



Thread Basics

Jacob Hallén
AB Strakt
Göteborg, Sweden



Overview

- Concurrent programming
- Running threads
- Communicating between threads
- Synchronising threads
- Implementing timers



Processes and threads

- A process implements concurrency at the operating system level
 - Several users can run jobs at the same time
 - A user can run several jobs at the same time
- A thread implements concurrency inside a process
 - The process can handle several different tasks at the same time



Why threads?

- To do something while we wait for I/O completion or external events
 - E.g. user interface responds even though we are doing a very long file operation
- To separate units of logic
 - Stronger separation than single threaded object oriented programs
 - Interfaces have to be stricter and more tightly defined



Deadlock and starvation

- **Deadlock:** Process A and B both want resource X and Y
 - Process A starts by grabbing X
 - Process B starts by grabbing Y
- **Starvation**
 - Process A wants resource X but will never get it



Other concurrency issues

- Fairness
- Protection from other processes/threads
- Overload
- Real time issues



Synchronisation mechanisms

- Making sure some operations are atomic
- Giving up the processor
- Passing messages across barriers



Interprocess communication

- Shared memory
- Pipes
- Named pipes
- Sockets
 - Unix domain and TCP/IP
- Signals
 - Communicate with OS



Thread communication

- Memory always shared
- Queues
- Locks
 - Mutex
 - Lock
 - Condition
 - Semaphore
 - Event



Symmetric multiprocessing

- Some machines have more than one CPU
- Some processes can use this to run several threads at the same time
- Python can't
 - Depends on the Global Interpreter Lock (GIL)



Python Modules

- **Standard Library**
 - Part of Optional Operating System Services
- **thread and threading**
 - thread is low level
 - Use threading
- **Queue**



Classes

- Thread
- Queue
- Event
- Lock
 - RLock
- Condition
- Semaphore



Creating new threads

- Inherit Thread
- Override `__init__()` if necessary
- Override `run()`
- Call the `start()` method
 - Call only once!
 - Restarting a thread is not possible

```
import threading
```

```
class CountCharacters(threading.Thread):
```

```
    def __init__(self, filename):
```

```
        self.filename = filename
```

```
        threading.Thread.__init__(self)
```

```
    def run(self):
```

```
        characters = 0
```

```
        try:
```

```
            fd = open(self.filename)
```

```
        except:
```

```
            print 'Failed to open file %s' % self.filename
```

```
            return
```

```
        for line in fd.readlines():
```

```
            characters += len(line)
```

```
        fd.close()
```

```
        print characters
```


```
# print is atomic!
```

```
c1 = CountCharacters('/etc/passwd')
```

```
c1.start()
```

```
c2 = CountCharacters('/etc/services')
```

```
c2.start()
```




```
import threading
import time
```

```
class Sleeper(threading.Thread):
    ''' A thread that sleeps for some time, prints and exits. '''
    def __init__(self, name, seconds):
        self.name = name
        self.seconds = seconds
        threading.Thread.__init__(self)

    def run(self):
        time.sleep(self.seconds)
        print '%s done' % self.name
```

```
c1 = Sleeper('Long', 6)
c1.start()
c2 = Sleeper('Short', 2)
c2.start()
```



```
import threading
import time
```

```
''' Thread can be used without subclassing. '''
```


```
def sleepFunction(name, seconds):
    time.sleep(seconds)
    print '%s done' % name
```

```
c1 = threading.Thread(target=sleepFunction, args=('Long', 6))
c1.start()
c2 = threading.Thread(target=sleepFunction, args=('Short', 2))
c2.start()
```



Main thread and daemons

- The initial thread is the main thread until it dies
 - It can't become a daemon
- Other threads can become daemons
 - Program exits when all non-daemons have died
 - `setDaemon(True)` must be called on thread object before `start()` is called



```
import threading
import time

class Sleeper(threading.Thread):

    def __init__(self, name, seconds):
        self.name = name
        self.seconds = seconds
        threading.Thread.__init__(self)

    def run(self):
        time.sleep(self.seconds)
        print '%s done' % self.name

c1 = Sleeper('Long', 6)
c1.setDaemon(1)
c1.start()
c2 = Sleeper('Short', 2)
c2.start()
```

```
import threading
import time

class Sleeper(threading.Thread):
    ''' Thread is daemonic by default. '''
    def __init__(self, name, seconds):
        self.name = name
        self.seconds = seconds
        threading.Thread.__init__(self)
        self.setDaemon(1)

    def run(self):
        time.sleep(self.seconds)
        print '%s done' % self.name


c1 = Sleeper('Long', 6)
c1.start()
c2 = Sleeper('Short', 2)
c2.setDaemon(0)
c2.start()
```



