# RossROS Design Principles and Lesson Plan

Ross L. Hatton

May 28, 2022

RossROS can be used directly (via the code in `rossros.py`) to support robot programming tasks or as a conceptual framework to provide a first experience in writing multitasked programs. In my "system stack" introductory graduate course, I use it in both ways—first assigning the students to convert a simple robot controller they have previously written as a procedural loop into a multitasked system, and then providing them with the `rossros.py` reference implementation to compare against their own efforts and use in later (higher-level) assignments.

The text below is adapted from the course manual for my "system stack" course. It provides a brief overview of concurrent programming (with links to discussion in greater depth), and provides suggested programming tasks on the path to implementing RossROS.

## 1 Concurrency

It is often useful to decouple the various operations that a robot performs, such that sensing, interpretation, and control run independently and in parallel. This notion of *concurrency* can be implemented via

- Multitasking, in which a single processor rapidly switches between the different functions it is executing, and
- Multiprocessing, in which the functions are assigned to dedicated processors.

Multitasking can be further characterized as being *cooperative* or *pre-emptive*, respectively based on whether the time-sharing of the processor is determined by the functions themselves, or is allocated externally by the computer's operating system. Pre-emptive multitasking is also called *threading*. More information on these concepts can be found at

`https://realpython.com/python-concurrency/`

For pre-emptive multitasking and multiprocessing, there is a risk that multiple threads and processes will attempt to operate on the same data structures at the same time. At a low level, consequences can include a task attempting to read a data from a location in the middle of another task writing to the location, and getting a mix of the bits in old values and new values, or two tasks attempting to write data to the same location at the same time, and ending up storing a mix of bits corresponding to the two values.

At a higher level, interleaving the execution of tasks can cause other problems even if bit-mixing is avoided. For example, the operation `x=x+1` in many languages does not mean "increment the value in `x` by `1`" but instead means "read the value in `x`, add `1` to it, then write the resulting

value into `x`". If two threads or processes attempt to execute this operation at the same time, the sub-operations may become interleaved such that the second thread reads the value in `x` before the first has incremented it. In this case both threads will write out a value to `x` that is `1` more than its *original* value, and an increment will be lost.

These difficulties with concurrency can be addressed by

1. Designing the system to minimize the opportunity for conflicts (e.g., by avoiding situations in which two operations write to the same location) and to be robust in the case that one such error does occur. This tends to be easier when the information being passed through the system is an approximation of an analog signal rather than an encoded text. (See also "Gray code" for an example of handling concurrency problems at a hardware level).

2. Ensuring that operations are *atomic* (not interruptible). Different programming languages handle atomicity at different levels. Some languages guarantee that reading and writing simple data types such as integers and floats is atomic, so that bit-mixing is not a worry, but do not guarantee that writing values to an array will not be interrupted partway through by a read operation. Python goes further, making read and write operations to many array types (including lists) atomic.

   Atomicity for higher-level operations generally needs to be specified by the programmer (to ensure that it is applied to operations for which interruption would be problematic are made atomic, while allowing other operations to be interruptible to keep the program flowing). Setting up the *locks* that ensure atomicity can quickly become complicated. If this were a CS course, we'd stop here and spend a good chunk of time going over various approaches to locking. Many of these approaches, however, are related to passing along discrete messages. By focusing our attention on tasks that broadcast their most-recent estimates of system states on dedicated channels, we can sidestep most of this extra complexity and leave it on a need-to-know basis.

A second effect to be careful of in pre-emptive multi-tasking is when the task scheduler interrupts a set of nominally simultaneous interactions with an external system. For example, if a program polls several sensors in succession to get a snapshot of the external world at a given time, the sensor measurements will be slightly offset in time from each other, which can cause "rolling shutter" distortion. This distortion will be amplified if the task scheduler switches to a different task in the middle of cycling through the sensors, increasing the delay between sensor readings. *Cooperative multitasking* can help to mitigate these effects, by having the programmer specify places where it is safe for threads to switch out (at the cost of requiring the programmer to think about and specify these points).

## 2 RossROS Components

The RossROS design specification implements multitasking in the form of *consumer-producer* objects that pass messages via *bus* containers that provide protections again common low-level problems with multitasking data flow.

## 2.1 Buses

When we assign functions to individual threads, we generally need to provide a means for them to pass information between threads, e.g., so that an interpreter function can see the most recently reported reading from the sensors, and a controller can see the most recent interpretation of the system state.

