THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

COMPSCI 340 & SOFTENG 370 2016
Operating Systems
Assignment 1 – Communicating Processes
due 15th of August 9:30 pm

In this assignment you will implement a process message-passing system. The system can be used to provide safe multi-processing applications that don't rely on shared resources. All information is conveyed between processes by message passing.

Introduction

Python provides a comprehensive number of thread and process synchronization techniques. In particular the standard library includes locks (threading.Lock and multiprocessing.Lock), safe queues (queue.Queue and multiprocessing.Queue), semaphores (threading.Semaphore and multiprocessing.Semaphore), and other useful constructs.

These synchronization techniques all work, but they suffer from programmer problems. Programmers are notorious at not using such synchronization techniques properly. Certainly safe queues are a vast improvement over locks and semaphores and this assignment can be seen as an extension of this style of programming.

Don't share data

If possible (and it usually is), it is preferable to keep data accessible to only one thread in one process. Then we can be absolutely sure that the program won't suffer from concurrent modifications. But of course, our threads have to have access to information that other threads or processes control. In this case a controlling process receives messages asking it to modify and/or return data.

You are going to make a library that adds a powerful but simple message passing synchronization system to Python based on the technique that Erlang uses. Then you will test that library with some common synchronization problems.

In this assignment the communication system will be between different processes. This has the advantage of making sure that one part of our program cannot interfere with the memory in another part of our program, but it has the disadvantage that starting up multiple processes and communicating between them is slower than multiple threads within the same process. In order to communicate between processes you can use a variety of techniques but I would recommend using named pipes (more on those in lectures and tutorials).

The idea

The most powerful idea is that a process can be waiting to receive a variety of different types of messages. When it receives a message it executes the code that is associated with that message.

We can represent sending a message as:

```
give(pid, "data", 7)
```

Figure 1: Sending a message

This sends a `"data"` message with the value of `7` to the `pid` process.

The message is received with a `receive` method. The `receive` can specify which message types it is waiting for and what to do when a matching message arrives.

Say we have a process that repeatedly waits for two types of messages: a `"data"` message, and a `"stop"` message. If it receives a `"data"` message, it prints the data value. If it receives a `"stop"` message it exits.

This could be represented like this:

```
loop do
 receive
   "data" message with value x do
     print x
   end
   "stop" message do
     exit
   end
end
```

Figure 2: Receiving a message

There is a list of messages associated with each process. As a message is sent to a process it is placed on the end of the list. Messages are removed from the list with the `receive` construct. The `receive` statement has to do several things. It must check the list of messages to see if there is one that matches a condition being waited for; if there are any `"data"` or `"stop"` messages in the example above. If it finds a match it removes the message and carries out any actions associated with the message. If a `"data"` message is received the value passed with the message is printed. If a `"stop"` message is received the process exits. In the example, the `receive` statement is inside a loop and so it repeats until it receives a `"stop"` message.

The `receive` statement blocks until it finds a match or there is a time-out (more on that later). Each `receive` can only receive one message, but of course the `receive` can be inside a loop.

So, `receive` checks for matches between messages sent to the process and the messages expected by the process. The messages sent to the process are checked in first come first served order. However if the first message does not match any of the `receive` conditions the second is checked and so on. If we have the following messages in the received message queue:

```
"one", "two", "three", "four"
```

and the receive statement is:

```
receive
  "two" message ...
  "one" message ...
end
```

The `"one"` message is matched, removed, and the associated action is carried out.

The queue would then be:

```
"two", "three", "four"
```

If the next receive was:

```
receive
  "three" message ...
  "four" message ...
end
```

The `"three"` message matches, the associated action is carried out and the queue becomes:

```
"two", "four"
```

There is also a special type of message, `any`, that matches any message sent to the process. If the list of messages sent to the process was:

```
"two", "four", "five", "six", "seven"
```

and the receive was:

```
receive
  "one" message do A
  "four" message do B
  any message do C
end
```

The message removed and acted upon would be `"two"`; the code executed would be `C`. If this same `receive` method was then called again, the next message received would be `"four"`. N.B. In this case the code executed as a result of receiving the message would be the code marked `B`, rather than `C`, because the first match in the `receive` for the message at the front of the queue is for a `"four"` message.

To summarize:

Messages in the queue are examined in order. For each message in the queue the different message types in the `receive` are also checked in order, looking for a match. If a match is found, the message is removed from the queue and the corresponding code in the `receive` is executed. If no match is found the `receive` blocks until a matching message arrives.

