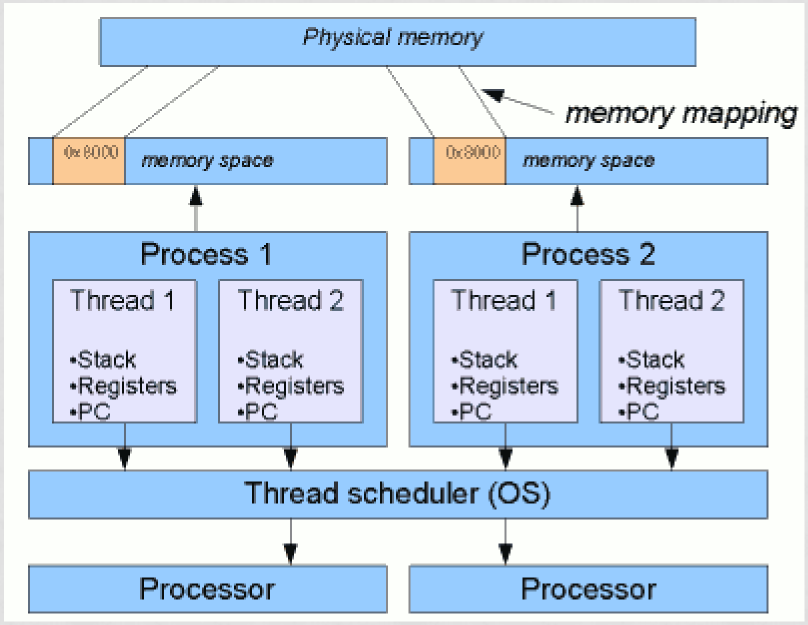
THREADING PROGRAMMING USING PYTHON

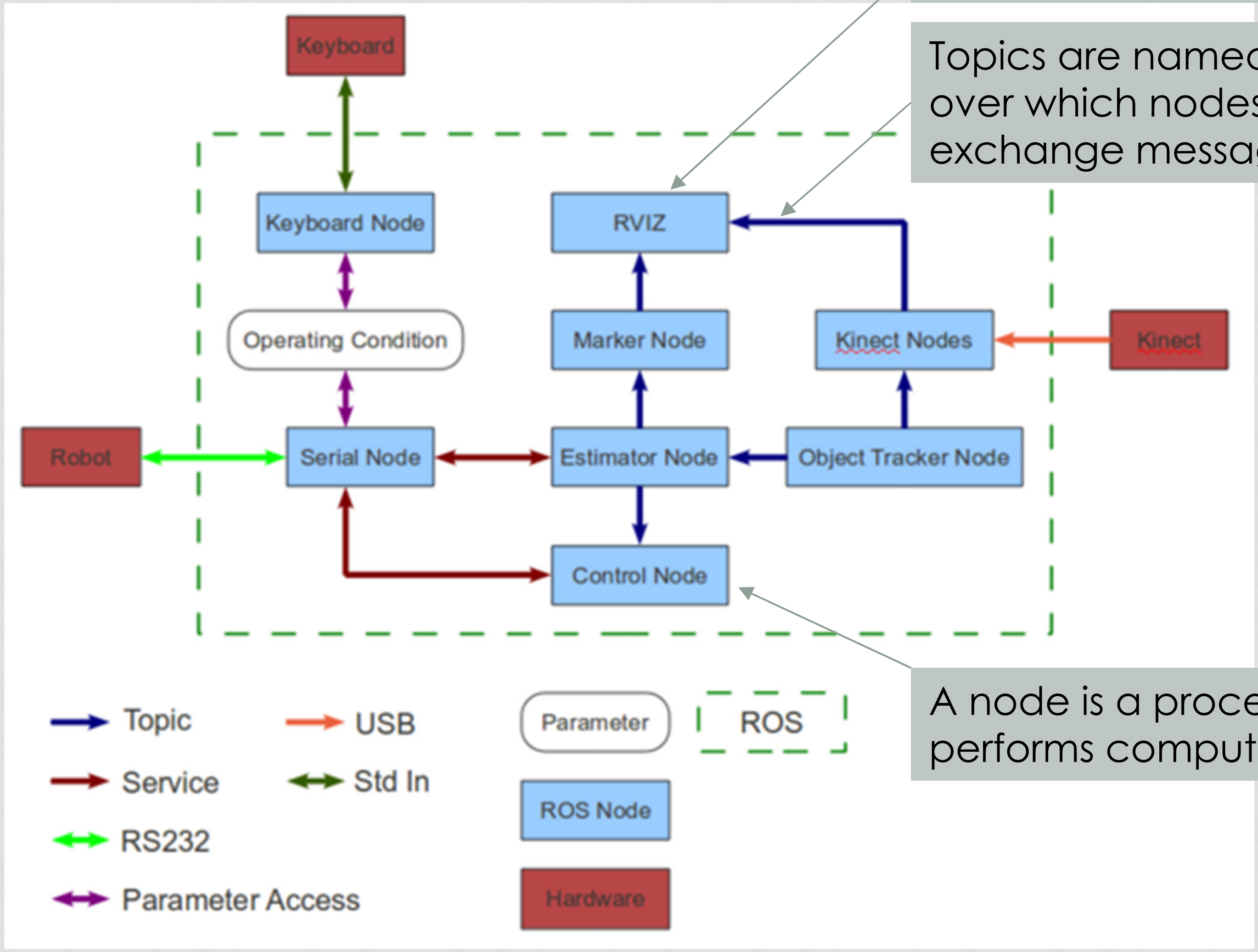
CUAUHTEMOC CARBAJAL
ITESM CEM
APRIL 06, 2013

PROCESS

- Background
  - A running program is called a "process"
  - Each process has memory, list of open files, stack, program counter, etc...
  - Normally, a process executes statements in a single sequence of control flow.
- Process creation with fork(),system(), popen(), etc...
  - These commands create an entirely new process.
  - Child process runs independently of the parent.
  - Has own set of resources.
  - There is minimal sharing of information between parent and child.
  - Think about using the Unix shell.



ROBOT OPERATING SYSTEM (ROS)



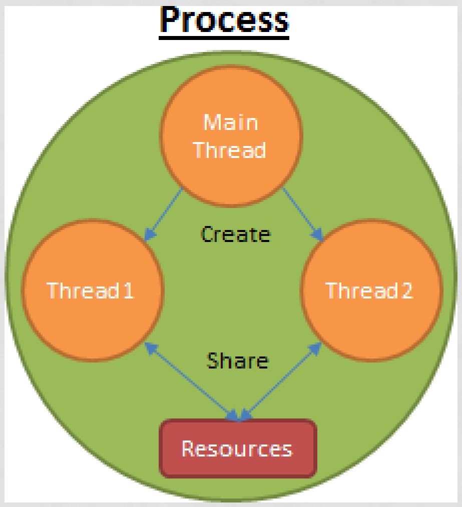
3D visualization tool

Topics are named buses over which nodes exchange messages.

