CIS 41B
Advanced Python Programming

Processes

De Anza College

Instructor: Clare Nguyen

Processes

- A process is an executable instance of a program.
- When a process starts, the OS allocates some memory to it so it can use the memory for temporary data storage. The allocated memory is only used by the process and is not shared with other processes.
- When a process runs, one of these scenarios happens:
  - It can run as a single entity on one processor.
  - It can spawn one or more child processes, and these child processes can run on the same processor.
  - It can spawn child processes to run on different processors if the system is a multi processor or multicore system.
- On a multicore system, typically the OS scheduler decides which process runs on a particular processor.
- The process can also ask to be run on a specific processor.
- Processes on a system or across systems can communicate with each other through sockets and pipes.

© 2019 C. Nguyen

Multiprocessing

- Just as with threads, we can use multiple processes to divide an application into smaller tasks and run the tasks separately and concurrently.
- Advantage of using multiple processes instead of multiple threads:
  - The GIL (Global Interpreter Lock), which limits the running of only one thread at a time, applies only to threads within one process. It does not apply across processes.
  - This means processes can run concurrently on different processors of a system, which means we can truly have parallel processing.
- Disadvantage of using multiple processes:
  - Each process has its own memory space so processes can't share data. Any data passing needs more code coordination and more time.
  - Each process runs on a different processor so communication between processes can take more time.
- Python has a multiprocessing module that helps us create and work with processes.

```
import  multiprocessing  as  mp
```

- The multiprocessing module can tell us how many cores there are on the current system:

```
mp.cpu_count( )
```

Process

- The Process class has an API that is intentionally similar to the Thread class API so it's easy to switch between multithreading and multiprocessing.
- To create a child process:

```
p = mp.Process( target = a_function, args = (arg_tuple), name = aName )
```

where: a_function is the function that the process will run
arg_tuple is a tuple of input arguments for a_function
name is the optional name for the process

- To start the process: `p.start()`
- To wait for the process to end: `p.join()`
- Each process in the system is identified by a unique PID or Process ID. To see the PID of the current process we use a method of the os module:

```
os.getpid()
```

- We can also get the process name:

```
mp.current_process().name
```

Creating a Child Process

- Depending on whether the OS is Windows or Linux/MacOS, a child process is created in a slightly different way.
- On Linux/MacOS, the OS forks a child process, in which the current parent process is duplicated and the copy becomes the child process. The memory allocated to the parent process is also duplicated and assigned to the child process.
- On Windows, the OS re-runs the current python script up to the point where the child process is being requested, thus creating a new process that is a duplicate of the parent, and has a duplicate of the parent’s memory space.
- On both types of OS, after the child process is created, the parent and child processes both run the same code and start the with same data, but each accesses its own memory space.
- In a multicore or multiprocessor system, the OS typically determines which core or processor the child process will run on. It may be the same core / processor or it may be a different core / processor, depending on the load balancing algorithm of the OS and what other processes are currently running in the system.

© 2019 C. Nguyen

The main Function (1 of 2)

- The .py file that contains the main function or main block of code, which is run to drive the rest of the code, is called the top level module.
- On Windows, if the parent process contains the main function, then during the child process creation, when the Python script is re-run, it means another main will be run. This will result in an error.
- Therefore, to make our multiprocessing code platform independent, we need to check whether the module's name is '__main__' before running the main block of code.
- When Python runs a .py file, it sets the __name__ attribute of the module to '__main__'. This makes the .py file the top level module.
- When a child process is created as a duplicate of its parent, its __name__ is '__multiprocessing_main__'.
- Therefore, to prevent the error of having 2 main blocks running at the same time, we wrap the main block in an if statement:

```
if  __name__  ==  '__main__'  :
      # code for the main function in the true block
```

- This means the main block will run when the parent process runs at the start of the application, and it will not run when the child process is created.

