Outline

MCS 260 Lecture 35
Introduction to Computer Science
Jan Verschelde, 8 April 2016

lifecycle of a thread
multithreaded programming

concurrency and parallelism

First some terminology:

- **concurrency**

  Concurrent programs execute multiple tasks independently.

  For example, a drawing application, with tasks:

  - receiving user input from the mouse pointer,
  - updating the displayed image.

- **parallelism**

  A parallel program executes two or more tasks in parallel with the explicit goal of increasing the overall performance.

  For example: a parallel Monte Carlo simulation for π, written with the multiprocessing module of Python.

Every parallel program is concurrent,
but not every concurrent program executes in parallel.

Parallel Processing
processes and threads

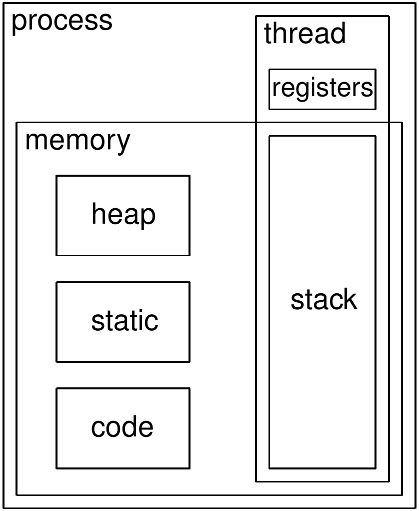
At any given time, many processes are running simultaneously on a computer.
The operating system employs ***time sharing*** to allocate a percentage of the CPU time to each process.

Consider for example the downloading of an audio file.
Instead of having to wait till the download is complete,
we would like to listen sooner.

Processes have their own memory space,
whereas threads share memory and other data.
Threads are often called lightweight processes.

A thread is short for ***a thread of execution***,
it typically consists of one function.
A program with more than one thread is *multithreaded*.

a single threaded process



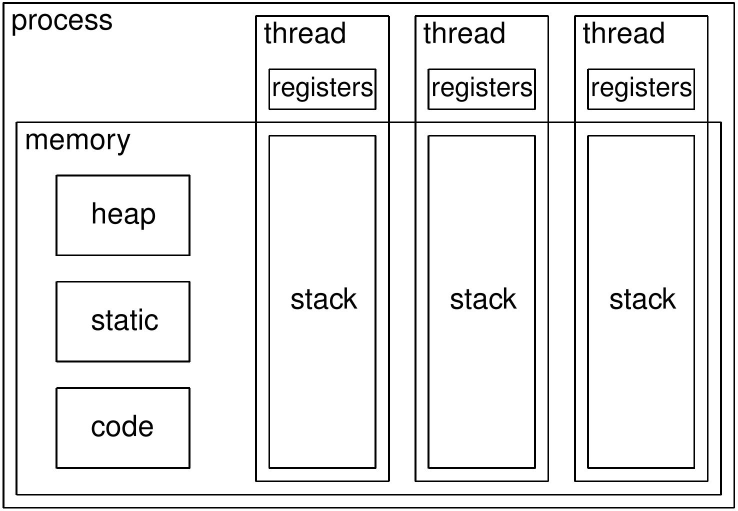
stack:

+ LIFO organization
+ for scratch space
+ fixed, limited size

heap

+ dynamic allocation
+ variable size
+ allocate at any time
+ deallocate at any time

a multithreaded process



processes and threads

A thread is a single sequential flow within a process.

Multiple threads within one process share

- heap storage, for dynamic allocation and deallocation,
- static storage, fixed space,
- code.

Each thread has its own registers and stack.

Difference between the stack and the heap:

- stack: Memory is allocated by reserving a block of fixed size on top of the stack. Deallocation is adjusting the pointer to the top.
- heap: Memory can be allocated at any time and of any size.