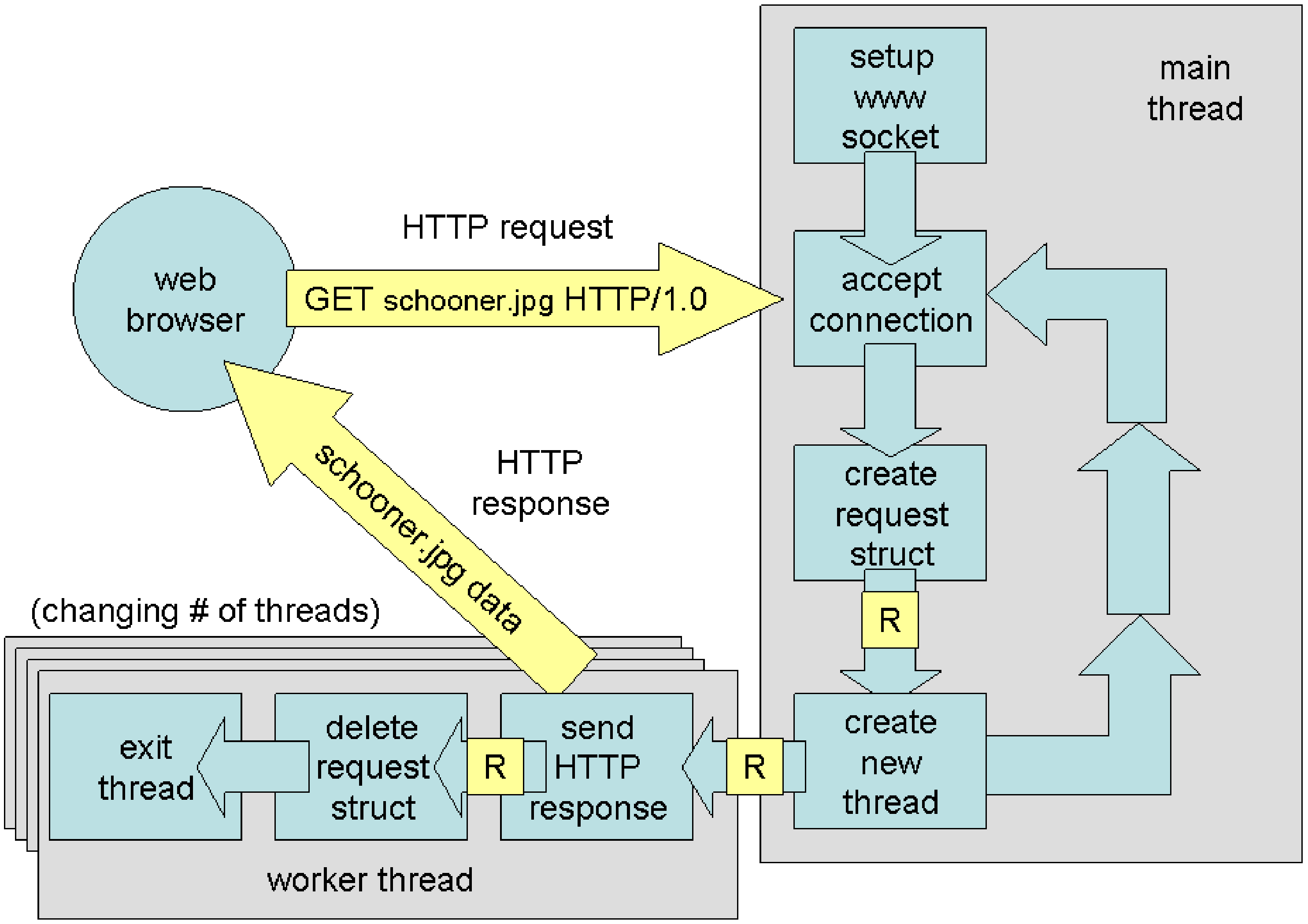
A node is a process that performs computation.

THREAD BASICS

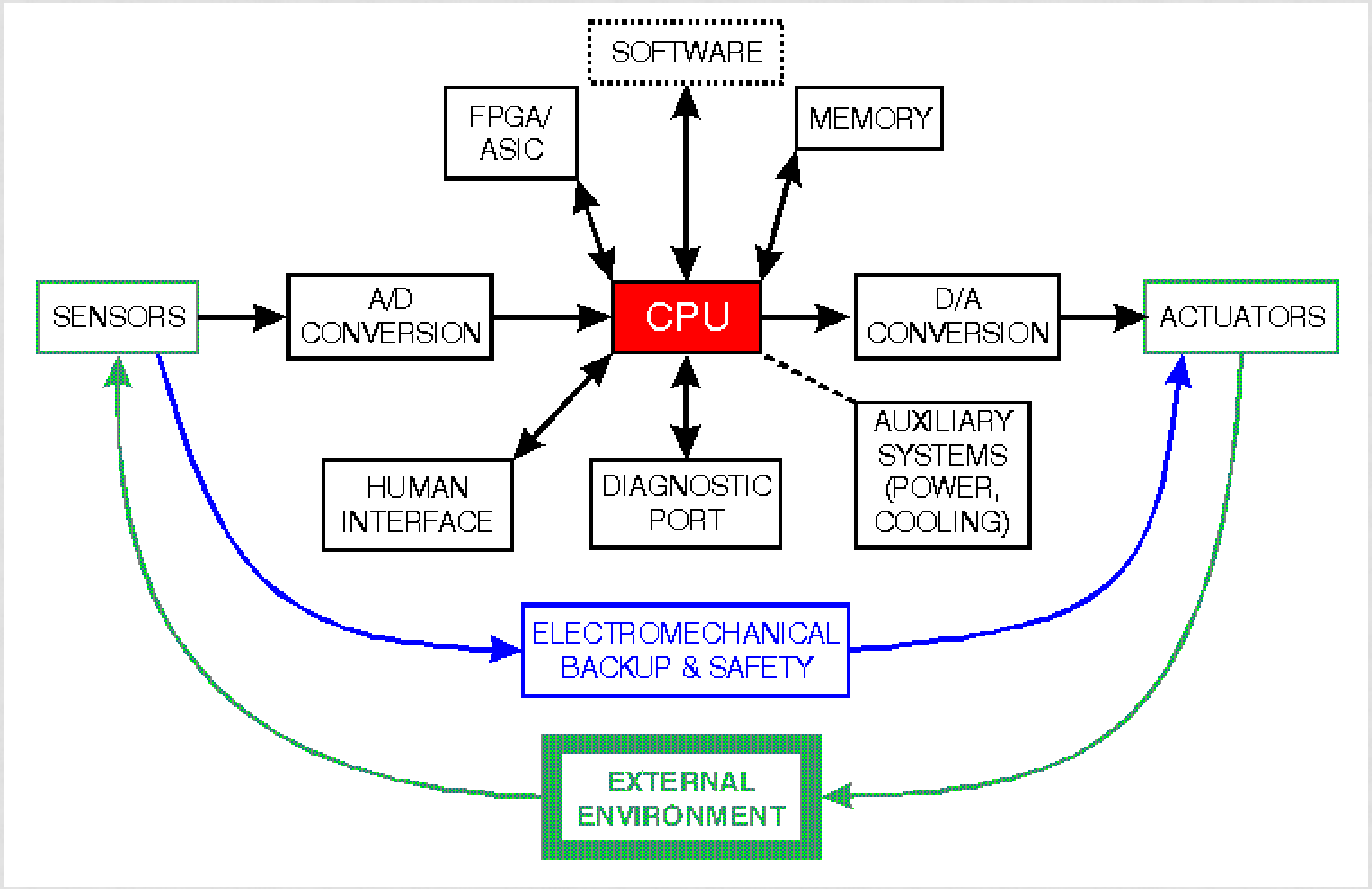
- Threads
  - A thread is a light-weight process (it's a sequence of control flow).
  - Except that it exists entirely inside a process and shares resources.
  - A single process may have multiple threads of execution.
  - Useful when an application wants to perform many concurrent tasks on shared data.
  - Think about a browser (loading pages, animations, etc.)



