

FIT2022 Laboratory Session 2

1. Objectives and Outcomes

1.1 Objectives

1. to build upon initial **understandings** of some basic operating system concepts, namely: processes and events;
2. to gain further **knowledge** and **skills** with the Python programming language;
3. to acquire an **attitude** that modelling operating systems gives insight into the behaviour of computer systems.

1.2 Outcomes

At the end of this lab session, you should:

1. be familiar with the basic concepts and ideas of computer system processes;
2. be familiar with how these can be modelled in Python;
3. be aware of the implications of concurrency.

Your demonstrator may ask you questions about these points, and will only give you a "satisfactory" for the lab if you can respond correctly. It is also worth pointing out what is not expected: you do not need to learn anything about literate programs! The program is presented that way as an approach to splitting it up to talk through it. You should be able to do all your coding on the resultant generated program text files. If you are interested in how this literate program works, visit my literate program literate program! (Written in itself!) **Don't forget to write up your lab journal!** This is what your tutor will want to see, to check your learning outcomes. Remember to check the last lab's work out of your svn subdirectory, add any new files, and commit them all at the end of the lab. A good thing to do in your lab journal is to write down the results you found for each example as you work through them, together with any observations you might have, such as "*as expected*", or "*this was a bit surprising*", or "*it took me a while to see what was happening here*". These comments will be very useful to you in your exam revision.

2. Literate Programs

The program you are to work with is presented as a **literate program**. Literate programming is an idea developed by Donald Knuth in the early 1980s as a way of documenting a program by "talking about it as though it were a story to be told". A literate program is written as a series of code or program text **chunks**, interspersed with documentation prose (and possibly diagrams, figures, images, whatever) that explains what the program text is about. This all goes into a single file (conceptually, at least: parts of the file may be included from other sources). One big advantage of this is that when editing the program text, it is a simple matter to update the program documentation as well. Two things happen to the literate program source file: it can be **woven**, meaning that it is assembled into a form suitable for processing as

a document, such as this web page, which has been woven from the same source file used to generate the program text you will use in the lab. Secondly, it can be **tangled**, which means that the program fragments are assembled in the correct order and placement, and written out to a file (again, possibly multiple files, such as .c and .h files) to define the program to be compiled and/or executed. Each of the programs below have already been tangled for you, and clicking on links like this will grab a copy for you that you can save and work on. (NB: You may want to tell your browser that you wish to save such files to disk by setting the preferences. In Netscape, go to the Edit pull-down, and choose Preferences. Select Navigator, and ensure that the arrow to the left is pointing downwards (click on it to change it). Then select Applications. Check if there is an entry for Python programs, and click new (if there isn't) or edit (if there is). Make sure the fields are Description: Python program, MIME Type: application/x-python, Suffixes: py; then click Save To Disk, OK (twice), and you are done!) In the document below, you will see chunks (program text fragments) with a pink background. These are the pieces of code that define the subcomponents of the program. You can follow the links as you read the program, using the **back** browser button to return to where you left off. The convention used here is that the description always follows the related code part. Key identifiers used in the program are also hotlinked to their point(s) of definition and use. At the end of the document are indices for the files, macros and identifiers defined in the document.

3. Introduction to the Process Model

This simulation is about modelling processes, and the issues that arise when we have a number of independently executing programs within the one environment. Basically, a **process** is an instance of an executing program. We use the term process, because there may be multiple processes in the one system that are all instances of the same **program**. (Usually, of course, there are multiple programs in the one system.) In this lab, we will create multiple process instances. You can think of a process as a program, plus state information about its progress in execution. This explains why we need to distinguish processes and programs. A program is a static thing, while processes are dynamic. You don't need to know anything about programs other than the strings of data that make up their code, and the input data supplied to the program. A process, on the other hand, has not only the data that constitutes the code being executed (and this does not change), and the data supplied to the program, but also has some data that identifies the values of data structures used by the program, as well as data that can be regarded as housekeeping data, such as whereabouts in the code we are currently executing (for example, the current value of the Program Counter). The heart of the simulation is this focus upon the key events that happen during a process's life cycle. We are interested in the effect of these events, and we seek to discover something about the real operating system environment by abstracting the events from that

real world system. So we are not building an operating system itself, but rather a model in which the events, the time at which they happen and their effect upon the system, parallel the real system, but all other unnecessary detail is removed. One of the tricky parts you will need to understand is that we are not dealing with the "real" processes in an operating system, but rather an "abstraction" of them. Python is particularly convenient in this respect, as it allows us to model a variety of abstractions directly in the language itself. Hence this first exercise is about building programmed representations of **processes** and **events**. Ultimately, these exercises will coalesce into one large program, that shows a complete "pseudo-operating system" at work!