Python version

The Python representations look a little different to the presentation above. All communicating processes have to be associated with the class you will write, the `MessageProc` class. To send a message to a process in Python you will use a `give` method in this class, similar to that shown in Figure 1: Sending a message.

```
self.give(pid, 'data', 7)
```

The process identifier will be a Linux process identifier. This number is returned by the `os.getpid()` method for the current process. The message identifier can be any Python object. Usually strings will be used because they are compact and readable. Similarly the data to be transferred with the message can be any Python object or objects; in this case the number `7`. You can easily pass several items of data in one message if you use Python's parameter expansion mechanism e.g. define your give method as:

```
def give(self, pid, label, *values):
```

If one of the values passed with a `give` is the process id of the current process the receiver can then send messages back to this process as a response to this message.

The `receive` example in Figure 2: Receiving a message, should be represented in a Python program as:

```
while True:
    self.receive(
        Message(
            'data',
            action=lambda x: print(x)),
        Message(
            'stop',
            action=lambda: sys.exit()))
```

In this representation the `receive` statement has become a method call. It takes any number of parameters; in the example there are two. In this case each parameter is a `Message` object. You have to write the `Message` class but it is just a container for the obvious fields. The `action=` shows that there is to be a named parameter in the `Message` constructor `__init__` (actually not really a constructor but for those programmers coming from Java it works in a similar way). The action parameter takes a function or method. In this case because the actions consist of only one expression we can use lambda expressions which create anonymous functions. In later examples you will see the more common case of passing ordinary methods.

The `receive` method returns the value of the action associated with the received message, e.g., in the following example the `receive` method returns the value `4` when the `'hi'` message is received.

```
value = self.receive(
    Message(
        'two',
        action=lambda: time.sleep(10)),
    Message(
        'hi',
        action=lambda: 2 * 2))
```

The special message identifier `any` is represented as the string `'any'` in Python, but you should define a constant variable with that value called `ANY`:

```
self.receive(
    Message(
        'one',
        action=self.a),
    Message(
        'four',
        action=self.b),
    Message(
        ANY,
        action=self.c))
```

Here is an entire producer/consumer program using this system.

```
from process_message_system import *
import sys

class Consumer(MessageProc):

    def main(self):
        super().main()
        while True:
            self.receive(
                Message(
                    'data',
                    action=lambda x: print(x)),
                Message(
                    'stop',
                    action=lambda: sys.exit()))

if __name__=='__main__':
    me = MessageProc()
    me.main()
    consumer = Consumer().start()
    for num in range(1000):
        me.give(consumer, 'data', num + 1)
    me.give(consumer, 'stop')
```

There are several things to note here. The `import` at the top loads in your message passing system, it must be called `process_message_system.py` and be in the same directory as this file.

We need a `MessageProc` instance in this process in order to be able to use the system. That is the reason for the `me` variable. This is a special case because this is the original running process and we do not need to start it running, hence we don't call `me.start()`.

The `main` method of `MessageProc` should set up the communication mechanism. In my implementation it is a named pipe in the `/tmp` directory. A named pipe is created with `os.mkfifo` and if it is named with the process id then other `MessageProc` processes will be able to open the pipe to send messages to it, as long as they know the process id of the destination process.

The `Consumer` class is a subclass of `MessageProc`. In this case the `main` method is overridden (but it does call the superclass version as you can see) and is called indirectly via the `start` method. The start method also starts up the new process and runs the `main` method in the new process. The easiest way to do this is using `os.fork`. The `start` method must return the process id of the new process to the parent process.

In this case the new process (running its `main` method) keeps printing out the data sent to it until it receives the `stop` message.

The main process is the producer and it sends the consumer process the data and then finally sends `stop`.

N.B. Whatever system you use for the message passing (e.g. named pipes, connections, sockets, multiprocessing.Queues) must be shutdown and tidied up when your programs finish. No extra files or open connections should be left around. Similarly all processes must terminate cleanly.

