PARALLEL PROGRAMMING

Adapted from David Beazley's paper:
"An Introduction to Python Concurrency"
Presented at USENIX Technical Conference
San Diego, June, 2009

David Beazley: http://www.dabeaz.com

Code Examples and Files

- Thanks Dave Beazley for contributing his fantastic set of source code examples on Python concurrency and parallel programming
- We have also added a few more examples and rephrased Dave's examples to suite our course objectives
- Our source code repository can be retrieved from: http://www.samyzaf.com/braude/OS/PROJECTS/PARALLEL_PROGRAMMING_LAB.zip
- Dave Beazley original resources can be retrieved from: http://www.dabeaz.com/usenix2009/concurrent/

Concurrent Programming

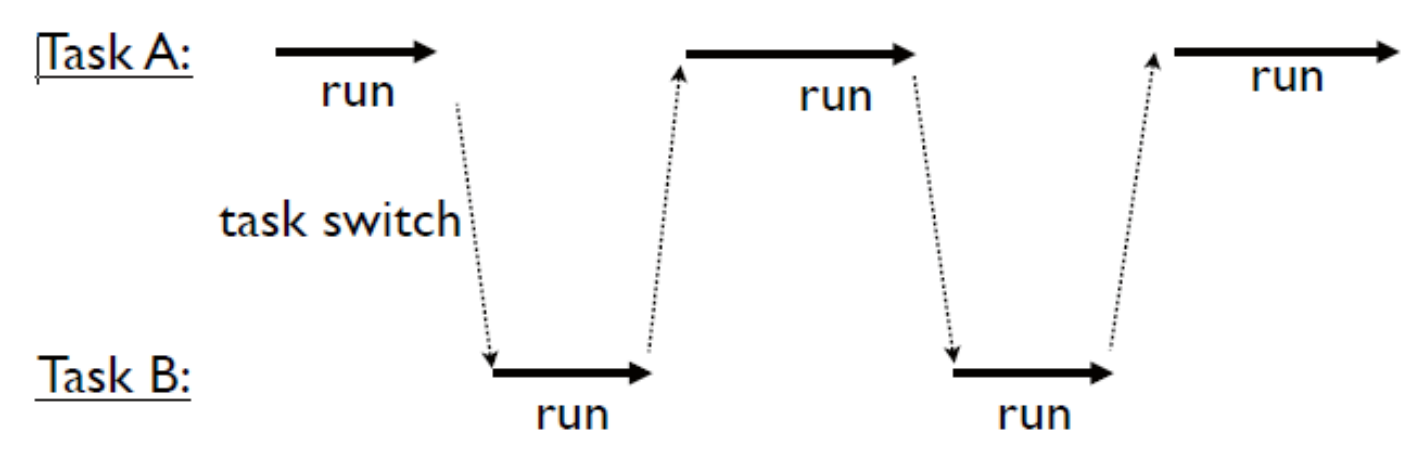
- Doing more than one thing at a time
- Writing programs that can work on more than one thing at a time
- Of particular interest to programmers and systems designers
- Writing code for running on “big iron”
- But also of interest for users of multicore desktop computers
- Goal is to go beyond the user manual and tie everything together into a "bigger picture."

Examples

- Web server that communicates with thousand clients (Google)
- Web client (Chrome or Firefox) that displays 10 or 20 tabs
- In the same process it may do the following tasks concurrently:
  - Download several images, audio files, movies (concurrently)
  - Display an image and a movie
  - Connect to several servers
- Microsoft Word can do several tasks at the same time
  - Let the user insert text with no waits or interrupts
  - Download/upload stuff
  - Backup the document every few seconds
  - Check spelling and grammar (and even mark words as the user is typing)
- Image processing software that uses 8 CPU cores for parallel intense matrix multiplications

Multitasking

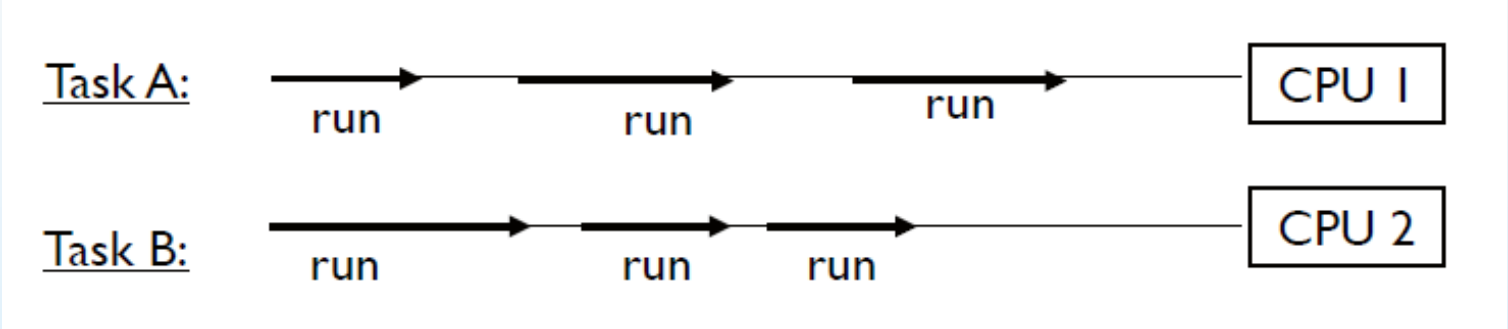
- Concurrency usually means multitasking



- If only one CPU is available, the only way it can run multiple tasks is by rapidly switching between them in one of two way:
  - Process context switch (two processes)
  - Thread context switch (two threads in one process)

Parallel Processing

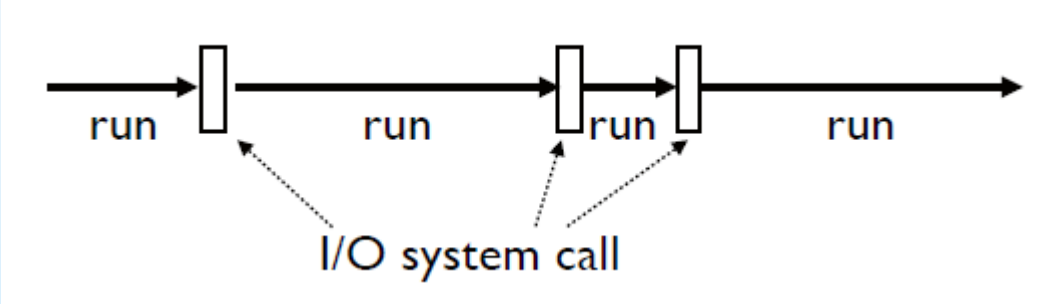
- If you have many CPU's or CORE's then you can have true parallelism: the two tasks run simultaneously



- If the total number of tasks exceeds the number of CPUs, then some CPU's must multitask (switch between tasks)

