Unified Engineering
Spring 2005
Problem Set #14
Solutions
Underlined sentences are important facts you should know about Ada. Other programming languages have different specifications. When compiling tasking programs it might be necessary to dictate the explicit order in which a task starts and executes.

**Problem C21. (Hello World)**

*a. Run the program several times. Is there a specific order in which the tasks start?*

Trick Question. Yes and No. The **Ada specification does NOT say in what order compilers should start tasks.** However, the compiler that comes with AdaGide starts the tasks in the order in which they are defined. **Starting** a task in a given order does NOT guarantee that they will continue **executing** in the same order.

These lines:

```
Task_A : Hello ('A', 10);
Task_B : Hello ('B', 10);
Task_C : Hello ('C', 10);
```

Dictate the starting order of the task. Once started, they are scheduled by the computer as needed.

Therefore, the output for most Ada compilers will start:

```
Hello from Task A
Hello from Task B
Hello from Task C
```

--...but after a while the execution order may change:

```
Hello from Task B
Hello from Task A
Hello from Task C
```
b. A delay statement is supposed to force a task to give up processor time for another task to execute. What happens when you delete the 'delay' statement?

The Ada specification allows but does not require time-slicing. Time-slicing says that one task will be allowed X amount of time to run before getting ‘paused’ and allowing another task to get started or continue execution for the same X amount of time. Without time-slicing, one task would be allowed to run until completion before another starts. The length of the time-slice can also vary. AdaGide does support time-slicing.

Time-slicing in AdaGide typically results in (answers may vary):

Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task B Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C

--newline

Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task B Hello from Task C
Hello from Task C
Hello from Task C

--newline

Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B

Note that sometimes the tasks are interrupted in the middle of execution. In this case, some of the executions were interrupted before the NewLine command could execute.
Interestingly, you can see the variation in time-slicing if you setup AdaGide to output to a file:

```
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
Hello from Task C
```

The keyword `delay` allows the task to give away a portion of its time share for another task to execute.
c. We would like the tasks to be started in a specific order. This is possible by using the keyword “accept.” Make the tasks start such that the first three outputs are:

“Hello from Task A”
“Hello from Task B”
“Hello from Task C”

... The line “accept Some_Label” tells the task to stop executing until it is called with the command “Task_Name.Some_Label.”

```ada
with Ada.Text_IO;
use Ada.Text_IO;

procedure Tasking_Hello is
  -- (a) ADDED is
  task type Hello (Message: Character; Num: Positive) is
    entry Startme;                    -- (a) ADDED entry Startme;
  end Hello;                           -- (a) ADDED end Hello;

  task body Hello is
    begin
      accept Startme;                   -- (a) ADDED accept Startme;
      for I in 1..Num loop
        Put("Hello from Task " & Message);
        New_Line;
        delay 0.1;
      end loop;
    end Hello;

    Task_A : Hello ('A', 10);
    Task_B : Hello ('B', 10);
    Task_C : Hello ('C', 10);

    begin
      Task_A.Startme;                     -- (a) ADDED accept Startme;
      Task_B.Startme;                     -- (a) ADDED accept Startme;
      Task_C.Startme;                     -- (a) ADDED accept Startme;
    end Tasking_Hello;
```
d. Now we want to make sure that the tasks all execute A followed by B followed by C, continuously in a loop (This can be done with semaphores – but there is a simpler implementation. If you would like to play with semaphores, take a look at Lecture 21).

There are a LOT of ways to do this. Here are 2 (Only the modifications significant to tasking are commented):

SOLUTION 1. A couple of new tasks were introduced. Each task starts the next task by triggering the accept in each task.