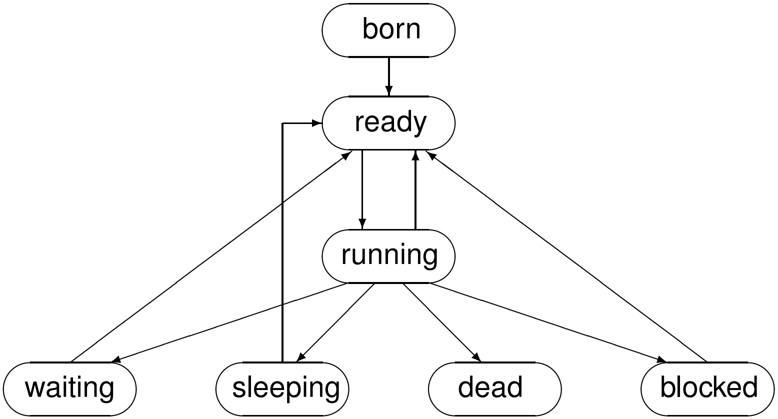
Threads share the same single address space and synchronization is needed when threads access same memory locations.

lifecycle of a thread
multithreaded programming

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
   - thread safety, critical sections, and deadlock
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer Consumer Relation
   - object-oriented design
   - classes producer and consumer

the Life Cycle of a Thread

a state diagram



lifecycle of a thread multithreaded programming

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
   - thread safety, critical sections, and deadlock
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer Consumer Relation
   - object-oriented design
   - classes producer and consumer

Thread Safety

Code is ***thread safe*** if its simultaneous execution by multiple threads is correct.

Only one thread can change shared data.

Once only concern for operating system programmer ...

Some illustrations of thread safety concerns:

1. shared bank account
2. do not block intersection
3. access same mailbox from multiple computers
4. threads access same address space in memory

Deadlock

Deadlock occurs when two are more processes (or threads) are blocked from progressing because each is waiting for a resource allocated to another.

Example: two processes occupy each half of the available memory and need more memory to continue.

Three conditions must be met for deadlock to occur:

1. competition for nonshareable resources;
2. resources are requested on a partial basis;
3. once allocated, no forcible retrieval possible.

Application: handling requests in a web server.
High internet traffic should not deadlock the server.

The Dining Philosphers Problem

an example problem to illustrate synchronization

The problem setup, rules of the game:

1. Five philosophers are seated at a round table.
2. Each philosopher sits in front of a plate of food.
3. Between each plate is exactly one chop stick.
4. A philosopher thinks, eats, thinks, eats, ...
5. To start eating, every philosopher
    1. first picks up the left chop stick, and
    2. then picks up the right chop stick.

Why is there a problem?

The problem of the starving philosophers:

- every philosoper picks up the left chop stick, at the same time,
- there is no right chop stick left, every philosopher waits, ...

lifecycle of a thread
multithreaded programming

have threads say hello

with the thread module

Our first multithreading Python script will do the following:

1. Import the `_thread` module.
2. Start three threads using `_thread.start_new_thread`.
3. Each thread will say hello and sleep for a number of seconds.
4. After starting the threads we must wait long enough for all threads to finish.

the program hello_threads.py with the thread module

```
import thread
import time

def say_hello(name, nsc):
    "says hello and sleeps n seconds"
    print "hello from " + name
    time.sleep(nsc)
    print name + " slept %d seconds" % nsc

print "starting three threads"
thread.start_new_thread(say_hello, ("1st thread", 3))
thread.start_new_thread(say_hello, ("2nd thread", 2))
thread.start_new_thread(say_hello, ("3rd thread", 1))
time.sleep(4)  # we must wait for all to finish!
print "done running the threads"
```

running hello_threads

At the command prompt $:

```
$ python hello_threads.py
starting three threads
hello from 1st thread
hello from 2nd thread
hello from 3rd thread
3rd thread slept 1 seconds
2nd thread slept 2 seconds
1st thread slept 3 seconds
done running the threads
$
```

Locks define a Critical Section

A ***critical section*** in code consists of instructions that may be executed by *only one* thread at any given time.
To ensure that ***only one*** thread executes code in critical section, we use a ***lock***:

```
import _thread
lock = _thread.allocate_lock()

lock.acquire()
# code in critical section
lock.release()
```

A lock is also called a ***semaphore***,
in analogy to railroad signals.

the Python interpreter lock

In CPython, the *global interpreter lock*, or GIL, prevents multiple native threads from executing Python bytecodes in parallel.

