Super-Duper Semaphoric Synchronization Solution

This assignment will acquaint you with semaphore implementations to code your own mutual exclusion/synchronization solutions using Python3.

Program 1: Implement Semaphores

You are to create a semaphore module that can be used to provide semaphores for synchronizing multi-threaded Python programs using the `_thread` module. You can use the semaphore algorithms in the book as a basic guide. You will need to figure out some way of making waiting threads sleep – the easiest and safest way to put a thread to sleep is to have it wait on a lock. (*Hint:* The thread that releases a lock may be different from the one that acquired it!)

Program 2: lockfight (using semaphores)

The purpose of writing this program is to test your Semaphore class.

For this program you should launch a number of threads that "fight" to gain control of a single mutex semaphore. Specifically, you should write a function that does the following **three times**.

- Try to acquire the global mutex semaphore.
- When successful, announce that it has the mutex and hold it by sleeping for a random amount of time between 0 and 5 seconds. (*Note:* You can use `random.randrange(5)` to generate the number of seconds to sleep.)
- Announce that the mutex is being released.

Your main program should accept the number of threads to launch as a command-line parameter, launch that many threads executing the above function, and then wait for all threads to complete before exiting. (*Hint:* You can have the main program wait on a single semaphore for the appropriate number of times; each thread can signal the done semaphore just before it terminates.) There should be no "extra" sleeps necessary in your program to get interleaving output – the first come first served nature of the semaphores should take care of this.

Some sample output:

```
[letsche@localhost semaphores]$ python lockfight.py 3
Thread 0 trying for the lock.
Thread 0 got the lock.
Thread 0 holding the lock for 0 seconds.
Thread 0 releasing the lock.
Thread 0 trying for the lock.
Thread 0 got the lock.
Thread 0 holding the lock for 2 seconds.
Thread 1 trying for the lock.
Thread 2 trying for the lock.
Thread 0 releasing the lock.
Thread 0 trying for the lock.
Thread 1 got the lock.
Thread 1 holding the lock for 4 seconds.
Thread 1 releasing the lock.
Thread 1 trying for the lock.
Thread 2 got the lock.
Thread 2 holding the lock for 3 seconds.
Thread 2 releasing the lock.
Thread 2 trying for the lock.
Thread 0 got the lock.
Thread 0 holding the lock for 3 seconds.
Thread 0 releasing the lock.
Thread 0 trying for the lock.
Thread 1 got the lock.
Thread 1 holding the lock for 4 seconds.
```

```
Thread 1 releasing the lock.
Thread 1 trying for the lock.
Thread 2 got the lock.
Thread 2 holding the lock for 2 seconds.
Thread 2 releasing the lock.
Thread 2 trying for the lock.
Thread 0 got the lock.
Thread 0 holding the lock for 0 seconds.
Thread 0 releasing the lock.
Thread 0 trying for the lock.
Thread 1 got the lock.
Thread 1 holding the lock for 4 seconds.
Thread 1 releasing the lock.
Thread 1 trying for the lock.
Thread 2 got the lock.
Thread 2 holding the lock for 2 seconds.
Thread 2 releasing the lock.
Thread 2 trying for the lock.
Thread 0 got the lock.
Thread 0 holding the lock for 3 seconds.
Thread 0 releasing the lock.
Thread 0 exiting.
Thread 1 got the lock.
Thread 1 holding the lock for 4 seconds.
Thread 1 releasing the lock.
Thread 1 trying for the lock.
Thread 2 got the lock.
Thread 2 holding the lock for 0 seconds.
Thread 2 releasing the lock.
Thread 2 trying for the lock.
Thread 1 got the lock.
Thread 1 holding the lock for 3 seconds.
Thread 1 releasing the lock.
Thread 1 exiting.
Thread 2 got the lock.
Thread 2 holding the lock for 1 seconds.
Thread 2 releasing the lock.
Thread 2 exiting.
Main thread exiting.
```

Program 3: Naïve Philosphers

Write a program that simulates the Dining Philosophers using the naïve approach of simply protecting each fork as a semaphore. Your program should do the following:

- Get the number of philosophers as a command-line parameter (see the first assignment)
- Create a graphical display of the problem.
- Start the philosophers.
- Pause for the user to click the window.

Each philosopher will be an infinite loop that does the following:

- Think (sleep) for a random time between 0 and 6 seconds.
- Attempt to grab its left fork.
- Sleep for 1 second.
- Attempt to grab its right fork.
- Eat (sleep) for one or two seconds.
- Release the right fork.
- Release the left fork.

You will have to update the graphical display at the appropriate points in the code to show what is happening in your simulation. The code in `dp-shell.py` may give you some idea of how to get started. The GUI is already done for you and can be found in `DPgui.py`. It requires the file `graphics-threaded.py`.

Program 4: Deadlock-Free Dining Philosophers

Fix your solution to program 3 to prevent deadlock. Make sure to document your code to indicate how deadlock has been avoided. In class we talked about a number of ways to prevent, avoid, or detect deadlock. You can use any of those techniques on this problem.

_thread

index
(built-in)
Module Reference

```
This module provides primitive operations to write multi-threaded programs.
The 'threading' module provides a more convenient interface.
```

LockType = class lock(builtins.object)

```
A lock object is a synchronization primitive.  To create a lock,
call the PyThread_allocate_lock() function.  Methods are:

acquire() -
- lock the lock, possibly blocking until it can be obtained
release() -- unlock of the lock
locked() -- test whether the lock is currently locked

A lock is not owned by the thread that locked it; another thread
may
unlock it.  A thread attempting to lock a lock that it has alread
y locked
will block until another thread unlocks it.  Deadlocks may ensue.
```

acquire(...)

```
acquire([wait]) -> None or bool

Lock the lock.  Without argument, this blocks if the lock is already
locked (even by the same thread), waiting for another thread to release
the lock, and return None once the lock is acquired.
With an argument, this will only block if the argument is true,
and the return value reflects whether the lock is acquired.
The blocking operation is interruptible.
```

locked(...)

```
locked() -> bool

Return whether the lock is in the locked state.
```

release(...)

```
release()

Release the lock, allowing another thread that is blocked waiting for
the lock to acquire the lock.  The lock must be in the locked state,
but it needn't be locked by the same thread that unlocks it.
```

Functions

allocate_lock(...)

```
allocate_lock() -> lock object
(allocate() is an obsolete synonym)

Create a new lock object.  See help(LockType) for information
about locks.
```

exit(...)

```
exit()
(exit_thread() is an obsolete synonym)

This is synonymous to ``raise SystemExit''.  It will cause the
current thread to exit silently unless the exception is caught.
```

start_new_thread(...)

```
start_new_thread(function, args[, kwargs])

Start a new thread and return its identifier.  The thread will
call the
function with positional arguments from the tuple args
and keyword arguments
taken from the optional dictionary kwargs.
  The thread exits when the function returns; the return value
is ignored.  The thread will also exit
when the function raises
an unhandled exception; a stack trace will be printed unless
the exception is SystemExit.
```