Threads

Multithreaded programming

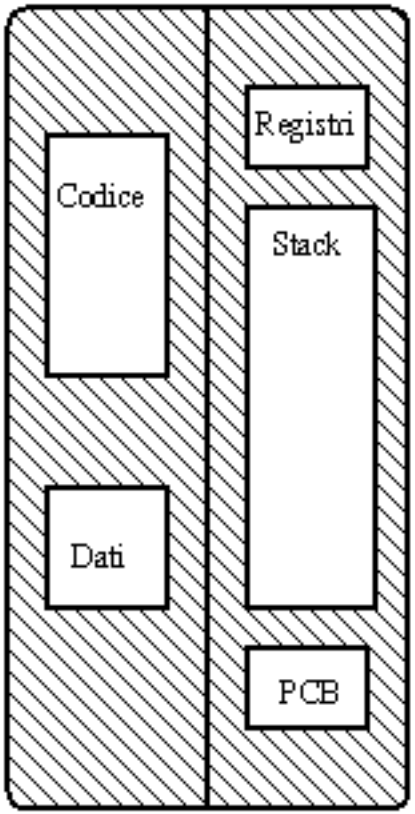
- A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system. Threads are normally created by a fork of a computer script or program in two or more parallel (which is implemented on a single processor by multitasking) tasks.

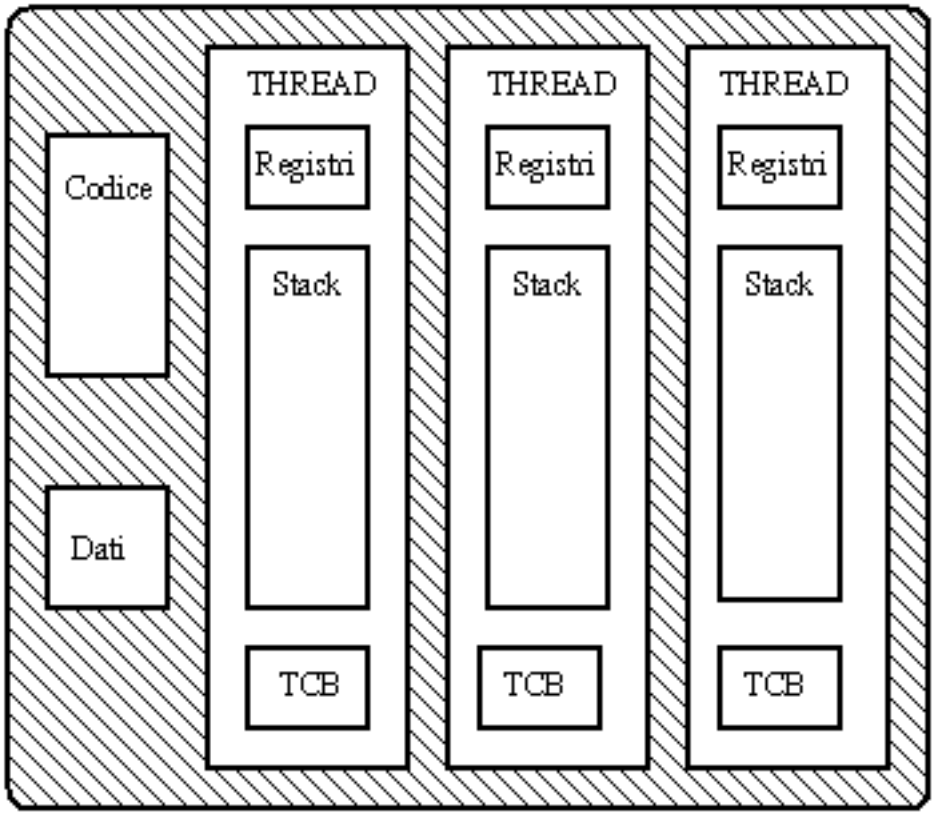
- Threads are usually contained in processes. More than one thread can exist within the same process. These threads **share the memory and the state of the process**. In other words: They share the code or instructions and the values of its variables.

- Every process has at least one thread, i.e. the process itself. A process can start multiple threads. The operative system executes these threads like parallel "processes". On a single processor machine, this parallelism is achieved by thread scheduling or timeslicing.