MULTITHREADED WEB SERVER

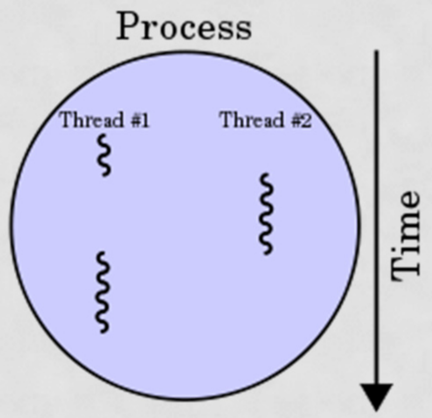


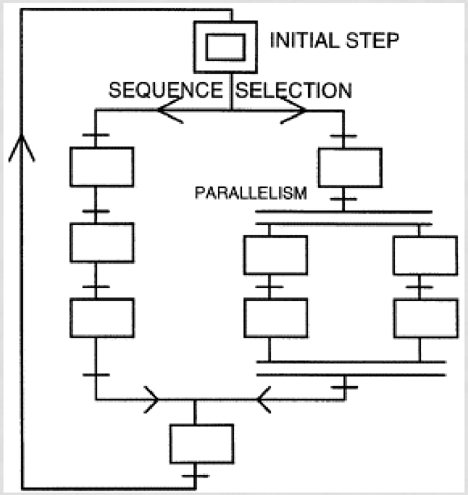
MULTITASKING EMBEDDED SYSTEM



ADVANTAGES OF THREADING

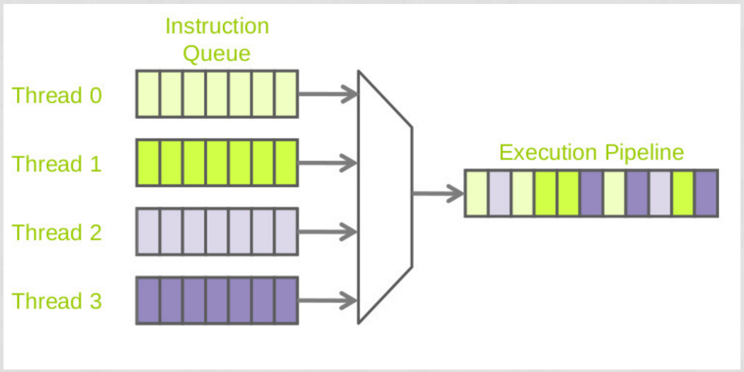
- Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be truly concurrent.
- A program can remain responsive to input. This is true both on single and on multiple CPUs.
- Allows to do something else while one thread is waiting for an I/O task (disk, network) to complete.
- Some programs are easy to express using concurrency which leads to elegant solutions, easier to maintain and debug.
- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

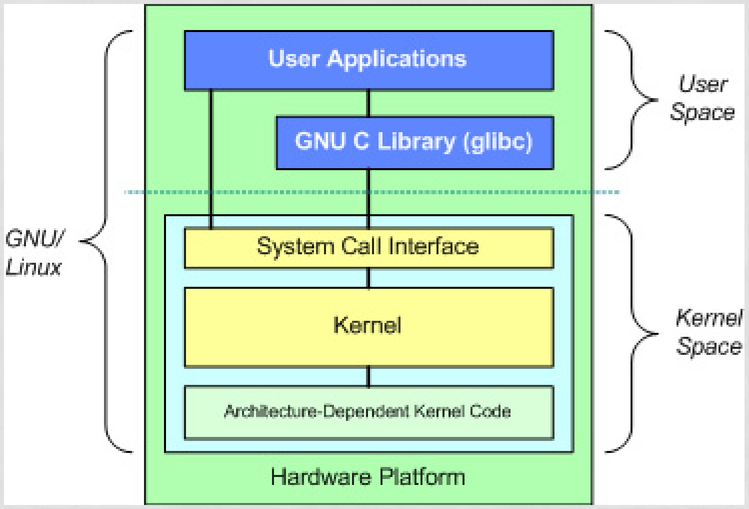




THREADS ISSUES…

- Scheduling
  - To execute a threaded program, must rapidly switch between threads.
    - This can be done by the user process (user-level threads).
    - Can be done by the kernel (kernel-level threads).
- Resource Sharing
  - Since threads share memory and other resources, must be very careful.
  - Operation performed in one thread could cause problems in another.
- Synchronization
  - Threads often need to coordinate actions.
  - Can get "race conditions" (outcome dependent on order of thread execution)
  - Often need to use locking primitives (mutual exclusion locks, semaphores, etc...)





PYTHON THREADS

- Python supports threads on the following platforms
  - Solaris
  - Windows
  - Systems that support the POSIX threads library (pthreads)
- Thread scheduling
  - Tightly controlled by a **global interpreter lock** and scheduler.
    - Only a single thread is allowed to be executing in the Python interpreter at once.
    - Thread switching only occurs between the execution of individual byte-codes.
    - Long-running calculations in C/C++ can block execution of all other threads.
    - However, most I/O operations do not block.

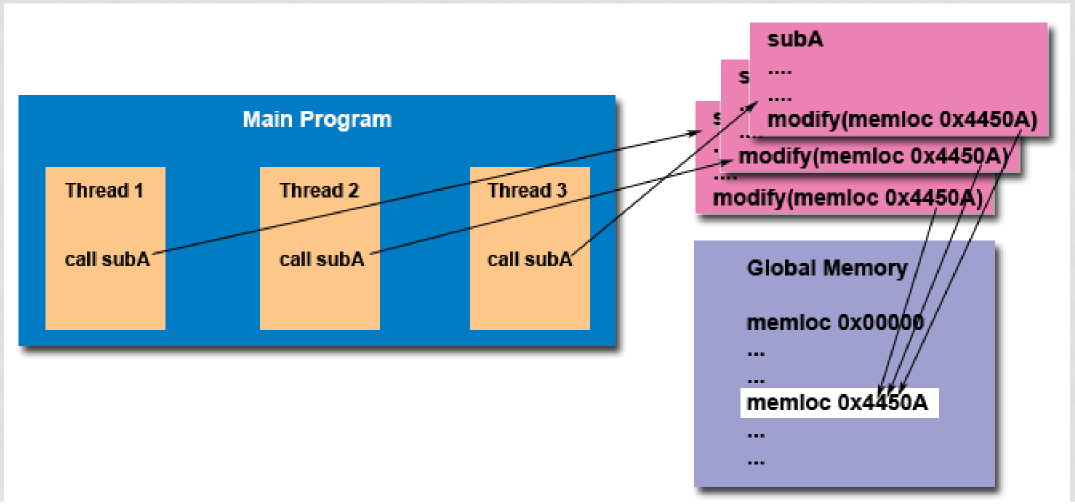
```
>>> import dis
>>> def my_function(string1):
          return len(string1)

>>> dis.dis(my_function)
  2       0 LOAD_GLOBAL          0 (len)
          3 LOAD_FAST            0  (string1)
          6 CALL_FUNCTION        1
          9 RETURN_VALUE
>>>
```

```
>>> import sys
>>> sys.getcheckinterval()
100
```

PYTHON THREADS (CONT)

- Comments
  - Python threads are somewhat more restrictive than in C.
  - Effectiveness may be limited on multiple CPUs (due to interpreter lock).
  - Threads can interact strangely with other Python modules (especially signal handling).
  - Not all extension modules are thread-safe.
    - Thread-safe describes a program portion or routine that can be called from multiple programming threads without unwanted interaction between the threads.



´PYTHON THREAD MODULE

THREAD MODULE

- The thread module provides low-level access to threads
  - Thread creation.
  - Simple mutex locks.
- Creating a new thread
  - `thread.start_new_thread(func,[args [,kwargs]])`
  - Executes a function in a new thread.

```
import thread
import time
def print_time(delay):
  while 1:
    time.sleep(delay)
    print time.ctime(time.time())
# Start the thread
thread.start_new_thread(print_time,(5,))
# Go do something else
# statements
while (1): pass
```

Convert a time expressed in seconds since the epoch to a string representing local time

Return the time in seconds since the epoch

/threading/ex7.py C3PO

```python
#!/usr/bin/python

import thread
import time

# Define a function for the thread
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

/threading/ex1.py C3PO

THREAD MODULE (CONT)

- Thread termination
  - Thread silently exits when the function returns.
  - Thread can explicitly exit by calling **`thread.exit()`** or **`sys.exit()`**.
  - Uncaught exception causes thread termination (and prints error message).
  - However, other threads continue to run even if one had an error.
- Simple locks
  - **`allocate_lock()`**. Creates a lock object, initially unlocked.

```
import thread
lk = thread.allocate_lock()
def foo():
  lk.acquire() # Acquire the lock
  # critical section
  lk.release() # Release the lock
```

  - Only one thread can acquire the lock at once.
  - Threads block indefinitely until lock becomes available.
  - You might use this if two or more threads were allowed to update a shared data structure.

