```
#\input texinfo
1
2
   #
   # Module implementing synchronization primitives
3
   #
4
   # multiprocessing/synchronize.py
5
   #
6
   # Copyright (c) 2006-2008, R Oudkerk
7
   # Licensed to PSF under a Contributor Agreement.
8
   #
9
10
   all = [
11
        'Lock', 'RLock', 'Semaphore',
12
        'BoundedSemaphore', 'Condition', 'Event'
13
       1
14
   import threading
15
   import sys
16
   import tempfile
17
   import _multiprocessing
18
19
   from time import time as _time
20
21
   from . import context
22
23
   from . import process
   from . import util
24
25
   # Try to import the mp.synchronize module cleanly,
26
   if it fails
   # raise ImportError for platforms lacking a
27
   working sem_open implementation.
   # See issue 3770
28
   try:
29
       from _multiprocessing import SemLock,
30
       sem_unlink
   except (ImportError):
31
```

```
raise ImportError("This platform lacks a
32
       functioning sem_open" +
                            implementation, therefore,
33
                         the required" +
                             synchronization primitives
34
                         needed will not" +
                           " function, see issue
35
                         3770.")
36
37
   #
   # Constants
38
39
   #
40
   RECURSIVE MUTEX, SEMAPHORE = list(range(2))
41
   SEM VALUE MAX =
42
   _multiprocessing.SemLock.SEM_VALUE_MAX
43
   #
44
   # Base class for semaphores and mutexes; wraps
45
    _multiprocessing.SemLock`
   #
46
47
   class SemLock(object):
48
49
       rand = tempfile. RandomNameSequence()
50
51
       def __init__(self, kind, value, maxvalue, *,
52
       ctx):
53
            if ctx is None:
                ctx =
54
                context._default_context.get_context()
            name = ctx.get_start_method()
55
            unlink_now = sys.platform == 'win32' or
56
            name == 'fork'
            for i in range(100):
57
```

```
58
                try:
                    sl = self._semlock =
59
                    multiprocessing.SemLock(
                         kind, value, maxvalue,
60
                         self._make_name(),
                         unlink_now)
61
                except FileExistsError:
62
63
                    pass
                else:
64
65
                    break
            else:
66
                raise FileExistsError('cannot find
67
                name for semaphore')
68
            util.debug('created semlock with handle
69
            %s' % sl.handle)
            self. make methods()
70
71
            if sys.platform != 'win32':
72
                def _after_fork(obj):
73
                    obj._semlock._after_fork()
74
                util.register_after_fork(self,
75
                _after_fork)
76
            if self. semlock.name is not None:
77
                # We only get here if we are on Unix
78
                with forking
                # disabled. When the object is
79
                garbage collected or the
                # process shuts down we unlink the
80
                semaphore name
                from .semaphore_tracker import
81
                register
                register(self._semlock.name)
82
                util.Finalize(self, SemLock._cleanup,
83
```

```
(self. semlock.name,),
                                exitpriority=0)
84
85
        @staticmethod
86
        def _cleanup(name):
87
             from .semaphore_tracker import unregister
88
             sem unlink(name)
89
             unregister(name)
90
91
        def make methods(self):
92
             self.acquire = self._semlock.acquire
93
             self.release = self._semlock.release
94
95
        def enter (self):
96
             return self. semlock. enter ()
97
98
        def __exit__(self, *args):
99
             return self._semlock.__exit__(*args)
100
101
        def __getstate__(self):
102
             context.assert_spawning(self)
103
             sl = self. semlock
104
             if sys.platform == 'win32':
105
                 h =
106
                 context.get_spawning_popen().duplicate
                 for child(sl.handle)
             else:
107
                 h = sl.handle
108
             return (h, sl.kind, sl.maxvalue, sl.name)
109
110
        def __setstate__(self, state):
111
             self._semlock =
112
             _multiprocessing.SemLock._rebuild(*state)
             util.debug('recreated blocker with handle
113
            %r' % state[0])
```

```
self. make methods()
114
115
        @staticmethod
116
        def make name():
117
             return '%s-%s' %
118
             (process.current_process()._config['sempre
             fix'l.
