**Project 3 EGR 423**

**Paul Pace**

**6/19/2020**

**Introduction:**

Concurrency is a method of creating multiple running processes in parallel. Parallelism has several uses from reducing process times to real-time systems. Concurrent programing has found common usage in embedded systems such as air traffic control, engines, environmental monitors and many more. As we grow into a world with where multiprocessor hardware is common concurrency will become not only more important but necessary to take advantage of many advantageous properties that come with concurrency.

Concurrent systems will make extensive use of the packages Ada.Calendar and Ada.Real\_Time. Ada.Calender provides a type time and various subtypes to give time notation that we are familiar with. This Package does acknowledge leap-seconds, years, and other adjustments within the Gregorian Calendar. The monotonic package Ada.Real\_Time though like Ada.Calendar does not subdivide its types into days, weeks, and years. There are additional Packages that provide for advance arithmetic on the Ada.Calendar type time (Ada.Calendar.Arithmetic) and to handle time zone conversion (Ada.Calendar.Time\_Zones).

The time type becomes useful when using delay statements which come in two forms. The simply delay statement,

Delay 5.0;

Will delay a minimum of 5 seconds with no necessary upper bound. This form of delay can cause something know as drift which is the amount of difference from the actual delay to the desired delay. Drift can add up overtime especially in repeated delay structures. But consider a factory that must activate something at a specific delay repeatedly, without drift. This can be solved using the delay until.

Delay until 5.0;

The delay until must have the time type but can have relative statmentes so,

Delay Begin; --illegal

Delay until begin; --Legal

This allows for a variety actions that can be taken about the amount of delay to give with various types.

Here is an example program that takes advantage of the various delays. Do note that some output might depend on your system and operation conditions.

Example 3.1

with Ada.Integer\_Text\_IO;

with Ada.Text\_IO; use Ada.Text\_IO;

with Ada.Calendar; use Ada.Calendar;

procedure Example is

--package renmes

package Int\_IO renames Ada.Integer\_Text\_IO;

--Task declartion

task Increase; --increase X

task Decrease; --Decreases X

task Speak; --Outputs X

--Variables

Done: Boolean:=false; --Variables to end all loops in the tasks

X: Integer :=0; --X variables modified by the tasks

Start:Time :=Clock; --Staart of the system used for timing

--Task bodies

--Increasees X by one on every itteration of a loop

--Has a minimum delay of 0.2 seconds

task body increase is

begin

loop

X:=X+1;

exit when done;

delay 0.2;

if X > 100 then done:= True; end if;

end loop;

end increase;

--Decreases X by 2

--Has a standard repeated delay of 1 second

task body Decrease is

Interval: Duration:= 1.0;

begin

loop

X:=X-2;

exit when done;

delay until Start + Interval;

Interval := Interval + 1.0;

if X < -100 then done:= True; end if;

end loop;

end Decrease;

--outputs X

--Has a set repeated delay of 1 second

task body Speak is

Interval: Duration:= 2.0;

begin

loop

Int\_IO.put(X);

delay until Start + Interval;

Interval := Interval + 1.0;

if X mod 5 = 0 then

new\_line;

end if;

if X > 100 then done:= True; end if;

exit when done;

end loop;

end Speak;

begin

--Minmum delay of 5 seconds then outputs DONE

delay 5.0;

done := true;

new\_line;

put\_line("DONE");

null;

end Example;

**Getting Started:**

Concurrent programing in Ada is provided throug the task structure which is similar to the package format. Here is a task from the previous example.

task Increase; --Specification

task body increase is --body

begin

loop

X:=X+1;

exit when done;

delay 0.2;

if X > 100 then done:= True; end if;

end loop;

end increase;

This is the basic task body which will run alongside your main program and other tasks. We will start with the body of task. The body of a task is like that of a procedure not containing its own return statement but has various computations. Task bodies will also contain, as we will learn later, the structure for communication between tasks. A task can be considered a active entity which will spontaneously do its own actions unlike its reactive entity counterparts the function and procedure.