```
#!/usr/bin/env python

import time
import thread

def myfunction(string,sleeptime,lock,*args):
  while 1:
    #entering critical section
    lock.acquire()
    print string," Now Sleeping after Lock acquired for ",sleeptime
    time.sleep(sleeptime)
    print string," Now releasing lock and then sleeping again "
    lock.release()
    # exiting critical section
    time.sleep(sleeptime) # why?

if __name__ == "__main__":

  lock = thread.allocate_lock()
  thread.start_new_thread(myfunction,("Thread No:1",2,lock))
  thread.start_new_thread(myfunction,("Thread No:2",2,lock))

  while 1:
    pass
```

An asterisk (*) is placed before the variable name that will hold the values of all nonkeyword variable arguments.

/threading/ex2.py C3PO

THREAD MODULE (CONT)

- The main thread
  - When Python starts, it runs as a single thread of execution.
  - This is called the "main thread."
  - On its own, it's no big deal.
  - However, if you launch other threads it has some special properties.
- Termination of the main thread
  - If the main thread exits and other threads are active, the behavior is system dependent.
  - Usually, this immediately terminates the execution of all other threads without cleanup.
  - Cleanup actions of the main thread may be limited as well.
- Signal handling
  - Signals can only be caught and handled by the main thread of execution.
  - Otherwise you will get an error (in the signal module).
  - Caveat: The **`KeyboardInterrupt`** can be caught by any thread (non-deterministically).

THREAD MODULE (CONT)

- Currently, The Python Interpreter is not fully thread safe.
- There are no priorities, no thread groups.
- Threads cannot be stopped and suspended, resumed or interrupted.
- That is, the support provided is very much basic.
- However a lot can still be accomplished with this meager support, with the use of the threading module.

PYTHON THREADING MODULE

THREADING – HIGHER-LEVEL THREADING INTERFACE

- Python manages to get a lot done using so little.
  - The Threading module uses the built in thread package to provide some very interesting features that would make your programming a whole lot easier.
  - There are in built mechanisms which provide critical section locks, wait/notify locks etc.
- Major Components of the Threading module are:
  - Thread Object
  - Lock object
  - RLock object
  - Semaphore Object
  - Condition Object
  - Event Object

THREADING – HIGHER-LEVEL THREADING INTERFACE (CONT)

- It is a high-level threads module
  - Implements threads as classes (similar to Java)
  - Provides an assortment of synchronization and locking primitives.
  - Built using the low-level thread module.
- Creating a new thread (as a class)
  - Idea: Inherit from the "Thread" class and provide a few methods

```
import threading, time
class PrintTime(threading.Thread):
  def __init__(self,interval):
    threading.Thread.__init__(self) # Required
    self.interval = interval
  def run(self):
    while 1:
      time.sleep(self.interval)
      print time.ctime(time.time())
t = PrintTime(5) # Create a thread object
t.start() # Start it
# Do something else
while(1): pass
```

/threading/ex5.py

C3PO

THREAD CLASS

- There are a variety of ways you can create threads using the Thread class.
- We cover three of them here, all quite similar.
- Pick the one you feel most comfortable with, not to mention the most appropriate for your application and future scalability:
    1. Create Thread instance, passing in function
    2. Create Thread instance, passing in callable class instance
    3. Subclass Thread and create subclass instance

1 CREATE THREAD INSTANCE, PASSING IN FUNCTION

```
#!/usr/bin/env python
import threading
from time import sleep, time, ctime
loops = [ 4, 2 ]
def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime(time())
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime(time())
def main():
    print 'starting threads...'
    threads = []
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop,
            args=(i, loops[i]))
        threads.append(t)
    for i in nloops:                # start threads
        threads[i].start()
    for i in nloops:                # wait for all
        threads[i].join()           # threads to finish
    print 'all DONE at:', ctime(time())
if __name__ == '__main__':
    main()
```

/threading/mtsleep3.py

C3PO

2 CREATE THREAD INSTANCE, PASSING IN CALLABLE CLASS INSTANCE

```
#!/usr/bin/env python
import threading
from time import sleep, time, ctime
loops = [ 4, 2 ]

class ThreadFunc:

    def __init__(self, func, args, name=''):
        self.name = name
        self.func = func
        self.args = args

    def __call__(self):
        apply(self.func, self.args)

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime(time())
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime(time())
```

```
def main():
    print 'starting threads...'
    threads = []
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread( \
                target=ThreadFunc(loop, (i, loops[i]),
                loop.__name__))
        threads.append(t)
    for i in nloops:
        threads[i].start()
    for i in nloops:
        threads[i].join()
    print 'all DONE at:', ctime(time())

if __name__ == '__main__':
    main()
```

empty list

/threading/mtsleep4.py

C3PO

__CALL__

aclass.py

```
class Aclass:
  def __call__(self):
    print 'Hi I am __call__ed';
  def __init__(self, *args, **keyargs):
    print "Hi I am __init__ed";
```

```
x = Aclass()
Hi I am __init__ed
x()
Hi I am __call__ed
```

Executing `x = Aclass()` will call `__init__()` and just `x()` will call `__call__()`.

test.py

```
class Test(object):
  def __call__(self, *args, **kwargs):
    print args
    print kwargs
    print '-'*80
t = Test()
t(1, 2, 3)
t(a=1, b=2, c=3)
t(4, 5, 6, d=4, e=5, f=6)
callable(t)
```

```
(1, 2, 3)
{}
---...
()
{'a': 1, 'c': 3, 'b': 2}
---...
-
(4, 5, 6)
{'e': 5, 'd': 4, 'f': 6}
---...
```

3 SUBCLASS THREAD AND CREATE SUBCLASS INSTANCE

```python
#!/usr/bin/env python
import threading
from time import sleep, ctime
loops = [ 4, 2 ]

class MyThread(threading.Thread):

    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        apply(self.func, self.args)

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()
```

```python
def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))
    for i in nloops:
        t = MyThread(loop, (i, loops[i]),
                loop.__name__)
        threads.append(t)
    for i in nloops:
        threads[i].start()
    for i in nloops:
        threads[i].join()

    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

/threading/mtsleep5.py C3PO

CLASS THREADING.THREAD

```
class threading.Thread(group=None, target=None,
name=None, args=(), kwargs={})
```

- Arguments are:
  - ***group:*** should be `None`; reserved for future extension when a ThreadGroup class is implemented.
  - ***target:*** is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.
  - ***name:*** is the thread name. By default, a unique name is constructed of the form **"Thread-*N*"** where *N* is a small decimal number.
  - ***args:*** is the argument tuple for the target invocation. Defaults to `()`.
  - ***kwargs:*** is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.
- If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

CLASS THREADING.THREAD METHODS

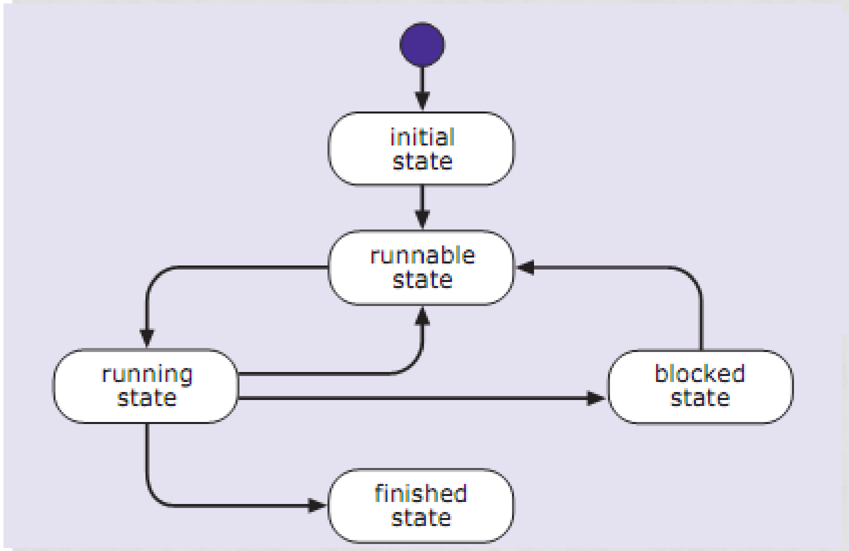
- **`start()`**
  - Start the thread's activity.
  - It must be called at most once per thread object.
  - It arranges for the object's **`run()`** method to be invoked in a separate thread of control.
  - This method will raise a **`RuntimeError`** if called more than once on the same thread object.
- **`run()`**
  - Method representing the thread's activity.
  - You may override this method in a subclass. The standard **`run()`** method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the **`args`** and **`kwargs`** arguments, respectively.