4. Processes

4.1 A Process Example

Perhaps an example will show what is meant. In the code following (which you do not need to understand fully, but take it a bit on trust at this stage!), we have three processes, two of which are instances of the same program. When you execute it, you should see the execution of the two instances interleaving with each other and with the third process.

```
# FIT2022 Lab 2 example1.py
import time
from threading import *

class ProgramA(Thread):
    def __init__(self, label):
        Thread.__init__(self)
        self.label = label
    def start(self):
        Thread.start(self)
    def run(self):
        for i in range(50):
            print "Program A", self.label, i
            time.sleep(0.5)

class ProgramB(Thread):
    def __init__(self, label):
        Thread.__init__(self)
        self.label = label
    def start(self):
        Thread.start(self)
    def run(self):
```

```

    for i in range(12):
        print " Program B",self.label,i*i
        time.sleep(2)

process1 = ProgramA("process 1"); process1.start()
process2 = ProgramA("process 2"); process2.start()
process3 = ProgramB("process 3"); process3.start()

process1.join()
process2.join()
process3.join()

```

Exercise 1

If you clicked on the link above labelled `example1.py`, you should have a copy of `example1.py`, which you can try running with `python example1.py`.

1. How many times does process 3 print a message, compared to processes 1 and 2?
2. Is it always the same number of Program A steps between each Program B message? Explain where this ratio comes from.
3. Feel free to change some of the values in the program to see what effect these might have.

4.2 The Process Example Dissected

There are a number of important issues in the example above, so let's look at it again, this time breaking it down into various parts, and explaining each of them by means of a literate program. First of all, we define the program. There are a number of slight changes, which we shall also explain as we go.

```

#!/usr/bin/env python
# FIT2022 Lab 2 example1a.py

```

The first line is an addition to `example1.py`, and has nothing to do with processes! It's there to allow us to call the program directly from the command line. It uses the Unix shell convention that program scripts may define which interpreter is to be used to execute them, in this case the Python interpreter. (Note that you may have to check with your tutor as to whether the path name is correct.) Putting this line in, and then changing the permission bits on the file to allow execution (`chmod 755 example1a.py`) allows us to call the program directly, as in `example1a.py`, rather than `python example1.py`. Subsequent lines are a literate program

device to defer definition of the program fragments. Each chunk is referenced by a name, which is given in italics within the angle bracket symbols (also known as "less than" and "greater than"). At the end of the name is a cross-reference to the chunk number(s) that define this chunk. Note that clicking on the chunk name will take your browser to the first so-named chunk, and you can use this device for browsing the code. We now proceed to define the nested code fragments.

```
import time
from threading import *
```

Python allows programs to be constructed from a range of **modules**. You will learn more about modules in other subjects, but note for the moment that modules allow code sections to be written independently, and brought together as required. (A bit like literate programming does in another way.) We rely in this example on two other modules, the `threading` module and the `time` module. These modules are **imported** for use in this program. Note that when imported, we can refer to variables and procedures defined in the module by **qualifying** the name, that is, putting the name of the module before the variable or procedure name, with a dot separating them. We import the `time` module in this fashion. Sometimes this is a bit inconvenient, so we can also import the names in an **unqualified** way. This allows us to use the names **without** the module name in front. We import the `threading` module in this fashion. The `*` means import all names. We could substitute `Thread` for the star, and the program would work just as well, since `Thread` is the only name actually used from the module `threading`.

```
class ProgramA(Thread):
    label = "Program A"
```

Those of you who have studied Java (which should be most of you) will be aware that classes are important entities in **object oriented** programming. We are not about to embark upon explaining object-orientation here, but there are two aspects of OO that are relevant, not only to understand the code, but also because they help understand the (OS) paradigm as well. A **class** is a template for creating instances called **objects**. From the one class, you can create many objects. Each object has its own existence, but all objects behave in similar ways, as defined by the class. Sort of like programs (classes) and processes (objects), really. In this very example, Program A is a "class" that creates two process "objects", process 1 and process 2. Program B is a "class" that has only one "object" instance, process 3. Note that classes and objects are **not** the same thing as programs and processes, but the analogy is very strong. (See also Lab 2 Objective 3, and Subject Objective 8!)