Task will activate at the begin statement of larger body they are in. Here is a function that we can add to our previous example.

separate (Example)

function taskExmpl (A: in out Integer) return Integer is

task change1;

task change2;

task body change1 is

begin

A:= A+5;

end change1;

task body change2 is

begin

delay 0.5;

A:= A\*\*2;

end change2;

begin

delay 1.0;

return A;

end;

This function will not modify A until the begin statement then it will follow this timeline. To the Left.

Notice that a portion of a program will not terminate until all of its tasks terminate this becomes more important when using several subprograms that could end a deadlock (a situation that ends in multiple tasks preventing each other from terminate.

Ada is a concurrent language that is concurrent entities can be clearly identified within the code. But not all languages have the same structure below is the basic structure of a thread (similar to a task) in Java and a short comparison between them.

|  |  |  |
| --- | --- | --- |
| **Java**  Runnablemy\_ my\_task = () -> {  //Do something  };  My\_task.run(); | | **Ada** |
| task my\_task;  task body my\_task is  begin  --Do something  end increase; |
| **Topic** |  |  |
| **Activation** | My\_task.start(); | Implicitly |
| **Communication** | Not Applicable | Rendezvous |
| **Relative Delay** | Sleep(X) [X in milliseconds] | Delay X [X n seconds] |
| **Limited Delay** | Not Applicable | Delay until X (of type time) |
|  |  |  |

This is just a short comparison, but one can see Javas basic threading system does lack some Ada features (Java does have a Concurrency API with additional features that we will not get into here). Yet Java does some features such as limiting startup explicitly which can have its uses.

Another way to implement a task is with Task types. The task type can be used to create multiple tasks that operate in an identical manner and/or to allow a task to inherit from an interface. Task type are objects similar to a record, fully capable of having discriminates (but only of discrete or access types), being put into data structures like arrays or records. Here is a simple task type.

Task type TT (Data: Integer) is

--Entries for communication

End TT;

Task body TT is

--Private variables, tasks, or subprograms

Begin

--body of the task type

End TT;

**Communication:**

Example 3.1 shows three separate tasks that outside of how they influence X. Theses tasks influence on X is a type of indirect communication but this has some rather inconsistent results, as seen by the table below.

|  |  |  |
| --- | --- | --- |
| **Run #** | **Operation time** | **Output** |
| **1** | 7.29s | -1 19 22 25  30 31  DONE |
| **2** | 7.28s | -1 19 22 27 28 33  DONE |
| **3** | 6.27s | -1 21 24  25  30  DONE |
| **4** | 7.31s | -1 19 22 27  28 31  DONE |

Although our final output is consistently around or above 30 and our run time is never below 6 second consider a mission critical system, such as a ventilator, our inconsistent compilation time could result in server damage to equipment or in some cases the loss of life.

One method to solve our problem is to let our task have knowledge through a process know as Synchronization. But the use of Synchronization does have one dangerous pitfall, the deadlock. A deadlock often occurs when multiple tasks are in a system such that one waits on another’s action, and in that process a miscommunication occurs such that multiple tasks are waiting on each other to finish so that they can start, hence the term deadlock. When a deadlock occurs the only way to end it is to hold the program and restart.

The second danger of synchronization is the lockdown this often effects only one process at a time preventing it from accessing a variable or critical section (set of statements used to manipulate a shared variable). This might occur when higher priority tasks permanently (or for indeterminately long time) postpone one tasks access to a critical section. Luckily this can commonly be prevented by also adding a upper bound on how long a task can wait for access to be granted.

Rendezvous Model:

A strong form of direct task communication is through the rendezvous model. In Ada this could be viewed like a client/server interaction. This means that a task provides a service often called an Entry.

Here is an example task with an Entry.

Task Entry\_task is

Entry GetX(A: out Integer);

End Entry\_task;

Task Body Entry\_task is

X: Integer := 3;

Begin

Loop

--Things to do before Accepting the entry

Accept GetX (A: out Integer) do

A:=X;

End GetX;

--Things to do after Accepting GetX

End loop;

End Entry\_task;

As you can see the structure of an entry is common to the procedure with its input discriminates variables each having a tag for read-write capabilities. The major difference is that the body of an entry begins with Accept and rather than the declaration of the Accept statement ending in is, the accept statement declaration ends in do.

