Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

MCS 275 Lecture 30
Programming Tools and File Management
Jan Verschelde, 27 March 2017

Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

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.

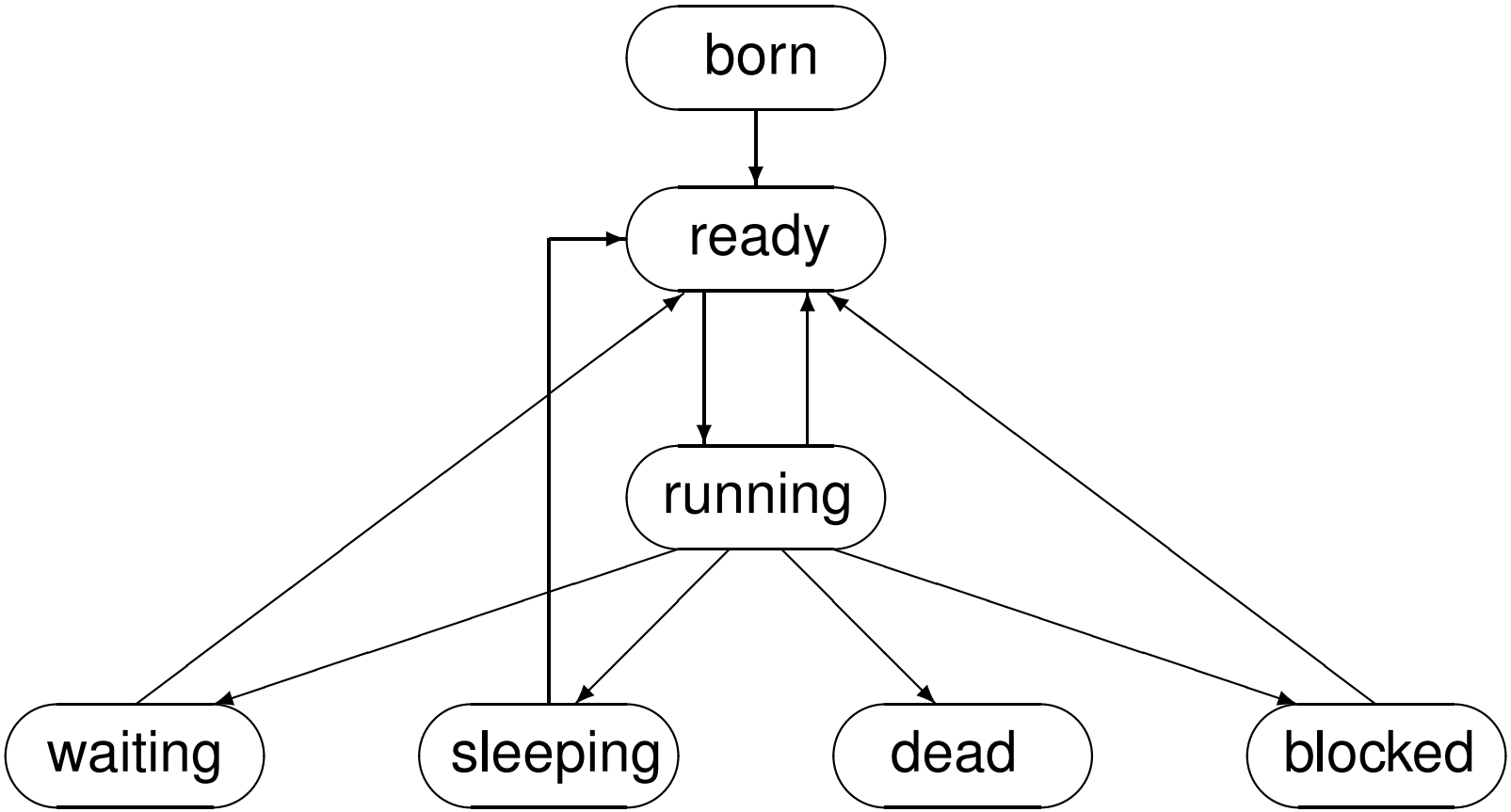
A thread is short for *a thread of execution*, it typically consists of one function.
A program with more than one thread is *multithreaded*.

Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

the Life Cycle of a Thread

a state diagram



Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

have threads say hello

with the `_thread` module

Our first multithreading Python code will

1. import the `_thread` module
2. start three threads using `_thread.start_new_thread`
3. each thread will say hello and sleep for `n` seconds
4. after starting the threads we must wait long enough for all threads to finish

hello_threads.py uses the _thread module

```
import _thread
from time import sleep

def say_hello(name, nsec):
    """
    Says hello and sleeps nsec seconds.
    """
    print("hello from " + name)
    sleep(nsec)
    print(name + " slept %d seconds" % nsec)

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))
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
$
```

Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

using the Thread class

an object oriented approach

The `threading` module exports the `Thread` class.

