Things I Wish They Told Me About Multiprocessing in Python

By: Pamela McA'Nulty 27 February, 2019 in development

Framing the problem

“Some people, when confronted with a problem, think ‘I know, I’ll use multithreading’. Nothhw tpe yawrve o oblems.” (Eiríkr Åsheim, 2012)

If multithreading is so problematic, though, how do we take advantage of systems with 8, 16, 32, and even *thousands,* of separate CPUs? When presented with large Data Science and HPC data sets, how to you use all of that lovely CPU power without getting in your own way? How do you tightly coordinate the use of resources and processing power needed by servers, monitors, and Internet of

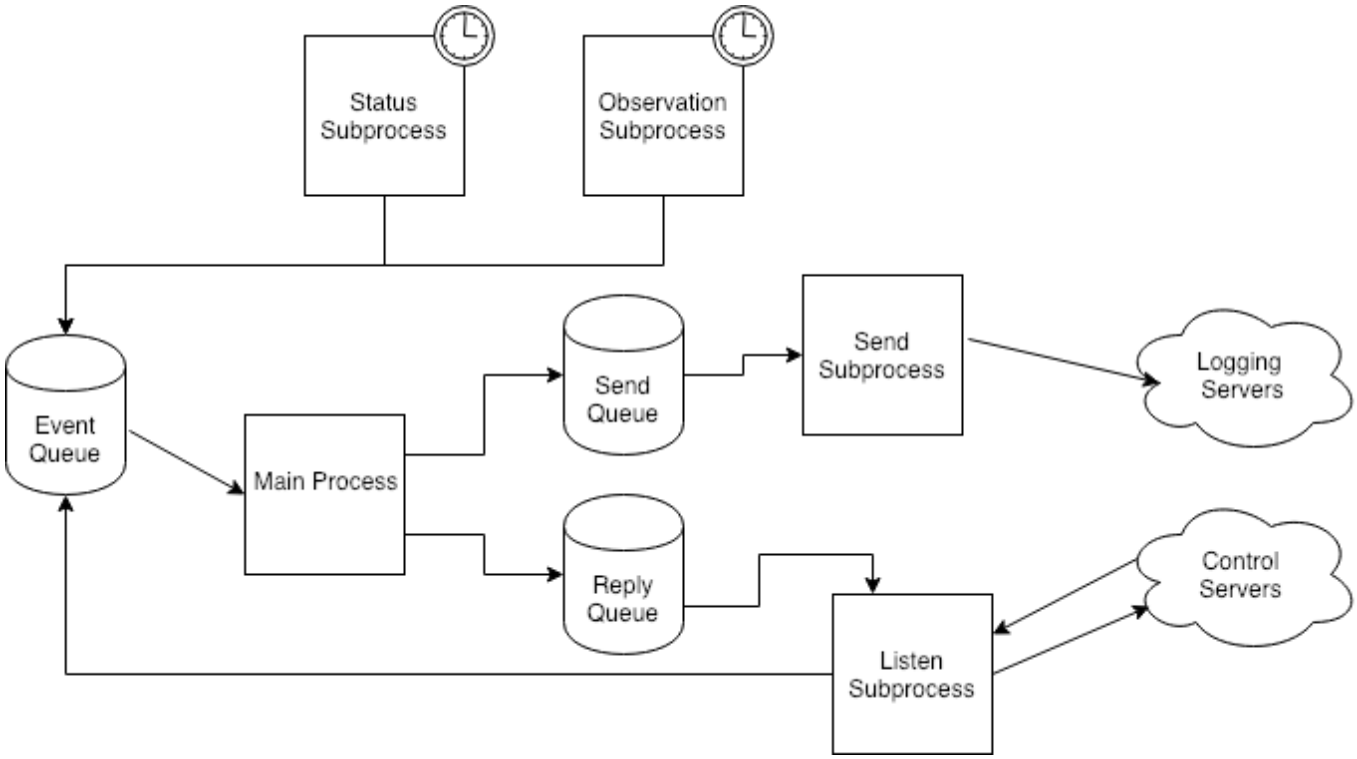
Things applications - where there can be a lot of waiting for I/O, many distinct but interrelated operations, and non-sharable resources - and you *still* have to crank through the preprocessing of data before you send it somewhere?

