Carl Haddick
PO Box 1586
Mexia TX 76667
carl@carlhaddick.com

October 30, 2020

Coolthreads

This module defines two classes that greatly simplify running Python functions in parallel via threading. Note that the threading used here is Python's in-interpreter threading and is subject to limitations such as the Global Interpreter Lock. It does not always result in faster overall throughput to run code in parallel at this level, but it often does.

For instance, where real-world delays are involved this level of threading is easy to use, easy to debug, and will result in significant time savings.

In one case, a Perl script was used to search authoritative name servers for zones starting with the root name servers. The script was not particularly fast and had a habit of hanging on some zones. For the full inventory of zones under investigation, runtime was around 26 hours with constant vigilance required to restart it with a trimmed-down inventory every time it hung.

A rerun of the script was needed and the 26 hour delay was going to prove difficult. The zones that hung and silently paused the process were going to prove fatal to a deadline.

While the unthreaded job was running, I wrote a threaded solution, handing the problematic Perl script one zone at a time. Since there was a time limit of 20 seconds on each instance of the Perl script, if it hung on one zone, all others remained viable. One zone would not report information and the logs generated by the threading code would alert the situation.

With 100 instances constantly running, the job took about 25 minutes. Writing the threaded code took about 30 minutes to write, test, and adjust. Less than one hour - including coding and testing! - accomplished what would have taken 26 hours.

Threading is spectacular when a function has to run a large number of times with real-world delays.

Theory of operation

One thread runs a single instance of a CoolThreads object. This background object handles metered dispatching, thread population monitoring, logging, and control. Worker threads are instances of class CoolThreadOp. Each instance of CoolThreadOp runs in a separate thread.

Python's main thread is independent of CoolThreads.

This class has some internal complexity, which probably explains taking about one full day to write and debug, and another day to test and document.

From the outside, though, it's not complex at all. Don't let the internal gears and wheels intimidate.

Basically, you say, "I want a thread dispatcher." That's one line of code.

Then, you say, "Here, run this function as a thread." That's another line of code.

Then, you clean up with the request, "Let the dust settle and close up shop." That's a final line of code.

To run threads with collated logging, thread-safety as needed, and exception logging, it's two lines of code plus whatever it takes to identify what runs as threads. I've seen remote query of 800+ servers threaded up in less than ten lines of code, and that was with a threading wrapper much less refined than this one.

Almost every line of source has a comment, and the external topology is simple. If you need threading, this is a solid solution.

CoolThreads

This is the dispatcher object. It does its work in a thread, so it remains responsive to worker thread termination and logging requests. All members are named with a prefix and capitalization in the form CTabcXyz. This makes it easy to avoid name collisions in keyword arguments or subclassing. There is one exception to that naming convention. Python's threading.Thread requires a method called "run," and that requirement is respected.

CoolThreads inherits threading.Thread and signals.SigIntercept.

CoolThreads.__init__

Arguments to the CoolThreads constructor are:

- thrdName - a name to identify this CoolThreads instance. Worker threads have their own names.
- logFile - a file object for logging. Defaults to None.
- threadMax - defaults to 10. This is the maximum number of active threads the dispatcher will allow.
- collate - a boolean, defaulting to True. If set, logging to logFile is collated by thread with in-memory buffering. Each thread's log output is in chronological order. The threads are grouped in the log file in the order they terminated.
- queueSleep - defaults to 0. Can be set higher to save useless loop iterations in the dispatcher. If nothing is found in the command and log queue, queueSleep sets a number of seconds to wait before trying again. This uses the time.sleep function, so fractional seconds are supported.
- consoleLog - a boolean defaulting to True. If set, logging to logFile is duplicated on the console with threadsafe output.
- **kwargs - additional arguments passed to user-supplied functions. More below in the description of individual members. Not used by the CoolThreads dispatcher. Kwargs is included for use by subclasses. Normally, subclassing CoolThreads is not needed.

CoolThreads attributes of interest:

- CTlogFile - the file object log entries are written to.
- CTthreadMax - maximum allowed threads.
- CTcollate - set True if logging collated by thread is required.
- CTqueueSleep - delay after detecting an empty command and log queue.
- CTconsoleLog - set to True to echo log activity to console.

- kwargs - not used. The **kwargs parameters from __init__'s argument list. Included for subclassing needs.

- CTthreadLimit - a bounded semaphore used to limit how many threads can run at any one time.

- CTnoThread - a reentrant lock object used to provide execution windows for non thread-safe functions. Use of a reentrant lock allows a thread to lock multiple things (including itself, see CoolThreadOp).

- CTlogQueue - a command and log queue. More detail is provided below.

- CTjobLog - a dictionary of log entry lists. Used for deferred and collated logging.