Classes can define variables, which will create a separate instance of a variable with that name in each object created from the class. The variable is in the scope of the object, meaning that it can only be accessed within that object. Program A has a class variable called `label`. Classes have other important components to them, called **methods**. Methods provide a procedure-like interface to objects. Both Program A and B are defined by classes which have three methods associated with them. These methods reflect the behaviour as processes. One very important characteristic of object orientation is **inheritance**. Inheritance allows us to *reuse* methods (and other things) from a parent or super class. For example, if `vehicle` was a class defining the behaviour and functionality of vehicles, then both cars and trucks display behaviour common to vehicles. So we could define sub classes, `car` and `truck`, that inherited from `vehicle`, and this would save us redefining all the methods required that were common to both. Differences could be then defined locally to each of the sub classes. In this example, our processes inherit from a generic model of a process, called a `Thread`, indicated by that name appearing after the class name in parentheses (it looks like a parameter, but isn't). Recall further above how we said that programs were static and processes dynamic. What makes something dynamic? The importance of **time**. A process has a *start time* and an *end time*. In between, it is said to be *running* (leaving aside questions of preemption for now). But before anything can happen to it, it must be *created* and *initialized*. Hence the three methods we define are to *create and initialize* the process, to *start* the process, and to *run* the process.

```
def __init__(self, label):
    Thread.__init__(self)
    self.label = self.label+" "+label
```

This method initializes Program A as a process. The method name, `__init__`, is a special name recognized by the Python interpreter as an object initialization method. (A bit like *constructor* functions in C++, for those of you who know C++.) The body of this method is a call on the `Thread` class initialization method -- which is not entirely surprising, since we want this object instance to behave like a process instance, and threads are a way of modelling processes. The parameter passed to the `Thread` initialization is a reference to this object, `self` (which will be a reference ultimately, when this Python program executes, to the "process 1" or "process 2" object instances. Since all methods need to know the object that invoked them, Python has an implied `self` reference passed as first parameter to all methods. By convention, this is usually given the formal parameter name of `self`. (You can read all about the methods that the `Thread` class supports in the `Thread Objects` reference page.) But we add another twist. When the processes are running, we'd like to know which one is which, so we add a label to each process/object, given by the second parameter. This is copied into the local variable of the process instance *itself*, `self.label` (i.e., the variable defined in the class).

```
def start(self):
    Thread.start(self)
```

To start the process, we call the super class Thread method `start`. (Sort of obvious, really!)

```
def run(self):
    for i in range(50):
        print self.label,i
        time.sleep(0.5)
```

This is what you might think of as the code of the process, or the program proper. It is the code that is executed when the process is running. We make it a very trivial example, since our purpose in this lab is to understand processes, not Python programming (which was Lab 1!) What does the program do? It loops 50 times, printing a message saying who it is (both as a program, and as a process), and the loop counter value. But notice also the `sleep` instruction. This is very important to our understanding of processes.

Exercise 2

One way to understand what it does is to see what happens without the `sleep` instruction. Go to your downloaded code for `example1.py`, and take the `sleep` statements out from ProgramA and ProgramB. Then run the program. Can you explain what happens? This is an example of **process scheduling**, a topic we will visit much later in the unit. If nothing else is stated, when we start a process executing, it will usually run until completion. If we want all three processes to make progress together, there needs to be some way of each process saying that it is prepared to **relinquish control**, and allow some other process to run instead. The simplest way to do this is to just "send the process to sleep", when the other processes can then gain control until they also "put themselves to sleep". This is not all that artificial. As we shall see, I/O activities (like the `print` operations here) take a finite amount of time, and it is usual to allow other processes to run while I/O is taking place.

```
class ProgramB(Thread):
    label = "Program B"

    def run(self):
        for i in range(12):
            print " ",self.label,i*i
            time.sleep(2)
```