An excellent solution is to use multiprocessing, rather than multithreading, where work is split across separate processes, allowing the operating system to manage access to shared resources. This also gets around one of the notorious Achilles Heels in Python: the Global Interpreter Lock (aka theGIL). This lock constrains all Python code to run on only one processor at a time so that Python multi-*threaded* applications are largely only useful when there is a lot of waiting for IO. If your application is I/O bound and doesn't require large blocks of CPU time, then, as of Python version 3.4, the *asyncio* system is the preferred approach.

Python ships with the multiprocessing module which provides a number of useful functions and classes to manage subprocesses and the communications between them. One interface the module provides is the `Pool` and `map()` workflow, allowing one to take a large set of data that can be broken into chunks that are then mapped to a single function. This is extremely simple and efficient for doing conversion, analysis, or other situations where the same operation will be applied to each data. But `Pool` and `map()` aren't suited to situations that need to maintain state over time or, especially, situations where there are two or more different operations that need to run and interact with each other in some way. For these kinds of problems, one needs to make use of somewhat more complex features of the multiprocessing module, such as `Process`, `Queue` and `Event`. Using these features introduces a number of complicated issues that now need to be managed, especially with regard to cleanly starting and stopping subprocesses, as well as coordinating between them.

Adding context

Since Python multiprocessing is best for complex problems, we'll discuss these tips using a sketched out example that emulates an IoT monitoring device. This example is based on an implementation of an HVAC system that I worked on in 2018.



The application consists of a "Main Process" - which manages initialization, shutdown and event loop handling - and four subprocesses.

- "Observation Subprocess" runs every 10 seconds and collects data about the HVAC systems being monitored, queuing an "Observation" event message to the Event Queue. In this example, it is assumed that "Observation Subprocess" is CPU intensive as it performs a significant amount of data processing on the observation before sending it to the server.
- "Status Subprocess" also runs every 10 seconds, and collects data about the IoT system, including such things as uptime, network settings and status, and storage usage. It also queues a "Status" event message to the Event Queue.
- "Send Subprocess" accepts "Send" event messages from the Send Queue, and then sends those messages over the network to a central server that will store, analyze, present, and automatically act upon the data it receives,
- "Listen Subprocess" listens on a network port for incoming requests from the central command servers, queues up a "Command" event message to the Event Queue, and then waits for a "Reply" message from the Main Process. Such commands are largely focused on managing the HVAC system and obtaining

detailed IoT system operating data. Note that, in the real world, this interface is much more complex, and relies upon a security layer.

- "Main Process", besides managing the startup and shutdown of the entire system, also manages the Event Queue, routing messages to the proper subprocess: "Status" and "Observation" messages result in a "Send" message to the Send Queue, and "Command" events are handled and then a "Reply" message is queued to the Reply Queue.

Note on code examples: These examples use the mptools{:} Python library that I developed while writing this blog post. This is the only example where the library interface is directly referenced.

```
def main():
    with MainContext() as main_ctx:
        init_signals(main_ctx.shutdown_event, default_signal_handler

        send_q = main_ctx.MPQueue()
        reply_q = main_ctx.MPQueue()

        main_ctx.Proc("SEND", SendWorker, send_q)
        main_ctx.Proc("LISTEN", ListenWorker, reply_q)
        main_ctx.Proc("STATUS", StatusWorker)
        main_ctx.Proc("OBSERVATION", ObservationWorker)

        while not main_ctx.shutdown_event.is_set():
            event = main_ctx.event_queue.safe_get()
            if not event:
                continue
            elif event.msg_type == "STATUS":
                send_q.put(event)
            elif event.msg_type == "OBSERVATION":
                send_q.put(event)
            elif event.msg_type == "ERROR":
                send_q.put(event)
```

```
        elif event.msg_type == "REQUEST":
            request_handler(event, reply_q, main_ctx)
        elif event.msg_type == "FATAL":
            main_ctx.log(logging.INFO, f"Fatal Event received: {e
            break
        elif event.msg_type == "END":
            main_ctx.log(logging.INFO, f"Shutdown Event received:
            break
    else:
        main_ctx.log(logging.ERROR, f"Unknown Event: {event}"
```

