ETEC3702 – Concurrency

Inter-Process Communication 2

https://youtu.be/6rupQaM-nPk

Inter-process Communication 2

Review:

Previously we’ve looked at several methods for allowing processes to communicate with each other.

Here is a summary of the ones we’re looked at:

Queues – Object that allows multiple processes to put and get from the queue.

Pipes – Object that connects two processes.

Managers – Object that allows namespaces that can protect shared attributes / values.

Inter-process Communication 2

Other Methods:

Previously we also mentioned:

Value / Array Objects – Simple data objects that can be shared between processes.

Ctypes – Value / Array Objects are a wrapper for Ctypes shared memory.

Sockets – These allow the sending / receiving of data between processes. Can communicate between processes on the same or on different machines.

Other – Frameworks and protocols that can simplify and standardize this. (Ex: MPI)

We are going to look at a few more of these today.

Inter-process Communication 2

Value / Array Objects:

Built into the multiprocessing module is a special class Value( ).

This works as follows:

shared_variable=multiprocessing.Value([type or typecode],*args)

The first parameter is a python type of a ctypes typecode.
The second parameter is an argument list to be passed on to the constructor of the type.

Inter-process Communication 2

Value / Array Objects:

Here are some examples of creating Value() instances:

```
num=multiprocessing.Value('d',0.0) #double-precision float, initial value of 0.0
counter=multiprocessing.Value('i',0) #integer number, initial value of 0
```

Once a Value() instance has been created, it can be accessed by multiple processes as a shared memory variable.

Access to the shared data-value is performed as follows:

```
num.value=3.1415926
counter.value+=1
```

There are other possible typecode specifiers besides ‘d’ and ‘i’ ...

Inter-process Communication 2

Value / Array Objects:

Possible typecodes:

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2 (see note)
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

Note: The 'u' typecode corresponds to Python’s unicode character. On narrow Unicode builds this is 2-bytes, on wide builds this is 4-bytes.

Inter-process Communication 2

Value / Array Objects:

This same method can be used to make shared arrays of data:

```
sharedArray=multiprocessing.Array('i',range(100)) #make array of 100 integers.
dailyValues=multiprocessing.Array(‘d’,365) #make array of 365 doubles.
```

To access the values of the array, simply access them like a normal array:

```
sharedArray[10]+=5
dailyValues[31]=100.25
```

Note: These arrays are not like normal Python lists. You can’t change the type of elements. You can’t add and remove items from the list. Use a Manager() for that.

Inter-process Communication 2

Complete Value( ) example:

```
import time
import multiprocessing

def someFunc(counter):
    for i in range(10):
        time.sleep(0.001)
        counter.value+=1

if __name__ == '__main__':
    sharedVal = multiprocessing.Value('i', 0)
    procList=[]
    for i in range(10):
        procList.append(multiprocessing.Process(target=someFunc, args=(sharedVal,)))

    for p in procList:
        p.start()
    for p in procList:
        p.join()

    print("Final value:",sharedVal.value)
```

This program should create 10 processes that each increment the shared counter.value 10 times.

What should the final result be?

Increment the Value() object's value

create Value() object

Create 10 processes

Pass Value() object to each process

Start all 10 processes

Wait for all 10 processes

Print the final value

Inter-process Communication 2

We asked the question:

> This program should create 10 processes that each increment the shared counter.value 10 times.
>
> What should the final result be?

The answer should be 100, but here is the actual output across several attempts:

```
Final value: 75
Final value: 66
Final value: 55
Final value: 61
>>>
```

What is going on here?

Inter-process Communication 2

So, what is going on here? Why didn't we get 100?

According to the Python documentation for multiprocessing.Value():

> **"…a new recursive lock object is created to synchronize access to the value."**

So, the assumption might be that we don't have to worry about protecting access to the object since it is locked automatically during access.

So, if the object is locked, why ***is*** there a problem?

The problem is that the automatic lock that surrounds the access to the value is too fine-grained. Meaning that it only locks the object while it is being accessed.

When we perform an operation like `counter.value+=1` in our code, access is **not** performed in an atomic manner.

Inter-process Communication 2

Since `counter.value+=1` is **not** atomic, the execution sequence essentially looks like this:

Acquire Lock
Get counter.value
Release Lock

Add one to the value retrieved.

Acquire Lock
Set counter.value to the new value
Release Lock

Q: Why is this a problem?

Inter-process Communication 2

Since `counter.value+=1` is **not** atomic, the execution sequence essentially looks like this:

Acquire Lock
Get counter.value
Release Lock

Add one to the value retrieved

Acquire Lock
Set counter.value to the new value
Release Lock

Q: Why is this a problem?

A: Multiple processes can lock and get the same value.

They will then all add one to that same value.

And, one at a time, lock and update to the new value.