Waiting for thread to finish

- Call `join()` method on thread object
- Several threads can wait for a thread to finish





```
import threading
import time
```

```
''' Waiting for threads to exit. '''
```

```
class Sleeper(threading.Thread):
```

```
    def __init__(self, name, seconds):
        self.name = name
        self.seconds = seconds
        threading.Thread.__init__(self)
```

```
    def run(self):
        time.sleep(self.seconds)
        print '%s done' % self.name
```

```
c1 = Sleeper('Long', 6)
c1.setDaemon(1)
c1.start()
c2 = Sleeper('Short', 2)
c2.start()
c2.join()
print 'Done waiting for c2'
c1.join()
print 'Done waiting for c1'
```



```
import threading
```

```
import time
```

```
''' Waiting with timeout. '''
```

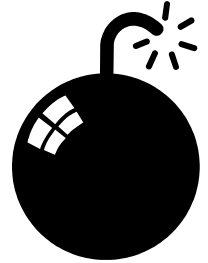
```
class Sleeper(threading.Thread):
```

```
    def __init__(self, name, seconds):  
        self.name = name  
        self.seconds = seconds  
        threading.Thread.__init__(self)
```

```
    def run(self):  
        time.sleep(self.seconds)  
        print '%s done' % self.name
```

```
c1 = Sleeper('Long', 6)  
c1.setDaemon(1)  
c1.start()  
c2 = Sleeper('Short', 2)  
c2.start()  
c1.join(4)  
print 'Done waiting for c1'
```

Killing threads



- You can't!
- The thread has to exit by itself

```
def run(self):  
    # done can either be a global variable or an internal  
    # condition in the loop  
    while not done:  
        # do work...
```



Using common resources

- **Must be controlled**
 - Files, sockets, objects, global variables, hardware
- **A. One thread owns the resource**
 - Other threads send messages to the owner
- **B. Threads can reserve the resource for a time**
 - Must release the resource when they are done

```
import threading # ERROR! Will only work most of the time
import time
import sys

class Producer(threading.Thread):

    def __init__(self, fd, spot):
        self.fd = fd
        self.spot = spot
        threading.Thread.__init__(self)
        self.setDaemon(1)

    def run(self):
        while 1:
            self.fd.seek(self.spot)           # These two lines
            self.fd.write('%d' % self.spot)  # are not an atom
            time.sleep(1)

fd = open('/tmp/foo', 'w')


p1 = Producer(fd, 0)
p2 = Producer(fd, 20)
p1.start()
p2.start()

time.sleep(5)
fd.close()
```



Sending messages

- **Queue.Queue**
 - `get([blocking])`
 - `put(obj [, blocking])`
 - `full()`, `empty()`
 - `qsize()`
- You can queue any object
- You can subclass Queue
 - Eg. Priority Queue (Python recipe by Simo Salminen)



```
import threading
import Queue
import time
```

```
class Producer(threading.Thread):
```

```
    def __init__(self, q, slot):
```

```
        self.q = q
```

```
        self.slot = slot
```

```
        threading.Thread.__init__(self)
```

```
        self.setDaemon(1)
```

```
    def run(self):
```

```
        msg = (self.slot, '%d' %self.slot)
```

```
        while 1:
```

```
            try: # Get exception if queue full
```

```
                self.q.put(msg, 0)
```

```
                self.q.put_nowait(msg) # Synonym to the line above
```

```
            except Queue.Full:
```

```
                print 'Queue full'
```

```
        self.q.put(msg, 1) # Wait if queue full
```

```
        time.sleep(1)
```

```
class Writer(threading.Thread):

    def __init__(self, filename, q):
        self.fd = open(filename, 'w')
        self.q = q
        threading.Thread.__init__(self)

    def run(self):
        while not done:
            spot, str = self.q.get(1) # Blocking
            self.fd.seek(spot)
            self.fd.write(str)
            self.fd.close()

done = 0
q = Queue.Queue(2)
w = Writer('/tmp/foo', q)
w.start()

p1 = Producer(q, 0)
p2 = Producer(q, 20)
p1.start()
p2.start()

time.sleep(5)
done = 1
```