Tip #1

Don’t Share Resources, Pass Messages

At first thought, it might seem like a good idea to have some sort of shared data structures that would be protected by locks. When there is only *one* shared structure, you can easily run into issues with blocking and contention. As such structures proliferate, however, the complexity and unexpected interactions multiply, potentially leading to deadlocks, and very likely leading to code that is difficult to maintain and test. The better option is to pass **messages** using `multiprocessing.Queue` objects. Queues should be used to pass all data between subprocesses. This leads to designs that “chunkify” the data into messages to be passed and handled, so that subprocesses can be more isolated and functional/task oriented. The Python `Queue` class is implemented on unix-like systems as a PIPE - where data that gets sent to the queue is serialized using the Python standard library `pickle` module. Queues are usually initialized by the main process and passed to the subprocess as part of their initialization.

```
event_q = multiprocessing.Queue()
send_q = multiprocessing.Queue()
# ...
event_q.put(“FOO”)
```

```
# … in another subprocess ...
event = event_q.get(block=True, timeout=timeout)

# …
queue.close()
queue.join_thread()
```

Tip #2

Always clean up after yourself

Subprocesses can hang or fail to shutdown cleanly, potentially leaving some system resources unavailable, and, potentially worse, leaving some messages un-processed. For this reason, a significant percentage of one's code needs to be devoted to cleanly stopping subprocesses.

The first part of this problem is telling subprocesses to stop. Since we're using Queues and messages, the first, and most common, case is to use "END" messages. When it's time to stop the application, one queues up "END" messages to each queue in the system, equal to the number of subprocesses *reading* from that Queue. Each of the subprocess should be looping on messages from the queue, and once it receives an "END" message, it will break out of the loop and cleanly shut itself down.

"END" messages are very useful, but they aren't the only method available, and they don't help if a subprocess isn't reading from a queue. I recommend using a `multiprocessing.Event` object. An `Event` object is a `True`/`False` flag, initialized as `False`, that can be safely set to `True` in a multiprocess environment while other processes can check it with `is-set()` and wait on for it to change to `True`. Best practice is to have only one "shutdown-requested" Event object in an application which is passed to all subprocesses. Subprocesses should then loop using that `Event` as their boolean check - so that if the `shutdown-requested Event` is set to `True`, the loop terminated.

```
while not shutdown_event.is_set():
    try:
        item = work_queue.get(block=True, timeout=timeout)
    except queue.Empty:
        continue

    if item == "END":
        break
    # …
```

Once a subprocess needs to end - be it via “END” message, **`shutdown_event`** Event flag, or an exception of some sort - it is the subprocess’s ***duty*** to clean up after itself by releasing any resources it owns. Python does a pretty great job of cleaning things up during garbage collection, but some resources need to be closed cleanly (pipes, files), and some will hang for some unknown timeout period, thus preventing a clean exit of the process. Network resources can not only tie up local resources, they can also tie up resources on the remote server systems while they wait for timeouts. So final cleanup is vital.

```
def startup(self):
    # -- Called during worker process start up sequence
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1
    self.socket.bind(('127.0.0.1', 9999))
    self.socket.settimeout(self.SOCKET_TIMEOUT)
    self.socket.listen(1)

def shutdown(self):
    # -- Called when worker process is shutting down
    self.socket.close()
```

