Communication Between Processes

As with threads, a common use pattern for multiple processes is to divide a job up among several workers to run in parallel. Effective use of multiple processes usually requires some communication between them, so that work can be divided and results can be aggregated.

Passing Messages to Processes

A simple way to communicate between process with `multiprocessing` is to use a `Queue` to pass messages back and forth. Any pickle-able object can pass through a `Queue`.

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessi
        print 'Doing something fa


def worker(q):
    obj = q.get()
    obj.do_something()


if __name__ == '__main__':
    queue = multiprocessing.Queue

    p = multiprocessing.Process(t
    p.start()

    queue.put(MyFancyClass('Fancy

    # Wait for the worker to fini
    queue.close()
    queue.join_thread()
    p.join()
```

This short example only passes a single message to a single worker, then the main process waits for the worker to finish.

```
$ python multiprocessing_queue.py

Doing something fancy in Process-
```

Page Contents

Navigation

Table of Contents
Previous: multiprocessing Basics
Next: Implementing MapReduce with multiprocessing

This Page

Show Source

Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

If you are looking for examples that work under Python 3, please refer to the PyMOTW-3 section of the site.

A more complex example shows how to manage several workers consuming data from a **`JoinableQueue`** and passing results back to the parent process. The *poison pill* technique is used to stop the workers. After setting up the real tasks, the main program adds one "stop" value per worker to the job queue. When a worker encounters the special value, it breaks out of its processing loop. The main process uses the task queue's **`join()`** method to wait for all of the tasks to finish before processin the results.

```
import multiprocessing
import time

class Consumer(multiprocessing.Pr

    def __init__(self, task_queue
        multiprocessing.Process._
        self.task_queue = task_qu
        self.result_queue = resul

    def run(self):
        proc_name = self.name
        while True:
            next_task = self.task
            if next_task is None:
                # Poison pill mea
                print '%s: Exitin
                self.task_queue.t
                break
            print '%s: %s' % (pro
            answer = next_task()
            self.task_queue.task_
            self.result_queue.put
        return


class Task(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __call__(self):
        time.sleep(0.1) # pretend
        return '%s * %s = %s' % (
    def __str__(self):
        return '%s * %s' % (self.


if __name__ == '__main__':
    # Establish communication que
    tasks = multiprocessing.Joina
    results = multiprocessing.Que

    # Start consumers
    num_consumers = multiprocessi
    print 'Creating %d consumers'
    consumers = [ Consumer(tasks,
                  for i in xrange
    for w in consumers:
        w.start()

    # Enqueue jobs
    num_jobs = 10
```

```
    for i in xrange(num_jobs):
        tasks.put(Task(i, i))

    # Add a poison pill for each
    for i in xrange(num_consumers
        tasks.put(None)

    # Wait for all of the tasks t
    tasks.join()

    # Start printing results
    while num_jobs:
        result = results.get()
        print 'Result:', result
        num_jobs -= 1
```

Although the jobs enter the queue in order, since their execution is parallelized there is no guarantee about the order they will be completed.

```
$ python -u multiprocessing_produ

Creating 16 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-5: 4 * 4
Consumer-6: 5 * 5
Consumer-7: 6 * 6
Consumer-8: 7 * 7
Consumer-9: 8 * 8
Consumer-10: 9 * 9
Consumer-11: Exiting
Consumer-12: Exiting
Consumer-13: Exiting
Consumer-14: Exiting
Consumer-15: Exiting
Consumer-16: Exiting
Consumer-1: Exiting
Consumer-4: Exiting
Consumer-5: Exiting
Consumer-6: Exiting
Consumer-2: Exiting
Consumer-3: Exiting
Consumer-9: Exiting
Consumer-7: Exiting
Consumer-8: Exiting
Consumer-10: Exiting
Result: 0 * 0 = 0
Result: 3 * 3 = 9
Result: 8 * 8 = 64
Result: 5 * 5 = 25
Result: 4 * 4 = 16
Result: 6 * 6 = 36
Result: 7 * 7 = 49
Result: 1 * 1 = 1
Result: 2 * 2 = 4
Result: 9 * 9 = 81
```

Signaling between Processes

The **`Event`** class provides a simple way to communicate state information between

processes. An event can be toggled between set and unset states. Users of the event object can wait for it to change from unset to set, using an optional timeout value.

```
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be s
    print 'wait_for_event: starti
    e.wait()
    print 'wait_for_event: e.is_s

def wait_for_event_timeout(e, t):
    """Wait t seconds and then ti
    print 'wait_for_event_timeout
    e.wait(t)
    print 'wait_for_event_timeout


if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(


    w1.start()

    w2 = multiprocessing.Process(


    w2.start()

    print 'main: waiting before c
    time.sleep(3)
    e.set()
    print 'main: event is set'
```

When **`wait()`** times out it returns without an error. The caller is responsible for checking the state of the event using **`is_set()`**.