Program B has a very similar structure to Program A, so we won't dissect this one any further. But note that since the initialization and starting are exactly the same as for Program A, we can just "reuse" the code from that definition!

```
process1 = ProgramA("process 1")
process2 = ProgramA("process 2")
process3 = ProgramB("process 3")
```

To create the process instances, we create an object of the appropriate class (2 instances of Program A and 1 of Program B, remember). Pass in the label by which each process is to be known.

```
process1.start()
process2.start()
process3.start()
```

These calls start each process instance. Each then proceeds to execute its run method (in turn: note the behaviours described above about running to completion).

```
process1.join()
process2.join()
process3.join()
```

Before terminating the simulation, make sure that each "process" has completed its task.

Exercise 3

Modify `example1a.py` in the following ways, and run each modified version to see the various behaviours.

1. Change the permission bits so that you can run it directly, without having to invoke the python interpreter (check with your demonstrator that the path is correct).
2. Change the parameters by which each process is known (its label). For example, change "process 1" to "process tom".
3. Add a line to the top of the file `from random import *`. Now change the time each process sleeps by calling the random number generator `random()`. You can keep the same relative speeds (roughly) by appending (for example) `*random()` to each parameter. Run it several times. Do you get the same behaviour each time?
4. `random()` gives a uniformly distributed random number between 0.0 and 1.0. You can try other fancy ones, see for example the random module reference page. For example,

```
time.sleep(expovariate(8))
```

!

We pointed out above that I/O takes a finite amount of time, and it is usual to switch between processes at this point. The module `fit2022io.py` provides a `printf` equivalent that you should use in all your programs from here on, which has a built-in wait (sleep) proportional to the length of the string being printed. To use it, do an unqualified import of the module, and call `printf` as you would the equivalent C call:

```
from fit2022io import *
...
printf("format string\n", var1, var2, ...)
```

Exercise 4

Rewrite `example1a.py` to use this `printf` function, removing the `time.sleep()` calls, and check the new behaviour. Don't forget the new lines! (The `\n` character at the end of the format string.)

4.3 Building Processes in Python

If you followed all the above, you should now have a fair idea about how to build a basic process implementation in Python. But wait, there's more! The processes we ran above did not communicate with each other, and they also suffered from the limitation that they were *statically defined*. In operating systems, we need to be able to create new processes dynamically, without having to compile every program we are ever going to run into the operating system code itself! How can we build a model of process that can be defined dynamically? Let's look at this latter issue first. Remember that we defined a process as a program with state. What is that state information? You will see from the lectures that in real life systems it includes things like program counters, stack pointers, buffers, context information, etc., as well as data stored in main memory. What is it in our Python model of a process? Python has a very subtle way of describing the data state of an executing program (process). All the variables are defining in a dictionary data structure, where the current value of a variable may be found by looking up the dictionary with the name of a variable.

Exercise 5

Run the Python interpreter interactively. Type the following commands and observe the behaviour:

```
>>> a
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: a
>>> a=2
>>> a
2
>>> globals()
```

```

{'__doc__': None, 'a': 2, '__name__':
 '__main__', '__builtins__': <module '__builtin__'
 (built-in)>}
>>> globals()['a']
2
>>> b=5
>>> globals()
{'__doc__': None, 'a': 2, 'b': 5, '__name__':
 '__main__', '__builtins__': <module '__builtin__'
 (built-in)>}

```

Do you see how, as variables `a` and `b` are defined, they get added to the global name space, defined by the builtin function `globals()`? (You can ignore the identifiers surrounded by double underscores.) Now you will appreciate that we can put all that into a file and execute it directly:

```

# FIT2022 Lab 2 example2.py
a = 2
print a
print globals()
print globals()['a']
b = 5
print globals()

```

Exercise 6

Download `example2.py` and execute it directly:

```
python example2.py
```

Compare its output with that of the previous exercise. We can also store that program in a string and execute it indirectly:

```

# FIT2022 Lab 2 example3.py
code = """a = 2
print a
print globals()
print globals()['a']
b = 5
print globals()
"""
exec(code)

```

It gets a bit hairy, and is complicated by the fact that the variable `code` is in there as well, but it does work! OK, what about this one?

```

# FIT2022 Lab 2 example4.py
codefile = open("example2.py")
code = codefile.read()
codefile.close()
exec(code)

```

Exercise 7

Download and run programs `example3.py` and `example4.py`. Compare the output of `example2.py` with that of `example4.py`. Spend a little time making sure you understand what is going on with each example. Whoo-Hoo! Time to go and stretch your legs, or do some isometric exercises, while you wrap your brain around that one!!