Not only do subprocesses need to clean up after themselves, the main process *also* needs to clean up the subprocesses, **`Queue`** objects, and other resources that it might have control over. Cleaning up the subprocesses involves making sure that

each subprocess gets whatever termination messaging that it might need, *and* that the subprocesses are actually terminated. Otherwise, stopping the main process could result in either a hang, or an orphaned zombie subprocess. The normal procedure involves setting the shutdown flag, waiting for all the processes to stop normally within some reasonable amount of time, and then terminating any that haven't stopped yet.

```
def stop_procs(self):
    self.shutdown_event.set()

    end_time = time.time() + self.STOP_WAIT_SECS
    num_terminated = 0
    num_failed = 0

    # -- Wait up to STOP_WAIT_SECS for all processes to complete
    for proc in self.procs:
        join_secs = max(0.0, min(end_time - time.time(), STOP_WAIT_S
        proc.proc.join(join_secs)

    # -- Clear the procs list and _terminate_ any procs that
    # have not yet exited
    while self.procs:
        proc = self.procs.pop()
        if proc.proc.is_alive():
            proc.terminate()
            num_terminated += 1
        else:
            exitcode = proc.proc.exitcode
            if exitcode:
                num_failed += 1

    return num_failed, num_terminated
```

Python's Queue objects also need a bit of special handling to be fully cleaned up: they need to be drained of any items that have been left there (and those items may or may not need to be dealt with somehow - perhaps by saving them to disk), closed, and, importantly, have `Queue.join_thread()` called so that the associated monitoring thread gets cleaned up and doesn't generate a misleading exception message during final Python garbage collection.

```
item = work_queue.get(block=False)
while item:
    try:
        work_queue.get(block=False)
    except Empty:
        break
work_queue.close()
work_queue.join_thread()
```

Tip #3

Always handle TERM and INT signals

We discussed how to be sure to end subprocesses, but how does one determine *when* to end them? Applications will often have a way to determine that they have nothing left to process, but server processes usually receive a TERM signal to inform them that it's time to stop. Also, especially during testing, one often finds oneself using the INT signal (aka `KeyboardInterrupt`) to stop a runaway test. More often than not, one desires the same behavior from TERM and INT signals, though INT might also always want to generate a stack trace so the user can see more precisely what they interrupted.

The approach I recommend is to have signal handlers set the `shutdown_event` Event flag the first two times they get called, and then raise an exception each time they're called thereafter. This allows one to hit control-C twice and be sure to stop code that's been hung up waiting or in a loop, while allowing "normal" shutdown processing to properly clean up. The example below uses a common signal handler function, using functools.partial to create the two functions, differing only in which

exception they will raise, that get passed as the signal handlers. An important detail is that signals need to be set up separately for *each* subprocess. Linux/Unix systems automatically propagate signals to all child processes, so those subprocesses also need to capture and handle the signals as well. An advantage to this is that subprocess signal handlers can potentially operate on resources specific to that subprocess. For example, I have written a signal handler that changed a ZeroMQ blocking socket into a non-blocking one. This allowed me to write code for a `get()` call that didn't have a timeout, and didn't really need one.

```
class SignalObject:
    MAX_TERMINATE_CALLED = 3

    def __init__(self, shutdown_event):
        self.terminate_called = 0
        self.shutdown_event = shutdown_event

def default_signal_handler(
        signal_object,
        exception_class,
        signal_num,
        current_stack_frame):
    signal_object.terminate_called += 1
    signal_object.shutdown_event.set()
    if signal_object.terminate_called = signal_object.MAX_TERMINATE_(
        raise exception_class()

def init_signal(signal_num, signal_object, exception_class, handler)
    handler = functools.partial(handler, signal_object, exception_cl
    signal.signal(signal_num, handler)
    signal.siginterrupt(signal_num, False)

def init_signals(shutdown_event, int_handler, term_handler):
    signal_object = SignalObject(shutdown_event)
    init_signal(signal.SIGINT, signal_object, KeyboardInterrupt, int_
```

```
    init_signal(signal.SIGTERM, signal_object, TerminateInterrupt, te
    return signal_object
```