- CTopenForBiz - set True if the dispatcher is in a state to accept new threads. Note that when a CoolThreads instance has been shut down, it should not be restarted. This is a limitation of the way Python's threading.Thread class is implemented.

- CTterminated - set True when all threads and deferred logging is complete.

Note that the CoolThreads constructor takes are of calling the inherited Thread.start function to begin execution of the dispatcher loop.

CoolThreads.CTaddThread

Adds a new thread to the execution queue. Returns True if thread was added, False otherwise. The only condition that will block adding a thread is if CTopenForBiz is False.

Exceptions are trapped and logged.

Arguments are a thread id and a class or subclass of CoolThreadOp to execute in a thread.

It's not a bad idea to wrap CTaddThread with a try-except block. There's no problem with untrapped exceptions inside CTaddThread, but there might be with some methods of calling CTaddThread.

For example, you might have a dictionary of CoolThreadOp classes to run in threads. That could be done in this code snippet:

```
runQ.CTaddThread(opKey, opInventory[opKey](opKey, **opParams[opKey]))
```

Here's the assumptions to understand that line of code:

- runQ - an instance of CoolThreads
- opKey - a dictionary key derived elsewhere in the code
- opInventory - a dictionary of CoolThreadOp subclasses
- opParams - a dictionary of dictionaries, used as **kwargs

All very cool, except what happens if opKey isn't in one of those dictionaries?

That will cause a key value exception. Without catching that, your main thread will terminate. All your threads will run happily to their normal terminations. Your runQ dispatcher will wait for new threads that will never be added, or a call to CTwaitForComplete that will never come.

Your CoolThreads based application will crash. Not optimal.

When I'm handing pretty guaranteed data to CTaddThread, I don't wrap it in a try-except stanza but I remain aware of what happens if there's a problem in my CTaddThread parameter list. The Ghostbusters streams cross, the known universe concludes existence, yada, yada, yada.

Consider trapping exceptions. I will try to listen to my own advice in the future.

CoolThreads.CTlogger

This function sends log messages to their appropriate destinations (CTjobLog, console, logFile), and queues threading commands. This function is where dispatcher commands originate.

Log output is formatted with a prefix of "YYYY-MM-DD HH:MM:SS threadId:" followed by the log message itself.

A string to log is the only required argument. If present, a second argument is a threadID to credit the log entry to, and a third argument is a logging directive bit map

- 0x100 - log to console.
- 0x200 - log immediately to log file.

- 0x400 - deferred logging to CTjobLog dictionary.

CoolThreads.CTthreadSafe

This provides protection for anything not thread safe. Note that other threads remain in execution. A cooperative lock is used as protection against collisions.

Two arguments are required, the thread id making the call and a reference to the target function to run. The thread id is used for logging in the case of untrapped exceptions.

Additionally, *args and **kwargs are collected and passed to the target function. The target function's return value is returned unless an untrapped exception is encountered. In that case, the thread will be terminated, the exception logged, and other threads will not be affected.

If you need to reference CoolThreads resources from within a protected call, it's suggested to add kwargs['CT'] = self.CT to what is passed to CTthreadSafe.

Beware use of CTlogger from within a CTthreadSafe call, because CTlogger uses the thread lock to serialize console output.

CoolThreads.CTprintConsole

Thread safe print function. The thread protection is cooperative, using a lock.

Four arguments are supported. Only the first two, a thread id and a message to print, are required.

- threadId - the id of the running thread, used for logging.
- outstr - the string to print.
- getInput - defaults to False. If True, the user will be prompted to enter something. Note that if other threads make unprotected print calls, output can be scrambled.
- rePrompt - defaults to None. prompt used if user's first attempt isn't an expected response. If a second prompt is needed and rePrompt is set to None, the original outstr will be reprinted.
- allowedInput - defaults to None. Can be set to a list of allowed input values. . Note that this should be a list instead of a string.

If user input was collected, it's returned from CTprintConsole and logged. Otherwise, None is returned without triggering a log entry.

CoolThreads.CTactiveCount

This function returns the number of threads running. If you want to let all current threads complete, you could use:

```
while CT.CTactiveCount(): time.sleep(1)
```

Note that this does not terminate threading. After this line runs, you can add additional threads with CTaddThread.

The count returned by CTactiveCount is for the number of threads running under the current dispatcher. Other threads running for other CoolThreads instances, a GUI, or any other purpose are not included in the count.

CoolThreads.CTwaitForComplete

CTwaitForComplete calls CTstop and waits for CTterminated to become True. This is the most common way to shut down threading.

CoolThreads.CTdumpThreadLog

This function dumps the CTjobLog for a single thread to logFile for collated logging. Note that since this runs in the dispatcher thread, logging is inherently thread-safe from a worker thread perspective.