An Entry will make a queue for tasks that wish to use them. Then number of tasks in the queue can be accessed by the attribute ‘Count. Only one task can be processed at a time so this attribute can be very important with multiple entry tasks which will be processed in order.

Entries can also be made in arrays or families below is an example entry family.

|  |  |
| --- | --- |
| **Specification** | **Body** |
| Task EFT is  Entry E(1..4) (A: in Integer, B out Float)  End EFT; | Task body EFT is  Begin  Loop  For I in 1..4 loop  Accept E(I) (A: in Integer, B out Float)  --Do something  End E(I);  End loop |

Each of the entries in the family can also have its own body and will be called individually. Since the task will go through the accept statements in order as it goes through its loop the entry family is useful for setting the order of operations and priority communication channels.

For another task to join an entry queue it works like accessing a procedure from a package.

EFT.E(1)(5, 3.72);

Unlike packages the use clause cannot be used on a task, but if one wishes to simplify their entries name (commonly used when entries are nested), they can use a procedure to rename the entry from the uppermost level.

Procedure enterEFT (A: in integer, B: out Float) renames EFT.E(1);

Select Statements

As was mentioned before when a task has multiple accept statements with queues it will take one entry from each accept statement sequentially. This may not be the optimal solution for your system. An alternative is the select statement.

A select statement can be found in an accept statement with this structure.

Select

Accept E1;

--Statements to do after E1

Or

Accept E2;

--Statement to do after E2

End select;

This form of select statement will arbitrarily choose which accept statement is used with rather limited logic. But logic can easily be added.

` Select

Accept E1;

--Statements to do after E1

Or

Accept E2;

--Statement to do after E2

Or

When E2’Count > 5 =>

Accept E3;

End select;

This select statement will then arbitrarily choose between E1 and E2 but not E3 unless E2 has more than 5 tasks waiting on it.

But consider we wanted a task to wait for some time to occur (such as a error detection that uses long times of silence in other accept statement to call an error). This statement uses a delay instead of when. It is important to not that although a select statement can have multiple delayed responses they can only use relative delay (delay X) or set delays (delay until X). Here is an example of such a select statement.

Select

Accept E1;

--Statements to do after E1

Or

Accept E2;

--Statement to do after E2

Or

When E2’Count > 5 =>

Accept E3;

Or

Delay 1.05

--Declare an error message or contingency in case of timeout (or another accept statement)

End select;

The final addition to the select statement is the terminate form. This can include any of the additions such as delay or logic addition. Here is our example with a termination block.

Select

Accept E1;

--Statements to do after E1

Or

Accept E2;

--Statement to do after E2

Or

When E2’Count > 5 =>

Accept E3;

Or

Delay 1.05

--Declare an error message or contingency in case of timeout (or another accept statement)

Or

When E1’Count > 5 =>

Delay 5.01;

Terminate;

End select;

In the case that no entries come in and one wishes to have some code to be active in this circumstance, one can use the else block by simply using else instead of or. Like in an if statement the else statement will always be the last block in the select statement body. Adding to our example.

Select

Accept E1;

--Statements to do after E1

Or

Accept E2;

--Statement to do after E2

Or

When E2’Count > 5 =>

Accept E3;

Or

Delay 1.05

--Declare an error message or contingency in case of timeout (or another accept statement)

Or

When E1’Count > 5 =>

Delay 5.01;

Terminate;

Else

--Busy work

End select;

This has been a discussion over the various forms of communication between tasks. Shared variables can be used but are rarely recommended due to inaccuracies between tasks interactions with a variable. The rendezvous model is an excellent substitution allowing for tasks to queue up to access a variable or interact with each other and these interaction can be strongly controlled using a select statement.

Requeue

Another method of communication is the Requeue statement this statement will redired an entry call to another entry (not necessarily in the same task or protected object) and place its caller (the task requesting entrance) to the other entry queue. The optional with abort tag also informs the currently called task what to do after the caller is required. When aborted the called upon task may continue as normal while the caller must wait for it to leave its new queue.