A basic means of passing messages between threads or processes is to create "message buses" that processes can read from or write to. The messages on these buses can be "state updates" that act as "one-way broadcasts", in that reading the message does not remove it from the bus, or "queued", in that messages are "cleared" as they are read.

The core RossROS code uses "broadcast" messages. Conceptually, this is similar to how global variables can be used to pass information between functions, but with a bit more structure.

### Coding assignment

Define a simple Python class to serve as your bus structure. It should have:

1. A "message" attribute to store values
2. A "write" method that sets the message
3. A "read" method that returns the message

Ultimately, the bus class will also incorporate a "lock" to make "write" and "read" operations atomic. Python actually forces atomicity for operations on basic scalars and arrays, so we can hold off implementing this lock until later (see §2.3 "Concurrent execution" below).

## 2.2 Consumer-producers

The core elements of a robot control program are operations that carry out tasks such as polling sensors, interpreting data, and controlling motors. When these processes exchange data via busses, we can think of them as

- Producers, writing newly-created data to busses;
- Consumer-producers, reading information from some busses, processing that data, and writing the output to other busses; and
- Consumers, reading information from busses and acting on it without publishing data to busses.

These categories closely correspond to our previous notions of sensor, interpretation, and control functions.

### Coding assignment

Define producer, consumer-producer, or consumer functions for the sensor, interpretation and control processes in the previous lesson.

1. Each consumer/consumer-producer/producer should be defined as a function that takes instances of your bus class and a delay time as arguments

2. Each consumer/consumer-producer/producer function should contain a while loop

3. The sensor, interpretation, or control function should be executed inside the loop function, with data read from or written to bus classes as appropriate

4. The loop should `sleep` by the delay amount once each cycle

### 2.3 Concurrent execution

Once the busses and consumer-producers have been created, we can have the consumer-producer functions execute concurrently. Some basic steps to get concurrent execution up and running are:

1. Import the `concurrent.futures` module with:

   ```
   import concurrent.futures
   ```

2. Tell the system to run your system components concurrently. For example, to run your `sensor_function` with inputs of the `sensor_values_bus` to write data to and the `sensor_delay` to set the polling rate together with your `interpreter_function`, you can run

   ```
   with concurrent.futures.ThreadPoolExecutor(max_workers=2) as
      executor:
           eSensor = executor.submit(sensor_function,
              sensor_values_bus, sensor_delay)
           eInterpreter = executor.submit(interpreter_function,
              sensor_values_bus, interpreter_bus,interpreter_delay)
   ```

   Note that to cancel execution of a program running under `concurrent.futures` you will need to hit `ctrl-C` multiple times. Additionally, any errors thrown by the functions will not be displayed in the terminal; this problem can be worked around by adding a line

   ```
   eSensor.result()
   ```

   outside of the `with` block.

3. Make sure that writing and reading bus messages is properly locked. The suggested locking code here is an "writer-priority read-write lock", which means that:

   (a) Only one thread is allowed to write the bus message at any given time, and may not start writing while any other thread is reading the message

   (b) Any number of threads may read the bus message at any given time, but may not start reading while a thread is writing the message

   (c) Write operations are given priority over read operations

   A physical analogy of this situation is a whiteboard at the front of a room, with instructors taking turns writing information on the board, and students able to read the information on the board whenever an instructor is not blocking their view. Writer-priority means that the students may have to wait for information, but it will always be the most up-to-date information (see `https://en.wikipedia.org/wiki/Readers-writer_lock` for more details).

(a) Install the `readerwriterlock` Python module by entering

```
python3 -m pip install -U readerwriterlock
```

at the command line.

(b) Put

```
from readerwriterlock import rwlock
```

in your header

(c) Put

```
self.lock = rwlock.RWLockWriteD()
```

inside your `__init__` method for the `bus` class

(d) Use

```
with self.lock.gen_wlock():
    self.message = message
```

inside your "write" method, and

```
with self.lock.gen_rlock():
    message = self.message
```

inside your "read" method.

### 2.4 Further refinements

RossROS implements several refinements on the ideas above. In particular:

1. Consumer-producers are implemented as a generic wrapper class that takes in a "task" function, ties its inputs and outputs to message buses, and places it within a while loop. Producers and Consumers are then implemented via special cases of the Consumer-producer class.

2. The Consumer-producer class is set up so that the while loop monitors one or more "termination buses" that signal the loop to end. These termination buses can be configured to trigger based on system state or, using a provided "Timer" producer object, after a given elapsed time.