The exception to that would be if a single log file were used simultaneously by multiple CoolThread instances. Recommendation - use separate log files for multiple CoolThreads instances, or subclass CoolThreads for log file lock exclusion.

One argument is passed, the thread ID to dump.

CoolThreads.CTdumpLog

Dump all threads in CTjobLog, using CTdumpThreadLog for each job found.

CoolThreads.CTlogQueue

This is a queue structure of unlimited size (within the limit of available memory) used to pass log entries from worker threads to the CoolThreads dispatcher for serial processing.

Each entry in the queue is a three member tuple, (command, thread id, log message). The first is a bit mapped value passing a log directive and a dispatch command. The lower byte of the first log tuple element contains these flags for dispatcher behavior:

- bit 0 - CTcontinueThreading - this bit is actually not checked in the default CoolThreads run loop. It is included for future subclassing needs. If CTcontinueThreading is set, it means to process the attached log entry
- bit 1 - CTjobStop - this bit indicates the job identified in the second element in the log tuple has concluded operations, either naturally or through untrapped exception.
- bit 2 - CTthreadingStop - request dispatcher shutdown. This is a one way process. The dispatcher should not be restarted once its thread has been terminated.

The next byte (masked by 0xff00) carries three flags:

- bit 8 - CTconsoleMsg - send message to console (with lock for thread safety).
- bit 9 - CTimmedateLog - send message to logFile immediately for chronological logging.
- bit 10 - CTdeferedLog - send message to CTjobLog.

The second element in the log/command tuple is the string used to identify the thread originating a log message.

The third element is the log message itself. Log messages are guaranteed to be written with a trailing newline. If a log message ends with a newline, it will be written that way. Multiple newlines are permitted.

A log line without a newline will have one added.

Please note that console output is done via the CTnoThread lock. Unresolvable blocking will occur if a worker thread runs a function under CTthreadSafe that in turn calls for logging to the console.

CoolThreads.run

This function is required to exist by Python's threading.Thread class. It takes no arguments.

This function will run continuously as long as CTopenForBiz is true and there are worker threads still executing.

CoolThreads.run's primary function is processing the command and log queue.

A queue command of CTjobStop will trigger a dump of deferred logging for the indicated thread and release its semaphore in CTthreadLimit.

When CTopenForBiz becomes False and no more threads are running, any remaining log entries in CTjobLog will be dumped (shouldn't be any), the CTterminated flag will be set True, and the dispatcher's thread will end.

Any further threaded execution should be done with a fresh instance of CoolThreads.

CoolThreads.CTcleanThreads

This method is called on detection of control-C, and is a candidate for overriding in a subclass.

The default implementation sets CTopenForBiz false, which triggers a shutdown in the dispatchers run loop.

CoolThreads.CTannounceTerminate

This can be called by the user if a special case can be constructed to require it. Otherwise, this is called automatically. The example code below illustrates this.

CTannounceTerminate queues a threading command of CTjobStop and returns. The default implementation of this function has no other task to perform. Nothing else is required to gracefully retire a thread.

A single argument is passed to CTannounceTerminate, the thread ID of the terminating thread.

CoolThreads.CTstop

This function is similar to CTannounceTerminate. It queues a command, in this case a CTthreadingStop command.

When the run loop processes that command it will set CTopenForBiz to False, wait for all threads to terminate, and dump deferred logging.

This is another function not generally called by user code. CTwaitForComplete is more commonly used to shut down threading.

CoolThreadOp

CoolThreadOp.__init__

One parameter is required, a thread name. In the case of a subclass of CoollThreadOp, that may be the only argument needed.

If CoolThreadOp is not subclassed, CTrunFunction is a constructor keyword argument that will specify a Python function to run in a thread.

Functions run in that manner will be expected to tolerate an argument CT, which will be a reference to the CoolThreads controlling instance.

That's not always convenient, though, so if the constructor receives an argument of CT, then it will strip out the CT parameter from arguments passed to the CTrunFunction.

In all cases, CTrunFunction is stripped out of the keyword arguments passed to the called procedure.

In other words, if you are calling a function that is OK with getting an extra CT argument, or a function that knows how to use CoolThreads resources, don't pass a CT parameter to CoolThreadOp's constructor.

If you're threading a Python function that doesn't know about CoolThreads, add CT = None to your command line parameters.

Since CT gets set to a CoolThreads reference, if you actually need to pass a CT argument that has nothing to do with CoolThreads, then subclass CoolThreadOp and do anything you need.

Note that CTrunFunction and CT, if used, must be keyword arguments to keep them from disappearing into *args.

CoolThreadOp also recognizes an argument CTlock, which defaults to False.

If True, the new thread will wait to execute until it can acquire the CTnoThread lock.