Accept Entry1 is

--Do something

If Some\_condition then

Requeue Entry1;

End If;

End Entry1;

Accept Entry2 is

--Do something

Requeue Entry1;

End Entry2;

In this example Entry1 will send a object back into its queue when a certain condition is met. And Entry2 will always send a request into Entry1. Since the Entry Queue is first-in first out the requeue is useful when some kind of service cannot be used currently (such as when there is no space in memory).

**Objects and Tasking:**

Ada has some interesting mechanics when it comes to OOP and taksing. The first was mentioned above the task type.

Task type TT (Data: Integer) is

--Entries for communication

End TT;

Task body TT is

--Private variables, tasks, or subprograms

Begin

--body of the task type

End TT;

Task types can inherit from interfaces to have certain procedures and function. But this is not the only modification to the tagged type used in concurrent programing. These are the synchronized, protected and task interfaces, all of which are an extension of the limited interface. The protect and task interface are also extensions of the synchronized interface.

The protected interface can only be used by a protected type and the task interface can only be used with a task type.

A strong connection from OOP to concurrency is the use of the protected object. The protected object has the same inheritance limitations as task type inheritance, that is they can inherit only from interfaces. Here is a sample protected object.

Protected type P\_Object is

Function Get return Data;

Procedure Set (Value: in Data);

Private

Info: Data;

End P\_Object;

Protected body P\_Object is

Function Get return Data is

Begin

Return Info;

End Get;

Procedure Set(Value: in data) is

Begin

Info:=Value;

End Set;

End P\_Object;

One might notice that the protected type does not have an internal loop structure as many of our previous tasks which had entries had. This is because the protected object has the select and looping structures prebuilt into its form and because the protected object is passive it does not have any internal termination capabilities as it will automatically terminate at the end of its scope.

Protected object can also use entries with one addition, they have a condition for them to execute. These Entry conditions are called barriers. The entry barrier works like the logic addition to a select block.

Protected type P\_Object is

Function Get return Data;

Entry Set (Value: in Data);

Private

Info: Data;

Getting: Boolean:=false;

End P\_Object;

Protected body P\_Object is

Function Get return Data is

Begin

Getting:=True;

Return Info;

Getting:=false;

End Get;

Entry Set(Value: in data)

When not getting is

Begin

Info:=Value;

End Set;

End P\_Object;

As with task entries the protected entry has the ‘Count Attribute. There can also be entry families but these cannot be enumerated like a task entry family thus a basic entry family declaration might be.

Type Channels is range 1..6; --used for simplification and potentially reduction of errors

Protected type P\_Object2 is

Entry Receive(Channels)(X: in Data);

…

End P\_Object2;

Protected body P\_Object2 is

Entry Receive (for I in channels) is

When Barrior\_Condition(I) is

Benign

…

End

…

End P\_Object2;

There can be access types to a protected type which work as normal though these can also access a subprogram (except entries).

Type PO2\_Ptr is assess all P\_Object2;

Type Get\_PO is access protected function Get return Data;

**Pragmas:**

There are two prominent pragmas when it come to Concurrency, the Volatile and Atomic Pragmas.

The Pargma Volatile is a Pragma used to prevent a complier to optimize code by saving data into the register and instead will allow all reads and writes to be send to memory.

Explosive: Data;

Pragma Volatile(Explosive); --the object Explosive is volatile

The extension of the Volatile pragma Volatile component works the same way but for arrays setting each component as volatile.

The Atomic and Atomic\_Componets pragmas limit reading and writing to be direct to memory like the volatile but includes capabilities to keep the variable or array consistent.

Type Protons is array (Integer <>) of Data;

Pragma Atomic\_Components (Protons); --All components of the Protons type will be atomic

**Program**

This is a sample version of the sieve of Eratosthenes it takes inspiration form a Sieve from John Barns Programing in Ada 2005. His version can be found on pages 541 to 544. Each File name is its header and can be copied directly in the GPS IDE.