The next example is a more sophisticated program with multiple consumers reading input from a shared `Buffer` process.

```
from process_message_system import *
import sys
import os

class Buffer(MessageProc):

    def main(self, main_proc):
        super().main()
        self.main_proc = main_proc
        buffer_space = []
        while True:
            self.receive(
                Message(
                    'put',
```

```
                    action=lambda data: buffer_space.append(data)),
                Message(
                    'get',
                    guard=lambda: len(buffer_space) > 0,
                    action=lambda consumer: self.give(consumer, 'data', buffer_space.pop(0))),
                Message(
                    'stop',
                    guard=lambda: len(buffer_space) == 0,
                    action=self.finish))

    def finish(self):
        self.give(self.main_proc, 'finished')
        sys.exit()

class Consumer(MessageProc):

    def main(self, which, main_proc, buffer):
        super().main()
        self.which = which
        self.main_proc = main_proc
        self.count = 0
        while True:
            self.give(buffer, 'get', os.getpid())
            self.receive(
                Message(
                    'stop',
                    action=self.finish),
                Message(
                    ANY,
                    action=self.handle_input))

    def handle_input(self, data):
        print('{} from consumer {}'.format(data, self.which))
        self.count += 1

    def finish(self):
        self.give(self.main_proc, 'completed', self.count)
        sys.exit()

def add_to_total(n):
    global total
    total += n

if __name__=='__main__': # really do need this
    me = MessageProc()
    me.main()
    main_proc = os.getpid()
    buffer = Buffer().start(main_proc)

    consumers = []
    for i in range(10):
        consumer = Consumer().start(i, main_proc, buffer)
        consumers.append(consumer)

    for i in range(1000):
        me.give(buffer, 'put', i*i)
    me.give(buffer, 'stop')

    me.receive(
        Message(
            'finished'))
    total = 0
    for consumer in consumers:
        me.give(consumer, 'stop')
        me.receive(
            Message(
                'completed',
                action=add_to_total))

    print('The total number processed was', total)
```

The consumers request data as they need it from the buffer. It also shows the use of guards in the message parameters (see below).

Calls to `start` can take any number of parameters as in `Consumer().start(i, main_proc, buffer)`. Any parameters passed here are sent to the `main` method of the new process when it starts running.

Guards

Later in the course you will learn about condition variables. They are queues associated with given conditions. A process waiting on a condition variable is waiting for some condition to change. The same sort of thing is available in our message passing system using message guards. A guard is associated with a particular message type in a `receive` statement. When a message from the queue is being checked to see if it matches a `receive` message type, the guard is evaluated; if it is true then the match is successful; if it is false, the match fails.

A guard is sent as a function (or lambda) to the `Message` class `__init__` method using the `guard=` named parameter. e.g. In the `Buffer` class the buffer will only carry out the `give` message if there is something in the buffer to give to the requesting process because of the `guard=lambda: len(buffer_space) > 0`.

A message without an explicit guard is always ready to work. It has an implicit guard of `lambda: True`.

Time-outs

The last addition to our message system is the concept of a time-out. A time-out is some code that is executed when a certain period of time has passed during which no messages have been matched by the `receive`. In Python a time-out is like a message, but it doesn't use an identifier, instead it accepts a time in seconds.

```
TimeOut(2,
   action=lambda: None)
```

There is no guard on time-outs.

Each `receive` can use only one time-out, if the programmer makes a mistake and includes more than one, the first one in the list of parameters is the one that is used.

If there is no matching message the time-out countdown should start, but this is interruptible in the sense that if a matching message arrives during the countdown the time-out is abandoned and the matching message action is performed instead. This means you will need to use a thread to catch messages and notify the time-out if necessary.

A time-out with a value of `0` means don't wait at all. In this case if there are no queued messages which match the `message` parameters in the `receive` call the action for the time-out is executed immediately. This can be used in lots of interesting ways. The following example throws away all messages sent to the process until one final message is sent.

```
from process_message_system import *

class Flusher(MessageProc):

    def main(self):
        super().main()
        print('before start message')
        self.receive(
            Message(
                'start',
                action=self.flush))
        print('after start message')
        self.receive(
            Message(
                ANY,
                 action=lambda data:print('The first thing in the queue after the flush is',
data)))

    def flush(self, *args):
        self.receive(
            Message(
                ANY,
                action=self.flush), # recursively call the flush method
            TimeOut(
                0,
                action=lambda: None)) # when no more messages return
```

```
if __name__=='__main__':
    me = MessageProc()
    me.main()
    flusher = Flusher().start()
    # put a few things in the queue
    me.give(flusher, 'something', 'wow')
    me.give(flusher, 'something else', 'gosh')
    me.give(flusher, 'HHGTTG', 42)

    # then start the flush method
    me.give(flusher, 'start')

    time.sleep(1) # to give the flusher time to throw away existing messages

    me.give(flusher, 'last', ['this array', 1, 2, 3, 8])
```

What you have to do

process_message_system.py

In a file called `process_message_system.py` you need to write the `MessageProc` class and the `Message` and `TimeOut` classes.

In the Canvas assignment section you will find the example programs given above. Your system must work correctly with these and similar programs.