Single Thread

Multiple Threads





Advantages of Threading

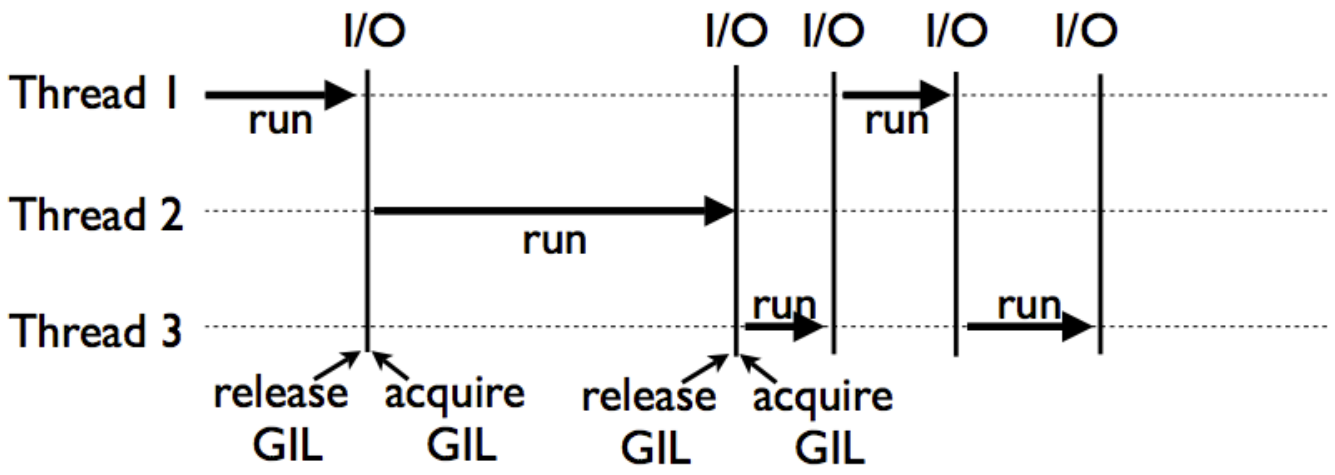
- Multithreaded programs can run faster on computer systems with multiple CPUs, because these threads can be executed truly concurrently.
- A program can remain responsive to input. This is true both on single and on multiple CPU
- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

Python Threads

- Python threads are REAL system threads, fully managed by the host operating system (Posix threads pthreads and Windows threads)
- Represent threaded execution of the Python interpreter process (written in C)
- **BUT**: **parallel execution is forbidden**! There is a "Global Interpreter Lock" (GIL from now), that ensures that only one thread runs in the interpreter at once
- This simplifies many low-level details (memory management, callouts to C extension…)

GIL

With the GIL, you get what you can call a "**cooperative multitasking**": when a thread is running, it holds the GIL, and **release it on I/O** (read, write, send, recv, etc…)



As you can imagine, CPU-bond threads that NEVER perform I/O must be handled as a special case…
Every 100 "ticks" a check occurs. Ticks loosely map to interpreter instructions, and are not related to timing

GIL

- The periodic check is really simple. The currently running thread:
  1. Reset the tick counter
  2. Runs signal handlers if the main thread
  3. Releases the GIL
  4. Reacquires the GIL

- The GIL is at least controversial: it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems in certain situations.

- Note that potentially blocking or long-running operations, such as I/O, image processing, and Numpy number crunching, happen ***outside*** the GIL. **Therefore it is only in multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a bottleneck**.

GIL

- However the GIL degrades performance even when it is not a bottleneck. The system call overhead is significant, especially on multicore hardware. Two threads calling a function may take twice as much time as a single thread calling the function twice. The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered.

- There are also non-CPython implementations
  1. Jython and IronPython have no GIL and can fully exploit multiprocessor systems
  2. PyPy currently has a GIL like Cpython
  3. In Cython the GIL exists, but can be released temporarily using a "with" statement

The Threading module

To get informations:

```
>>> import threading
>>> dir(threading)
>>> help(threading)
>>> dir(threading.Thread)
>>> help(threading.Thread)
>>> import pydoc
>>> pydoc.gui()
```

There are various ways to use the thread class, here we show two of them :

1. Create a Thread instance with a custom function
2. Create a subclass of Thread and then instantiate it

Obviously the second method is more suitable if you want an interface more object-oriented.