4.4 Executing Dynamic Processes in Python

What are going to do now is to modify `example1` to make it read in the various programs to be executed, and then execute them. Sort of like a real-life operating system! Here's the basic skeleton, you can download it as `example5.py`.

```

#!/usr/bin/env python
# FIT2022 Lab 2 example5.py
from threading import *
class Program(Thread):
    def __init__(self,code):
        Thread.__init__(self)
        self.code = code
        self.d = {}
    def start(self):
        Thread.start(self)
    def run(self):
        exec(self.code,self.d)
joblist = []
while :

    for instance in joblist:
        instance.start()
    for instance in joblist:
        instance.join()

```

The instance variable `d` is the dictionary used for each process. It maintains the data local to each process. Note that the various code fragments labelled "`example5: ...`" are **not** defined by this version of the program: you will need to edit code for

these operations into `example5.py` yourselves. In the following exercise, and all exercises marked as *Group Exercises*, you should work in pairs. Your tutor will assist in arranging groups, but you should be seated next to your partner. When the exercise asks you to save work in your *group directory*, you should agree with your partner as to whose directory you should use, and you should save the group work in a subdirectory called `groupwork`. For example, suppose your student id is `mememe`, and your partner's is `himher`. You agree that `himher` is where you will store your group work. `himher` creates a subdirectory in his/her working directory called `groupwork`, adds and commits it. Then `mememe` can make a new working directory `groupwork2` (which should be outside the individual SVN working directory), and make it svn managed by:

```
svn                                     checkout
                                        groupwork2
```

Don't try to make both your `groupwork` directories subdirectories in your own workspaces, or `svn` won't see them as shared! In your group, discuss how to implement the various parts of this program that are shown as `literate` program fragments. Allocate the various parts to different members of the group (the first two, and should be allocated to the person setting up the `groupwork` directory, and all the other parts allocated to the other person in the group). Use your SVN `groupwork` directory to save your group's code.

```
# to be written
# to be written
# to be written
# to be written
# to be written
```

5. Communicating Processes

Suppose we want to run a number of processes, but we want them to regulate their execution, so that each takes turns in executing, then handing back control to the next in a well defined sort of way. For example, in `example 1`, we might want the two programs to alternate their print operations. We will rewrite `example 1` to show what we mean.

```
#!/usr/bin/env python
# FIT2022 Lab 2 example6.py
import time
from threading import *
from fit2022io import *
```

```

class ProgramA(Thread):
    label = "Program A"
    def __init__(self,label):
        Thread.__init__(self)
        self.label = self.label+" "+label
    def start(self):
        Thread.start(self)
    def run(self):
        for i in range(20):
            printf("%s %d\n",self.label,i)

class ProgramB(Thread):
    label = "Program B"
    def __init__(self,label):
        Thread.__init__(self)
        self.label = self.label+" "+label
    def start(self):
        Thread.start(self)
    def run(self):
        for i in range(12):
            printf(" %s %s\n",self.label,i*i)

process1 = ProgramA("process 1")
process2 = ProgramB("process 2")

process1.start()
process2.start()

process1.join()
process2.join()

```

Exercise 8

Run `example6.py` and note the interleaving of the two processes. If we want the two processes to interleave on a strict alternating basis, we have to get them to communicate with each other. Think about how you might do this. One way is to use a flag variable, saying whose "turn" it is to go next. The trouble with this, as we shall see in lectures, is that a process has to keep checking whose turn it is, and this can use up processing time better spent doing something else. We would prefer that each process, when it relinquishes control, to "sleep" until its turn comes around. But how do we know when that happens? This problem arises so frequently in operating system design that there is a special mechanism for it, and the Python `threading` module has a special class to handle it. The mechanism is known as an **event**.