                                next(SemLock._rand))
119
120
121
    #
    # Semaphore
122
123
    #
124
    class Semaphore(SemLock):
125
126
        def __init__(self, value=1, *, ctx):
127
             SemLock.__init__(self, SEMAPHORE, value,
128
             SEM_VALUE_MAX, ctx=ctx)
129
        def get value(self):
130
             return self._semlock._get_value()
131
132
        def __repr__(self):
133
134
             try:
                 value = self. semlock. get value()
135
             except Exception:
136
                 value = 'unknown'
137
             return '<%s(value=%s)>' %
138
             (self.__class__.__name__, value)
139
140
    #
    # Bounded semaphore
141
142
143
    class BoundedSemaphore(Semaphore):
144
```

```
145
        def __init__(self, value=1, *, ctx):
146
             SemLock.__init__(self, SEMAPHORE, value,
147
             value, ctx=ctx)
148
        def __repr__(self):
149
             try:
150
                 value = self._semlock._get_value()
151
             except Exception:
152
                 value = 'unknown'
153
             return '<%s(value=%s, maxvalue=%s)>' % \
154
                    (self.__class__._name__, value,
155
                    self. semlock.maxvalue)
156
157
158
    # Non-recursive lock
159
160
    class Lock(SemLock):
161
162
        def __init__(self, *, ctx):
163
             SemLock.__init__(self, SEMAPHORE, 1, 1,
164
             ctx=ctx)
165
        def __repr__(self):
166
             try:
167
                 if self._semlock._is_mine():
168
169
                     name =
                     process.current_process().name
                     if threading.current_thread().name
170
                      != 'MainThread':
                          name += '|' +
171
                          threading.current_thread().nam
                 elif self. semlock. get value() == 1:
172
```

```
173
                     name = 'None'
                 elif self._semlock._count() > 0:
174
                     name = 'SomeOtherThread'
175
                 else:
176
                     name = 'SomeOtherProcess'
177
             except Exception:
178
                 name = 'unknown'
179
             return '<%s(owner=%s)>' %
180
             (self.__class__._name__, name)
181
    #
182
183
    # Recursive lock
184
    #
185
    class RLock(SemLock):
186
187
        def __init__(self, *, ctx):
188
             SemLock.__init__(self, RECURSIVE_MUTEX, 1,
189
             1, ctx=ctx)
190
        def repr (self):
191
             try:
192
                 if self._semlock._is_mine():
193
194
                     name =
                     process.current process().name
                     if threading.current thread().name
195
                      != 'MainThread':
                          name += '|' +
196
                          threading.current_thread().nam
                     count = self._semlock._count()
197
                 elif self._semlock._get_value() == 1:
198
                     name, count = 'None', 0
199
                 elif self._semlock._count() > 0:
200
                     name, count = 'SomeOtherThread',
201
```

```
'nonzero'
                 else:
202
                     name, count = 'SomeOtherProcess',
203
                     'nonzero'
             except Exception:
204
                 name, count = 'unknown', 'unknown'
205
             return '<%s(%s, %s)>' %
206
             (self.__class__._name__, name, count)
207
    #
208
    # Condition variable
209
210
    #
211
    class Condition(object):
212
213
        def __init__(self, lock=None, *, ctx):
214
             self._lock = lock or ctx.RLock()
215
             self._sleeping_count = ctx.Semaphore(0)
216
             self._woken_count = ctx.Semaphore(0)
217
             self._wait_semaphore = ctx.Semaphore(0)
218
             self. make methods()
219
220
        def getstate (self):
221
             context.assert_spawning(self)
222
             return (self._lock, self._sleeping_count,
223
                     self. woken count,
224
                     self._wait_semaphore)
225
        def __setstate__(self, state):
226
             (self._lock, self._sleeping_count,
227
              self._woken_count, self._wait_semaphore)
228
              = state
             self. make methods()
229
230
        def enter (self):
231
```

```
return self._lock.__enter__()
232
233
        def exit (self, *args):
234
             return self._lock.__exit__(*args)
235
236
        def _make_methods(self):
237
             self.acquire = self._lock.acquire
238
             self.release = self._lock.release
239
240
        def __repr__(self):
241
             try:
242
                 num waiters =
243
                 (self._sleeping_count._semlock._get_va
                 lue() -
244
                          self._woken_count._semlock._g
                          et_value())
             except Exception:
245
                 num waiters = 'unknown'
246
             return '<%s(%s, %s)>' %
247
             (self.__class__._name__, self._lock,
             num waiters)
248
        def wait(self, timeout=None):
249
             assert self._lock._semlock._is_mine(), \
250
                    'must acquire() condition before
251
                    using wait()'
252
            # indicate that this thread is going to
253
             sleep
             self._sleeping_count.release()
254
255
            # release lock
256
             count = self._lock._semlock._count()
257
             for i in range(count):
258
```

```
self. lock.release()
259
260
             try:
261
                 # wait for notification or timeout
262
263
                 return
                 self._wait_semaphore.acquire(True,
                 timeout)