Table 4-2 `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`[a]	Creates a "barrier," at which a specified number of threads must all arrive before they're all allowed to continue

http://bit.do/t1-py

```
import time
from threading import Thread

loops=[4,2,3]

def sleeper(nloop, nsec):
    print "thread %d sleeps for %s seconds" % (nloop, nsec)
    sleep(nsec)
    print "thread %d woke up" % nloop


def main():
    threads =[]
    nloops=range(len(loops))
    for i in nloops:
        t=threading.Thread(target=sleeper,args=(i,loops[i]))
        threads.append(t)

    for i in nloops:
        threads[i].start()
    for i in nloops:
        threads[i].join()

main()
```

Create the Thread objects to which pass the function to run

Start the threads together

You have to call `join()`

The class threading.Thread has a method `start()`, which can start a Thread.
It triggers off the method `run()`, which has to be overloaded.
The `join()` method makes sure that the main program waits until all threads have terminated.

http://bit.do/t2-py

```
import time
from threading import Thread

loops=[4,2,3]

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):
        self.func(*self.args)

def sleeper(nloop, nsec):
   print "thread %d sleeps for %s seconds" % (nloop, nsec)
   sleep(nsec)
   print "thread %d woke up" % nloop

def main():
    threads =[]
    nloops=range(len(loops))
    for i in nloops:
        t=MyThread(sleeper,(i,loops[i]),sleeper.__name__)
        threads.append(t)
    for i in nloops:
        threads[i].start()
    for i in nloops:
        threads[i].join()
main()
```

A subclass allows more flexibility

You have to explicitly call the constructor of the parent class

```
>>> sleeper.__name__
'sleeper'
```

Synchronizing Access to Shared Resources

- Until now, however, we have not seen how to **synchronize** threads: some operations, such as changing a database or update a file, can not be performed in a consistent manner from multiple threads simultaneously, unless the threads are not synchronized with each other to avoid occurrence of **race conditions**

- If you're not careful, overlapping accesses or modifications from multiple threads may cause all kinds of problems, and what's worse, those problems have a tendency of appearing only under heavy load, or on your production servers, or on some faster hardware that's only used by one of your customers.

Atomic operations

- The simplest way to synchronize access to shared variables or other resources is to rely on atomic operations in the interpreter. An atomic operation is an operation that is carried out in a single execution step, without any chance that another thread gets control.

- In general, this approach only works if the shared resource consists of a single instance of a core data type, such as a string variable, a number, or a list or dictionary. Here are some thread-safe operations:
  - reading **or** replacing a single instance attribute
  - reading **or** replacing a single global variable
  - fetching an item from a list
  - modifying a list in place (e.g. adding an item using **append**)
  - fetching an item from a dictionary
  - modifying a dictionary in place (e.g. adding an item, or calling the **clear** method)

Locks

- Note that operations that read a variable or attribute, modifies it, and then writes it back are not thread-safe. Another thread may update the variable after it's been read by the current thread, but before it's been updated.

- Locks are the most fundamental synchronization mechanism provided by the **threading** module. At any time, a lock can be held by a single thread, or by no thread at all. If a thread attempts to hold a lock that's already held by some other thread, execution of the first thread is halted until the lock is released.

- Locks are typically used to synchronize access to a shared resource. For each shared resource, create a **Lock** object. When you need to access the resource, call **acquire** to hold the lock (this will wait for the lock to be released, if necessary), and call **release** to release it.

Locks

- When multiple threads contend for a lock on a resource, the first blocks it, and is allowed to use it. The other threads that come after are **locked** until the first thread exits the critical code section and releases the lock. At that point one of the waiting threads takes over.

- **There is NO order for locked thread, as FIFO or LIFO**: the selection of the next thread is not deteministic and may vary depending on the Python implementation.

- The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

```
lock = Lock()
lock.acquire() # will block if lock is already held
    ... access shared resource
lock.release()
```

For proper operation, it's important to release the lock even if something goes wrong when accessing the resource. You can use **try-finally** for this purpose:

```
lock.acquire()
try:
    ... access shared resource
finally:
    lock.release() # release lock, no matter what
```

You can also use the **with** statement. **When used with a lock, this statement automatically acquires the lock before entering the block, and releases it when leaving the block**:

```
with lock:
    ... access shared resource
```

http://bit.do/t3-py

```
#!/usr/bin/env python
import threading
some_var = 0
class IncrementThread(threading.Thread):
    def run(self):
        global some_var
        read_value = some_var
        print "some_var in %s is %d" % (self.name, read_value)
        some_var = read_value + 1
        print "some_var in %s after increment is %d" % (self.name,
some_var)

def use_increment_thread():
    threads = []
    for i in range(50):
        t = IncrementThread()
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    print "After 50 modifications, some_var should have become 50"
    print "After 50 modifications, some_var is %d" % (some_var,)

use_increment_thread()
```

If you run the script multiple times you will find that **the result will vary**.
There is a race condition when two threads read the same value of the global variable, then increase it separately and write the same output value twice.

