Session 5 : Synchronization primitives

Razmig Kéchichian

Learning outcomes

The aim of this session is to familiarize you with tools that allow synchronized access and protection for shared resources in multiprocess and multithreaded Python programs.

I. Introduction

Python provides synchronization primitives for processes and threads in respective modules `multiprocessing` and `threading`. Recall that many Python inter-process and -thread communication primitives are already synchronized, such as shared objects created by `multiprocessing.Value()` and `multiprocessing.Array()`, and `Queue` objects provided in modules `multiprocessing` and `queue`. In this session, we turn our attention to situations where **explicit synchronization** is required, namely in the *critical code section* problem which can by solved via mutex locks, and the simultaneous access to a limited number of resources which can be synchronized via semaphores.

Synchronization primitives defined in modules `multiprocessing` and `threading` have the same API and semantics. We shall study two of these, namely mutex locks defined by `Lock` and semaphores defined by `Semaphore`. Instances of these classes behave the way you expect them to (if in doubt, review your lecture handout). Acquisition and release of objects of

both types is done via methods `acquire()` and `release()` respectively. Furthermore, both classes support the context management protocol allowing the automation of the pair of calls `acquire()` and `release()` in the beginning and at the end of a `with` block, if necessary. An example of mutual exclusion via `threading.Lock` looks like the following.

```
# shared definitions
counter = 0
lock = Lock()

# in threaded code
with lock:
  counter += 1
```

The use of the `with` statement is equivalent to calling `lock.acquire()` and `lock.release()` prior to and after updating the `counter` variable.

Here's a typical use of semaphores. Imagine a client-server system. The client is multi-threaded and connects to the server in separate threads. The server accepts a limited number of client connections, beyond which new connections will have to wait. A typical solution for the client looks like this.

```
# shared definitions
MAX_CONNECTIONS = 10
semaphore = Semaphore(MAX_CONNECTIONS)

# in threaded code
with semaphore:
  connection = connect_to_server()
  try:
    # use connection
  finally:
    connection.close()
```

Once again, the `with` statement can be replaced by `semaphore.acquire()` and `semaphore.release()` calls surrounding the enclosed code block if necessary.

Needless to say, the Python implementation makes sure that locks and semaphores are thread and process safe, i.e. that their methods are atomic. Have a look on the documentation of classes `Lock` and `Semaphore` before moving on to exercises.

Exercise 1: Critical π

In Exercise 1 of Session 3, you were asked to implement a randomization method to estimate the value of π. The implementation was based on a pair of threads, a worker thread that generated random points and counted in-circle occurrences, and a main thread that calculated the value of π.

We'd like to develop this solution to allow several worker threads, allowing the generation of a greater number of points in parallel hence improving the precision of the π estimate.

- Make sure to identify the critical section first and choose the appropriate synchronization primitive to protect against race conditions.
- Provide the number of threads and the number of points to generate per thread as arguments to your program (hint: `sys.argv[]`)
- Since the APIs of `threading` and `multiprocessing` are largely similar, you can opt for a multiprocessing solution for this problem. Pay attention however to the kind of shared variables to use in each case (if necessary, review the documentation of `Value`).

Exercise 2: Fibonacci producers and consumers

In this exercise, we tackle an instance of the Producers and Consumers problem with a **bounded buffer** (if necessary, review your lecture handouts to refresh your memory).

In Exercise 1 from Session 2, you were asked to implement a parent-child process communication via shared memory, whereby the child process generates and writes Fibonacci series to the shared buffer up to the n^{th} element while the parent waits for it to terminate before displaying results.

In this exercise, you are asked to implement a Producer-Consumer solution to this problem to allow the consumer process to read Fibonacci numbers **as soon as they are generated**. Reading and writing takes place from and to a shared buffer of a predefined size.

Here's few hints to assist you.

- Organize your program around 3 processes, a producer generating Fibonacci series, a consumer reading the series and displaying them, and the main process launching the previous two.
- You can opt for a multi-threaded or a multi-processing solution. Pay attention however to the kind of shared buffer to use in each case. For a multi-threaded solution, you need to share an array-type structure between your producer and consumer threads. The standard module `array` provides a data structure called `array` intended for the efficient storage of elements of identical type (as opposed to `list`). For example, you can create an `array` of 5 integers in the following manner.

```
a = array.array('i', range(5))
```

- Since the buffer is limited, you have to use modulo arithmetic to index it when writing to or reading from it (cf. lecture notes).
- Neither the producer nor the consumer processes loop indefinitely in a `while True:` loop; the producer terminates when it reaches n. To terminate the consumer, you can maintain a flag, initially set to `True`, on which the consumer iterates. The producer sets it to `False` upon terminating, allowing the termination of the consumer as well. There's a problem with this termination mechanism however. Can you identify it ? Pay attention to the type of the structure to use to store this flag for a multi-threaded or a multi-processing solution.
- **Optional**: generalize your solution to accept multiple producers and consumers.