Task Execution

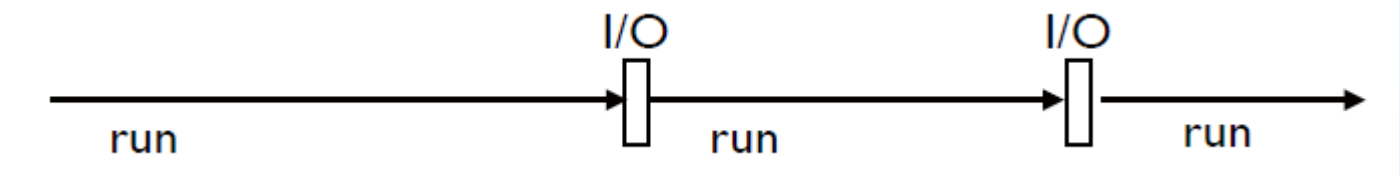
- Every task executes by alternating between CPU processing and I/O handling:
  - disk read/write
  - network send/receive



- For I/O, tasks must wait (sleep): the underlying system will carry out the I/O operation and wake the task when it's finished

CPU Bound Tasks

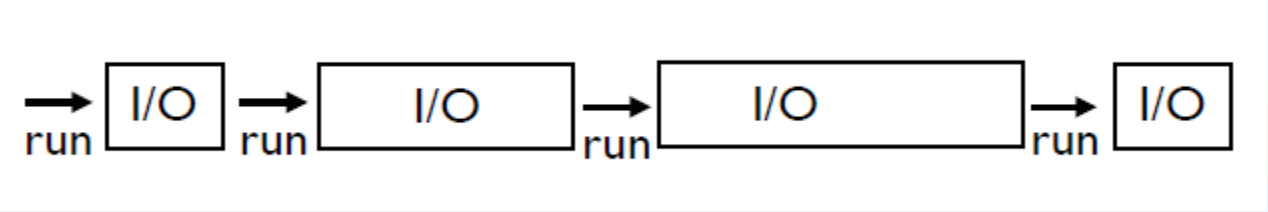
- A task is "CPU Bound" if it spends most of its time processing with little I/O



- Examples:
  - Image processing
  - Weather forecast system
  - Heavy mathematical computations

I/O Bound Tasks

- A task is "I/O Bound" if it spends most of its time waiting for I/O

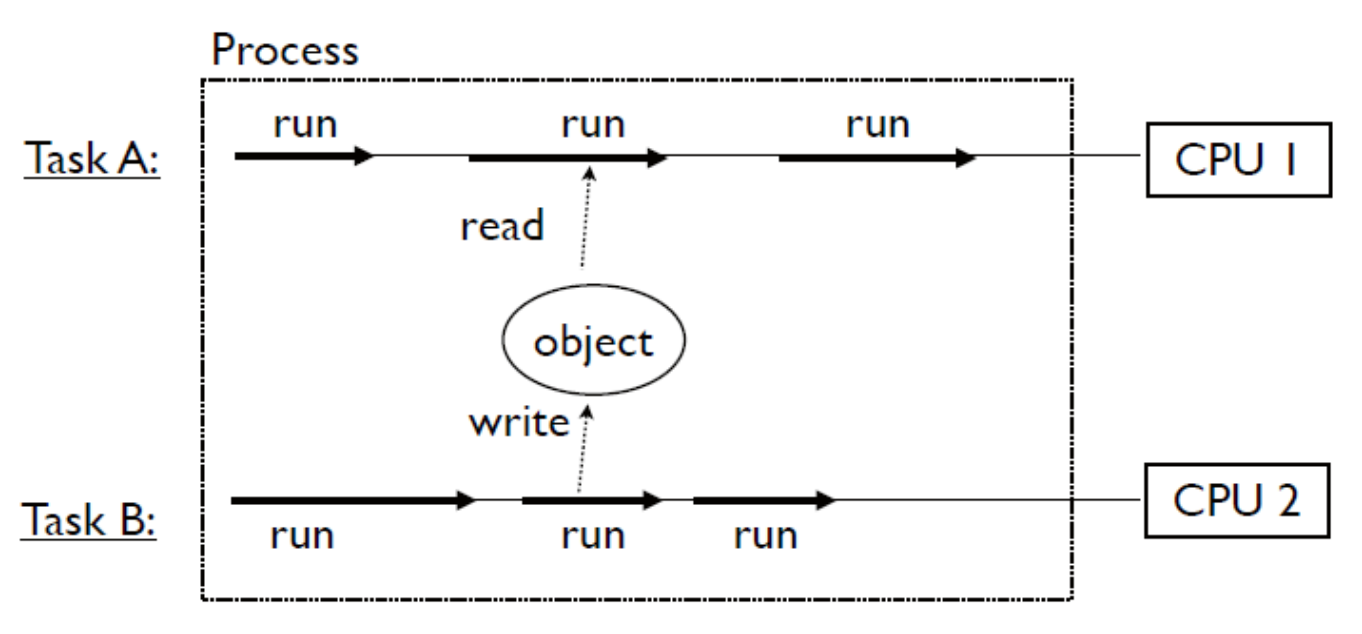


- Examples:
  - Reading input from the user (text processors)
  - Networking
  - File Processing

- Most "normal" programs are I/O bound

Shared Memory

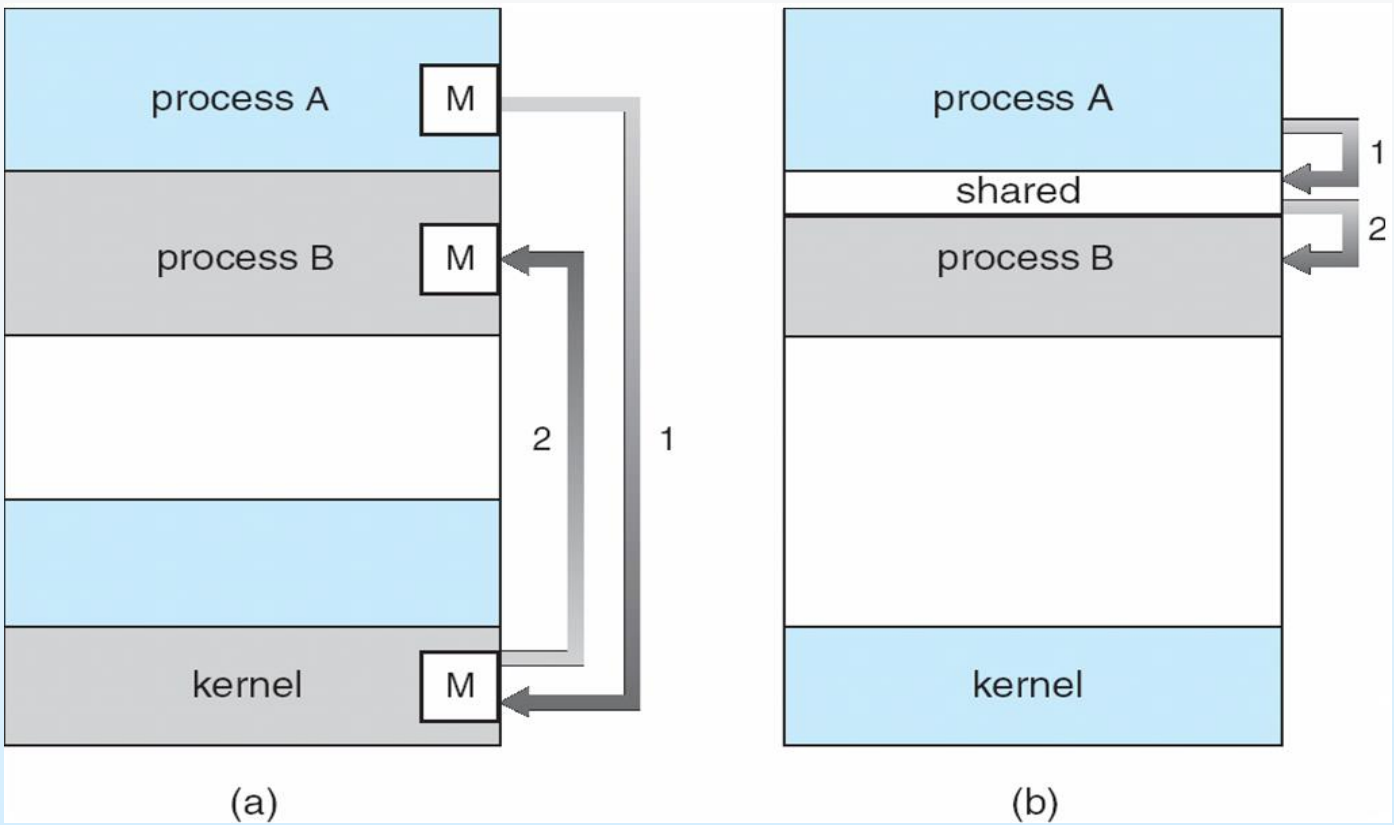
- In many cases, two tasks need to share information ("**cooperating tasks**") and access an object simultaneously
- Two threads within the same process always share all memory of that process
- Two independent processes on the other hand need special mechanisms to communicate between them