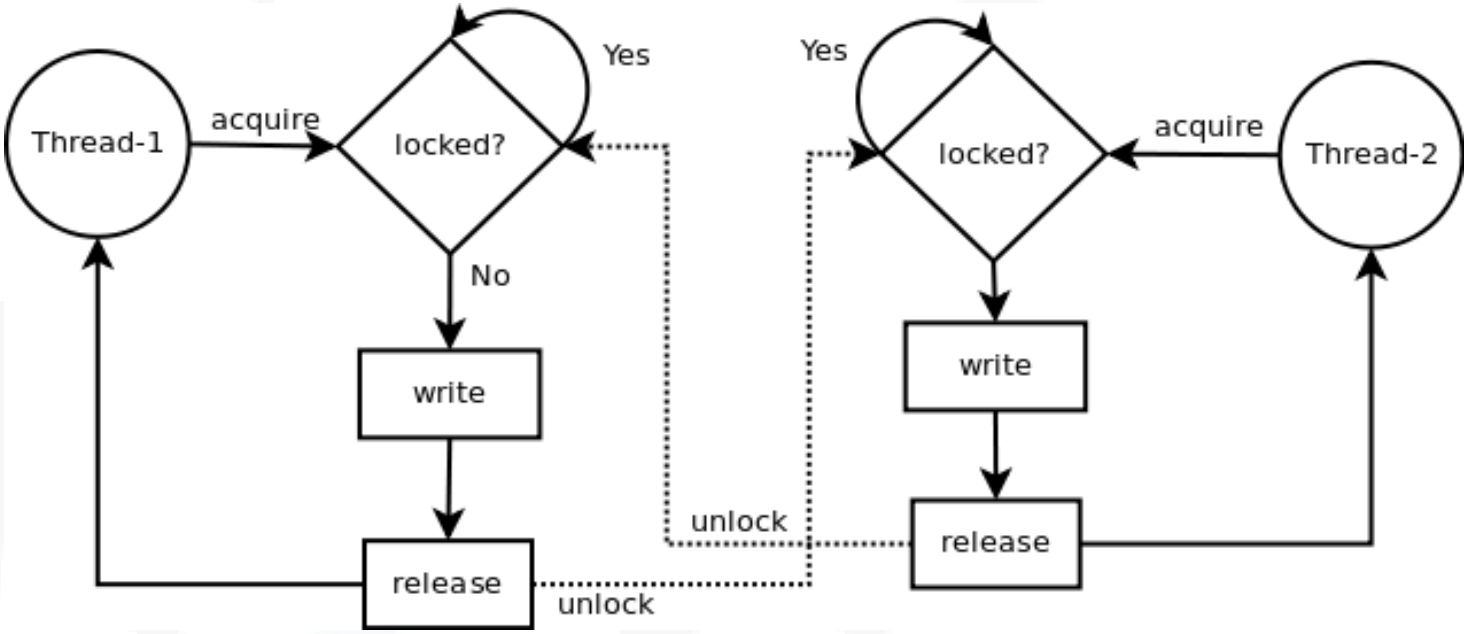
http://bit.do/t4-py

```
#!/usr/bin/env python
import threading
lock=threading.Lock()
some_var = 0
class IncrementThread(threading.Thread):
    def run(self):
        global some_var
        lock.acquire()
        read_value = some_var
        print "some_var in %s is %d" % (self.name, read_value)
        some_var = read_value + 1
        print "some_var in %s after increment is %d" % (self.name,
some_var)
        lock.release()

def use_increment_thread():
    threads = []
    for i in range(50):
        t = IncrementThread()
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    print "After 50 modifications, some_var should have become 50"
    print "After 50 modifications, some_var is %d" % (some_var,)

use_increment_thread()
```

