```python
"""
Scheduler and Io module

Copyright (c) 2009 John Markus Bjoerndalen <jmb@cs.uit.no>,
      Brian Vinter <vinter@diku.dk>, Rune M. Friborg <runef@diku.dk>.
See LICENSE.txt for licensing details (MIT License).
"""

# Imports
from greenlet import greenlet
import threading
import time

# Constants
ACTIVE, DONE, POISON, RETIRE = range(4)
READ, WRITE = range(2)
FAIL, SUCCESS = range(2)


# Decorators
def io(func):
    """
    @io decorator for blocking io operations.
    Execution is moved to seperate threads and the current greenlet is yielded.

    >>> from __init__ import *

    >>> @io
    ... def sleep(n):
    ...     import time
    ...     time.sleep(n)

    >>> @process
    ... def P1():
    ...     sleep(0.05)

    Sleeping for 10 times 0.05 seconds, which equals roughly half a second
    in the sequential case.
    >>> time_start = time.time()
    >>> Sequence([P1() for i in range(10)])
    >>> diff = time.time() - time_start
    >>> diff >= 0.5 and diff < 0.6
    True

    In parallel, it should be close to 0.05 seconds.
    >>> time_start = time.time()
    >>> Parallel([P1() for i in range(10)])
    >>> diff = time.time() - time_start
    >>> diff >= 0.05 and diff < 0.1
    True
    """
    def _call_io(*args, **kwargs):
        io_thread = Io(func, *args, **kwargs)

        if io_thread.p == None:
            # We are not executed from a greenlet
            # Run io code and quit
            return func(*args, **kwargs)

        io_thread.s.io_block_prepare(io_thread.p)
        io_thread.start()
        io_thread.s.io_block_wait(io_thread.p)

        # Return value from function, set by Io class.
        return io_thread.retval
    return _call_io


# Classes
class Io(threading.Thread):
    """ Io(fn, *args, **kwargs)
    It is recommended to use the @io decorator, to create Io instances.
    See io.__doc__
```

```
    """
    def __init__(self, fn, *args, **kwargs):
        threading.Thread.__init__(self)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.retval = None

        self.s = Scheduler()
        self.p = self.s.current

    def run(self):
        self.retval = self.fn(*self.args, **self.kwargs)
        self.s.io_unblock(self.p)


class Scheduler(object):
    """
    Scheduler is a singleton class.

    It is optimized for fast switching and is not fair.

    >>> A = Scheduler()
    >>> B = Scheduler()
    >>> A == B
    True
    """

    __instance = None  # the unique instance

    def __new__(cls, *args, **kargs):
        return cls.getInstance(cls, *args, **kargs)

    def __init__(self):
        pass

    def getInstance(cls, *args, **kargs):
        '''Static method to have a reference to **THE UNIQUE** instance'''
        if cls.__instance is None:
            # (Some exception may be thrown...)
            # Initialize **the unique** instance
            cls.__instance = object.__new__(cls)

            # Initialize members for scheduler
            cls.__instance.new = []
            cls.__instance.next = []
            cls.__instance.current = None
            cls.__instance.greenlet = greenlet.getcurrent()

            # Timer specific  value = (activation time, process)
            # On update we do a sort based on the activation time
            cls.__instance.timers = []

            # Io specific
            cls.__instance.cond = threading.Condition()
            cls.__instance.blocking = 0

        return cls.__instance
    getInstance = classmethod(getInstance)

    # Called by MainThread
    def timer_wait(self, p, seconds):
        new_time = seconds + time.time()

        inserted = False
        for i in xrange(len(self.timers)):
            if new_time < self.timers[i][0]:
                self.timers.insert(i,(new_time, p))
                inserted = True
                break

        if not inserted:
            self.timers.append((new_time, p))
```

```
    # Called by MainThread
    def timer_cancel(self, p):
        for i in xrange(len(self.timers)):
            if self.timers[i][1] == p:
                self.timers.pop(i)
                break

    # Called by threading.Timer()
    def timer_notify(self):
        self.cond.acquire()
        self.cond.notify()
        self.cond.release()

    # Called from MainThread
    def io_block_prepare(self, p):
        self.cond.acquire()
        self.blocking += 1
        p.setstate(ACTIVE)
        self.cond.release()

    # Called from MainThread
    def io_block_wait(self, p):
        p.wait()

    # Called from io thread
    def io_unblock(self, p):
        self.cond.acquire()
        p.notify(DONE, force=True)
        self.blocking -= 1
        self.cond.notify()
        self.cond.release()

    # Add a list of processes onto the new list.
    def addBulk(self, processes):

        # We reverse the list of added processes, if the total amount of new process
es exceeds 1000.
        if len(self.new) + len(processes) > 1000:
            processes.reverse()

        self.new.extend(processes)

    # Main loop
    # When all queues are empty all greenlets have been executed.
    # Queues are new, next, timers and "blocking io counter"
    # Greenlets that are either executing, blocking on a channel or blocking on io i
s not in any lists.
    def main(self):
        while True:
            if self.timers and self.timers[0][0] < time.time():
                _,self.current = self.timers.pop(0)
                self.current.greenlet.switch()
            elif self.new:
                if len(self.new) > 1000:
                    # Pop from end, if the new list might be large.
                    self.current = self.new.pop(-1)
                else:
                    # Pop from beginning to be more fair
                    self.current = self.new.pop(0)
                self.current.greenlet.switch()
            elif self.next:
                # Pop from the beginning
                self.current = self.next.pop(0)
                self.current.greenlet.switch()

            # We enter a critical region, since timer threads or blocking io threads
,
            # might try to update the internal queues.
            self.cond.acquire()
            if not (self.next or self.new):
                # Waiting on blocking processes or all processes have finished!
                if self.timers:
```

```
                    # Set timer to lowest activation time
                    seconds = self.timers[0][0] - time.time()
                    if seconds > 0:
                        t = threading.Timer(seconds, self.timer_notify)

                        # We don't worry about cancelling, since it makes no differe
nce if timer_notify
                        # is called one more time.
                        t.start()

                        # Now go to sleep
                        self.cond.wait()

                elif self.blocking > 0:

                    # Now go to sleep
                    self.cond.wait()
                else:
                    # Execution finished!
                    self.cond.release()
                    return
            self.cond.release()


    # Join is called from _parallel and will block the greenlet until
    # greenlet processes has been executed.
    def join(self, processes):
        if self.greenlet == greenlet.getcurrent():
            # Called from main greenlet
            self.main()
        else:
            # Called from child greenlet
            for p in processes:
                while not p.executed:
                    # p, not executed yet, switch to any waiting greenlet
                    self.getNext().greenlet.switch()



    # Get next greenlet available for scheduling
    def getNext(self):
        if self.new:
            # Returning scheduler, to avoid exceeding the recursion limit.
            # All new greenlets must be started from the scheduler, to have the
            # scheduler as parent greenlet.
            # Switch to main loop
            return self
        elif self.next:
            # Quick choice
            self.current = self.next.pop(0)
            return self.current
        else:
            # Some processes are blocking or all have been executed.
            # Switch to main loop.
            return self


    def activate(self, process):
        self.next.append(process)


# Run tests
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```