The markers will use Python version 3 on Linux to run your assignment and so you should check that your solution runs under that environment on Linux before submitting it.

If you are working on the assignment on Linux in the lab you may need a place on the local file system where you can create named pipes. The location you should use is `/tmp`. WARNING: Any files you place in `/tmp` will be deleted when you log out. You can hard code in the `/tmp` part of the named pipes filenames if you want. So the line to create a named pipe could look like this:

```
os.mkfifo('/tmp/pipe' + str(os.getpid())
```

Useful Unix commands

Type these at the command line:

`top` – this shows you the current processes, with the ones using the CPU at the top (by default).

You can use this to keep an eye on all the Python processes you start (which may be more than you expect :)).

`ps` - produces a list of your processes by default

`killall` – you follow this with the name of the command associated with the processes you want to kill, e.g. `killall python3` should kill all of your Python processes.

Occasionally you may get really unlucky and end up with processes in the uninterruptible sleep state. There is nothing you can do to get rid of these processes (even root can't delete them), they only go away when the machine is rebooted (believe it or not).

`pstree` – (if it is installed) prints out a nice tree showing all processes, or if you add a process id then all processes descending from that one.

`rm /tmp/pipe* -` removes all files in the tmp directory starting with the name `pipe`. Probably necessary if your processes don't clean up properly after themselves.

Useful Python classes methods

To send Python objects from one process to another they have to be serialized. The way to do this is with the `pickle` module, in particular `pickle.dump` and `pickle.load`.

Extra section for SE370 students (and any CS340 students who want to)

The example programs start all of their processes from one initial process. Using this technique we always have the process IDs of the processes we want to communicate with. Obviously this limits the way the system can be used. To solve this you need to provide a simple naming server (using named pipes or some other method) where a service can register itself with a

name. Any clients which need a service go to this naming server to receive the process id of the server they want. You may use a constant name for the naming server's pipe. You also need to define a simple message passing protocol to register with the name server and to query it. You will need to submit files which demonstrate how your name server functions (with instructions to the marker how to start it all running). You also need to provide simple (and short) documentation explaining how your name server works and the calling interface (i.e. give example code showing how a service registers itself with the name server and how a client requests that information from the name server).

Package this in a different directory from the normal `process_message_system.py` when you submit your zip file.

Submitting

Make sure your name and login is included in every file you submit.

Use the assignment drop box to submit a zip file with all of your source files for your programs (for CS students probably only your `process_message_system.py` file).

Submit your answers to the questions as a single text file, called `a1Answers.txt`. Please make sure your name and login is in this file too.

Any work you submit must be your work and your work alone

Marking guide

Questions

1. Named pipes are useful as a means of communication between processes on the same machine however there can be a problem with messages from different senders getting interleaved. Explain under what circumstances you can be sure that such messages will NOT be interleaved. (2 marks)
2. In the multiple consumer example the buffer space for the Buffer is a simple Python list which is not protected with a lock. Explain why it is safe for access to this space not to be controlled by a lock? (2 marks)
3. Discuss the security issues involved in using the process id as the message passing identifier. (2 marks)
4. (SE 370 students) Explain, giving an example, how your name server works. (2 marks)

Code inspection

Code in `process_message_system.py` is clear, readable and well formatted. (2 marks)

Execution

`MessageProc start` method starts another process executing the code in the `main` method.

e.g. The following program produces two different process IDs.

```
from process_message_system import *

import os

class SecondProc(MessageProc):

    def main(self):
        super().main()
        print(os.getpid())

if __name__=='__main__': # really do need this
    example = SecondProc().start()
    print(os.getpid())
```

(2 marks)

Messages can be sent with parameters.

e.g. `single_producer_single_consumer.py` program works.

(2 marks)

The order of receiving messages is correct (normally FIFO unless there is no matching message identifier in the `receive` method).

e.g. The `order.py` program prints:

```
one
two
three
four
```
(2 marks)

Guards in messages work correctly as in the `single_producer_multiple_consumer.py` program.

(2 marks)

Timeouts in messages work correctly including as in the `flusher.py` program.

(2 marks)

A more complicated program using all components works correctly.

(2 marks)

(SE370) There is an example program written by the student (a simple client/server system is enough) which demonstrates the use of the naming server and the naming server example works correctly.

(3 marks)

You will lose 2 marks if any garbage is left hanging either as files or processes when a program has finished.

Total marks

CS 340 students (20)

SE 370 students (25 possibly adjusted back to 20 for Canvas purposes)

The assignment is worth 7% of the course mark for both CS and SE students.