IPC – Inter Process Communication

- Processes within a system may be **independent** or **cooperating**
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need inter-process communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Two Types of IPC

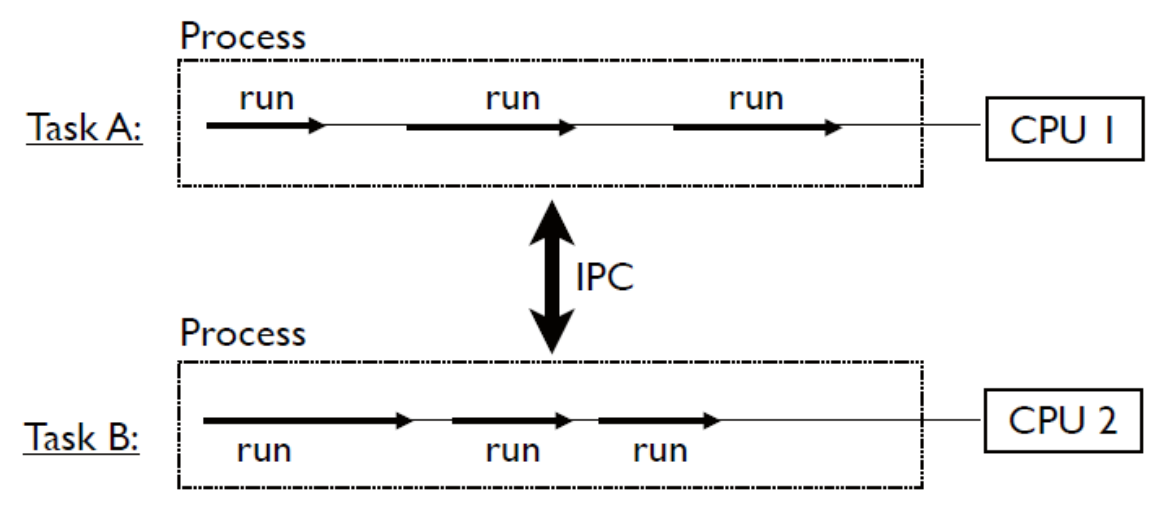


(a) Kernel shared memory: Pipe, Socket, FIFO, mailboxes

(b) Process shared memory (OS is not involved here!)

IPC – Inter Process Communication

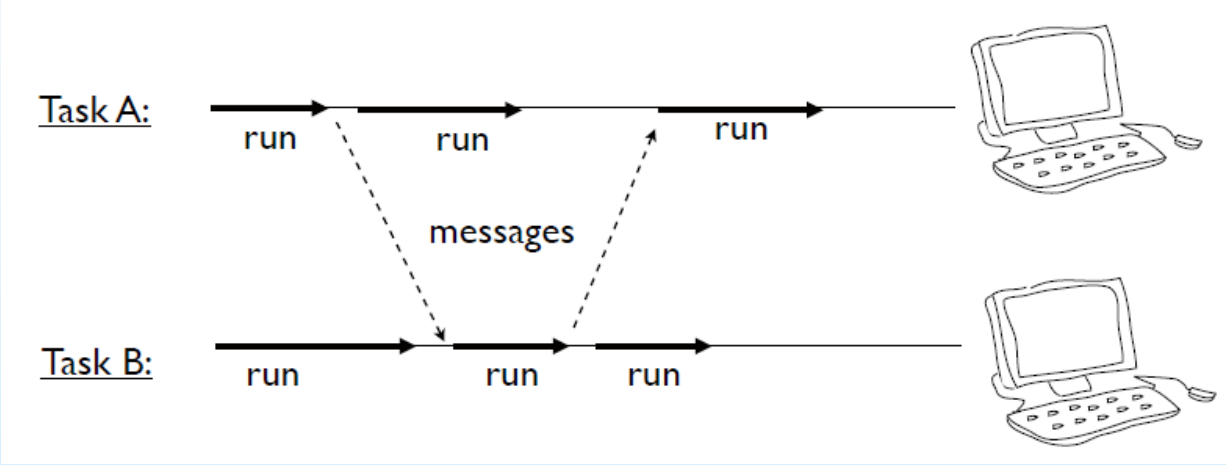
- The simplest mechanism for two processes to communicate are
  - Pipe
  - FIFO
  - Shared memory (memory mapped regions)
  - Socket



- Processes can also communicate through the file system, but it tends to be too slow and volatile (disk is full, unwritable, or busy)

Distributed Computing

- Tasks may be running on distributed systems
- Sometimes on two different continents



- Cluster of workstations
- Usually: communication via sockets (or MPI)

Programmer Performance

- Programmers are often able to get complex systems to "work" in much less time using a high-level language like Python than if they're spending all of their time hacking C code
- In some cases scripting solutions might be even competitive with C++, C# and, especially, Java
- The reason is that when you are operating at a higher level, you often are able to find a better, more optimal, algorithm, data structures, problem decomposition schema, or all of the above

"The best performance improvement is the transition from the nonworking to the working state."
- John Ousterhout

"Premature optimization is the root of all evil."
- Donald Knuth

"You can always optimize it later."
- Unknown

Intel VLSI Tools as an Example

- In recent years, a fundamental transition has been occurring in the way industry developers write computer programs
- The change is a transition from system programming languages such as C or C++ to scripting languages such as Perl, Python, Ruby, JavaScript, PHP, etc.

Tool	C/C++ lines	TCL/PERL/PYTHON lines
VLSI CAD TOOL 1	510,946	644,272
VLSI CAD TOOL 2	581333	368435
VLSI CAD TOOL 3	1,517,917	1,421,400
VLSI CAD TOOL 4	422,239	1,332,767

Performance is Irrelevant!