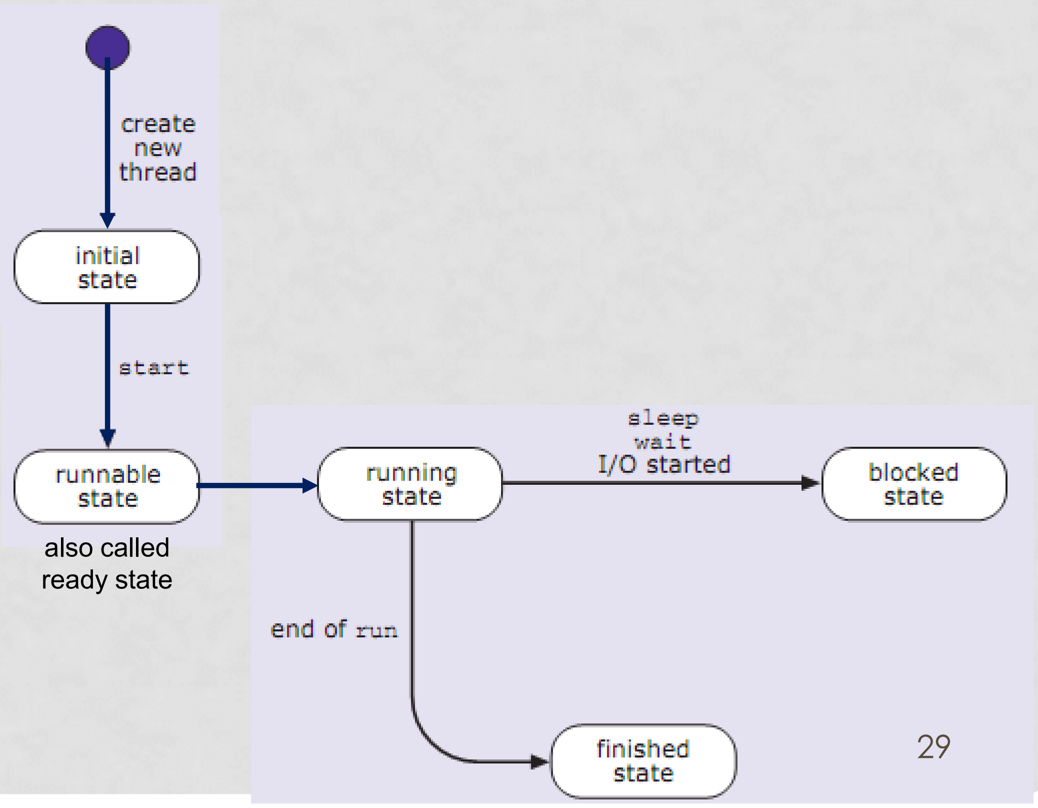
CLASS THREADING.THREAD METHODS (CONT)

- **`join([timeout])`**
  - Wait until the thread terminates. This blocks the calling thread until the thread whose **`join()`** method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.
  - When the timeout argument is present and not **`None`**, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).
  - As **`join()`** always returns **`None`**, you must call **`isAlive()`** after **`join()`** to decide whether a timeout happened – if the thread is still alive, the **`join()`** call timed out.
  - When the timeout argument is not present or **`None`**, the operation will block until the thread terminates.

THREAD STATES

- When the **`start()`** method is called, the thread is pushed into the Ready state; after that, it is controlled by the scheduler.
- When the thread get chance to run in running state then Scheduler call the **`run()`** method.





THREAD STATES (CONT)

- **Initial State.** After the creations of Thread instance the thread is in this state but before the `start()` method invocation. At this point, the thread is considered not alive.
- **Runnable state.** A thread starts its life from Runnable state. A thread first enters runnable state after invoking the `start()` method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state.** The thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from the runnable pool.
- **Dead state.** A thread can be considered dead when its `run()` method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked.** A thread can enter in this state because of waiting the resources that are hold by another thread.

CLASS THREADING.THREAD METHODS (CONT)

- The Thread class
  - When defining threads as classes all you need to supply is the following:
    - A constructor that calls `threading.Thread.__init__(self)`
    - A `run()` method that performs the actual work of the thread.
- A few additional methods are also available

```
t.getName()          # Get the name of the thread
t.setName(name)      # Set the name of the thread
t.isAlive()          # Return 1 if thread is alive.
t.isDaemon()         # Return daemonic flag
t.setDaemon(val)     # Set daemonic flag
```

- Daemon threads
  - Normally, interpreter exits only when all threads have terminated.
  - However, a thread can be flagged as a daemon thread (runs in background).
  - Interpreter really only exits when all non-daemonic threads exit.
  - Can use this to launch threads that run forever, but which can be safely killed.

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

DAEMON THREAD

Logs a message with level DEBUG on the root logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

/threading/daemon.py C3PO

```
(daemon    ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

SYNCHRONIZATION PRIMITIVES

- The threading module provides the following synchronization primitives
  - Mutual exclusion locks
  - Reentrant locks
  - Conditional variables
  - Semaphores
  - Events
- Why would you need these?
  - Threads are updating shared data structures
  - Threads need to coordinate their actions in some manner (events).
  - You need to regain some programming sanity.

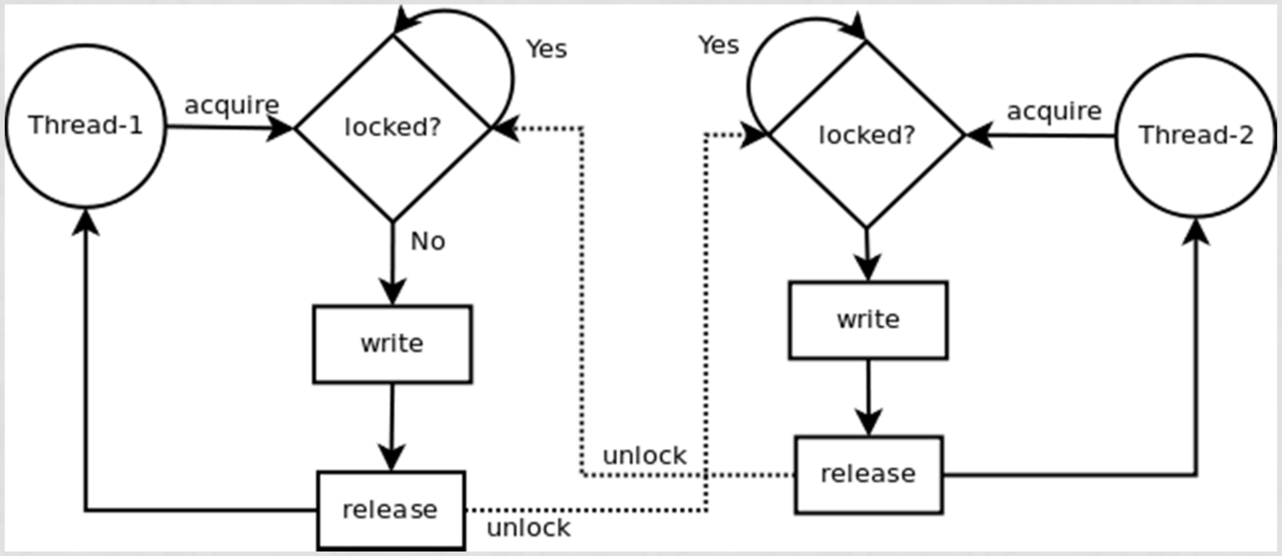
Threading Module Objects

Object	Description
Thread	Object that represents a single thread of execution
Lock	Primitive lock object (same lock as in thread module)
RLock	Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
Condition	Condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value
Event	General version of condition variables, whereby any number of threads are waiting for some event to occur and all will awaken when the event happens
Semaphore	Provides a "counter" of finite resources shared between threads; block when none are available
BoundedSemaphore	Similar to a Semaphore but ensures that it never exceeds its initial value
Timer	Similar to Thread, except that it waits for an allotted period of time before running
Barrier*	Creates a "barrier," at which a specified number of threads must all arrive before they're all allowed to continue

* New in Python 3.2

LOCK

- Locks have 2 states: locked and unlocked.
- 2 methods are used to manipulate them: **acquire()** and **release()**.
- Those are the rules:
  - if the state is unlocked: a call to **`acquire()`** changes the state to locked.
  - if the state is locked: a call to **`acquire()`** blocks until another thread calls **`release()`**.
  - if the state is unlocked: a call to **`release()`** raises a **`RuntimeError`** exception.
  - If the state is locked: a call to **`release()`** changes the state to unlocked().



LOCK OBJECTS

- The Lock object
  - Provides a simple mutual exclusion lock

```
import threading
data = []                          # Some data
lck = threading.Lock()             # Create a lock
def put_obj(obj):
  lck.acquire()
  data.append("object")
  lck.release()
def get_obj():
  lck.acquire()
  r = data.pop()
  lck.release()
  return r
```

  - Only one thread is allowed to acquire the lock at once
  - Most useful for coordinating access to shared data.

RLOCK

- RLock is a reentrant lock.
  - **`acquire()`** can be called multiple times by the same thread without blocking.
  - Keep in mind that **`release()`** needs to be called the same number of times to unlock the resource.
- Using Lock, the second call to **`acquire()`** by the same thread will block:

```
lock = threading.Lock()
lock.acquire()
lock.acquire() # block
```

- If you use RLock, the second call to **`acquire()`** won't block.

