `SecurityException` being thrown in the thread invoking `destroy()`. Then the thread whose `destroy()` was invoked is destroyed without any cleanup. Any monitors this thread has locked remain locked.

This method is not implemented in early versions of Java, through 1.1. (That is why we did not use this method in Example 2.4.)

Finally, we have the `exit()` method of class Runtime:

```
public void exit(int status) throws SecurityException
```

First, if there is a security manager, its `checkExit()` method is called with the status value as its argument. (A running Java program may have a security manager, which is an instance of class `SecurityManager`. The security manager contains a large number of methods that check whether certain sensitive operations are permitted.) The `checkExit()` method checks whether the thread is permitted to execute the `exit()` operation.

The `exit()` method terminates the currently running Java Virtual Machine M. This stops every thread being run by M. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination. This method never returns normally.

The problem of task dependence (like in Ada) does not arise in Java. If the scope of a thread variable is left, then only the reference to the thread object ceases to exist but the thread object may still be running. The only restriction is that the JVM may not terminate until all threads are terminated.

We conclude this section with an example showing the use the methods described above.

```
class actor extends Thread {
   private String s;
   public actor (String s) {
      this.s = s;
   }
   public void run() {
      try {
         for (int i=0; i<10000; i++)
            System.out.println (s);
      finally {
         System.out.println (“Something went wrong !”);
      }
   }
}

public class exiting_threads {
   public static void main (String args[])
     throws InterruptedException {
      actor a1 = new actor("Actor 1");
      actor a2 = new actor("Actor 2");
      actor a3 = new actor("Actor 3");
      a1.start();
      a2.start();
      a3.start();
      a1.stop();
      Thread.sleep (20);
      a2.join();
      a3.join();
      System.exit(-1);
      //This code will never be executed
      Thread.sleep (1000000000);
   }
}
```

Example 2.4: Termination of Java threads

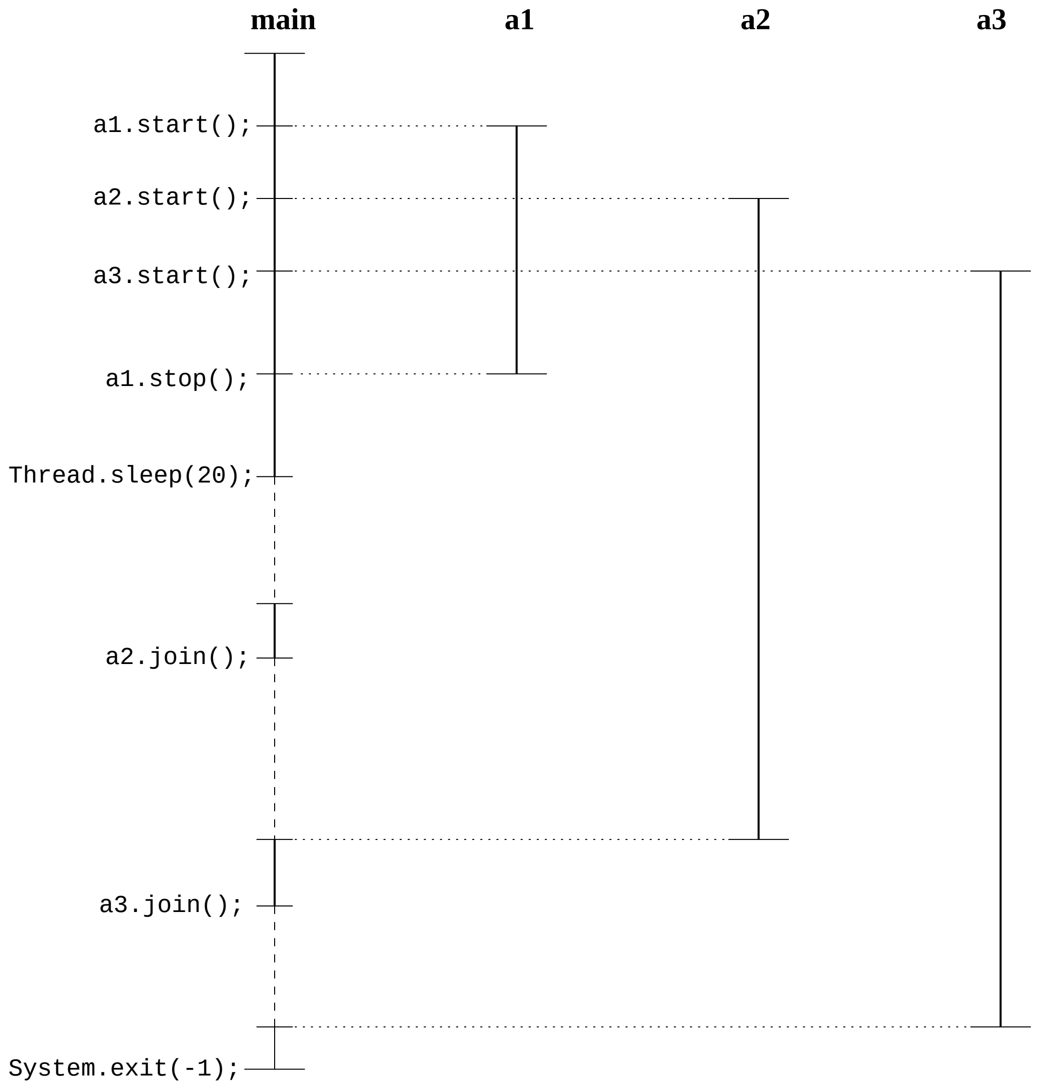

Figure 2.11: A possible execution of Example 2.4

Summary and Comparison

Our intention was to examine the constructs for suspending, resuming, and termination of actors.

Suspending and Resuming

When considering suspending and resuming of actors, we conclude that Java provides us with the most features.

A Java thread may suspend and resume any other threads that are visible to it and it can delay itself until a certain amount of time has elapsed by invoking the `sleep()` method of class `Thread`. Finally, a Java thread may wait for another thread to terminate by calling the `join()` method of that thread.

Ada has the package `Ada.Asynchronous_Task_Control` containing the procedures `Hold` and `Continue`. But there is the limitation that its support depends on the target environment.

CHILL has no features for suspending and resuming of process instances or tasks.

Termination

In each language, an actor can terminate by reaching its final end after normal execution or by causing an exception.

Now we consider the features for explicit termination of actors.

Again, Java contains the richest set of capabilities. Java allows threads to

- safely terminate other threads by invoking the `stop()` method that allows performing of cleanup and releasing of synchronization locks
- abruptly destroy other threads without any cleanup and without releasing synchronization locks by invoking the `destroy()` method
- exit the whole program by invoking the `exit()` method, which terminates the currently running Java Virtual Machine

Ada provides the ABORT statement that allows a task to terminate all other tasks it can name by making them abnormal. This kind of termination can be considered as safe termination and is similar to the `stop()` method in Java.

CHILL has no possibilities that allow an actor to terminate other actors. A CHILL process instance may only terminate itself by calling the STOP method.