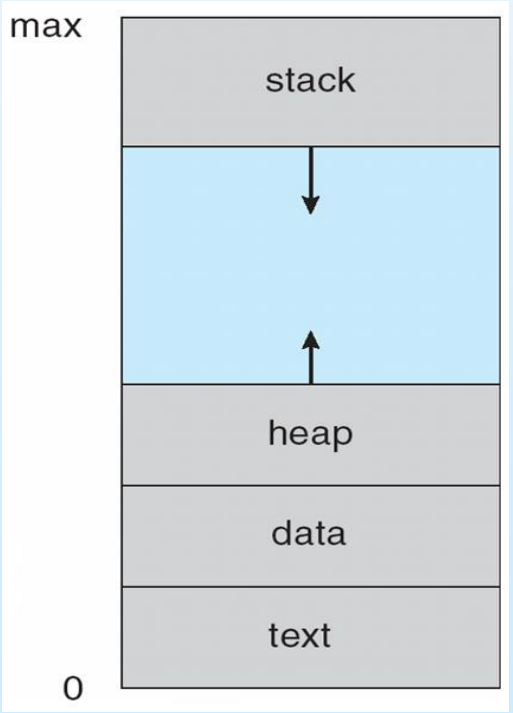
- Many concurrent programs are "I/O bound“
- They spend virtually all of their time sitting around waiting for
  - Clients to connect
  - Client requests
  - Client responses
- Python can "wait" just as fast as C
- One exception: if you need an extremely rapid response time as in real-time systems

You Can Always Go Faster

- Python can be extended with **C code**
- Look at **ctypes**, **Cython**, **Swig**, etc.
- If you need really high-performance, you're not coding Python -- you're using C extensions
- This is what most of the big scientific computing hackers are doing
- It's called: **"using the right tool for the job"**

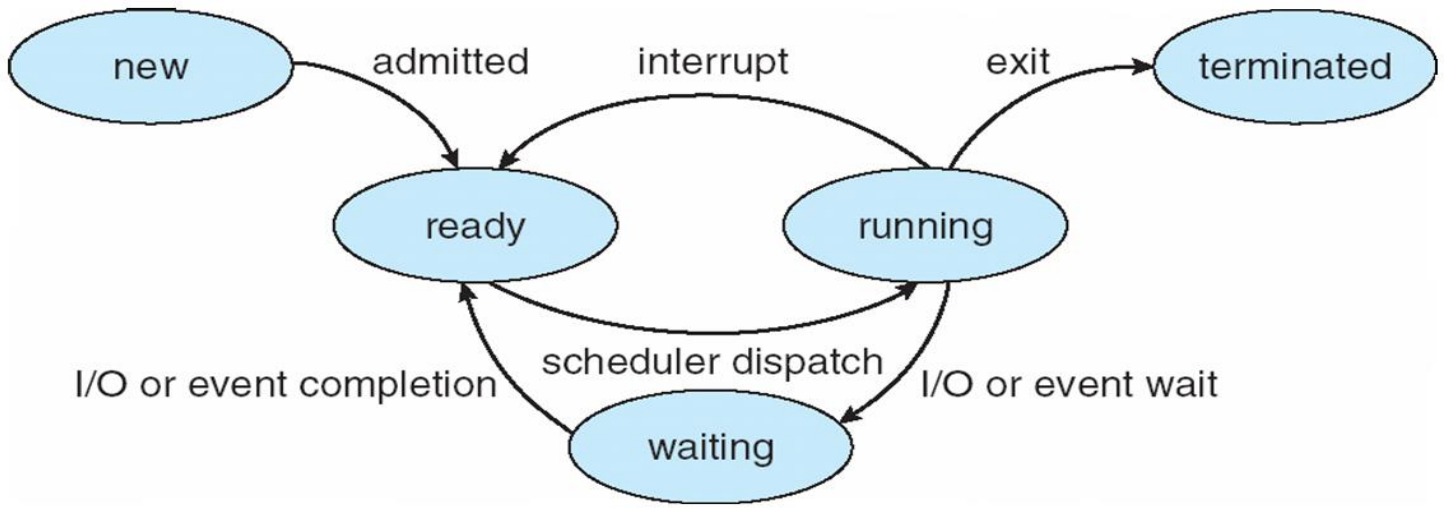
Process Concept Review

- A process (or job) is a program in execution
- A process includes:
  1. **Text** (program code)
  2. **Data** (constants and fixed tables)
  3. **Heap** (dynamic memory)
  4. **Stack** (for function calls and temporary variables)
  5. **Program counter** (current instruction)
  6. **CPU registers**
  7. **Open files table** (including sockets)
- To better distinguish between a program and a process, note that a single Word processor program may have 10 different processes running simultaneously
- Consider multiple users executing the same Internet explorer (each has the 6 things above)
- Computer activity is the sum of all its processes



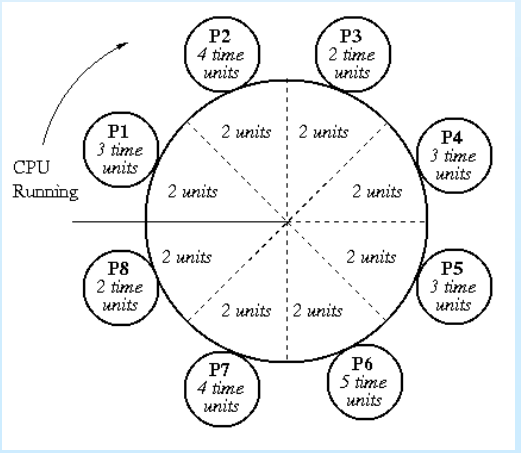
Process States

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



CPU Process Scheduling

- Modern operating systems can run hundreds (and even thousands) of processes in parallel
- Of course, at each moment, only a single process can control a CPU, but the operating system is switching processes every 15 milliseconds (on average) so that at 1 minute, an operating system can swap 4000 processes!
- The replacement of a running process with a new process is generated by an **INTERRUPT**