The first is primeFinder.adb this is the main unit.

primeFinder.adb

----------------------------------------------------------------

--Project: Prime Finder ----------------------------------------

--This project is a recreation of the sieve of eratosthenes ----

--Takes insparation from Programing in Ada 2005 by John Barnes--

--Author Paul Pace Writen For ECE 423 Mercer University ------

--Date of completion: 6/19/2020 --------------------------------

----------------------------------------------------------------

With Ada.Text\_IO; Use Ada.Text\_IO; --Text input output

With Ada.Characters.Latin\_1; Use Ada.Characters.Latin\_1; --Used to make colored text

With Ada.Integer\_Text\_IO; --Used to get Maximum seach number

with sieve; --Custom Package to run the sieve

Procedure Primefinder Is

package Int\_IO renames Ada.Integer\_Text\_IO; --Renaming to Int\_IO for easier output

--Maximum number to search for primes

Max: Integer:= -1;

Begin

Put(ESC & "[92m"); --Sets color to greeen

Put\_Line("Welcome to the Prime Finder v0.0"); --Greeting

--Loops until user inputs postivie number

while Max < 1 loop

Put\_Line("Put the maximum number to search for prime");

Put(ESC & "[93m"); --Color is Yellow

Int\_IO.Get(Item => Max);

--If input is less than 0 then output this

if Max < 1 then

Put(ESC & "[91m"); --Color is red

Put\_Line("INVALID INPUT: PLEASE USE A POSITIVE NUMBER");

end if;

end loop;

--Logic to tell if we need to start the sieve

if Max > 3 then

--Start sieve

sieve.start(Max);

else

--when less that 3 we can just output 1 to 3

for I in Integer range 1 .. Max loop

Int\_IO.Put(I);

end loop;

end if;

Put(ESC & "[0m"); --Set text color to white so that next start will look

--Corect

End Primefinder;

sieve.ads

--------------------------------------------------------------------------------------

--Project: primefinder ---------------------------------------------------------------

-- This package operates the sieve functionality in the primeFinder project-----------

--Takes inspiration from ------------------------------------------------------------

--John Barnes' Programing in ADa 2005 Sieve design pg 541-544-------------------------

--Date of Completion 6/19/2020 -------------------------------------------------------

--Writen by Paul Pace for EGR 423 Mercer University ----------------------------------

--------------------------------------------------------------------------------------

package sieve is

--Procedure to activate the sieve

procedure start(Max: in Integer);

--task type Prime with entries

task type Prime(Num : Integer) is

entry Recieve(Val : Integer); --Recieve a new number to test

entry Finish; --Closes Task

end Prime;

function new\_prime (Value: Integer) return access prime; --Creates new Prime

private

--Protected type that operates line separations

protected type lineMaker is

procedure Increase; --Increments state by 1

private

State: Integer:=3; --Current state 3

end lineMaker;

end sieve;

sieve.adb

-----------------------------------------------------------------------------

--Project: primefinder ------------------------------------------------------

-- This package operates the sieve functionality in the primeFinder project--

--Takes inspiration from ---------------------------------------------------

--John Barnes' Programing in ADa 2005 Sieve design pg 541-544----------------

--Date of Completion 6/19/2020 ----------------------------------------------

--Writen by Paul Pace for EGR 423 Mercer University -------------------------

-----------------------------------------------------------------------------

with Ada.Integer\_Text\_IO; use Ada.Integer\_Text\_IO; --For intger outpu

with Ada.Text\_IO; --For Text (New line) output

package body sieve is

--Renames Ada.Text\_IO to Text

package Text renames Ada.Text\_IO;

--LIne maker ptr type is access to a lineMaker

type lineMaker\_ptr is access all lineMaker;

LineMaster: aliased lineMaker; --LineMaster Alliased lineMaker

---------

--Start--

---------

--Runs sieve

--Variables

--Max: in Intger maximum number to search for Primes

procedure start(Max: in Integer) is

Top: access prime:= new\_prime(3); --Top most Prime

Current: Integer := 5; --Current Value to test for Primes

begin

Put(1); Put(2); Put(3); --Puts 1 2 3 since they will not output otherwise

--Main loop that will increment through primes eleminates need for

--Prime valued 2 by skiping over all even numbers

while Current <= max loop

Top.Recieve(Current); --Send current value down Prime List

Current:=Current + 2; --Increment by 2

end loop;

top.finish; --Have topmost task finish

end start;