5.1 Events

An event can be thought of as a moment in time when something important happens. The problem is, if we want to know when the event happens, we have to watch for it, and that can be expensive. Imagine a phone that, instead of ringing, flashed a light that you could see only if you were watching it directly! You would have to spend the day watching the phone, or else no-one could contact you! (Hmm, maybe that's not such a bad thing ...) Another thing about watching for an event to happen is, how do we know if the event happened before we started watching for it? This is like arranging to meet someone under the clocks at Flinders Street, and we arrive late to find them not there. Have they been and got fed up waiting for us and left? Or are they just later than us, and haven't got there yet? You should be able to see that what is needed is a value that we can test to see if it is set (indicating an event *has* happened), or, if it is not set, we can *wait* for it to get set, and go to sleep in the meantime, knowing that we will be woken up when it finally does get set. Enter the Event. Again, we shall work through an example to see how events work. Suppose we have four processes: one to compute an integer value, one to double it, one to subtract 1 from it, and one to print the result. Each process must wait for the previous one to do its stuff, and when the printing process prints the result, we resume the first process to find the next integer value. Now obviously, you could do this with a simple loop. But we want to see how to do it with processes, so here's how. We make one significant change: the one program generates all four processes, and we use a 4-way test and branch on the process name to decide what activity this process is to do.

```
#!/usr/bin/env python
# FIT2022 Lab 2 example7.py
import time
from threading import *
from fit2022io import *

running = 1; i = 0; di = 0; dim1 = 0;

class Program(Thread):
    def __init__(self, label):
        Thread.__init__(self, name=label)
        self.label = label
    def start(self):
        Thread.start(self)
    def run(self):
        global running, i, di, dim1, k
        global ev0, ev1, ev2, ev3
        while running:
```

```

    if self.label == "next i":

    elif self.label == "double i":

    elif self.label == "minus 1":

    else:

process1 = Program("next i")
process2 = Program("double i")
process3 = Program("minus 1")
process4 = Program("print")

process1.start()
process2.start()
process3.start()
process4.start()

process1.join()
process2.join()
process3.join()
process4.join()

```

You should be able to follow most of this by now. We create four instances of Program, giving each a process name that reflects the process task. That name is used internal to distinguish the four different process bodies, defined separately below.

```

ev0 = Event(); ev0.set()
ev1 = Event(); ev1.clear()
ev2 = Event(); ev2.clear()
ev3 = Event(); ev3.clear()

```

These are the new event variables. There are four of them, reflecting the events:

- + ev0: Either everything is just starting, or we have been through a complete cycle of processing;
- + ev1: The next value of i has been computed;
- + ev2: The doubled value of i has been computed;
- + ev3: $2*i-1$ has been computed.

Initially, we identify the event "just starting up". All other events "haven't happened".

```

ev0.wait(); ev0.clear(); i = i+1; ev1.set()

```

The first process body waits for the event representing "just starting/finished a cycle", resets the event to show that it is now responding to it, does its task (generate the next i), then flags the fact that the next event "next value of i has been computed" has now happened.

```
ev1.wait(); ev1.clear(); di = 2*i; ev2.set()
```

A similar story for the second process body ...

```
ev2.wait(); ev2.clear(); dim1 = di - 1; ev3.set()
```

... and the third ...

```
ev3.wait(); ev3.clear();  
printf("i=%d, 2*i=%d, 2*i-1=%d\n",i,di,dim1)  
if i>=20:  
    running = 0  
    ev0.set()
```

... and here we go full circle. To put some closure on the process, we terminate all processes once i reaches 20. Often in OS programs, loops run "forever", but that's not really appropriate in this example!

Exercise 9

1. Download `example7.py` and run it. Note that it needs the `io` module `fit2022io.py` to operate correctly.
2. Try removing some of the `evX` fragments of code, and see what happens. Good ones to try are the "wait" statements!

Extend the program you developed for Group Task 1 to use events to control the sequence of execution.

6. Reflection

Exercise 10

You **must** attempt this exercise! The groupwork directory owner should create an empty file called `Lab2Reflections.txt` in the directory `groupwork`, and add it to `svn`. Then

- + Think about those things that have worked well, and write them into your `Lab2Reflections.txt`. Commit.
- + Think about those things that haven't worked well, , and write them into your `Lab2Reflections.txt`. Commit.
- + `svn update` your file, and see what happens. Discuss in the file, and commit again!

7. Indices

7.1 Files Defined by this Document

7.2 Macros Defined by this Document

7.3 Identifiers Defined by this Document