Reserving resource

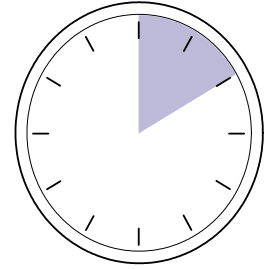
- Can be done with Lock, Rlock, Semaphore, Condition
- Queue is better!
 - Higher level
 - Fewer operations
 - Uncle Tim says so

```
''' Two threads sharing a file descriptor resource using Queue. '''
```

```
class Producer(threading.Thread):  
  
    def __init__(self, q, spot):  
        self.q = q; self.spot = spot; threading.Thread.__init__(self)  
        self.setDaemon(1)  
  
    def run(self):  
        while 1:  
            fd = self.q.get(1)           # Acquire file descriptor, blocking  
            fd.seek(self.spot)  
            fd.write('%d' % self.spot)  
            self.q.put(fd)              # Release file descriptor  
            time.sleep(1)  
  
fd = open('/tmp/foo', 'w')  
  
q = Queue.Queue(1)  
q.put(fd)  
  
p1 = Producer(q, 0)  
p2 = Producer(q, 20)  
p1.start()  
p2.start()  
  
time.sleep(5)  
q.get(1)                               # Wait for resource to be free  
fd.close()
```



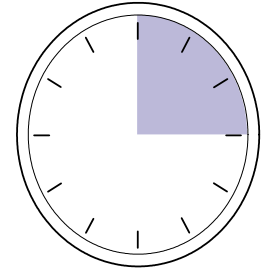
Doing timeouts



- Use Thread and Event
- Timeouts are polling
 - Costs some resources
 - Reason: Hard to do signal based due to differences between operating systems



Event



- `wait([timeout])`
 - `Flag == 0` -> block, `Flag == 1` -> NOP
- `set()`
 - `Flag = 1`, all in `wait()` will wake up
- `clear()`
 - `Flag = 0`, subsequent `wait()` will block
- `isSet()`

```
import threading
import time
```

```
class TimeoutThread(threading.Thread):
```

```
    """
```

```
    Release a file lock after a timeout, unless we get a call that
    resets the timeout.
```

```
    Timeout values are seconds in floating point. Magic values:
```

```
    -1.0 disabling the timeout
```

```
    -2.0 killing the thread
```

```
    0.0 indicates to the run() method that there actually
        was a timeout and not a reset of the timer
```

```
    You can either run the thread as a daemon, or as a non-daemon.
    In the latter case, you have to explicitly kill it by calling
    the terminate() method.
```

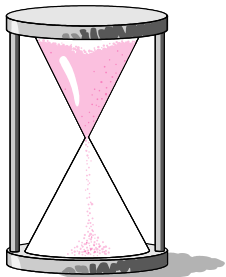
```
    """
```

```
def __init__(self, wantsNotification, *params):
    self.wantsNotification = wantsNotification
    self.params = params
```

```
    threading.Thread.__init__(self)
```

```
    self.timer = threading.Event()
```

```
    self.timeout = -1.0
```

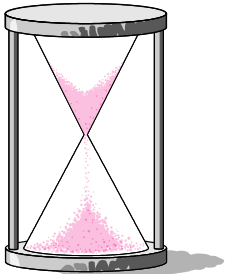


```
def run(self):
    while 1:
        self.timer.clear() # Makes us wait when calling timer.wait()
        if self.timeout > 0.0:
            time = self.timeout
            self.timeout = 0.0
            self.timer.wait(time) # Wait with timeout
        else:
            self.timer.wait() # Wait without timeout (means no lock)

        if self.timeout: # We came here through a reset of the timer
            if self.timeout < -1.0:
                break # Terminate thread
            else:
                pass # self.timeout == -1.0, timeout disabled
        else: # The timer timed out
            self.wantsNotification(*self.params) # Timeout operation

def setTimer(self, timeout):
    print 'Setting timer'
    self.timeout = timeout
    self.timer.set() # Allows the timerThread.run() to proceed

def terminate(self):
    self.timeout = -2.0
    self.timer.set()
```