THREADS

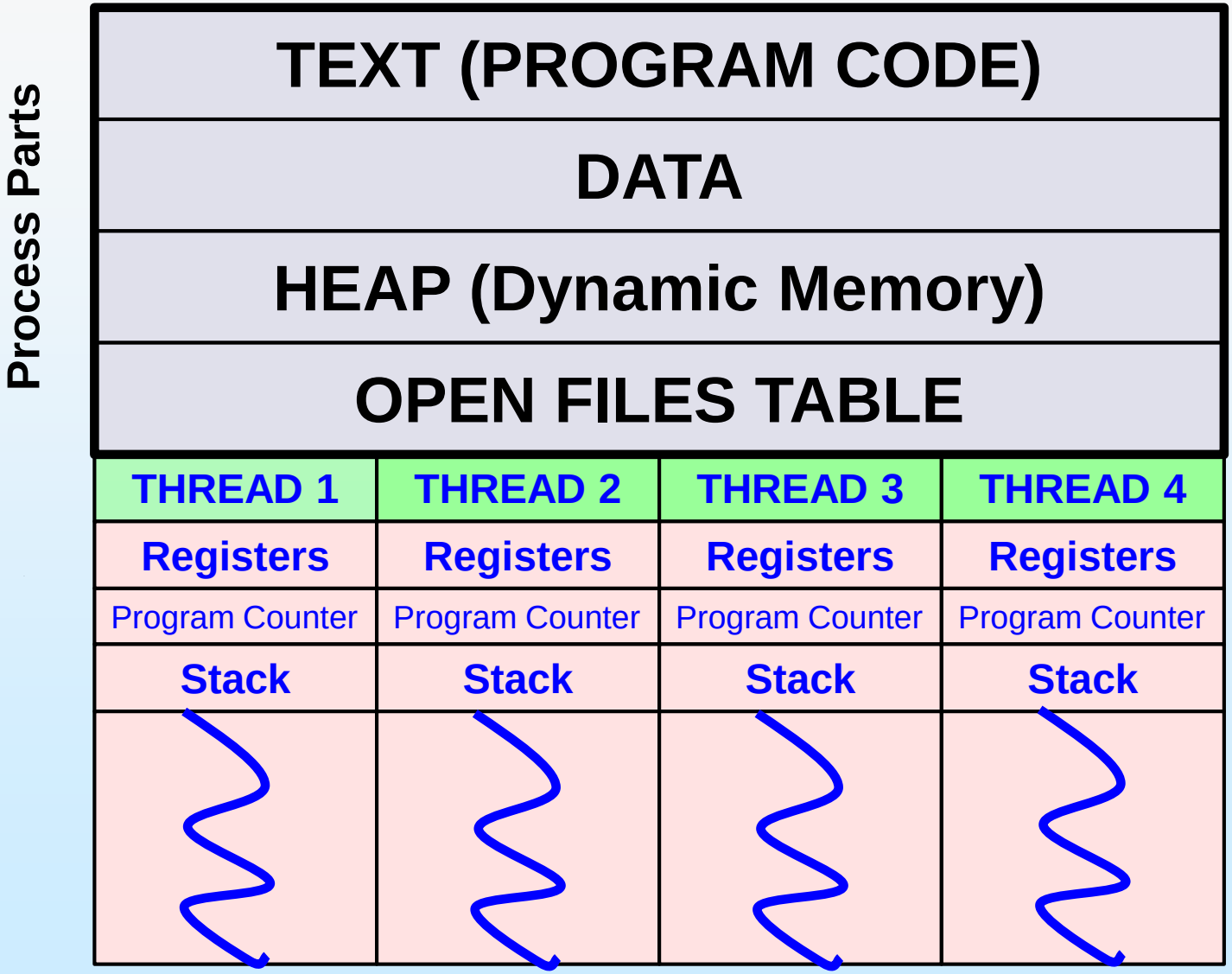
- What most programmers think of when they hear about **“concurrent programming”**
- A Thread is an **independent task** running inside a program
- Shares resources with the main program (and other threads)
  - **Memory** (Program text, Data, Heap)
  - **Files**
  - **Network connections**
- Has its own **independent** flow of execution
  - **Thread stack**
  - **Thread program counter**
  - **Thread CPU registers (context)**

Threads Example

```
import time
from threading import Thread

def apples(n):
    for i in range(n):
        print "Apple_%d" % i
        time.sleep(1)

def bananas(n):
    for i in range(n):
        print "Banana_%d" % i
        time.sleep(1)

def serial_run():               # Run time = 13 seconds
    apples(5)
    bananas(8)

def parallel_run():             # Run time = 8 seconds !
    t1 = Thread(target=apples, args=(5,))
    t2 = Thread(target=bananas, args=(8,))
    t1.start()
    t2.start()
```

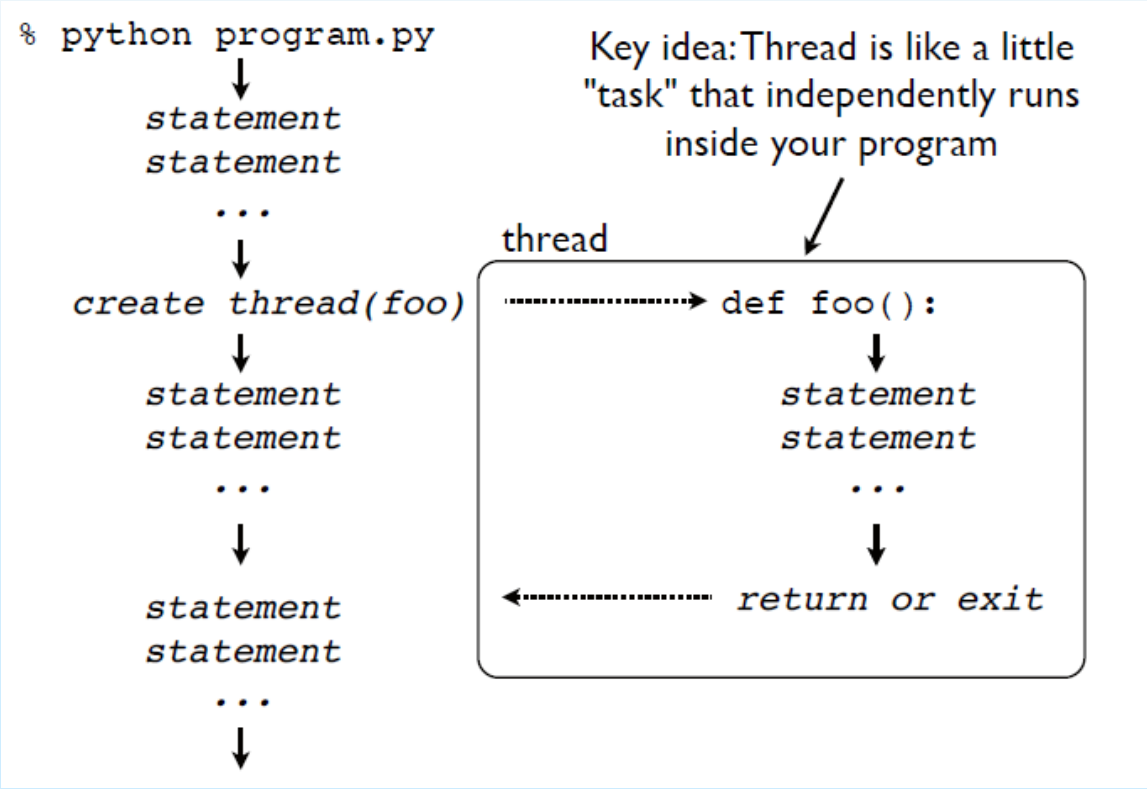
apples_and_bananas_1.py

One Process, Many Threads!