The main Function (2 of 2)

- The same if statement can be conveniently used when we write unit testing code for a .py file.
- Example:
  - We write a moduleA.py file and at the end of the file, there is a main block of unit testing code for the functions / methods of module A.
  - We wrap this main block of code in the same if statement that checks for __name__ being '__main__'.
  - When we run the main block of code to test moduleA, the name of the .py file is '__main__' and everything runs as usual.
  - If moduleA.py is ever imported in another Python moduleB.py, and moduleB.py is run, then the __name__ moduleA.py will not become 'moduleA'. This means the main block of moduleA.py will automatically not run, the programmer doesn't have to comment it out or remove it.

© 2019 C. Nguyen

Communication Between Processes

- Because a parent and child processes can run on different processors, and processors communicate with binary data, this means data that are passed between processes cannot be Python data types. They must be binary data.
- If a Python processA needs to send a dictionary to a Python processB, then processA first pickles the dictionary and then sends the pickle object. At the receiving side, processB un-pickles the binary bytes back into a dictionary to use.
- In addition to using pickle, the multiprocessing module also has some special data types that we can use to send data between processes. These data types will convert between Python data types and binary byte strings for us.
- However, it is best to avoid sending data between processes. The conversion between Python data types and binary data for each pass is costly.
- Instead the processes should work independently of each other’s data and communicate with each other by checking status signals such as using Events.

© 2019 C. Nguyen

Checking Process Status

- Just like with threads, when a parent process waits for a child process with a a join, the join can block for a long time.
- We can set a timer for the join method so that after a certain amount of time, the join will run and stop the block, even if the child process is not finished.

```
p.join(2.0)     # timer is in seconds
```

- If using a timer, then before continuing with the next task, the parent process can check whether it becomes unblocked due to the timer timing out or due to the child process being done:

```
p.is_alive( )
```

Return: True if the process is not done, False if done

- We can also check the exit status of a child process when it's done:

```
p.exitcode
```

Exit code:
  - 0: process completed with no error
  - Positive value: process terminated due to error, and the error code is the exit code.
  - Negative value: process was killed or terminated with a terminate signal

© 2019 C. Nguyen

Event

- An Event object can be used for two processes to synchronize their tasks.
- Just like with threads, an Event object can be set or unset.
- One process can set the Event object when some condition happens, and the other process waits for the Event to change state to take some action.
- To create an Event object:

```
e = mp.Event()
```

- To change state of the Event object:

```
e.set()
```

or

```
e.clear()
```

- To check whether the Event is set:

```
e.is_set()
```

- To wait for the Event to be set:
  - Blocking wait until Event is set

```
e.wait()
```

  - Blocking wait with timer:

```
e.wait(2.5)
```

Lock

- The multiprocessing module also has a Lock class that is used 2 processes when they share same resource.
- To create a lock:

```
lock = mp.Lock()
```

- To request a lock of the shared resource:

```
lock.acquire()
```

  The acquire method also has a block=True argument and a timeout argument

- To release the lock when done with the resource:

```
lock.release()
```

- We can also use the "with" construct instead of lock.acquire and lock.release:

```
lock.acquire()
do some task
lock.release()
```

is the same as

```
with lock :
    do some task
```

© 2019 C. Nguyen

Queue

- When two processes are a producer – consumer pair, then we can use the multiprocessing queue as the data buffer between these 2 asynchronous processes.
- Note that this is a *multiprocessing* queue and is not the same as the *threading* queue.
- The producer puts data into the queue, independently from the consumer getting data out of the queue.
- To create a queue:

```
q = mp.Queue()
```

- The Queue object is a FIFO queue, and it has a built-in lock mechanism so that only one process can access one end of it at one time.
- To put data in the queue:

```
q.put(data)
```

- To get data from the queue:

```
data = q.get()
```

  and the queue status can be checked with:

```
q.empty()
```

- When a Python data type is put into the queue, the queue serializes it into a binary byte string. When data is retrieved from the queue, the queue converts it back into a Python data type.

