**Operating Systems Lab 4**

Rutger Yager

4/29/2015

COSC 4321.01

Grade Option B

**C Option Output**

Below is an example output for the C option using a task to periodically print the time with a wait time of 35 seconds entered.

Enter Time to wait: 35  
Time: 4561.197 ( 35.00 seconds left)  
Time: 4571.198 ( 25.00 seconds left)  
Time: 4581.198 ( 15.00 seconds left)  
Time: 4591.199 ( 5.00 seconds left)  
Time: 4595.199 ( 0.00 seconds left)  
Completed!  
Actual Elapsed Time: 35.001709868  
Accumulated Error: 1.710 milliseconds

To contrast, below is the output of the same program, but with the time drift correction code taken out.

Enter Time to wait: 35  
Time: 5577.501 ( 35.00 seconds left)  
Time: 5587.502 ( 25.00 seconds left)  
Time: 5597.502 ( 15.00 seconds left)  
Time: 5607.505 ( 5.00 seconds left)  
Time: 5612.506 ( 0.00 seconds left)  
Completed!  
Actual Elapsed Time: 35.004304794  
Accumulated Error: 4.305 milliseconds

It is worth noting that the code with the error correction does not completely get rid of discrepancies in the actual elapsed time; there will still be some delay caused by external factors on the system. For example, the process scheduler could take slightly longer than normal to put the program back into execution when the process awakes from sleep. To get around this, a higher priority in the scheduler should be requested, or a Real-Time system that allows for the serial completion of executing processes should be used. One should also consider variances in the actual clock cycle, and the system-set time between clock interrupts. The standard frequency for Windows machines is usually 64 clock interrupts a second, which means the clock register is only updated every ~15.6 milliseconds. This means that, even accounting for drift in the program, the process will not wake up at exactly 10 seconds – it will wake up at ~10.0152 seconds. In order to get around this, one would have to manually override the clock interrupt frequency. Another things to consider is that the assembly code generated by the compiler may generate extra code around the sleep statement that could cause slight errors to accumulate in the actual amount of drift recorded. To avoid this, one should consider writing time sensitive code such as this in assembly to avoid any compiler clutter that may ruin.

That being said, it is still obvious that the simple inclusion of time drift correcting code in the delay statements causes a significantly less amount of error to accumulate. In the above example, although the actual error percentages are very small for both programs, the percent difference between the accumulated errors is over 86%, with the program not accounting for time drift accumulating over 2.5 times more error. Overtime, such a difference in accumulated error could lead to the incorrect program and correct program being hours apart from one another.

**C Option Code**

**Coption.adb**

This is the code for the correct version of the C option; this version accounts for time drift.

with Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar;

use Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar;

procedure Coption is

package DurationIO is new Ada.Text\_IO.Fixed\_IO(Duration); use DurationIO;

task type TimePrinter is

entry GiveCountdownTime(Time : in Duration);

entry Start;

end TimePrinter;

task body TimePrinter is

StartTime : Duration;

StopTime : Duration;

TotalTime : Duration;

TimeLeft : Duration;

Correction : Duration;

Done : Boolean := False;

begin

accept GiveCountdownTime(Time : in Duration) do

TotalTime := Time;

TimeLeft := Time;

end GiveCountdownTime;

accept Start do

Correction := Seconds(Clock);

Put("Time: ");

StartTime := Correction;

Put(Correction, 5, 3);

Put(" (");

Put(TimeLeft, 3, 2);

Put(" seconds left)");

New\_Line;

end Start;

while not Done loop

if TimeLeft = 0.0 then

Correction := Seconds(Clock);

done := True;

elsif TimeLeft < 10.0 then

delay (TimeLeft - (Seconds(clock) - Correction));

Correction := Seconds(Clock);

done := True;

else

delay (10.0 - (Seconds(clock) - Correction));

Correction := Seconds(Clock);

end if;

if Done then

StopTime := Correction;

Put("Time: ");

Put(Correction, 5, 3);

Put\_Line(" ( 0.00 seconds left)");

Put\_Line("Completed!");

else

TimeLeft := TimeLeft - 10.0;

Put("Time: ");

Put(Correction, 5, 3);

Put(" (");

Put(TimeLeft, 3, 2);

Put(" seconds left)");

New\_Line;

end if;

end loop;

Put("Actual Elapsed Time: ");

Put(StopTime - StartTime, 3);

New\_Line;

Put("Accumulated Error: ");

Put(((StopTime - StartTime) - TotalTime) \* 1000, 2, 3);

Put(" milliseconds");

end TimePrinter;

InputTime: Duration;

Timer : TimePrinter;

begin

Put("Enter Time to wait: ");

Get(InputTime);

Timer.GiveCountdownTime(InputTime);

Timer.Start;

end Coption;

**CoptionIncorrect.adb**

This is the code for the incorrect version of the C option; this version does NOT account for time drift. It was used for comparative purposes in the C option explanation.

with Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar;

use Ada.Text\_IO, Ada.Integer\_Text\_IO, Ada.Calendar;

procedure CoptionIncorrect is

package DurationIO is new Ada.Text\_IO.Fixed\_IO(Duration); use DurationIO;

task type TimePrinter is

entry GiveCountdownTime(Time : in Duration);

entry Start;

end TimePrinter;

task body TimePrinter is

StartTime : Duration;

StopTime : Duration;

TotalTime : Duration;

TimeLeft : Duration;

Correction : Duration;

Done : Boolean := False;

begin

accept GiveCountdownTime(Time : in Duration) do

TotalTime := Time;

TimeLeft := Time;

end GiveCountdownTime;

accept Start do

Correction := Seconds(Clock);

Put("Time: ");

StartTime := Correction;

Put(Correction, 5, 3);

Put(" (");

Put(TimeLeft, 3, 2);

Put(" seconds left)");

New\_Line;

end Start;

while not Done loop

if TimeLeft = 0.0 then

Correction := Seconds(Clock);

done := True;

elsif TimeLeft < 10.0 then

delay TimeLeft; --No correction for drift

Correction := Seconds(Clock);

done := True;

else

delay 10.0; --No correction for drift

Correction := Seconds(Clock);

end if;

if Done then

StopTime := Correction;

Put("Time: ");

Put(Correction, 5, 3);

Put\_Line(" ( 0.00 seconds left)");

Put\_Line("Completed!");

else

TimeLeft := TimeLeft - 10.0;

Put("Time: ");

Put(Correction, 5, 3);

Put(" (");

Put(TimeLeft, 3, 2);

Put(" seconds left)");

New\_Line;

end if;

end loop;

Put("Actual Elapsed Time: ");

Put(StopTime - StartTime, 3);

New\_Line;

Put("Accumulated Error: ");

Put(((StopTime - StartTime) - TotalTime) \* 1000, 2, 3);

Put(" milliseconds");

end TimePrinter;

InputTime: Duration;

Timer : TimePrinter;

begin

Put("Enter Time to wait: ");

Get(InputTime);

Timer.GiveCountdownTime(InputTime);

Timer.Start;

end CoptionIncorrect;

**B Option Output**

I chose to do Option 4 for the B option, which is to create a program that handles at least one hardware interrupt and one user defined interrupt. In this example, the hardware interrupt handled is the SIGINT interrupt, which is a UNIX signal generated when Ctrl-C is pressed on the keyboard. To implement a user defined interrupt, I used Ada’s exception routine functionality and caused the program to generate a user defined exception after a certain number of SIGINT interrupts had been generated, terminating the program. Below is an example output from the program running:

Task running...  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Handled a SIGINT Interrupt. (CTRL-C)  
Handled a SIGINT Interrupt. (CTRL-C)  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Handled a SIGINT Interrupt. (CTRL-C)  
Task running...  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Task running...  
Handled a SIGINT Interrupt. (CTRL-C)  
Handled a SIGINT Interrupt. (CTRL-C)  
User Interrupt generated. Terminating Program...

The lines signaling the SIGINT interrupt handling are manually generated by pressing Ctrl-C. After 10 presses, a user exception is generated which ends the program. To keep the program from immediately executing to completion, a task is placed in the background that periodically prints “Task running…” and checks to see if 10 or more SIGINT interrupts have been handled.

**B Option Code**

The following package creates a protected procedure, defines the procedure as an interrupt handler, and then links the procedure to an interrupt, in this case SIGINT.

**Sigint\_Handler.ads**

with Ada.Interrupts, Ada.Interrupts.Names;

use Ada.Interrupts, Ada.Interrupts.Names;

package Sigint\_Handler is

protected Handler is

procedure Handle;

function Check(Num : in Integer) return Boolean;

pragma Interrupt\_Handler(Handle);

pragma Attach\_Handler(Handle, Sigint);

private

Interrupt\_Count : Integer := 0;

end Handler;

end Sigint\_Handler;

**Sigint\_Handler.adb**

with Ada.Text\_IO;

use Ada.Text\_IO;

package body Sigint\_Handler is

protected body Handler is

procedure Handle is

begin

Put\_Line("Handled a SIGINT Interrupt. (CTRL-C)");

Interrupt\_Count := Interrupt\_Count + 1;

end Handle;

function Check(Num : in Integer) return Boolean is

begin

return Interrupt\_Count >= Num;

end Check;

end Handler;

end Sigint\_Handler;

This is the B option code that uses the Sigint\_Handler package, and creates a user defined exception.

**Boption.adb**

with Ada.Text\_IO, Sigint\_Handler;

use Ada.Text\_IO, Sigint\_Handler;

procedure Boption is

User\_Exception : Exception;

procedure CheckMaxInterrupts is

begin

if Handler.Check(10) then

raise User\_Exception;

end if;

end CheckMaxInterrupts;

task Sig\_Handler;

task body Sig\_Handler is

begin

loop

CheckMaxInterrupts;

Put\_Line("Task running...");

delay 1.0;

end loop;

exception

when User\_Exception =>

Put\_Line("User Interrupt generated. Terminating Program...");

end Sig\_Handler;

begin

null;

end Boption;