THREADS

- Several threads within the same process can use and share
  - common variables
  - common open files
  - common networking sockets



Cooperating Threads

```
stop = False   # global variable

def apples(n):
    global stop
    for i in range(n):
        print "Apple_%d" % i
        time.sleep(1)
    stop = True

def bananas(n):
    for i in range(n):
        if stop: break
        print "Banana_%d" % i
        time.sleep(0.5)

def parallel_run():      # What happens here ??
    t1 = Thread(target=apples, args=(5,))
    t2 = Thread(target=bananas, args=(100,))
    t1.start()
    t2.start()
```

apples_and_bananas_2.py

Threading Module

- Python threads are defined by a class
- You inherit from Thread and redefine run()

```
from threading import Thread
Import time

class CountdownThread(Thread):
    def __init__(self, name, count):
        Thread.__init__(self)
        self.name = name
        self.count = count

    def run(self):
        while self.count > 0:
            print "%s:%d" % (self.name, self.count)
            self.count -= 1
            time.sleep(2)
        return
```

This code executes in the thread

countdown1.py

Launching a Thread

- To launch a thread: create a thread object and call start()

```
t1 = CountdownThread(10)    # Create the thread object
t2 = CountdownThread(20)    # Create another thread
t1.start()                  # Launch thread t1
t2.start()                  # Launch thread t2
```

countdown1.py

- Thread executes until their run method stops (return or exit)

Alternative way to launch threads

```
import time
from threading import Thread

def countdown(name, count):
    while count > 0:
        print "%s:%d" % (name, count)
        count -= 1
        time.sleep(2)
    return

t1 = Thread(target=countdown, args=("A", 10))
t2 = Thread(target=countdown, args=("B", 20))
t1.start()
t2.start()
```

countdown2.py

- Creates a Thread object, but its run() method just calls the countdown function

Joining a Thread

- Once you start a thread, it runs independently
- Use **t.join()** to wait for a thread to exit
- This only works from other threads
- A thread can't join itself!

```
t = Thread(target=foo, args=(a1,a2))
t.start()
# Do some work ...
t.join()    # Wait for the thread to exit
# Continue your work ...
```

Daemonic Threads

- If a thread runs forever, make it "daemonic“
- If you don't do this, the interpreter will lock when the main thread exits - waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

```
t.daemon = True
t.setDaemon(True)
```

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic
- (which is a really good way to have your head explode)

Accessing Shared Data

- Consider a shared object

```
x = 0
```

- And two threads that modify it

```
#Thread 1
x = x + 1
```

```
#Thread 2
x = x - 1
```

- It's possible that the resulting value will be unpredictably corrupted

Accessing Shared Data

- Is this a serious concern?
- YES! This is a dead serious matter!
- Look what happens in the following example !?

```
def foo():
    global x
    for i in xrange(1000000):
        x += 1

def bar():
    global x
    for i in xrange(1000000):
        x -= 1

t1 = Thread(target=foo)
t2 = Thread(target=bar)
t1.start()
t2.start()
```

RACE_WARS/race_1.py

The Therac-25 Accidents

- Machine for radiation therapy
  - Software control of electron accelerator and electron beam/Xray production
  - Software control of dosage
- **Software errors caused the death of several patients**
- A series of race conditions on shared variables and poor software design

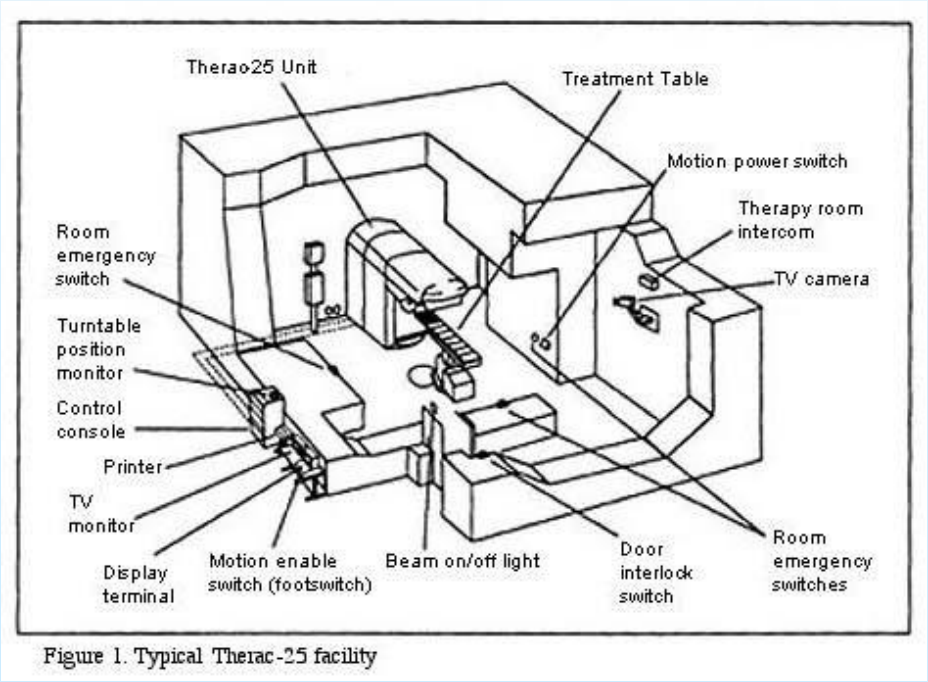


Figure 1. Typical Therac-25 facility

Race Conditions

- The corruption of shared data due to thread scheduling is often known as a "race condition."
- It's often quite diabolical - a program may produce slightly different results each time it runs (even though you aren't using any random numbers!)
- Or it may just flake out mysteriously once every two weeks

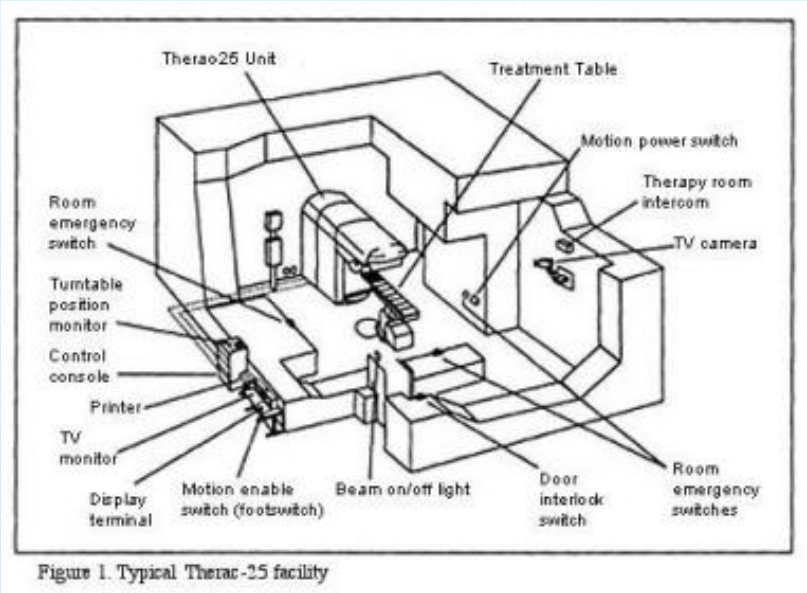


Figure 1. Typical Therac-25 facility

THREADS – Summary (1)

- Threads are easier to create than processes:
  - Threads do not require a separate address space!
- Multithreading requires careful programming!
- Threads share data structures that should only be modified by one thread at a time! (mutex lock)
- A problem in one thread can
  - Cause the parent process to block or crash
  - and thus kill all other threads!
- Therefore a lot of caution must be taken so that different threads don't step on each other!