```
$ python -u multiprocessing_event

main: waiting before calling Ever
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set(
main: event is set
wait_for_event: e.is_set()-> True
```

Controlling Access to Resources

In situations when a single resource needs to be shared between multiple processes, a **`Lock`** can be used to avoid conflicting accesses.

```
import multiprocessing
import sys

def worker_with(lock, stream):
```

```
    with lock:
        stream.write('Lock acquir

def worker_no_with(lock, stream):
    lock.acquire()
    try:
        stream.write('Lock acquir
    finally:
        lock.release()

lock = multiprocessing.Lock()
w = multiprocessing.Process(targe
nw = multiprocessing.Process(targ

w.start()
nw.start()

w.join()
nw.join()
```

In this example, the messages printed to the console may be jumbled together if the two processes do not synchronize their access of the output stream with the lock.

```
$ python multiprocessing_lock.py

Lock acquired via with
Lock acquired directly
```

Synchronizing Operations

`Condition` objects can be used to synchronize parts of a workflow so that some run in parallel but others run sequentially, even if they are in separate processes.

```
import multiprocessing
import time

def stage_1(cond):
    """perform first stage of wor
    name = multiprocessing.curren
    print 'Starting', name
    with cond:
        print '%s done and ready
        cond.notify_all()

def stage_2(cond):
    """wait for the condition tel
    name = multiprocessing.curren
    print 'Starting', name
    with cond:
        cond.wait()
        print '%s running' % name

if __name__ == '__main__':
    condition = multiprocessing.C
    s1 = multiprocessing.Process(
    s2_clients = [
        multiprocessing.Process(r
        for i in range(1, 3)
        ]
```

```
    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

In this example, two process run the second stage of a job in parallel, but only after the first stage is done.

```
$ python multiprocessing_conditio

Starting s1
s1 done and ready for stage 2
Starting stage_2[1]
stage_2[1] running
Starting stage_2[2]
stage_2[2] running
```

Controlling Concurrent Access to Resources

Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. A **`Semaphore`** is one way to manage those connections.

```
import random
import multiprocessing
import time

class ActivePool(object):
    def __init__(self):
        super(ActivePool, self)._
        self.mgr = multiprocessin
        self.active = self.mgr.li
        self.lock = multiprocessi
    def makeActive(self, name):
        with self.lock:
            self.active.append(na
    def makeInactive(self, name):
        with self.lock:
            self.active.remove(na
    def __str__(self):
        with self.lock:
            return str(self.activ

def worker(s, pool):
    name = multiprocessing.curren
    with s:
        pool.makeActive(name)
        print 'Now running: %s' %
        time.sleep(random.random(
```

```
            pool.makeInactive(name)

if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore
    jobs = [
        multiprocessing.Process(t
        for i in range(10)
        ]

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()
        print 'Now running: %s' %
```

In this example, the **`ActivePool`** class simply serves as a convenient way to track which processes are running at a given moment. A real resource pool would probably allocate a connection or some other value to the newly active process, and reclaim the value when the task is done. Here, the pool is just used to hold the names of the active processes to show that only three are running concurrently.

```
$ python multiprocessing_semaphor

Now running: ['0', '1', '2']
Now running: ['0', '1', '2']
Now running: ['0', '1', '2']
Now running: ['0', '1', '3']
Now running: ['4', '5', '6']
Now running: ['3', '4', '5']
Now running: ['1', '3', '4']
Now running: ['4', '7', '8']
Now running: ['4', '5', '7']
Now running: ['7', '8', '9']
Now running: ['1', '3', '4']
Now running: ['3', '4', '5']
Now running: ['3', '4', '5']
Now running: ['4', '5', '6']
Now running: ['7', '8', '9']
Now running: ['7', '8', '9']
Now running: ['7', '8', '9']
Now running: ['9']
Now running: ['9']
Now running: []
```

Managing Shared State

In the previous example, the list of active processes is maintained centrally in the **`ActivePool`** instance via a special type of list object created by a **`Manager`**. The **`Manager`** is responsible for coordinating shared information state between all of its users.

```
import multiprocessing

def worker(d, key, value):
    d[key] = value

if __name__ == '__main__':
    mgr = multiprocessing.Manager
    d = mgr.dict()
    jobs = [ multiprocessing.Proc
             for i in range(10)
             ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print 'Results:', d
```

By creating the list through the manager, it is shared and updates are seen in all processes. Dictionaries are also supported.

```
$ python multiprocessing_manager_

Results: {0: 0, 1: 2, 2: 4, 3: 6,
```