Why is the GIL necessary?
$\rightarrow$ The memory management in CPython is not thread safe.

Consequence: degrading performance on multicore processors.

For parallel code in Python, use

- the multiprocessing module (if sufficient memory available); or
- depend on multithreaded libaries that run outside the GIL.

using the Thread class

an object oriented approach

The `threading` module exports the `Thread` class.
From the module `threading` we import the `Thread` class.

We create new threads by inheriting from `Thread`,
overriding the `__init__` and `run`.

After creating a thread object, a new thread is born.

With `run`, we start the thread.

Main difference with the `thread` module is the explicit difference between the born and running state.

running `hello_threading`

At the command prompt `$`:

```
$ python hello_threading.py
1st thread is born
2nd thread is born
3rd thread is born
starting threads
hello from 1st thread
 hello from 2nd thread
hello from 3rd thread
threads started, waiting to finish
1st thread slept 3 seconds
3rd thread slept 4 seconds
2nd thread slept 6 seconds
$
```

the class HelloThread

```
from threading import Thread
from time import sleep
from random import randint

class HelloThread(Thread):
    """
    hello world with threads
    """
    def __init__(self, t):
        "initializes thread with name t"

    def run(self):
        "says hello and sleeps awhile"

def main():
    "starts three threads"

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

the methods of `HelloThread`

```
def __init__(self, t):
    "initializes thread with name t"
    Thread.__init__(self, name=t)
    print t + " is born "

def run(self):
    "says hello and sleeps awhile"
    name = self.getName()
    print "hello from " + name
    rnd = randint(3, 6)
    sleep(rnd)
    print name + " slept %d seconds" % rnd
```

the main function using `HelloThread`

```
def main():
    "starts three threads"
    one = HelloThread("1st thread")
    two = HelloThread("2nd thread")
    three = HelloThread("3rd thread")
    print "starting threads"
    one.start()
    two.start()
    three.start()
    print "threads started, waiting to finish"
    one.join()
    two.join()
    three.join()
```

The `start()` method executes the code define in the `run`.
With `join()` we wait for a thread to finish.

lifecycle of a thread
multithreaded programming

Producer/Consumer Relation

with threads

A very common relation between two threads is that of producer and consumer. For example, the downloading of an audio file is production, while listening is consumption.

Our producer/consumer relation with threads uses

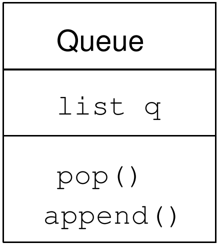
- an object of the class Producer is a thread that will append to a queue consecutive integers in a given range and at a given pace;
- an object of the class Consumer is a thread that will pop integers from the queue and print them, at a given pace.

If the pace of the produces is slower than the pace of the consumer, then the consumer will wait.

The Class Queue

class diagram

An object of the class Queue has three attributes: `list q`, `pop()`, `append()`.

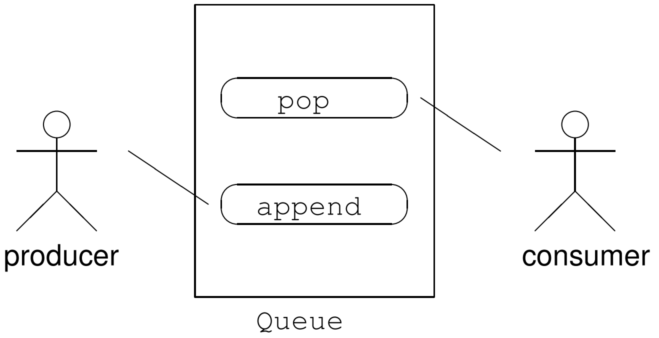


Two methods: `pop()` and `append()`.

Use Case Diagram for Queue

a behavior modeling diagram

Producer and consumer differ in their use of the Queue:



running the code