Notice:
1. In order to use the keyword accept in the task body, you need the keyword entry in the task specification.
2. The task specification has the parameters (Num:Positive) but the header for the task body does not.

```ada
with Ada.Text_Io;
use Ada.Text_Io;

procedure Tasking_Hello is
  task type Helloa (Num: Positive) is
    entry Startme;
  end Helloa;

  task type Hellob (Num: Positive) is
    entry Startme;
  end Hellob;

  task type Helloc (Num: Positive) is
    entry Startme;
  end Helloc;

  Task_A : Helloa (10);                -- (b) MOVED EARLIER
  Task_B : Hellob (10);
  Task_C : Helloc (10);

  task body Helloa is
    begin
      for I in 1..Num loop
        accept Startme;                -- (b) ADDED accept Startme;
        Put("Hello from Task A");    
        New_Line;
        Task_B.Startme;                -- (b) ADDED Task_B.Startme;
      end loop;
    end Helloa;

  task body Hellob is
    begin
      for I in 1..Num loop
        accept Startme;                -- (b) ADDED accept Startme;
        Put("Hello from Task B");
        New_Line;
        Task_C.Startme;                -- (b) ADDED Task_C.Startme;
      end loop;
    end Hellob;

  task body Helloc is
    begin
```
for I in 1..Num loop
    accept Startme;                -- (b) ADDED accept Startme;
    Put("Hello from Task C" );
    New_Line;
    Task_A.Startme;                -- (b) ADDED Task_A.Startme;
end loop;
end Helloc;

begin
    Task_A.Startme;                -- (b) ADDED accept Startme;
end Tasking_Hello;

SOLUTION 2. A loop is used to continuously in the main procedure to trigger the running of the next task. A protected type is used to make sure that the calls to Put and New_Line cannot be interrupted by another task.

Notice the way in which the protected type is formatted with a specification and a procedure specification inside.

with Ada.Text_IO;
use Ada.Text_IO;

procedure Tasking_Hello is
    -- (b) ADDED a protected type
    protected type Display is
        procedure Text(Message: Character);      -- (b) ADDED procedure specification
    end Display;

    protected body Display is
        procedure Text(Message: Character) is
            begin
                Put("Hello from Task " & Message);  -- (b) MODIFIED call to protected proc.
                New_Line;
            end Text;
        end Display;

    Text_Display: Display;           -- (b) Define instance of protected type

    task type Hello (Message: Character) is
        entry Startme;                   -- (b) CHANGED SPECIFICATION
    end Hello;

    task body Hello is
        begin
            loop
                accept Startme;             -- (b) ADDED accept Startme; (in loop)
                Text_Display.Text(Message); -- (b) MODIFIED call to protected proc.
            end loop;
        end Hello;

    Task_A : Hello ('A');
    Task_B : Hello ('B');
    Task_C : Hello ('C');

    begin
        for I in 1..10 loop                 -- (b) ADDED for I in 1..10 loop
            Task_A.Startme;                  -- (b) ADDED accept Startme;
            Task_B.Startme;                  -- (b) ADDED accept Startme;
            Task_C.Startme;                  -- (b) ADDED accept Startme;
        end loop;
    end Tasking_Hello;
Problem C22. The Lego Printer

Tasking is VERY important in real-time applications, but can be difficult to understand because it is like trying to coordinate multiple people, dependent on each other, yet all doing their own things. In the Mars Rover assignment, tasking would have been very useful to allow one task to measure the spectral quality of the floor while another takes care of navigation. Check out the Lego Printer in the Unified Lounge (starting Wednesday afternoon through finals week). It uses several tasks to accomplish some very basic printing tasks.

Follow the instructions next to the printer to watch it print some basic repeating output. Below is a simplified excerpt of code that is running on the printer.

a. Task Spool_Motor currently starts even before the sensors are initialized. This is a problem because the rotation sensor will not be accurately keeping track of where the paper is. We would like Spool_Motor to start after the ‘printer does some initialization of sensors.’ Implement this delayed start by modifying the task. (Hint: do not use ‘accept,’ use the keyword ‘new’ in the main procedure)

```with Lego;
use Lego;

procedure Main is
  task Stop_Over_Spool; -- Specification
  task body Stop_Over_Spool is -- Body
  begin
    while True loop
      if Spool_Rot_Sensor<59
        Stopalltasks();
      end if;
    end loop;
  end Stop_Over_Spool;

  Spool_Motor_Speed : Integer := 0;

  task type Spool_Motor; -- Specification
  type Spool_Motor_Ref is access Spool_Motor;
  S_Ref : Spool_Motor_Ref; -- Body
  task body Spool_Motor is
  begin
    while True loop
      if Spool_Motor_Speed == 0 then
        Float(S_Motor);
      else
        if Spool_Motor_Speed > 0 then
          OnRev(S_Motor);
        else
          OnFwd(S_Motor);
        end if;
        Wait(abs(Spool_Motor_Speed));
        Float(Out_A);
        Wait(32 - abs(Spool_Motor_Speed));
      end if;
    end loop;
  end Spool_Motor;
```
end Spool_Motor;

-- More code to take care of motor speed control follows. . .

begin

-- Printer does some initialization of sensors

-- We start writing!