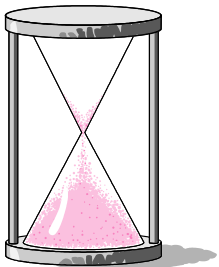


```
def lockRemove():
    lock = 0

    lock = 1
    timeoutThread = TimeoutThread(lockRemove) # Create thread
    timeoutThread.setDaemon(1)
    timeoutThread.start()
    timeoutThread.setTimer(5.0) # Set a timeout
    time.sleep(2)
    timeoutThread.setTimer(5.0) # Set a new timeout before timing out

while lock:
    print "Locked"
    time.sleep(1)


print "Unlocked"
```





GUIs in client programs

- GUI in one thread
- I/O in another thread
 - Non-polling `select()`
- Alternate solution
 - Polling `select()` in GUI thread
 - Tkinter `after()` or equivalent



```
import Tkinter
import time
import threading
import random
import Queue

class GuiPart:
    def __init__(self, master, queue, endCommand):
        self.queue = queue
        # Set up the GUI
        console = Tkinter.Button(master, text='Done', command=endCommand)
        console.pack()
        # Add more GUI stuff here

    def processIncoming(self):
        """
        Handle all the messages currently in the queue (if any).
        """
        while self.queue.qsize():
            try:
                msg = self.queue.get(0)
                # Check contents of message and do what it says
                # As a test, we simply print it
                print msg
            except Queue.Empty:
                pass
```



```
class ThreadedClient:
```

```
    """ Launch the main part of the GUI and the worker thread. """
```

```
    def __init__(self, master):
```

```
        """
```

```
        Start the GUI and the asynchronous threads. We are in the main
        (original) thread of the application, which will later be used by
        the GUI. We spawn a new thread for the worker.
```

```
        """
```

```
        self.master = master
```

```
        # Create the queue
```

```
        self.queue = Queue.Queue()
```

```
        # Set up the GUI part
```

```
        self.gui = GuiPart(master, self.queue, self.endApplication)
```

```
        # Set up the thread to do asynchronous I/O
```

```
        # More can be made if necessary
```

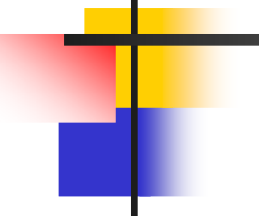
```
        self.running = 1
```

```
        self.thread1 = threading.Thread(target=self.workerThread1)
```

```
        self.thread1.start()
```

```
        # Start periodic call in the GUI to check if the queue contains
        # anything
```


```
        self.periodicCall()
```



```
def periodicCall(self):
    """
    Check every 100 ms if there is something new in the queue.
    """
    self.gui.processIncoming()
    if not self.running:
        # This is the brutal stop of the system. You may want to do
        # some cleanup before actually shutting it down.
        import sys
        sys.exit(1)
    self.master.after(100, self.periodicCall)

def workerThread1(self):
    """
    This is where we handle the asynchronous I/O. For example,
    it may be a 'select()'. One important thing to remember is
    that the thread has to yield control.
    """
    while self.running:
        # To simulate asynchronous I/O, we create a random number at
        # random intervals.
        time.sleep(rand.random() * 0.3) # Replace with real thing
        msg = rand.random()
        self.queue.put(msg)

def endApplication(self):
    self.running = 0
```



```
rand = random.Random()  
root = Tkinter.Tk()  
  
client = ThreadedClient(root)  
root.mainloop()
```



Resources

- Python Library documentation
- Python C API
 - 8.1 Thread state and Global Interpreter Lock
- Tutorial by Aahz
 - <http://starship.python.net/crew/aahz/>
- Python in a Nutshell
 - Alex Martelli, O'Reilly 2002
- Python Cookbook
 - <http://aspn.activestate.com/ASPN/Cookbook/Python>