Now reading and writing of global variable is protected by lock, and there cannot be race conditions



- If the status is *unlocked*: *acquire()* changes it in locked
- If the status is *locked*: *acquire()* is blocked until another thread calls *release()*
- If the status is *unlocked*: *release()* raises an exception
- If the status is *locked*: *release()* changes it in *unlocked()*

http://bit.do/Thread1-py

```
class FetchUrls(threading.Thread):
    ...
    def run(self):
        while self.urls:
            url = self.urls.pop()
            req = urllib2.Request(url)
            try:
                d = urllib2.urlopen(req)
            except urllib2.URLError, e:
                print 'URL %s failed: %s' % (url, e.reason)
            self.output.write('\n\nwrite START by %s\n\n' %  self.name)
            self.output.write(d.read())
            self.output.write('\n\nwrite END by %s\n\n' %  self.name)
```

This is an example were two threads write in the same output file. This first code does not use locking.

You can check the correct writes searching 'Thread' in the output file.
You will easily find situation like this:

write START by Thread-1
write START by Thread-2
write END by Thread-1
write END by Thread-2

http://bit.do/Thread2-py

```
class FetchUrls(threading.Thread):
    ...
    def __init__(self, urls, output, lock):
        ...
        self.lock=lock

    def run(self):
            ...
            self.lock.acquire()
            print 'lock acquired by %s' % self.name
            self.output.write('\n\nwrite START by %s\n\n' %  self.name)
            self.output.write(d.read())
            self.output.write('\n\nwrite END by %s\n\n' %  self.name)
            print 'write done by %s' %  self.name
            print 'lock released by %s' % self.name
            self.lock.release()
            print 'URL %s fetched by %s' % (url, self.name)
def main():
    lock = threading.Lock()
    ...
    t1 = FetchUrls(urls1, f, lock)
    t2 = FetchUrls(urls2, f, lock)
    ...
```

Now the write and read sequences are always correct, the access to the shared resource "file" is protected by lock

http://bit.do/Thread3-py

```
class FetchUrls(threading.Thread):
    ...
    def run(self):
        ...
        while self.urls:
            ...
            with self.lock:
                print 'lock acquired by %s' % self.name
                ...
                print 'write done by %s' % self.name
                print 'lock released by %s' % self.name
            ...
```

Under the `with` block you can write the code protected in the previous file by the acquire and release methods. `acquire()` is called when the execution enter into the `with` code, and `release()` is called when the execution exits.

Note that the lock is relative only to the file access, not to the url fetching.

RLock

- The **RLock** class is a version of simple locking that only blocks if the lock is held by *another* thread. While simple locks will block if the same thread attempts to acquire the same lock twice, a re-entrant lock only blocks if another thread currently holds the lock. If the current thread is trying to acquire a lock that it's already holding, execution continues as usual.

- The main use for this is nested access to shared resources. Note that this lock keeps track of the recursion level, so you still need to call **release** once for each call to **acquire**.

Semaphores

- A semaphore is a more advanced lock mechanism. A semaphore has an internal counter rather than a lock flag, and it only blocks if more than a given number of threads have attempted to hold the semaphore. Depending on how the semaphore is initialized, this allows multiple threads to access the same code section simultaneously. Semaphores are typically used to limit access to resource with limited capacity, such as a network connection or a database server.

- The semaphores are essentially **counters**, which are decremented when the resource control is "consumed", and incremented when it is "released"

- Python's **threading** module provides two semaphore implementations; the **Semaphore** class provides an unlimited semaphore which allows you to call **release** any number of times to increment the counter.
- To avoid simple programming errors, it's usually better to use the **BoundedSemaphore** class, which considers it to be an error to call **release** more often than you've called **acquire**.

http://bit.do/Semaphores-py

```
import threading, urllib2, time, random
class GrabUrl(threading.Thread):
    def __init__(self, arg0, pool):
        threading.Thread.__init__(self)
        self.host = arg0
        self.pool = pool
    def run(self):
        k = random.randint(10, 20)
        print "Processing " + self.host + " waiting for : " + str(k)
        time.sleep(k)
        print "exiting " + self.host
        self.pool.release()

class Handler(threading.Thread):
    def __init__(self, pool):
        threading.Thread.__init__(self)
        self.pool = pool
    def run(self):
        for i in hosts:
            self.pool.acquire()
            graburl = GrabUrl(i, self.pool)
            graburl.setDaemon(True)
            graburl.start()

maxconn = 2
pool = threading.BoundedSemaphore(value=maxconn)
hosts = ["http://yahoo.com", "http://google.com", "http://amazon.com",
"http://ibm.com", "http://apple.com"]
handler = Handler(pool)
handler.start()
handler.join()
print "exiting main"
```

This is only a placeholder…

The main thread can exit without waiting (no join here)

Max # of connections

There is 1 Handler thread that launches the GrabURL daemon threads.