THREADS – Summary (2)

- Unlike threads, **processes** do not share the same address space and thus are truly independent of each other.
- Threads are considered lightweight because they use far less resources than processes (no need for a full context switch)
- Threads, on the other hand, share the same address space, and therefor are interdependent
- Always remember the golden rules:
  - **Write stupid code and live longer (KISS)**
  - **Avoid writing any code at all if you don't have to! (Bjarn Stroustrup, inventor of C++)**

Thread Synchronization

- Identifying and fixing a race condition will make you a better programmer (e.g., it "builds character")
- However, you'll probably never get that month of your life back …
- To fix : You have to synchronize threads
- **Synchronization Primitives:**
  - Lock
  - Semaphore
  - BoundedSemaphore
  - Condition
  - Event

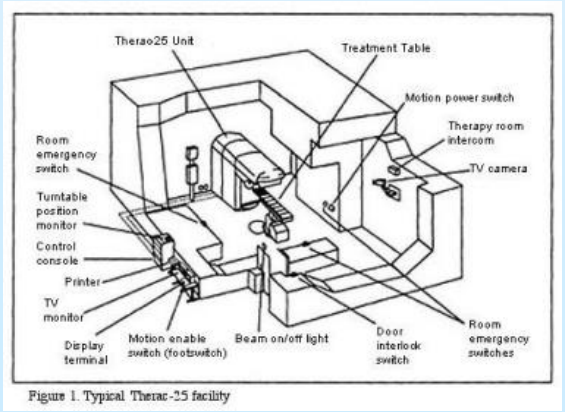


Figure 1. Typical Therac-25 facility

Mutex Locks

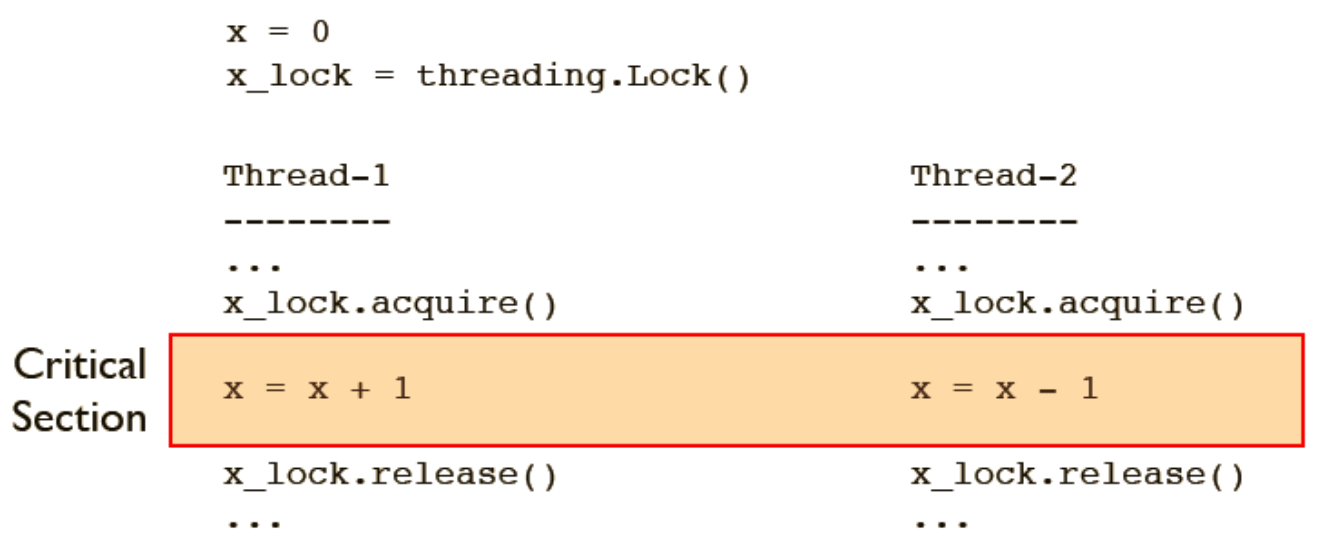
- Probably the most commonly used synchronization primitive
- Mostly used to synchronize threads so that only one thread can make modifications to shared data at any given time
- Has only two basic operations

```
from threading import Lock

m = Lock()
m.acquire()
m.release()
```

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

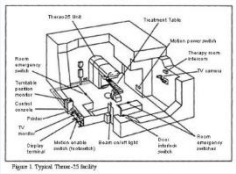
Use of Mutex Locks

- Commonly used to enclose critical sections



- Only one thread can execute in critical section at a time (lock gives exclusive access)

Using a Mutex Lock



- It is your responsibility to identify and lock all "critical sections“ !

```
x = 0
x_lock = threading.Lock()

Thread-1
--------
...
x_lock.acquire()
x = x + 1
x_lock.release()
...
```

```
Thread-2
--------
...
x = x - 1
...
```

If you use a lock in one place, but not another, then you're missing the whole point. All modifications to shared state must be enclosed by lock acquire()/release().

Lock Management

- Locking looks straightforward
- Until you start adding it to your code …
- Managing locks is a lot harder than it looks!
- Acquired locks must always be released!
- However, it gets evil with exceptions and other non-linear forms of control-flow
- Always try to follow this prototype:

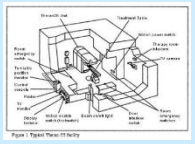
```
x = 0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```

Lock and Deadlock

- Avoid writing code that acquires more than one mutex lock at a time

```
mx = Lock()
my = Lock()

mx.acquire()
# statement using x
my.acquire()
# statement using y
my.release()
# ...
mx.release()
```

- This almost invariably ends up creating a program that mysteriously deadlocks (even more fun to debug than a race condition)
- Remember Therac-25 …



Semaphores

- A counter-based synchronization primitive
  - acquire() - Waits if the count is 0, otherwise decrements the count and continues
  - release() - Increments the count and signals waiting threads (if any)
- Unlike locks, acquire()/release() can be called in any order and by any thread

```
from threading import Semaphore
m = Semaphore(n) # Create a semaphore
m.acquire() # Acquire
m.release() # Release
```

Semaphores Uses

- **Resource control:** limit the number of threads performing certain operations such as database queries, network connections, disk writes, etc.
- **Signaling:** Semaphores can be used to send "signals" between threads
- For example, having one thread wake up another thread.

Resource Control

- **Using Semaphore to limit Resource:**

```
import requests
from threading import Semaphore

sem = Semaphore(5) #    Max: 5-threads
def get_link(url):
    sem.acquire()
    try:
        req = requests.get(url)
        return req.content
    finally:
        sem.release()
```

- Only 5 threads can execute the **`get_link`** function
- This make sure we do not put too much pressure on networking system

Thread Signaling

- **Using a semaphore to "send a signal":**