This is a typical “concurrent-update” problem. **Let’s fix it!!!!**

Inter-process Communication 2

Fixed Value( ) example:

```
import time
import multiprocessing

def someFunc(counter,lock):
    for i in range(10):
        time.sleep(0.001)
        with lock:
            counter.value+=1

if __name__ == '__main__':
    valLock=multiprocessing.Lock()
    sharedVal = multiprocessing.Value('i', 0)
    procList=[]
    for i in range(10):
        procList.append(multiprocessing.Process(target=someFunc, args=(sharedVal,valLock)))

    for p in procList:
        p.start()
    for p in procList:
        p.join()

    print("Final value:",sharedVal.value)
```

Use shared lock when accessing shared value

create Lock() object

Pass shared Lock() object to each process

Inter-process Communication 2

Further Improvement?

This implementation works, but it relies upon the processes to use the lock properly.

Imagine a case where a shared value() or array() item might have numerous different processes that need to access the shared data.

If only **one** of them doesn’t use the lock or uses the lock incorrectly, then we have potential for problems again.   That would be bad!

Is there a way that we can **always** protect access to the data with a lock?

YES! Create a class that uses the “monitor” concept!

Inter-process Communication 2

Monitor Concept:

The monitor concept uses OOP and encapsulation to protect the access to the underlying shared data.

All access to the data is only performed through the defined class methods.

These methods ensure that the lock is used where necessary.

```
class SharedValue(object):
    def __init__(self):
        self.lock=multiprocessing.Lock()
        self.sharedVal=multiprocessing.Value('i', 0)
    def add(self,amount):
        with self.lock:
            self.sharedVal.value+=amount
    def getVal(self):
        return self.sharedVal.value
```

encapsulated Lock() object

Use shared lock when accessing shared value

Inter-process Communication 2

Complete Example:

```
import time
import multiprocessing

class SharedValue(object):
    def __init__(self):
        self.lock=multiprocessing.Lock()
        self.sharedVal=multiprocessing.Value('i', 0)
    def add(self,amount):
        with self.lock:
            self.sharedVal.value+=amount
    def getVal(self):
        return self.sharedVal.value

def someFunc(sv):
    for i in range(10):
        time.sleep(0.001)
        sv.add(1)

if __name__ == '__main__':
    sharedVal=SharedValue()
    procList=[]
    for i in range(10):
        procList.append(multiprocessing.Process(target=someFunc, args=(sharedVal,)))

    for p in procList:
        p.start()
    for p in procList:
        p.join()

    print("Final value:",sharedVal.getVal())
```

Inter-process Communication 2

Value / Array Summary:

Value() and Array() objects allow us to create data that can be shared across multiple processes.

Objects have, by default, automatic locking to unsure that access(get) and update(set) operations are performed in a mutually exclusive manner.

But care must still be taken when these shared variables are accessed by non-atomic operations.

To ensure that safety and liveness properties are preserved, programmers must be cautious and perform process synchronization explicitly where needed.

To reduce chances of problems, I’d recommend using an object to encapsulate shared data.   No matter what, in short, with shared variables, BE CAREFUL!

Inter-process Communication 2

Shared Ctypes Objects:

Ctypes is a library that allows the allocation of c-language compatible blocks of data.

These blocks of data can be allocated to be shared memory.

This is essentially what’s “behind the curtain” when using Value() and Array() objects.

Shared Ctypes memory can be allocated and used directly, but since we’ve already covered a more structured way of using that same functionality we’re not going to cover it in detail here.

Python, starting with Py3.8, introduced a mechanism for direct access to allocated block of shared memory: multiprocessing.shared_memory.

Inter-process Communication 2

multiprocessing.shared_memory

This new sub-module allows allocation and management of truly shared memory blocks.

Here is some of the description from the documentation:

“This module provides a class, SharedMemory, for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine.

...

In this module, shared memory refers to “System V style” shared memory blocks (though is not necessarily implemented explicitly as such) and does not refer to “distributed shared memory”. This style of shared memory permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. Processes are conventionally limited to only have access to their own process memory space but shared memory permits the sharing of data between processes, avoiding the need to instead send messages between processes containing that data. Sharing data directly via memory can provide significant performance benefits compared to sharing data via disk or socket or other communications requiring the serialization/deserialization and copying of data.”

So this new feature allows the allocation of truly shared memory.

Inter-process Communication 2

multiprocessing.shared_memory has some useful classes:

class multiprocessing.shared_memory.SharedMemory(name=None, create=False, size=0)

Creates a new shared memory block or attaches to an existing shared memory block. Each shared memory block is assigned a unique name. In this way, one process can create a shared memory block with a particular name and a different process can attach to that same shared memory block using that same name.

