

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

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

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

```

            (self._semlock.name,),
            exitpriority=0)
84
85
86     @staticmethod
87     def _cleanup(name):
88         from .semaphore_tracker import unregister
89         sem_unlink(name)
90         unregister(name)
91
92     def _make_methods(self):
93         self.acquire = self._semlock.acquire
94         self.release = self._semlock.release
95
96     def __enter__(self):
97         return self._semlock.__enter__()
98
99     def __exit__(self, *args):
100         return self._semlock.__exit__(*args)
101
102     def __getstate__(self):
103         context.assert_spawning(self)
104         sl = self._semlock
105         if sys.platform == 'win32':
106             h =
107                 context.get_spawning_popen().duplicate
108                 _for_child(sl.handle)
109         else:
110             h = sl.handle
111         return (h, sl.kind, sl.maxvalue, sl.name)
112
113     def __setstate__(self, state):
114         self._semlock =
115             _multiprocessing.SemLock._rebuild(*state)
116         util.debug('recreated blocker with handle
117 %r' % state[0])
```

```
114         self._make_methods()
115
116     @staticmethod
117     def _make_name():
118         return '%s-%s' %
119             (process.current_process()._config['sempre
120              fix'],
121              next(SemLock._rand))
122
123     #
124     # Semaphore
125     #
126
127     class Semaphore(SemLock):
128
129         def __init__(self, value=1, *, ctx):
130             SemLock.__init__(self, SEMAPHORE, value,
131                             SEM_VALUE_MAX, ctx=ctx)
132
133         def get_value(self):
134             return self._semlock._get_value()
135
136         def __repr__(self):
137             try:
138                 value = self._semlock._get_value()
139             except Exception:
140                 value = 'unknown'
141             return '<%s(value=%s)>' %
142                 (self.__class__.__name__, value)
143
144     #
145     # Bounded semaphore
146     #
147
148     class BoundedSemaphore(Semaphore):
```

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

```
173         name = 'None'
174     elif self._semlock._count() > 0:
175         name = 'SomeOtherThread'
176     else:
177         name = 'SomeOtherProcess'
178 except Exception:
179     name = 'unknown'
180 return '<%s(owner=%s)>' %
    (self.__class__.__name__, name)

181
182 #
183 # Recursive lock
184 #
185
186 class RLock(SemLock):
187
188     def __init__(self, *, ctx):
189         SemLock.__init__(self, RECURSIVE_MUTEX, 1,
190             1, ctx=ctx)
191
192     def __repr__(self):
193         try:
194             if self._semlock._is_mine():
195                 name =
196                     process.current_process().name
197                     if threading.current_thread().name
198                         != 'MainThread':
199                         name += '|' +
200                             threading.current_thread().name
201
202                 count = self._semlock._count()
203             elif self._semlock._get_value() == 1:
204                 name, count = 'None', 0
205             elif self._semlock._count() > 0:
206                 name, count = 'SomeOtherThread',
```

```

    'nonzero'
202     else:
203         name, count = 'SomeOtherProcess',
    'nonzero'
204 except Exception:
205     name, count = 'unknown', 'unknown'
206 return '<%s(%s, %s)>' %
    (self.__class__.__name__, name, count)
207
208 #
209 # Condition variable
210 #
211
212 class Condition(object):
213
214     def __init__(self, lock=None, *, ctx):
215         self._lock = lock or ctx.RLock()
216         self._sleeping_count = ctx.Semaphore(0)
217         self._woken_count = ctx.Semaphore(0)
218         self._wait_semaphore = ctx.Semaphore(0)
219         self._make_methods()
220
221     def __getstate__(self):
222         context.assert_spawning(self)
223         return (self._lock, self._sleeping_count,
224                 self._woken_count,
225                 self._wait_semaphore)
226
227     def __setstate__(self, state):
228         (self._lock, self._sleeping_count,
229          self._woken_count, self._wait_semaphore)
230         = state
231         self._make_methods()
232
233     def __enter__(self):
```



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

```
259         self._lock.release()
260
261     try:
262         # wait for notification or timeout
263         return
264         self._wait_semaphore.acquire(True,
265                                     timeout)
266     finally:
267         # indicate that this thread has woken
268         self._woken_count.release()
269
270         # reacquire lock
271         for i in range(count):
272             self._lock.acquire()
273
274 def notify(self):
275     assert self._lock._semlock._is_mine(),
276     'lock is not owned'
277     assert not
278     self._wait_semaphore.acquire(False)
279
280     # to take account of timeouts since last
281     notify() we subtract
282     # woken_count from sleeping_count and
283     rezero woken_count
284     while self._woken_count.acquire(False):
285         res =
286         self._sleeping_count.acquire(False)
287         assert res
288
289     if self._sleeping_count.acquire(False): #
290     try grabbing a sleeper
291         self._wait_semaphore.release() #
292         wake up one sleeper
293         self._woken_count.acquire() #
```

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

```

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

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

```
378         wrapper =
379             BufferWrapper(struct.calcsize('i') * 2)
380         cond = ctx.Condition()
381         self.__setstate__((parties, action,
382                             timeout, cond, wrapper))
383         self._state = 0
384         self._count = 0
385
386     def __setstate__(self, state):
387         (self._parties, self._action,
388          self._timeout,
389          self._cond, self._wrapper) = state
390         self._array =
391             self._wrapper.create_memoryview().cast('i'
392             )
393
394     def __getstate__(self):
395         return (self._parties, self._action,
396                 self._timeout,
397                 self._cond, self._wrapper)
398
399     @property
400     def _state(self):
401         return self._array[0]
402
403     @_state.setter
404     def _state(self, value):
405         self._array[0] = value
406
407     @property
408     def _count(self):
409         return self._array[1]
410
411     @_count.setter
412     def _count(self, value):
```

---

```
407 |         self._array[1] = value
408 |
```