```
$ python prodcons.py
producer starts...consumption starts...

consumer sleeps 1 seconds
producer sleeps 2 seconds
wait a second...
appending 1 to queue
producer sleeps 3 seconds
popped 1 from queue
consumer sleeps 1 seconds
wait a second...
wait a second...
appending 2 to queue
producer sleeps 2 seconds
popped 2 from queue
consumer sleeps 1 seconds
wait a second...
appending 3 to queue
production terminated
popped 3 from queue
consumption terminated
$
```

structure of `Producer` class

```
from threading import Thread
from random import randint
from time import sleep

class Producer(Thread):
    """
    Appends integers to a queue.
    """
    def __init__(self, t, q, a, b, p):
        """
        Thread t to add integers in [a,b] to q,
        sleeping between 1 and p seconds.
        """

    def run(self):
        """
        Produces integers at some pace.
        """
```

the constructor of the `Producer` class

```
def __init__(self, t, q, a, b, p):
    """
    Thread t to add integers in [a,b] to q,
    sleeping between 1 and p seconds.
    """
    Thread.__init__(self, name=t)
    self.queue = q
    self.begin = a
    self.end = b
    self.pace = p
```

the production method

```
def run(self):
    """
    Produces integers at some pace.
    """
    print self.getName() + " starts..."
    for i in range(self.begin, self.end+1):
        rnd = randint(1, self.pace)
        print self.getName() + \
            " sleeps %d seconds" % rnd
        sleep(rnd)
        print "appending %d to queue" % i
        self.queue.append(i)
    print "production terminated"
```

a small test program

```
def main():
    """
    Creates a producer object.
    """
    que = []
    prd = Producer('producer', que, 3, 9, 10)
    prd.start()
    prd.join()

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

structure of Consumer class

```
from threading import Thread
from random import randint
from time import sleep

class Consumer(Thread):
    """
    Pops integers from a queue.
    """
    def __init__(self, t, q, n, p):
        """
        Thread t to pop n integers from q.
        """

    def run(self):
        """
        Pops integers at some pace.
        """
```

constructor of `Consumer` class

```
def __init__(self, t, q, n, p):
    """
    Thread t to pop n integers from q.
    """
    Thread.__init__(self, name=t)
    self.queue = q
    self.amount = n
    self.pace = p
```

consuming elements

```
def run(self):
    """
    Pops integers at some pace.
    """
    print "consumption starts..."
    for i in range(0, self.amount):
        rnd = randint(1, self.pace)
        print self.getName() + \
            " sleeps %d seconds" % rnd
        sleep(rnd)
        while True:
            try:
                i = self.queue.pop(0)
                print "popped %d from queue" % i
                break
            except IndexError:
                print "wait a second..."
                sleep(1)
    print "consumption terminated"
```

a small test program

```
def main():
    """
    Simulates consumption on some queue.
    """
    que = range(5)
    cns = Consumer('consumer', que, 5, 10)
    cns.start()
    cns.join()

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

the main program in `prodcons.py`

Code for the class `Producer` and `Consumer` is in modules `class_producer` and `class_consumer` respectively.

```
from class_producer import Producer
from class_consumer import Consumer

QUE = []      # queue is shared list
PRD = Producer('producer', QUE, 1, 3, 4)
CNS = Consumer('consumer', QUE, 3, 1)
PRD.start()   # start threads
CNS.start()
PRD.join()    # wait for thread to finish
CNS.join()
```

Summary + Assignments

Background reading:
§3.3,4 & §6.6 in *Computer Science, an overview*.

Assignments:

1. Modify the producer/consumer relationship into card dealing. The producer is the card dealer, the consumer stores the received cards in a hand.
2. When running a large simulation, e.g.: testing the distribution of a random number generator, it is useful to consider the evolution of the histogram. Design a multithreaded program where the producer generates random numbers that are then classified by the consumer.