Tip #4

Don't Wait Forever

Proper shutdown behavior requires that every process in the system be resilient against getting "stuck". This means that not only will loops have terminating conditions, but that other system calls that could block and wait will need to use timeouts if at all possible: `Queue` objects allow for timeouts when doing both `put()` and `get()` calls, sockets can be configured to time out, etc.. This ends up being a form of polling where there are 2 events being checked: the system call, and the termination event (i.e. `shutdown_event` is `True`). You'll need to determine how long you can afford to wait on the system call before checking if the loop needs to terminate. The goal is to check for termination frequently enough that the system will respond promptly to a termination/shutdown request, while spending most of the process's time waiting on the resource (queue, event, socket, etc) It's important to not wait very long because for server processes started with systemd, systemd will eventually (90 seconds by default) decide that your application isn't stopping and send SIGKILL signal, and you no longer have a chance to clean up.

Here are some examples of waiting:

Polling against queues: `get()` from the `Queue` with block set to `True` and a short timeout. If `queue.Empty` is raised, go back to the top of the loop and try again, otherwise, process the returned item.

```
while not shutdown_event.is_set():
    try:
        item = work_queue.get(block=True, timeout=0.05)
    except queue.Empty:
        continue
    # ...
```

Poll against sockets: Call `settimeout()` on the socket with a short timeout. If the `socket.timeout` is raised, go back to the top of the loop, check for `shutdown_event`, and try again, otherwise, process handle the accepted client connection (which will *also* need to have `settimeout()` called on it, so *its* operations don't hang)

```
self.socket.settimeout(0.1)
self.socket.listen(1)
# …
while not shutdown_event.is_set():
    try:
        clientsocket, address = self.socket.accept()
    except socket.timeout:
        continue
```

Poll while waiting for a timer: Loop as long as `shutdown_event` is not set, firing a timer every `INTERVAL_SECS` seconds. Each pass through the loop sleeps for the time remaining until `next_time`, up to the max of `MAX_SLEEP_SECS` (0.02) seconds (which, of course, means that it *usually* sleeps 0.02 seconds). If the code comes out of the `sleep()` before `next_time`, go back to the top of the loop and try again, otherwise do something (in this case, put "TIMER EVENT" on the `event_queue`), and re-calculate the time for the *next* timer event.

```
class TimerProcWorker(ProcWorker):
    INTERVAL_SECS = 10
    MAX_SLEEP_SECS = 0.02

    def main_loop(self):
        next_time = time.time() + self.INTERVAL_SECS
        while not self.shutdown_event.is_set():
            sleep_secs = max(0.0, min(next_time - time.time(), self.I
            time.sleep(sleep_secs)
            if time.time() = next_time:
                event_queue.put("TIMER EVENT")
                next_time = time.time() + self.INTERVAL_SECS
```

Tip # 5

Report exceptions, and have a time based, shared logging system.

Logging in an application is vitally important, and even more so in a multiprocessing app, where a combined log shines at reporting events in time-based order. Happily, Python provides good logging facilities. Sadly, Python doesn't really provide a great way to sync subprocess log messages. Because there are so many moving parts, each log message needs 2 key pieces of data: which process is generating the log message, and how long it's been since the application started. I generally *name* my processes. If there are multiple copies of the same process, then they'll take the name "WORKER-1", "WORKER-2", etc.

Besides logging, each subprocess can send Error and Shutdown messages to the main event queue. Allowing the event handler to recognize and deal with unexpected events, such as retrying failed sends, or starting a new subprocess after one has failed.