```
rlock = threading.RLock()
rlock.acquire()
rlock.acquire() # no block, execution continues as usual
```

- RLock also uses **`thread.allocate_lock()`** but it keeps track of the owner thread to support the reentrant feature.
  - Following is the RLock **`acquire()`** method implementation. If the thread calling **`acquire()`** is the owner of the resource then the counter is incremented by one. If not, it tries to acquire it. First time it acquires the lock, the owner is saved and the counter is initialized to 1.

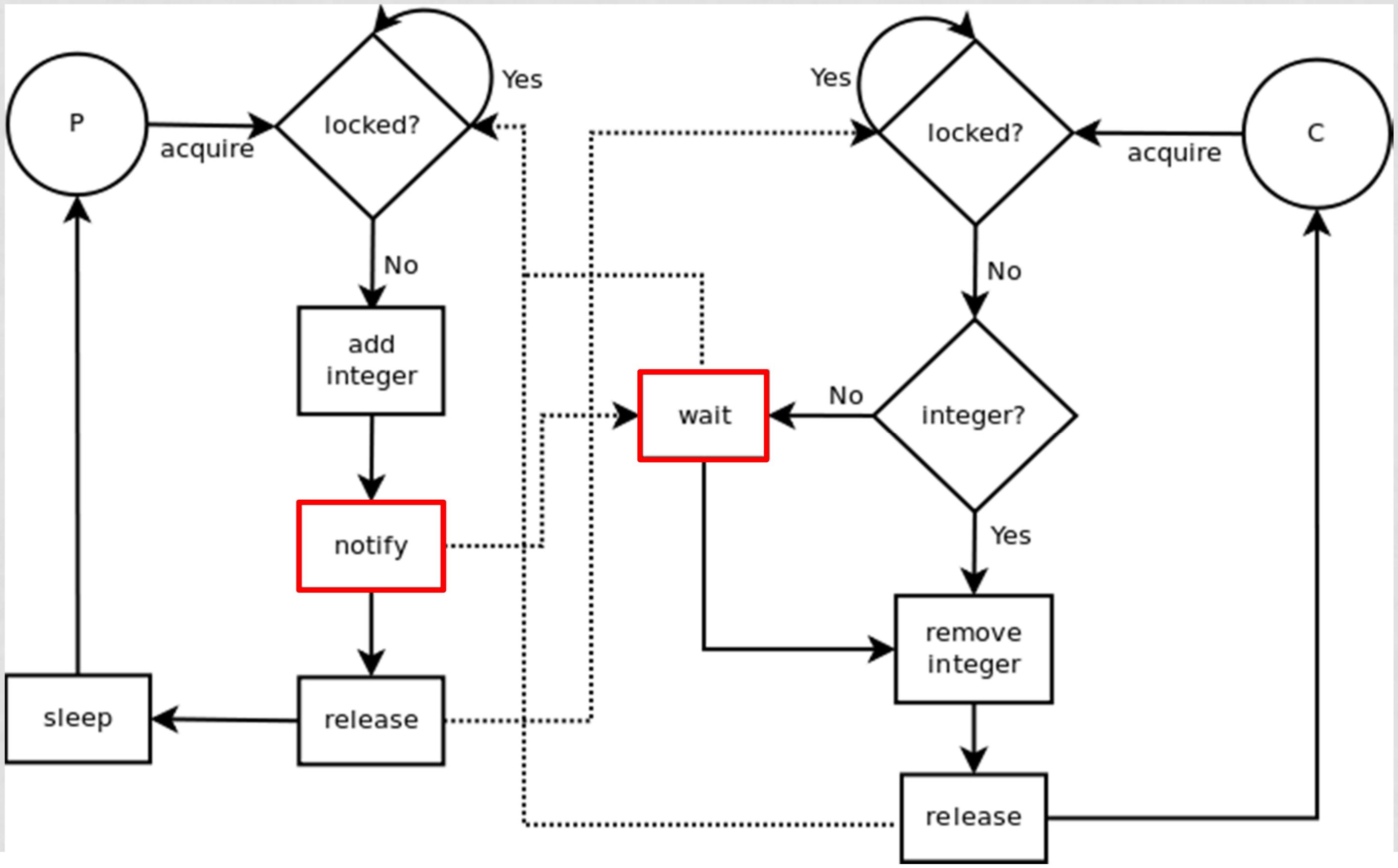
RLOCK OBJECTS

- A mutual-exclusion lock that allows repeated acquisition by the same thread
- Allows nested **acquire()**, **release()** operations in the thread that owns the lock.
- Only the outermost **release()** operation actually releases the lock.

```
import threading
data = []                          # Some data
lck = threading.Rlock()            # Create a reentrant lock
def put_obj(obj):
  lck.acquire()
  data.append("object")
  ...
  put_obj(otherobj)                # Some kind of recursion
  ...
  lck.release()
def get_obj():
  lck.acquire()
  r = data.pop()
  lck.release()
  return r
```

CONDITION

- This is a synchronization mechanism where a thread waits for a specific condition and another thread signals that this condition has happened. Once the condition happened, the thread acquires the lock to get exclusive access to the shared resource.



CONDITION VARIABLES

- Creates a condition variable.
- Synchronization primitive typically used when a thread is interested in an event or state change.
- Classic problem: producer-consumer problem.

```
# Create data queue and a condition variable
data = []
cv = threading.Condition()

# Consumer thread
def consume_item():
  cv.acquire()                     # Acquire the lock
  while not len(data):
    cv.wait()                      # Wait for data to show up;
  r = data.pop()
  cv.release()                     # Release the lock
  return r

# Producer thread
def produce_item(obj):
  cv.acquire()                     # Acquire the lock
  data.append("object")
  cv.notify()                      # Notify a consumer
  cv.release()                     # Release the lock
```

releases the lock, and then blocks until it is awakened by **`notify()`**

wakes up one of the threads waiting for the condition variable, if any are waiting; does not release the lock

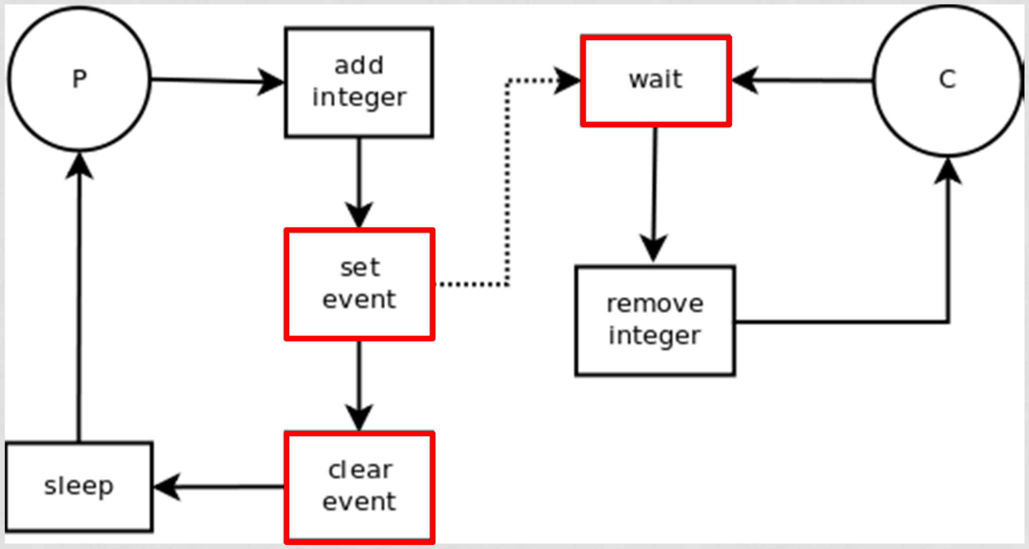
SEMAPHORE OBJECTS

- Semaphores
  - A locking primitive based on a counter.
  - Each **`acquire()`** method decrements the counter.
  - Each **`release()`** method increments the counter.
  - If the counter reaches zero, future **`acquire()`** methods block.
  - Common use: limiting the number of threads allowed to execute code