Is it silly to provide thread protection for a thread that isn't thread-safe?

No, because what happens is that other threads continue to run, but will defer writing to the console until the locked thread completes. Also, any other threads that are launched while a locked thread is executing.

A consequence of thread level locking highlights the fact the user doesn't really have control over the order in which threads start. Once an operation has been handed to Python's threading engine, it's up to Python's scheduler when to run the thread, and when to swap to other threads.

If you need synchronization between threads, use Python's event objects or queues.

CoolThreadOp.CTrun

This is the function that does the work of the thread. The default implementation of CTrun calls the function passed as CTrunFunction in the constructor's arguments.

If CT exists as a keyword argument for the constructor, it is removed from kwargs before passing them on to a target function identified by the keyword constructor argument CTrunFunction. This allows functions that wouldn't allow an extra CT argument to be run.

If CT doesn't exist in the original arguments, it becomes a reference to the CoolThreads dispatcher.

If you want CT be non-existent to the target CTrunFunction, set CT=None when calling CoolThreadOp.__init__.

CTthreadExternal

This is outside of the CoolThreadOp class but is closely related. It exists to run external OS commands inside a thread.

Call CoolThreads.CTaddThread with command line arguments to run, followed by keyword parameters CTgetLock, CTrunFunction, and CT keywords. CT should be set to None, and CTrunFunction should be set to CTthreadGeneric.

CTgetLock should evaluate to True if a lock is required before running the external command.

For example, this would run uname -a in a thread:

```
CTaddThread('threadid', CoolThreadOp('threadid', 'uname', '-a',
            CTrunnFunction = CTthreadGeneric, CT = None)
```

Because CT=None in the construction of a CoolThreadOp object, CT won't be passed to uname, which wouldn't know what to do with it.

Standard output and standard error will both be logged.

Example use

In general terms, CoolThreads is used by creating a CoolThreads object, adding threads, and waiting for them to terminate.

Then call CTaddThread with instances of CoolThreadOp (or a subclass thereof).

Once all the threads are launch, call CTwaitForComplete to harvest all the thread outputs and dump any deferred logging.

There is no requirement for threads to be doing the same thing, or even related things.

Here's an example of simple threading:

```
def sayHello(CT, outString, rwait):
    CT.CTprintConsole(outString + '\n')
    time.sleep(rwait)

with open('testq.log', 'wt') as logFile:
    testQ = CoolThreads('testqueue', logFile, consoleLog = False)
    for I in ('first, 'second', 'third'):
```

```
        testQ.CTaddThread(i,
                    CoolThreadOp(i,
                        CTrunFunction = sayHello,
                        rwait = int(random.random() * 10 + 3),
                        outString = f'Hello from {i}'))
    testQ.CTwaitForComplete()
```

Here's a walkthrough on that code.

First, it's assumed you've imported CoolThreads and random.

CoolThreads is part of a larger package called admintools, so a good way to use CoolThreads is with the import statement "from admintools.coolthreads import *".

The first step is to define a function to run in threads. Note that you don't have to run the same function in every thread. Here we are, but that's not a requirement.

The function we're going to run in parallel is sayHello. It takes three arguments:

- CT - a reference to the controlling CoolThreads object (the dispatcher)
- outString - something to print
- rwait - a duration to sleep after printing the output
- kwargs - a **kwargs was passed, but it would have been empty. It's OK to omit it in the called function's argument list in this case.

Of course, you could have used a single **kwargs parameter for sayHello, and then referenced arguments in the form of kwargs['outString'].

Next, we open a text output file in a with statement, which will be our log file.

Inside the with, a CoolThreads object is created with these parameters:

- testqueue - the name for this thread dispatcher
- logFile - the file we're using for logging
- consoleLog - set to False, so only what we explicitly print goes to the screen

Note that CoolThreads calls its own start method to initiate the dispatcher thread. You don't have to call start(). Worker threads are also automatically started.

The for loop adds threads for each instance of CoolThreadOp, parameterized with CTrunFunction to specify sayHello, and rwait and outString parameters to satisfy sayHello's requirements.

TestQ.CTwaitForComplete lets the threads finish, dumps collated logging, and stops the dispatcher thread.

You can also subclass CoolThreadOp for more flexibility. For example:

```
class TestOp(CoolThreadOp):
  def CTrun(self, **kwargs):
    self.CT.CTprintConsole(kwargs['outString'] + '\n')
    time.sleep(kwargs['rwait'])
```

Then, instead of adding threads of type CoolThreadOp, add instances of TestOp:

```
testQ.CTaddThread('testthread', TestOp('testthread',
                                       rwait = 6,
                                       outString = 'From subclass'))
```

You could also mix threads of different classes, just so long as they are subclasses of CoolThreadOp or a work-alike class of your design.