             finally:
264
                 # indicate that this thread has woken
265
                 self. woken count.release()
266
267
268
                 # reacquire lock
                 for i in range(count):
269
                     self. lock.acquire()
270
271
        def notify(self):
272
             assert self._lock._semlock._is_mine(),
273
             'lock is not owned'
             assert not
274
             self. wait semaphore.acquire(False)
275
             # to take account of timeouts since last
276
             notify() we subtract
             # woken_count from sleeping_count and
277
             rezero woken count
             while self._woken_count.acquire(False):
278
                 res =
279
                 self._sleeping_count.acquire(False)
280
                 assert res
281
             if self._sleeping_count.acquire(False): #
282
             try grabbing a sleeper
                 self._wait_semaphore.release()
283
                                                        #
                 wake up one sleeper
                 self._woken_count.acquire()
                                                        #
284
```

```
wait for the sleeper to wake
285
                 # rezero _wait_semaphore in case a
286
                 timeout just happened
                 self._wait_semaphore.acquire(False)
287
288
        def notify_all(self):
289
             assert self._lock._semlock._is_mine(),
290
             'lock is not owned'
291
             assert not
             self. wait semaphore.acquire(False)
292
            # to take account of timeouts since last
293
            notify*() we subtract
            # woken_count from sleeping_count and
294
             rezero woken_count
            while self._woken_count.acquire(False):
295
296
                 self. sleeping count.acquire(False)
                 assert res
297
298
             sleepers = 0
299
            while self._sleeping_count.acquire(False):
300
                 self. wait semaphore.release()
301
                 # wake up one sleeper
                 sleepers += 1
302
303
             if sleepers:
304
                 for i in range(sleepers):
305
                     self._woken_count.acquire()
306
                     # wait for a sleeper to wake
307
                 # rezero wait semaphore in case some
308
                 timeouts just happened
                 while
309
```

```
self. wait semaphore.acquire(False):
310
                      pass
311
         def wait for(self, predicate, timeout=None):
312
             result = predicate()
313
             if result:
314
                  return result
315
             if timeout is not None:
316
                 endtime = _time() + timeout
317
318
             else:
                 endtime = None
319
320
                 waittime = None
             while not result:
321
                  if endtime is not None:
322
                      waittime = endtime - time()
323
324
                      if waittime <= 0:
                          break
325
                 self.wait(waittime)
326
                  result = predicate()
327
             return result
328
329
330
    #
    # Event
331
332
    #
333
    class Event(object):
334
335
         def __init__(self, *, ctx):
336
             self. cond = ctx.Condition(ctx.Lock())
337
             self._flag = ctx.Semaphore(0)
338
339
        def is_set(self):
340
             with self. cond:
341
                  if self._flag.acquire(False):
342
                      self._flag.release()
343
```

```
return True
344
                  return False
345
346
         def set(self):
347
             with self._cond:
348
                  self._flag.acquire(False)
349
                  self._flag.release()
350
                  self. cond.notify all()
351
352
         def clear(self):
353
             with self._cond:
354
355
                  self._flag.acquire(False)
356
         def wait(self, timeout=None):
357
             with self. cond:
358
                  if self._flag.acquire(False):
359
                      self._flag.release()
360
                  else:
361
                      self. cond.wait(timeout)
362
363
                  if self._flag.acquire(False):
364
                      self._flag.release()
365
                      return True
366
367
                  return False
368
369
    #
    # Barrier
370
371
    #
372
    class Barrier(threading.Barrier):
373
374
         def __init__(self, parties, action=None,
375
         timeout=None, *, ctx):
             import struct
376
             from .heap import BufferWrapper
377
```

```
378
             wrapper =
             BufferWrapper(struct.calcsize('i') * 2)
             cond = ctx.Condition()
379
             self.__setstate__((parties, action,
380
             timeout, cond, wrapper))
             self._state = 0
381
             self._count = 0
382
383
        def __setstate__(self, state):
384
             (self._parties, self._action,
385
             self. timeout.
              self._cond, self._wrapper) = state
386
             self. array =
387
             self. wrapper.create memoryview().cast('i'
388
        def getstate (self):
389
             return (self._parties, self._action,
390
             self._timeout,
                     self._cond, self._wrapper)
391
392
393
        @property
        def _state(self):
394
             return self. array[0]
395
396
        @ state.setter
397
        def _state(self, value):
398
             self._array[0] = value
399
400
        @property
401
        def _count(self):
402
             return self._array[1]
403
404
405
        @ count.setter
        def _count(self, value):
406
```

self.\_array[1] = value
408