S_Ref := new Writer;
Spool_Motor_Speed := 20;
Wait(20);
Spool_Motor_Speed := 30;
Wait(20);

-- more code is used to vary the speed of the spool motor and the
-- motor that controls the print head.

end Main;

b. When does the Stop_Over_Spool task start?

It starts immediately after the keyword begin of the Main procedure.

c. Task Spool_Motor implements a Floyd-Steinberg control of the motor, faking a PWM (a method for controlling motor RPM by turning it on and off quickly). Modify the body of this task so it will spin the motor in the opposite direction when variable Spool_Motor_Speed is negative.

Some useful subprograms to use are:

Integer=abs(Integer)
OnFwd(Motor_Name)

-- Turns on the motor that spools the paper through the printer
task body Spool_Motor is
begin

while True loop

if Spool_Motor_Speed == 0 then
  Float(S_Motor);
else

if Spool_Motor_Speed > 0 then
  OnRev(S_Motor);
else
  OnFwd(S_Motor);
end if;

Wait(abs(Spool_Motor_Speed));
Float(Out_A);
Wait(32 - abs(Spool_Motor_Speed));
end if;
end loop;
end Spool_Motor;
Solution:

1. The signal is plotted below:

![Signal Plot](image)

The signal is very smooth, almost like a Gaussian. Therefore, I expect that the duration bandwidth product will be close to the theoretical lower bound.

2. \[
\left( \frac{\Delta t}{2} \right)^2 = \frac{\int t^2 g^2(t) \, dt}{\int g^2(t) \, dt}
\]

The two integrals are easily evaluated for the given \( g(t) \). The result is

\[
\int t^2 g^2(t) \, dt = \frac{7}{2}
\]

\[
\int g^2(t) \, dt = \frac{5}{2}
\]

Therefore,

\[
\Delta t = 2 \sqrt{\frac{7}{5}}
\]

3. The time domain formula for the bandwidth is

\[
\left( \frac{\Delta \omega}{2} \right)^2 = \frac{\int g^2(t) \, dt}{\int g^2(t) \, dt}
\]

The numerator integral is

\[
\int g^2(t) \, dt = \frac{1}{2}
\]
Solution:

I used Mathematica to find some of the integrals, although you could use tables or integrate by parts.

(a) 

\[ \bar{t} = \int t^2 g^2(t) \, dt = \int_0^\infty t^2 e^{-2t/\tau_{au}} \, dt = \frac{315}{16} \tau^8 \]

\[ \bar{\tau} = \int \tau^2 g^2(t) \, dt = \int_0^\infty \tau^6 e^{-2\tau/\tau_{au}} \, dt = \frac{45}{8} \tau^7 \]

Therefore, 

\[ \bar{\tau} = \frac{7}{2} \tau \]

(b) 

\[ \int (t - \bar{t})^2 g^2(t) \, dt = \frac{315}{32} \tau^9 \]

Therefore, 

\[ \Delta t = \sqrt{7} \tau \]

(c) 

\[ \int \dot{g}^2(t) \, dt = \frac{9}{8} \tau^5 \]

Therefore, 

\[ \Delta \omega = \frac{2}{\sqrt{5} \tau} \]

(d) The duration-bandwidth product is 

\[ \Delta t \Delta \omega = 2 \sqrt{\frac{7}{5}} \approx 2.366 \]

which compares favorably with the theoretical lower bound 

\[ \Delta t \Delta \omega \geq 2 \]
Therefore,

\[ \Delta \omega = \frac{2}{\sqrt{5}} \]

4. The duration-bandwidth product is

\[ \Delta t \Delta \omega = \frac{4\sqrt{7}}{5} \approx 2.1166 \]

which is very close to the theoretical lower limit of 2. This is not surprising, since the shape of \( g(t) \) is close to a gaussian.