We create new threads by inheriting from `threading.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
first thread is born
second thread is born
third thread is born
starting threads
hello from first thread
hello from second thread
hello from third thread
threads started
third thread slept 1 seconds
second thread slept 4 seconds
first thread slept 5 seconds
$
```

the class `HelloThread`

```
import threading

class HelloThread(threading.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.
    """
```

the constructor and run method

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

def run(self):
    """
    says hello and sleeps awhile"
    """
    name = self.getName()
    print("hello from " + name)
    nbr = randint(1, 6)
    sleep(nbr)
    print(name + " slept %d seconds" % nbr)
```

the `main()` in `HelloThread`

```
def main():
    """
    Starts three threads.
    """
    first = HelloThread("first thread")
    second = HelloThread("second thread")
    third = HelloThread("third thread")
    print("starting threads")
    first.start()
    second.start()
    third.start()
    print("threads started")

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

Multithreading

1. Concurrent Processes
   - processes and threads
   - life cycle of a thread
2. Multithreading in Python
   - the thread module
   - the Thread class
3. Producer/Consumer Relation
   - two different algorithms run concurrently

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

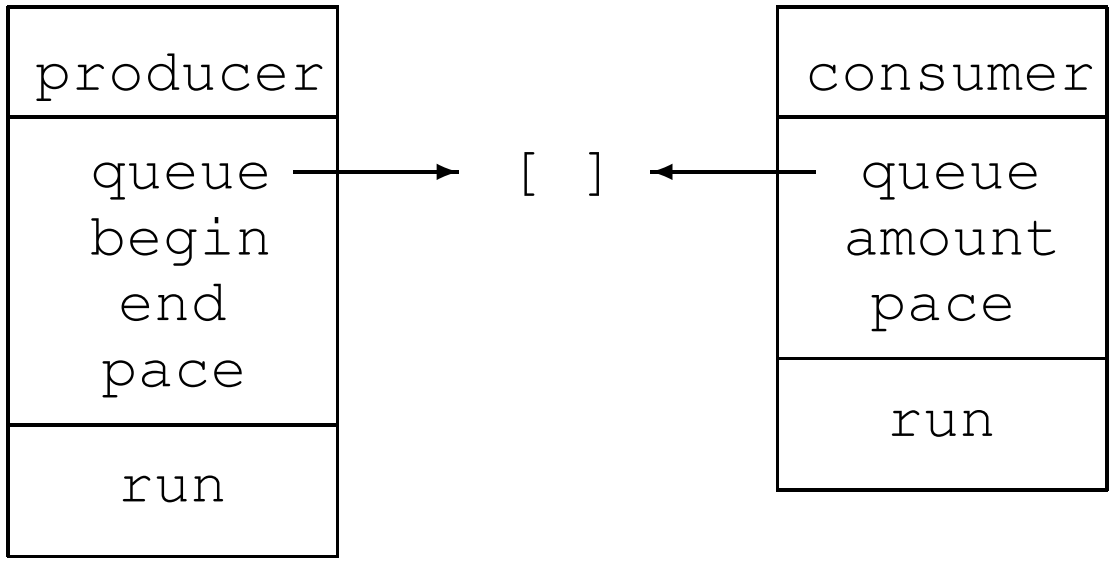
- an object of the class Producer is a thread
  that will append to a queue consecutive integers
  in a given range and at a given pace;
- an object of the class Consumer is a thread
  that will pop integers from the queue and print them,
  at a given pace.

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

running the code

```
$ python prodcons.py
producer starts...
producer sleeps 1 seconds
consumption starts...
consumer sleeps 1 seconds
appending 1 to queue
producer sleeps 4 seconds
popped 1 from queue
consumer sleeps 1 seconds
wait a second...
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
```

UML class diagrams



The `queue` in each class refer to the same list:

- The producer appends to the queue.
- The consumer pops from the queue.

structure of the class `Producer`

```
import threading

class Producer(threading.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 method

```
def __init__(self, t, q, a, b, p):
    """
    Thread t to add integers in [a, b] to q,
    sleeping between 1 and p seconds.
    """
    threading.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):
        nbr = randint(1, self.pace)
        print(self.getName() + \
            " sleeps %d seconds" % nbr)
        sleep(nbr)
        print("appending %d to queue" % i)
        self.queue.append(i)
    print("production terminated")
```

constructor and run method of the class `Consumer`

```
import threading

class Consumer(threading.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.
        """
```

the constructor of the class `Consumer`

```
def __init__(self, t, q, n, p):
    """
    Thread t to pop n integers from q.
    """
    threading.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):
        nbr = randint(1, self.pace)
        print(self.getName() + \
            " sleeps %d seconds" % nbr)
        sleep(nbr)
        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")
```

the main program in `prodcons.py`

Code for the class Producer and Consumer in modules `classproducer` and `classconsumer` respectively.

```
from classproducer import Producer
from classconsumer import Consumer

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

Summary + Assignments

Read chapter 13 in *Python Programming in Context*.

Assignments:

1. Implement the secret guessing with client/server network programming of lecture 24 using threads.
2. Modify the producer/consumer relationship into card dealing. The producer is the card dealer, the consumer stores the received cards in a hand.
3. 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.