Synchronization Between Threads: Conditions

- A condition represents some kind of state change in the application, and a thread can wait for a given condition, or signal that the condition has happened.

- To associate the condition with an existing lock, pass the lock to the **Condition** constructor. This is also useful if you want to use several conditions for a single resource.

- A great way to show the mechanism is to use a classic design pattern, "*producer / consumer*": the producer adds a number to a list of random integers at **random intervals**, and the consumer must retrieve these integers from the list.

http://bit.do/Conditions-py

```
class Producer(threading.Thread):
    def __init__(self, integers, condition):
        threading.Thread.__init__(self)
        self.integers = integers
        self.condition = condition

    def run(self):
        while True:
            integer = random.randint(0, 256)
            self.condition.acquire()
            print 'condition acquired by %s' % self.name
            self.integers.append(integer)
            print '%d appended to list by %s' % (integer, self.name)
            print 'condition notified by %s' % self.name
            self.condition.notify()
            print 'condition released by %s' % self.name
            self.condition.release()
            pause=random.randint(1,5)
            time.sleep(pause)
```

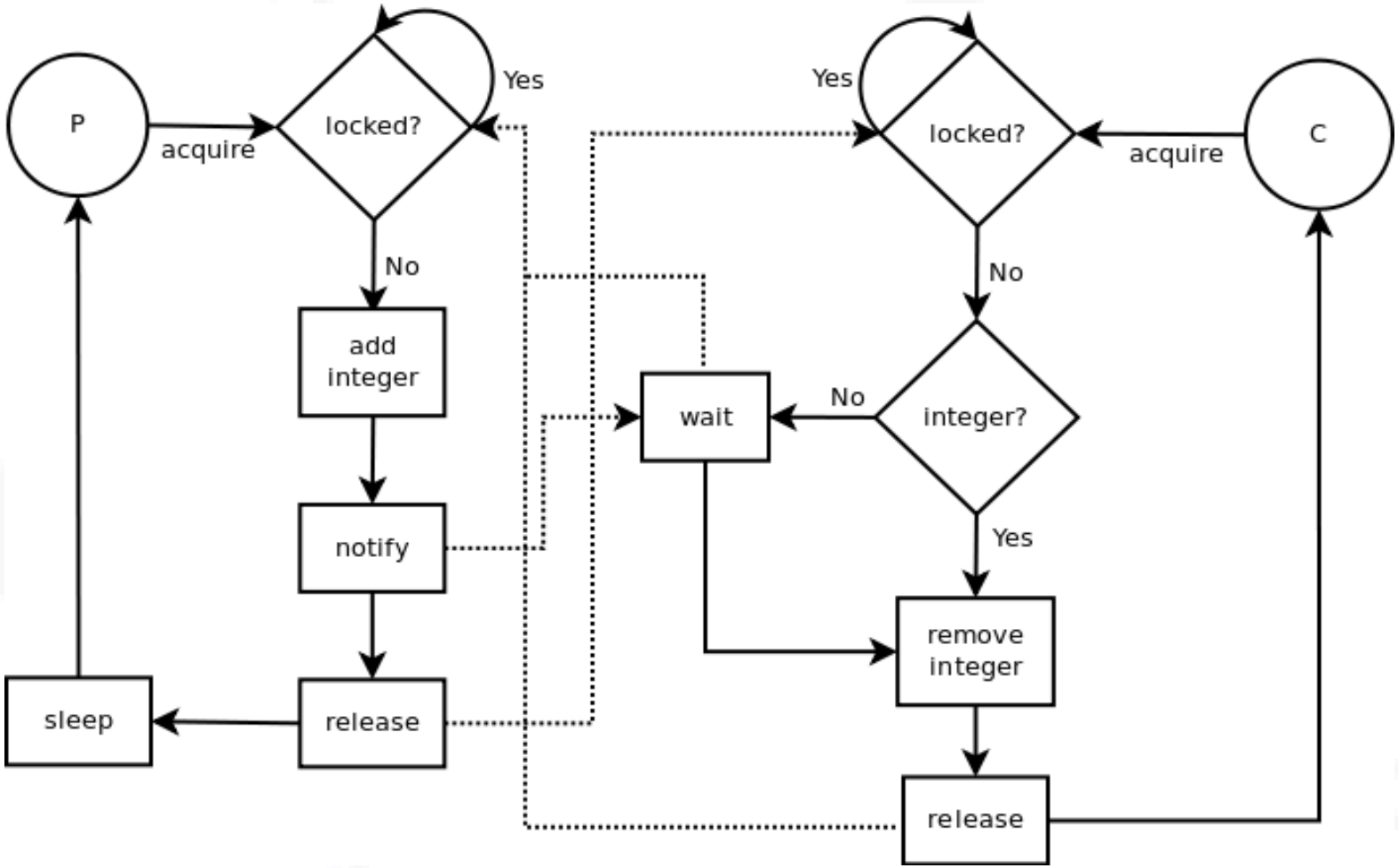
The `producer` acquires the lock, adds an integer, warns the consumer thread that something new to eat and releases the lock.
It goes on like this endlessly, with random intervals between 1 and 5 seconds between subsequent insert.