```
sem = threading.Semaphore(5)      # No more than 5 threads allowed
def fetch_file(host,filename):
   sem.acquire()                  # Decrements count or blocks if zero
   ...
   blah
   ...
   sem.release()                  # Increment count
```

EVENT

- This is a simple mechanism. A thread signals an event and the other thread(s) wait for it.
- An event object manages an internal flag that can be set to true with the **`set()`** method and reset to false with the **`clear()`** method. The **`wait()`** method blocks until the flag is true.



EVENT OBJECTS

- Events
  - A communication primitive for coordinating threads.
  - One thread signals an "event"
  - Other threads wait for it to happen.

```
# Create an event object
e = Event()
# Signal the event
def signal_event():
  e.set()
# Wait for event
def wait_for_event():
  e.wait()
# Clear event
def clear_event():
  e.clear()
```

  - Similar to a condition variable, but all threads waiting for event are awakened.

LOCKS AND BLOCKING

- By default, all locking primitives block until lock is acquired
  - In general, this is uninterruptible.
- Fortunately, most primitives provide a non-blocking option

```
if not lck.acquire(0):
    # lock couldn't be acquired!
```

  - This works for Lock, RLock, and Semaphore objects
- Timeouts
  - Condition variables and events provide a timeout option

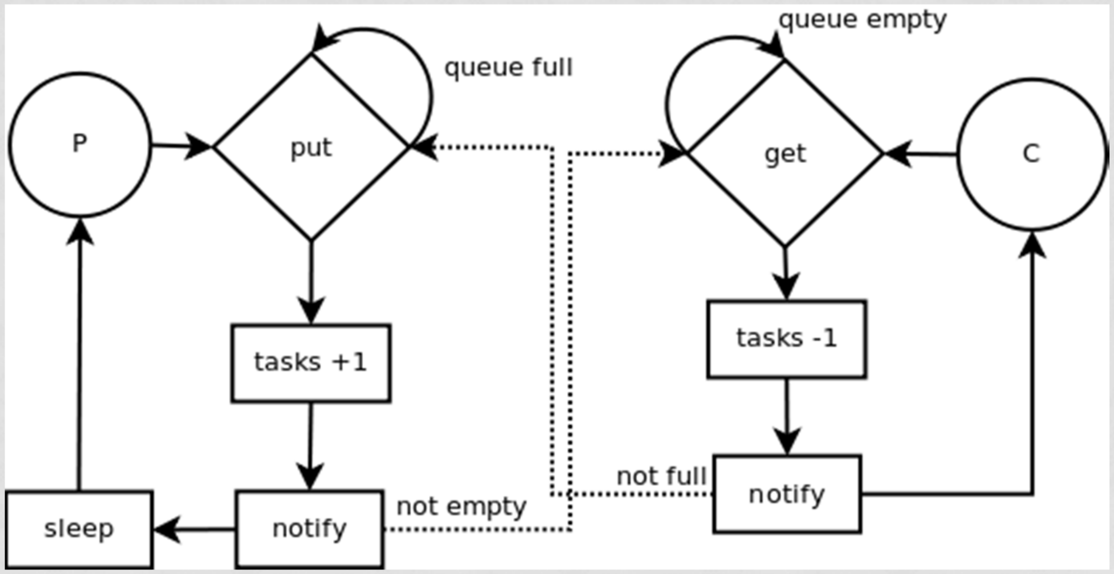
```
cv = Condition()
...
cv.wait(60.0)  # Wait 60 seconds for notification
```

- On timeout, the function simply returns. Up to caller to detect errors.

PYTHON QUEUE

QUEUE

- Queues are a great mechanism when we need to exchange information between threads as it takes care of locking for us.
- We are interested in the following 4 Queue methods:
    - **`put`**: Put an item to the queue.
    - **`get`**: Remove and return an item from the queue.
    - **`task_done`**: Needs to be called each time an item has been processed.
    - **`join`**: Blocks until all items have been processed.



THE QUEUE MODULE

- Provides a multi-producer, multi-consumer FIFO queue object
  - Can be used to safely exchange data between multiple threads

```
q = Queue(maxsize)    # Create a queue
q.qsize()             # Return current size
q.empty()             # Test if empty
q.full()              # Test if full
q.put(item)           # Put an item on the queue
q.get()               # Get item from queue
```

- Notes:
  - The Queue object also supports non-blocking put/get.

```
q.put_nowait("object")
q.get_nowait()
```

  - These raise the **`Queue.Full`** or **`Queue.Empty`** exceptions if an error occurs.
  - Return values for **`qsize()`**, **`empty()`**, and **`full()`** are approximate.

BASIC FIFO QUEUE

- The Queue class implements a basic first-in, first-out container. Elements are added to one "end" of the sequence using **`put()`**, and removed from the other end using **`get()`**.

```
import Queue

q = Queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get()
```

basic_fifo_queue.py C3PO

LIFO QUEUE

- In contrast to the standard FIFO implementation of Queue, the **`LifoQueue`** uses last-in, first-out ordering (normally associated with a stack data structure).

```
import Queue

q = Queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print q.get()
```

lifo_queue.py C3PO

PRIORITY QUEUE

- Sometimes the processing order of the items in a queue needs to be based on characteristics of those items, rather than just the order they are created or added to the queue.
- For example, print jobs from the payroll department may take precedence over a code listing printed by a developer.
- **`PriorityQueue`** uses the sort order of the contents of the queue to decide which to retrieve.

PRIORITY QUEUE (2)

```
import Queue

class Job(object):
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print 'New job:', description
        return
    def __cmp__(self, other):
        return cmp(self.priority, other.priority)

q = Queue.PriorityQueue()

q.put( Job(3, 'Mid-level job') )
q.put( Job(10, 'Low-level job') )
q.put( Job(1, 'Important job') )

while not q.empty():
    next_job = q.get()
    print 'Processing job:', next_job.description
```

priority_queue.py C3PO

FINAL COMMENTS ON THREADS

- Python threads are quite functional
  - Can write applications that use dozens (or even hundreds) of threads
- But there are performance issues
  - Global interpreter lock makes it difficult to fully utilize multiple CPUs.
  - You don't get the degree of parallelism you might expect.
- Interaction with C extensions
  - Common problem: I wrote a big C extension and it broke threading.
  - The culprit: Not releasing global lock before starting a long-running function.
- Not all modules are thread-friendly
  - Example: **`gethostbyname()`** blocks all threads if nameserver down.

EXAMPLES

THREAD EXAMPLE USING THE RPI (PUSH-BUTTON CIRCUIT)

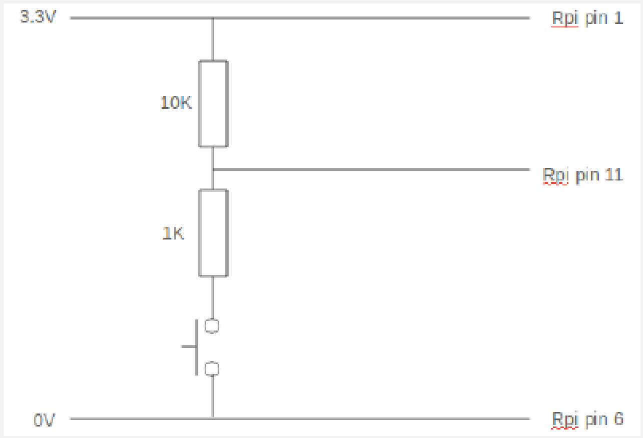
```
import threading
import time
import RPi.GPIO as GPIO

class Button(threading.Thread):
 """A Thread that monitors a GPIO button"

  def __init__(self, channel):
    threading.Thread.__init__(self)
    self._pressed = False
    self.channel = channel