Pool map Method (1 of 2)

- When we have a task that needs to be run multiple times, and each run is independent of the other runs (no exchange or sharing of data), then we can create a Pool of multiple processes.
- The Pool object accepts the task (a function) and the number of processes that we want to use. Then the Pool object distributes the number of runs of the task among the processes, coordinates them as they run, and stores all their results into a list.
- To create a pool of worker processes, where N is the number of workers:

```
pool = mp.Pool(processes = N)
```

- Use the Pool object's map method to divide the work among the workers

```
results = pool.map( a_function, arg_list )
```

  where:
  - a_function is the name of the task that needs to run multiple times. a_function can accept one input argument, which can be a container.
  - *each element* of arg_list is the argument(s) *for one run* of a_function. This means the length of arg_list controls the number of times that a_function will run.
  - results is a list of returned values from all the runs.

© 2019 C. Nguyen

Pool map Method (2 of 2)

- The map method will use each process in the pool to run a_function and pass to it one element of the arg_list.
- If there are more processes in the pool than there are number of runs, then some processes will not do any work.
  If there are fewer processes than the number of runs, then some processes will run multiple times.
- Once they start running, the processes in the pool run in parallel.
- The map method is blocking, which means that the main process will be blocked until the map call is done, which means when all the processes are done.
- This ensures that the order of resulting data will be the same as the order of the input arguments.
  For example, the first result is from the call to a_function with the first input argument(s), and the last result is from the call to a_function with the last input argument(s).

© 2019 C. Nguyen

Pool apply_async Method (1 of 2)

- If we have several tasks that we want to run in parallel, and each task is data independent (no sharing or exchanging data), then we can use the Pool apply_async method:

```
result = pool.apply_async(a_function, args=(tuple of args), callback=fct)
```

  where:
  - a_function is the name of the task that a process needs to run
  - args is a tuple of input arguments for a_function.
  - result is an ApplyResult object, and we can use:

    ```
    data = result.get()
    ```

    to retrieve the resulting data, or we can provide a callback function.
  - callback is the function that runs when a_function is done. It can be used instead of result.get to do something with the resulting data.
    The callback function can only accept one input argument, which is the returned result from a_function.
- Each apply_async method works with one task, so to have multiple tasks run in parallel, we call apply_async in a loop. In each iteration of the loop, apply_async will use an available process in the pool to run a_function, passing to it the tuple of arguments.

Pool apply_async Method (2 of 2)

- Each apply_async call is non-blocking, which means the main process can continue to run after the loop that starts all the apply_async methods.
- This also means that the ApplyResult objects that are returned from each task can be in a different order from the order that the tasks are run.
  For example, the third result that is received may be from the first task.
- To work around this problem, we need to add code in each task which will append to the result some unique way to identify the function.
- Since apply_async is non-blocking, the main process needs to wait for all the processes to be done before terminating. This is done in 2 steps:

`pool.close()`	Don't allow any more process to be added to the pool
`pool.join()`	Wait for all current processes in the pool to be done

Going further…

- There are other methods and data types in the multiprocessing module for further considerations.
- Both threads and processes can make the application run faster if we can separate the application into independent tasks to run concurrently.
- Cases to use multiprocessing:
  - Tasks that are CPU intensive will benefit from using of multiple cores / processors simultaneously. Some of these tasks include image and video processing, scientific / financial modeling, compiling.
    However, for Python, image / video processing and math modeling are done with C packages (such as numpy), which use their own parallel processing and don't require us to do any threading or multiprocessing. It's another instance where Python takes care of the heavy lifting for us.
  - Tasks that are time consuming *and* work independently from each other, so there is no waiting for data and not much data are exchanged.
- Cases to use multithreading:

  In almost all other circumstances related to IO, most notably making GUI applications responsive, threads are generally faster.

© 2019 C. Nguyen

Recap of Threads and Processes

- When an application first runs, it runs as a process and is a main thread.
- We can create multiple threads from one process.
  All threads will share the same memory space that's allocated to the process.
- Each process has its own memory space, it doesn't share the memory space with any other process.
- It may be possible to create a process from a thread, but no OS guarantees it and the OS documentation says the outcome is "undetermined."
- Finally, here's a write up that gives a gentle overview of threads and processes, to fill in any general gap from the hands-on class discussion and exercise.

Up Next: Network

© 2019 C. Nguyen