Below is the log output from a sample run. Note how the timing being based on application start time provides a clearer picture of what's going on during the all important startup process.

```
$ python multiproc_example.py

 DEBUG:root:  0:00.008 SEND Worker          Proc.__init__ starting : S
 DEBUG:root:  0:00.013 SEND Worker          Entering QueueProcWorker.r
 DEBUG:root:  0:00.013 SEND Worker          Entering init_signals
 DEBUG:root:  0:00.014 SEND Worker          Entering QueueProcWorker.m
 DEBUG:root:  0:00.014 SEND Worker          Proc.__init__ starting : S
 DEBUG:root:  0:00.015 LISTEN Worker        Proc.__init__ starting : L
 DEBUG:root:  0:00.019 LISTEN Worker        Entering init_signals
 DEBUG:root:  0:00.022 LISTEN Worker        Entering main_loop
 DEBUG:root:  0:00.022 LISTEN Worker        Proc.__init__ starting : L
```

```
DEBUG:root:  0:00.024 STATUS Worker         Proc.__init__ starting : S
DEBUG:root:  0:00.029 STATUS Worker         Entering init_signals
DEBUG:root:  0:00.033 STATUS Worker         Entering TimerProcWorker.r
DEBUG:root:  0:00.033 STATUS Worker         Proc.__init__ starting : S
DEBUG:root:  0:00.035 OBSERVATION Worker    Proc.__init__ starting : O
DEBUG:root:  0:00.039 OBSERVATION Worker    Entering init_signals
DEBUG:root:  0:00.040 OBSERVATION Worker    Entering startup
DEBUG:root:  0:00.042 OBSERVATION Worker    Entering TimerProcWorker.r
DEBUG:root:  0:00.043 OBSERVATION Worker    Proc.__init__ starting : O
^C (ed: user hit Control-C)
INFO:root:  0:03.128 OBSERVATION Worker    Normal Shutdown
INFO:root:  0:03.128 STATUS Worker         Normal Shutdown
DEBUG:root:  0:03.130 OBSERVATION Worker    Entering shutdown
DEBUG:root:  0:03.130 STATUS Worker         Entering shutdown
ERROR:root:  0:03.131 MAIN                  Unknown Event: OBSERVATIOI
DEBUG:root:  0:03.132 SEND Worker           QueueProcWorker.main_loop
INFO:root:  0:03.133 SEND Worker           Normal Shutdown
INFO:root:  0:04.025 LISTEN Worker         Normal Shutdown
DEBUG:root:  0:04.028 MAIN                  Process OBSERVATION ended
DEBUG:root:  0:04.028 MAIN                  Process STATUS ended with
DEBUG:root:  0:04.028 MAIN                  Process LISTEN ended with
DEBUG:root:  0:04.028 MAIN                  Process SEND ended with ex
```

Conclusion

Python's `mutliprocessing` module allows you to take advantage of the CPU power available on modern systems, but writing and maintaining *robust* multiprocessing apps requires avoiding certain patterns that can lead to unexpected difficulties, while also spending a fair amount of time and energy focusing on details that aren't the primary focus of the application. For these reasons, deciding to use multiprocessing in an application is not something to take on lightly, but when you do, these tips will make your work go more smoothly, and will allow you to focus on your core problems.

Other resources:

"Programming guidelines{:}" section from the Python docs.

My work-in-progress mptools library {:}

Photo by Patrick Tomasso{:} on Unsplash{:}.

Five Multiprocessing Python Tips from PyCon 2019

Pamela McA'Nulty
May 14 2019

Parsing logs 230x faster with Rust

André Arko
November 8 2018

Caching and The Web: One Site's Optimal Pattern is Another App's Slough

Aria Stewart
June 13 2017

Extreme Programming Explained —Bridge the Business-Technology Divide

Chuck Fitzpatrick
October 30 2013

We're here & ready to listen.

Talk to us today about your lean design, user experience, full-stack development, and product delivery needs.

415.796.6434

152 2nd st
SF, CA 94105

results@cloudcity.io

twitter follow us

facebook follow us

linkedin join us

Contact us