    # set up pin as input
    GPIO.setup(self.channel, GPIO.IN)

    # A program will exit when only daemon threads are left alive
    self.daemon = True
    # start thread running
    self.start()
```



Push-button circuit experiment wiring diagram

```
def pressed(self):
  if self._pressed:
    # clear the pressed flag now we have detected it
    self._pressed = False
    return True
  else:
    return False

def run(self):
  previous = None
  while 1:
    # read gpio channel
    current = GPIO.input(self.channel)
    time.sleep(0.01) # wait 10 ms

    # detect change from 1 to 0 (a button press)
    if current == False and previous == True:
      self._pressed = True

      # wait for flag to be cleared
      while self._pressed:
        time.sleep(0.05) # wait 50 ms
    previous = current
```

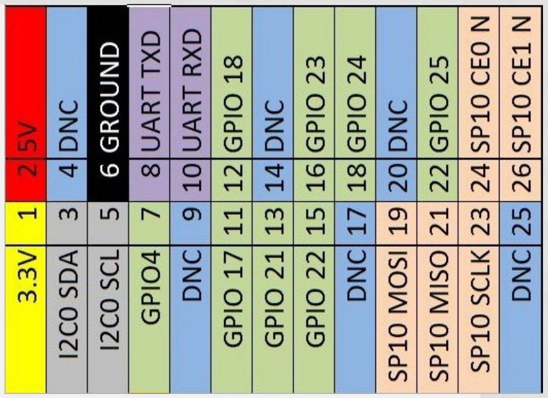
THREAD EXAMPLE USING THE RPI (CONT)

```
def onButtonPress():
  print('Button has been pressed!')

# create a button thread for a button on pin 11
button = Button(11)

try:
  while True:
    # ask for a name and say hello
    name = input('Enter a name (or Q to quit): ')
    if name.upper() == ('Q'):
      break
    print('Hello', name)
    # check if button has been pressed
    if button.pressed():
      onButtonPress()

except KeyboardInterrupt:  # trap a CTRL+C keyboard interrupt
    GPIO.cleanup()
GPIO.cleanup()
```

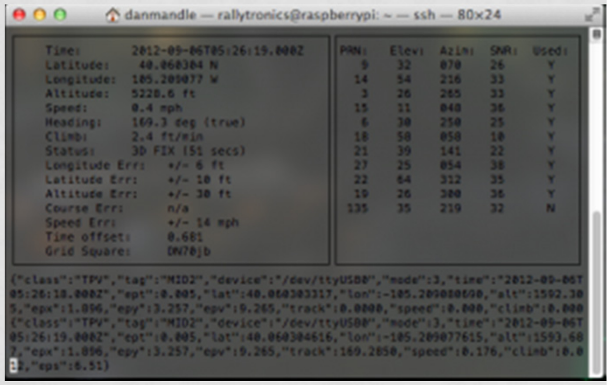


3.3V	1	2	5V
I2C0 SDA	3	4	DNC
I2C0 SCL	5	6	GROUND
GPIO4	7	8	UART TXD
DNC	9	10	UART RXD
GPIO 17	11	12	GPIO 18
GPIO 21	13	14	DNC
GPIO 22	15	16	GPIO 23
DNC	17	18	GPIO 24
SP10 MOSI	19	20	DNC
SP10 MISO	21	22	GPIO 25
SP10 SCLK	23	24	SP10 CE0 N
DNC	25	26	SP10 CE1 N

GPS TRACKER/RECORDER

- Required Hardware
  - Raspberry Pi with Debian Wheezy installed
  - GPSd compatible GPS Receiver
- Getting the Software
  - Enter the following command to install Python, GPSd, and the Python modules to bring them together:
  - **`sudo apt-get install python gpsd gpsd-clients`**
- Plugging in the USB receiver should start GPSd automatically.
- To make sure that GPSd is playing nice, you can open cgps to see what data it is receiving.
  - **`cgps`**

GlobalSat BU-353 USB GPS Navigation Receiver



```
    Time:        2012-09-06T05:26:19.000Z        PRN:   Elev:   Azim:   SNR:   Used:
    Latitude:     40.060304 N                       9     32     070     26       Y
    Longitude:   105.209077 W                      14     54     216     33       Y
    Altitude:    5220.6 ft                          3     26     265     33       Y
    Speed:       0.4 mph                           15     11     040     36       Y
    Heading:     169.3 deg (true)                   6     30     250     25       Y
    Climb:       2.4 ft/min                        18     58     050     10       Y
    Status:      3D FIX (51 secs)                  21     39     141     22       Y
    Longitude Err:   +/- 6 ft                      27     25     054     38       Y
    Latitude Err:    +/- 10 ft                     22     64     312     35       Y
    Altitude Err:    +/- 30 ft                     19     26     300     36       Y
    Course Err:      n/a                          135     35     219     32       N
    Speed Err:       +/- 14 mph
    Time offset:     0.681
    Grid Square:     DN70jb

{"class":"TPV","tag":"MID2","device":"/dev/ttyUSB0","mode":3,"time":"2012-09-06T
05:26:18.000Z","ept":0.005,"lat":40.060303317,"lon":-105.209080690,"alt":1592.30
5,"epx":1.096,"epy":3.257,"epv":9.265,"track":0.0000,"speed":0.000,"climb":0.000
{"class":"TPV","tag":"MID2","device":"/dev/ttyUSB0","mode":3,"time":"2012-09-06T
05:26:19.000Z","ept":0.005,"lat":40.060304616,"lon":-105.209077615,"alt":1593.68
2,"epx":1.096,"epy":3.257,"epv":9.265,"track":169.2850,"speed":0.176,"climb":0.0
12,"eps":6.51}
```

http://www.danmandle.com/blog/getting-gpsd-to-work-with-python/

```
import os
from gps import *
from time import *
import time
import threading

gpsd = None                                   #seting the global variable

os.system('clear')                            #clear the terminal (optional)

class GpsPoller(threading.Thread):
  def __init__(self):
    threading.Thread.__init__(self)
    global gpsd                               #bring it in scope
    gpsd = gps(mode=WATCH_ENABLE)             #starting the stream of info
    self.current_value = None
    self.running = True                       #setting the thread running to true

  def run(self):
    global gpsd
    while gpsp.running:
      gpsd.next() #this will continue to loop and grab EACH set of gpsd info to clear the buffer
```

```
if __name__ == '__main__':
  gpsp = GpsPoller()                                          # create the thread
  try:
    gpsp.start()                                              # start it up
    while True:
      #It may take a second or two to get good data
      os.system('clear')
      print
      print ' GPS reading'
      print '----------------------------------------'
      print 'latitude    ' , gpsd.fix.latitude
      print 'longitude   ' , gpsd.fix.longitude
      print 'time utc    ' , gpsd.utc,' + ', gpsd.fix.time
      print 'altitude (m)' , gpsd.fix.altitude
      print 'eps         ' , gpsd.fix.eps
      print 'epx         ' , gpsd.fix.epx
      print 'epv         ' , gpsd.fix.epv
      print 'ept         ' , gpsd.fix.ept
      print 'speed (m/s) ' , gpsd.fix.speed
      print 'climb       ' , gpsd.fix.climb
      print 'track       ' , gpsd.fix.track
      print 'mode        ' , gpsd.fix.mode
      print
      print 'sats        ' , gpsd.satellites
      time.sleep(5)                                           #set to whatever
  except (KeyboardInterrupt, SystemExit):                     #when you press ctrl+c
    print "\nKilling Thread..."
    gpsp.running = False
    gpsp.join()  # wait for the thread to finish what it's doing
    print "Done.\nExiting."
```