Shared Namespaces

In addition to dictionaries and lists, a **`Manager`** can create a shared **`Namespace`**.

```
import multiprocessing

def producer(ns, event):
    ns.value = 'This is the value
    event.set()

def consumer(ns, event):
    try:
        value = ns.value
    except Exception, err:
        print 'Before event, cons
    event.wait()
    print 'After event, consumer

if __name__ == '__main__':
    mgr = multiprocessing.Manager
    namespace = mgr.Namespace()
    event = multiprocessing.Event
    p = multiprocessing.Process(t
    c = multiprocessing.Process(t

    c.start()
    p.start()

    c.join()
    p.join()
```

Any named value added to the **`Namespace`** is visible to all of the clients that receive the **`Namespace`** instance.

```
$ python multiprocessing_namespac

Before event, consumer got: 'Name
After event, consumer got: This i
```

It is important to know that *updates* to the contents of mutable values in the namespace are *not* propagated automatically.

```
import multiprocessing

def producer(ns, event):
    ns.my_list.append('This is th
    event.set()

def consumer(ns, event):
    print 'Before event, consumer
    event.wait()
    print 'After event, consumer

if __name__ == '__main__':
    mgr = multiprocessing.Manager
    namespace = mgr.Namespace()
    namespace.my_list = []

    event = multiprocessing.Event
    p = multiprocessing.Process(t
    c = multiprocessing.Process(t

    c.start()
    p.start()

    c.join()
    p.join()
```

To update the list, attach it to the namespace object again.

```
$ python multiprocessing_namespac

Before event, consumer got: []
After event, consumer got: []
```

Process Pools

The **`Pool`** class can be used to manage a fixed number of workers for simple cases where the work to be done can be broken up and distributed between workers independently. The return values from the jobs are collected and returned as a list. The pool arguments include the number of processes and a function to run when starting the task process (invoked once per child).

```
import multiprocessing

def do_calculation(data):
    return data * 2
```

```
def start_process():
    print 'Starting', multiproces

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input   :', inputs

    builtin_outputs = map(do_calc
    print 'Built-in:', builtin_ou

    pool_size = multiprocessing.c
    pool = multiprocessing.Pool(p
                                i
                                )
    pool_outputs = pool.map(do_ca
    pool.close() # no more tasks
    pool.join()  # wrap up curren

    print 'Pool    :', pool_outpu
```

The result of the **`map()`** method is functionally equivalent to the built-in **`map()`**, except that individual tasks run in parallel. Since the pool is processing its inputs in parallel, **`close()`** and **`join()`** can be used to synchronize the main process with the task processes to ensure proper cleanup.

```
$ python multiprocessing_pool.py

Input   : [0, 1, 2, 3, 4, 5, 6, 7
Built-in: [0, 2, 4, 6, 8, 10, 12,
Starting PoolWorker-11
Starting PoolWorker-12
Starting PoolWorker-13
Starting PoolWorker-14
Starting PoolWorker-15
Starting PoolWorker-16
Starting PoolWorker-1
Starting PoolWorker-2
Starting PoolWorker-3
Starting PoolWorker-4
Starting PoolWorker-5
Starting PoolWorker-8
Starting PoolWorker-9
Starting PoolWorker-6
Starting PoolWorker-10
Starting PoolWorker-7
Pool    : [0, 2, 4, 6, 8, 10, 12,
```

By default **`Pool`** creates a fixed number of worker processes and passes jobs to them until there are no more jobs. Setting the *maxtasksperchild* parameter tells the pool to restart a worker process after it has finished a few tasks. This can be used to avoid having long-running workers consume ever more system resources.

```
import multiprocessing

def do_calculation(data):
```

```
    return data * 2

def start_process():
    print 'Starting', multiproces

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input   :', inputs

    builtin_outputs = map(do_calc
    print 'Built-in:', builtin_ou

    pool_size = multiprocessing.c
    pool = multiprocessing.Pool(p
                                i
                                m
                                )
    pool_outputs = pool.map(do_ca
    pool.close() # no more tasks
    pool.join()  # wrap up curren

    print 'Pool    :', pool_outpu
```

The pool restarts the workers when they have completed their allotted tasks, even if there is no more work. In this output, eight workers are created, even though there are only 10 tasks, and each worker can complete two of them at a time.

```
$ python multiprocessing_pool_max

Input   : [0, 1, 2, 3, 4, 5, 6, 7
Built-in: [0, 2, 4, 6, 8, 10, 12,
Starting PoolWorker-11
Starting PoolWorker-12
Starting PoolWorker-13
Starting PoolWorker-14
Starting PoolWorker-15
Starting PoolWorker-16
Starting PoolWorker-1
Starting PoolWorker-2
Starting PoolWorker-3
Starting PoolWorker-4
Starting PoolWorker-5
Starting PoolWorker-6
Starting PoolWorker-7
Starting PoolWorker-8
Starting PoolWorker-9
Starting PoolWorker-10
Pool    : [0, 2, 4, 6, 8, 10, 12,
```