http://bit.do/Conditions-py

```
class Consumer(threading.Thread):
    def __init__(self, integers, condition):
        threading.Thread.__init__(self)
        self.integers = integers
        self.condition = condition

    def run(self):
        while True:
            self.condition.acquire()
            print 'condition acquired by %s' % self.name
            while True:
                if self.integers:
                    integer = self.integers.pop()
                    print '%d popped from list by %s' % (integer, self.name)
                    break
                print 'condition wait by %s' % self.name
                self.condition.wait()
            print 'condition released by %s' % self.name
            self.condition.release()
```

The consumer acquires the lock and check if there's an integer in the list. If in the list there is nothing it waits (`wait()`) for the notification of the producer. When he receives it download the integer from the list and releases the lock.
The **wait** method releases the lock, blocks the current thread until another thread calls **notify** or **notifyAll** on the same condition, and then reacquires the lock. If multiple threads are waiting, the **notify** method only wakes up one of the threads, while **notifyAll** always wakes them all up.


P
acquire
locked?
Yes
No
add integer
notify
release
sleep
wait
No
integer?
Yes
remove integer
release
Yes
locked?
acquire
C
No

http://bit.do/Conditions-py

```
def main():
    integers = []
    condition = threading.Condition()
    t1 = Producer(integers, condition)
    t2 = Consumer(integers, condition)
    t1.start()
    t2.start()
    t1.join()
    t2.join()

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

```
condition acquired by Thread-1
118 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
condition acquired by Thread-2
118 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
condition acquired by Thread-1
117 appended to list by Thread-1
condition notified by Thread-1
condition released by Thread-1
117 popped from list by Thread-2
condition released by Thread-2
condition acquired by Thread-2
condition wait by Thread-2
```

Thread1 "*producer*" adds 118 to the list, and notifies the thread2 "*consumer*".

Thread 2 acquires the lock, takes the 118 and releases the lock.

At that moment Thread1 is still waiting (random sleep), so Thread2 re-acquires the lock and waits.

The resource is then released, and when the Thread1 sleep ends Thread1 acquires the lock and adds 117.

Repeat…

http://bit.do/Conditions2-py

```
class Producer(threading.Thread):
    ...
    def run(self):
        while True:
            integer = random.randint(0, 256)
            with self.condition:
                print 'condition acquired by %s' % self.name
                self.integers.append(integer)
                print '%d appended to list by %s' % (integer, self.name)
                print 'condition notified by %s' % self.name
                self.condition.notify()
                print 'condition released by %s' % self.name
            pause=random.randint(1,5)
            time.sleep(pause)
class Consumer(threading.Thread):
    ...
    def run(self):
        while True:
            with self.condition:
                print 'condition acquired by %s' % self.name
                while True:
                    if self.integers:
                        integer = self.integers.pop()
                        print '%d popped from list by %s' % (integer, self.name)
                        break
                    print 'condition wait by %s' % self.name
                    self.condition.wait()
                print 'condition released by %s' % self.name
```