-------------

--New Prime--

-------------

--Creates and retures a new access type of a prime

--Value: is the value of the new prime

function new\_Prime(Value: Integer) return access prime is

begin

return new Prime(Value);

end new\_prime;

---------

--Prime--

---------

--Task body for the type Prime used to test if a number is a prime

--Test: Integer to evaluate

--Next: The Next Prime in the list,

task body prime is

Test: Integer; --Number to be evaluated

Next: access prime :=null; --Next Prime in the list

Lines: lineMaker\_ptr; --Line Maker

begin

Lines:= LineMaster'Access; --Sets lines to LineMaster

--Operative loop

loop

--Accept Recieve sets test to new value Val

select

--Val is new value for Test

accept Recieve (Val : in Integer) do

Test:= Val;

end Recieve;

--Evaluation of new value

if Test mod Num /= 0 then

if Next=Null then

lines.Increase; --incremnes lineMaster

Put(Test); --Outputs new Prime

Next:= new\_Prime(Test); --Puts new prime on the list

else

Next.Recieve(Test); --Sends Test value down the list

end if;

end if;

--Accept Finish :Exits loop

or

--Finish: sends Finish call down the list unless next is null

Accept Finish do

--Checks if Next is null

if Next /= null then

Next.Finish; --Sends finish call to next

end if;

end Finish;

--Exits loop after processing Finish

exit;

end select;

end loop;

end prime;

----------------

-- Line Maker --

----------------

--Protected type lineMaker used to operate lines of integers

protected body lineMaker

is

--Increase increments State and

-- if state Equals number of Columns (5 by deault)

--Will make a new line to put integers on

procedure Increase is

begin

--Check for state Equal to number of Columns

if State = 9 then

text.New\_Line;

State:=1; --Reseting of state

else

State:= State + 1; --Incremnatation of State

end if;

end Increase;

end lineMaker;

end sieve;

**Time Records**

Direct:

|  |  |  |
| --- | --- | --- |
| Non-Direct time | | |
| **Date** | **Time (in minuets)** | **Description** |
| 5/17/2020 | 31 | Going over syllabus, Evaluating ADA IDEs and installing GPS |
| 5/17/2020 | 113 | Started GNAT Tutorial, Ended on the Debug Section |
| 5/18/2020 | 99 | Finished GNAT Tutorial and read some of the guide. Ganed a basic understanding of the editor. |
| 5/19/2020 | 147 | Read bgining parts of Wikkibooks, Wrote Hello world program and began experamenting with various basic programing functions in Ada |
| 5/20/2020 | 102 | Read Ada for Java and C++ Devlopers Gave wonderful insights into how Ada works |
| 5/20/2020 | 82 | Practiced with functions and Get\_Line, gained a genral understading of method equivalents from Java |
| 5/20/2020 | 70 | Wrote simple programs for all types of loops and did addiontal research into String usage |
| 5/21/2020 | 135 | Started on Project 1 Wrten Portion, Brain Storming and history section, Also did reserch into file input |
| 5/23/2020 | 80 | Worked on editor section of Project 1 describing the views with the GPS ediotr |
| 5/24/2020 | 62 | Explored Records, Finished writing editor section or project report |
| 5/24/2020 | 218 | Wrote on Loops and control structures, designed first package |
| 5/25/2020 | 133 | Wrote on Conditional Logic and finished loops, read a bit from wikibooks |
| 5/26/2020 | 119 | Wrote Types section |
| 5/27/2020 | 136 | Reorganized project and edited sections |
| 5/27/2020 | 63 | Wrote opperator section, designed a simple record |
| 5/28/2020 | 139 | Rewrote Types sction, completed opperator section, wrote Example 1.1 |
| 5/29/2020 | 10 | Contating books sellers to find ETA of my textbook, no answer |
| 5/29/2020 | 152 | Restarted on an example driven version of part 2, this is what I will go with. |
| 5/29/2020 | 115 | Read first three chapters of Ada2005 by Josh Barns (It finnally came in() |
| 5/30/2020 | 204 | Wrote all of part 1 of my final report of project 1 |
| 5/30/2020 | 186 | Wrote most of part 2 of final version of project 1 |
| 5/31/2020 | 199 | Educated mysleft on access types and wrote on them |
| 5/31/2020 | 82 | Fnised writing project 1 |
| 6/1/2020 | 196 | Began development of a simple personal banking/bugeting program |
| 6/2/2020 | 72 | Read chapter 11 of Programing in Ada 2005 |
| 6/3/2020 | 59 | Reformated and commited my example program |
| 6/3/2020 | 92 | Read chapter 12 of Programming in Ada 2005 |
| 6/4/2020 | 113 | Read chapter 13 of Programing in Ada 2005 |
| 6/5/2020 | 46 | Read on taged types and chapters 14-16 in Ada 2005, also research a bit into Ada for game development |
| 6/5/2020 | 33 | Read on concurrent programing |
| 6/6/2020 | 65 | Read OOP techniques from Programing in Ada 2005 |
| 6/7/2020 | 90 | Read about tasking techniques and started designeing a program that takes advantage of it |
| 6/8/2020 | 157 | Brainstorming Project 2 topics and examples |
| 6/8/2020 | 98 | Began writing code for project 2 |
| 6/9/2020 | 49 | Writing and debugging code for Project 2 (More debugging than writing) |
| 6/10/2020 | 142 | Completed Example 2.1 and worte first part of project 2 |
| 6/11/2020 | 189 | Completed section 1 of Project 2 and started on section 2 |
| 6/12/2020 | 273 | Wrote section on OOP and Exceptions |
| 6/13/2020 | 362 | Wrote on general types and will start on tasking and techniques tommrow |
| 6/14/2020 | 76 | Started writing on tasking |
| 6/14/2020 | 372 | Finished project 2 |
| 6/15/2020 | 135 | Wrote intro to Project 3 |
| 6/16/2020 | 171 | Finished Intro, Basic of Task type and started on introproccess comunication |
| 6/16/2020 | 95 | Wrote on deadlocks and lockdowns/brainstormed remander of project |
| 6/17/2020 | 146 | Wrote on select statements and rendevous |
| 6/18/2020 | 260 | Finshed writing paper portion of project 3 |
| 6/19/2020 | 256 | Wrote Program for Project 3 |
| **Total** | **6224** |  |

Indirect:

|  |  |  |
| --- | --- | --- |
| Direct Time | | |
| **Date** | **Time (Minites)** | **Description** |
| 4/21/2020 | 4 | Posted on facebook about what IDE others are using had short discussion Sy Nu about GPS |
| 4/22/2020 | 11 | Recearched and Responded to a question posted by Alexander Rozeboom about one of the textbooks |
| 5/25/2020 | 36 | Watched Beginning of Video Tutorial sugested by Caleb Nelson |
| 5/27/2020 | 6 | Checked Facebook |
| 5/27/2020 | 88 | Watch Major sections of Tutorial sujested by Caleb Nelson |
| 5/28/2020 | 62 | Finsihed Tutorial Series Sugested by Caleb Nelson and commented on his comment that contained it |
| 5/28/2020 | 5 | Checked Facebook |
| 5/29/2020 | 12 | Checked Facebook for a sample project after the evaluating the Slyabus, then Email Dr. Choi |
| 5/30/20202 | 6 | Checked Facebook |
| 6/1/2020 | 3 | Checked Facebook |
| 6/2/2020 | 4 | Checked Facebook |
| 6/9/2020 | 56 | Researched and posted on facebook deeper information on Ada game development. |
| 6/9/2020 | 1 | Checked Facbook replied to Dr. Chois reply to post earlier in the day. |
| 6/16/2020 | 5 | Replied to Dr. Choi's facebook post on prefered compiler |
| 6/16/2020 | 7 | Checked facebook |
| 6/17/2020 | 12 | checked facebook and responnded to Sy Vu's comment on Dr. Choi's post |
| 6/18/2020 | 14 | Checked Facebook read other replies to Dr. Choi's Post |
| 6/19/2020 | 13 | Made farewell post and read additional comments |
| **Total** | **345** |  |