As a resource for sharing data across processes, shared memory blocks may outlive the original process that created them. When one process no longer needs access to a shared memory block that might still be needed by other processes, the **close()** method should be called. When a shared memory block is no longer needed by any process, the **unlink()** method should be called to ensure proper cleanup.

name is the unique name for the requested shared memory, specified as a string. When creating a new shared memory block, if None (the default) is supplied for the name, a novel name will be generated.

create controls whether a new shared memory block is created (True) or an existing shared memory block is attached (False).

size specifies the requested number of bytes when creating a new shared memory block. Because some platforms choose to allocate chunks of memory based upon that platform’s memory page size, the exact size of the shared memory block may be larger or equal to the size requested. When attaching to an existing shared memory block, the size parameter is ignored.

Inter-process Communication 2

Accessing SharedMemory():

multiprocessing.shared_memory.SharedMemory() has the following attributes:

buf - A memoryview of contents of the shared memory block.

name - Read-only access to the unique name of the shared memory block.

size - Read-only access to size in bytes of the shared memory block.

Note:
- The way to access the memory block directly is through the **buf** attribute.
- The **name** attribute can be thought of as a unique ID for the shared block.
- The **size** attribute is read-only and cannot be changed dynamically.

Inter-process Communication 2

Example:

```
import multiprocessing
import multiprocessing.shared_memory
import random

NUM_ITEMS=20
def scrambleProc(lock):
    sharedBlock=multiprocessing.shared_memory.SharedMemory(name="mem1",create=False)
    for i in range(10):
        i=random.randint(0,NUM_ITEMS-1)
        j=random.randint(0,NUM_ITEMS-1)
        with lock:
            temp=sharedBlock.buf[i]
            sharedBlock.buf[i]=sharedBlock.buf[j]
            sharedBlock.buf[j]=temp
    sharedBlock.close()

if __name__ == '__main__':
    sharedBlock=multiprocessing.shared_memory.SharedMemory(name="mem1",create=True, size=NUM_ITEMS)
    lock=multiprocessing.Lock()
    for i in range(NUM_ITEMS):
        sharedBlock.buf[i]=i
    procList=[]
    for i in range(10):
        procList.append(multiprocessing.Process(target=scrambleProc,args=(lock,)))

    for p in procList:
        p.start()
    for p in procList:
        p.join()
    for i in range(NUM_ITEMS):
        print(sharedBlock.buf[i])
    sharedBlock.close()
    sharedBlock.unlink()
```

SM accessed by name

SM block accessed as a byte-array

SM block created by name

Note: SM block not passed!

Inter-process Communication 2

Using the shared memory buffer

The Python struct module can be useful for placing structured data into and pulling structured data out of the memory block.

The primary ways of using this module are pack and unpack:

> **struct.pack()** - encodes data according to a structured format.
>
> **struct.unpack()** - decodes data according to a structured format.

This won’t be covered in detail here, but you should be aware of it and learn about it on your own…

One other option that can make things easier is to use the ShareableList class.

It essentially packs and unpacks the data for you.

Inter-process Communication 2

Other useful shared_memory classes:

class multiprocessing.shared_memory.ShareableList(sequence=None, *, name=None)

Provides a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to only the int, float, bool, str (less than 10M bytes each), bytes (less than 10M bytes each), and None built-in data types. It also notably differs from the built-in list type in that these lists **can not change their overall length** (i.e. no append, insert, etc.) and do not support the dynamic creation of new ShareableList instances via slicing.

sequence is used in populating a new ShareableList full of values. Set to None to instead attach to an already existing ShareableList by its unique shared memory name.

name is the unique name for the requested shared memory, as described in the definition for SharedMemory. When attaching to an existing ShareableList, specify its shared memory block's unique name while leaving sequence set to None.

Inter-process Communication 2

Other useful shared_memory classes:

class multiprocessing.shared_memory.ShareableList(sequence=None, *, name=None)

count(value) - Returns the number of occurrences of value.

index(value) - Returns first index position of value. Raises ValueError if value is not present.

format - Read-only attribute containing the struct packing format used by all currently stored values.

shm - The SharedMemory instance where the values are stored.

Using ShareableList() objects can simplify the conversion to / from a simple shared block of data.

Inter-process Communication 2

Summary:

Sharing memory between processes has both pros and cons:

Good:
- Fast since data isn't being sent back and forth between processes.
- Especially good when needing to share a large amount of data.
- With SharedMemory the processes can access the blocks by name.

Bad:
- Can be more difficult to control since processes are not communicating.
- Because the data is shared, access needs to be synchronized.
- We must provide process synchronization.
- Storing complex data in a shared "block" of memory can be non-trivial.
- Since the data is shared in local memory, transitioning to a cluster or distributed model isn't straight-forward.

Inter-process Communication 2

That’s all for today. Stay safe!