```
sem = Semaphore(0)
```

```
# Thread 1
...
statements
statements
statements
sem.release()
...
```

```
# Thread 2
...
sem.acquire()
statements
statements
statements
...
```

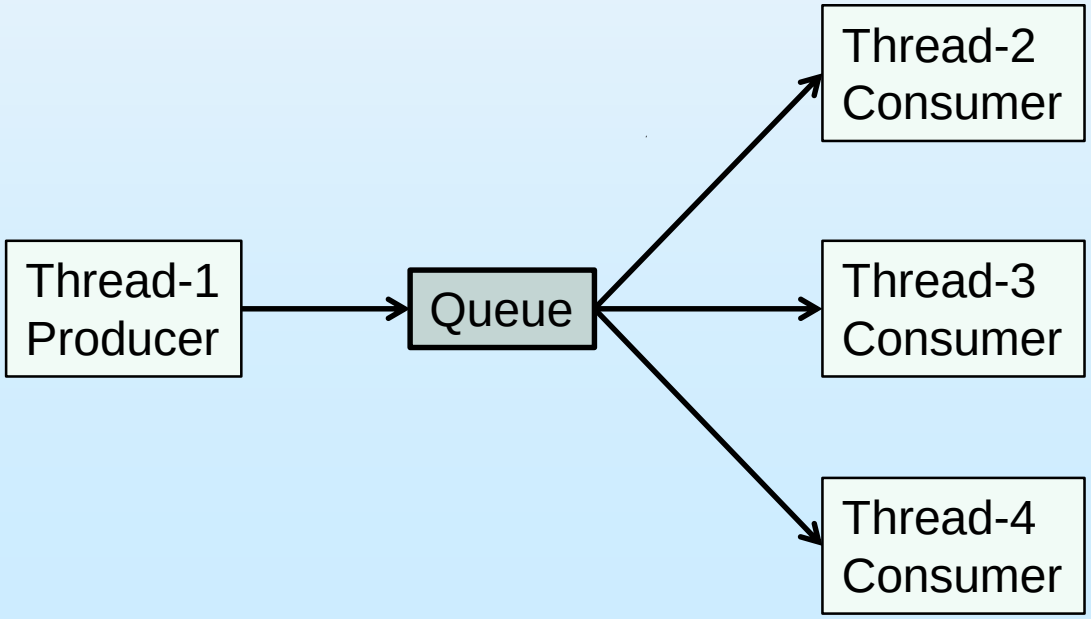
- Here, acquire() and release() occur in different threads and in a different order
- Thread-2 is blocked until Thread-1 releases "**sem**".
- Often used with producer-consumer problems

Threads Summary

- Working with all of the synchronization primitives is a lot trickier than it looks
- There are a lot of nasty corner cases and horrible things that can go wrong
  - Bad performance
  - deadlocks
  - Starvation
  - bizarre CPU scheduling
  - etc...
- All are valid reasons to not use threads, unless you do not have a better choice

Threads and Queues

- Threaded programs are easier to manage if they can be organized into producer/consumer components connected by queues
- Instead of "sharing" data, threads only coordinate by sending data to each other
- Think Unix "pipes" if you will...



Queue Library Module

- Python has a thread-safe queuing module
- Basic operations

```
from Queue import Queue

q = Queue([maxsize])    # Create a queue
q.put(item)             # Put an item on the queue
q.get()                 # Get an item from the queue
q.empty()               # Check if empty
q.full()                # Check if full
```

- Usage: Try to strictly adhere to get/put operations. If you do this, you don't need to use other synchronization primitives!

Queue Usage

- Most commonly used to set up various forms of producer/consumer problems

```
from Queue import Queue
q = Queue()
```

Producer Thread

```
for item in produce_items():
    q.put(item)
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
```

- Critical point : You don't need locks here !!! (they are already embedded in the Queue object)

Producer Consumer Pattern

```
import time, Queue
from threading import Thread, currentThread

que = Queue.Queue()

def run_producer():
    print "I am the producer"
    for i in range(30):
        item = "packet_" + str(i)   # producing an item
        que.put(item)
        time.sleep(1.0)

def run_consumer():
    print "I am a consumer", currentThread().name
    while True:
        item = que.get()
        print currentThread().name, "got", item
        time.sleep(5)

for i in range(10):      # Starting 10 consumers !
    t = Thread(target=run_consumer)
    t.start()

run_producer()
```

Code:
producer_consumers_que.py

Queue Signaling

- Queues also have a signaling mechanism

```
q.task_done()      # Signal that work is done
q.join()           # Wait for all work to be done
```

- Many Python programmers don't know about this (since it's relatively new)
- Used to determine when processing is done

Producer Thread

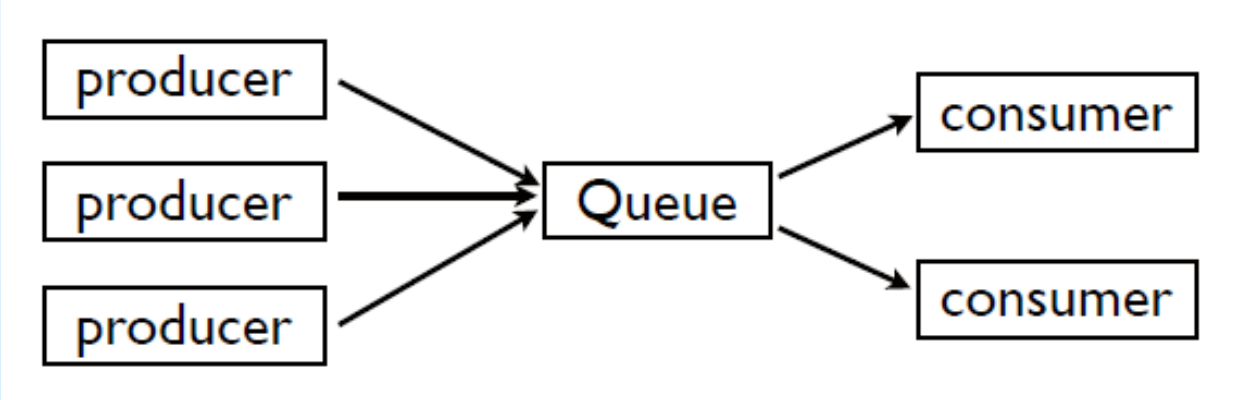
```
for item in produce_items():
    q.put(item)
# Wait for consumer
q.join()
```

Consumer Thread

```
while True:
    item = q.get()
    consume_item(item)
    q.task_done()
```

Queue Programming

- There are many ways to use queues
- You can have as many consumers/producers as you want hooked up to the same queue



- In practice, try to keep it simple !

Task Producer

- Can be defined in a function or in a class
- Here is a simple one in a function

```
# Keep producing unlimited number of tasks
# Every task is pushed to a task_que

def task_producer(id, task_que):
    while True:
        a = random.randint(0,100)   # random int from 0 to 100
        b = random.randint(0,100)   # random int from 0 to 100
        task = "%d*%d"  %  (a,b)    # multiplication task
        time.sleep(3)               # 3 sec to produce a task
        task_que.put(task)
        print "Producer %d produced task: %s" % (id, task)
```

Code:
producer_consumer_1.py

Worker (consumer)

- Can be defined in a function or in a class
- Here is a simple one in a function

```
# Accepts unlimited number of tasks (from task_que)
# It solves a task and puts the result in the result_que.

def worker(id, task_que, result_que):
    while True:
        task = task_que.get()
        t = random.uniform(2,3)  # Take 2-3 seconds to complete a task
        time.sleep(t)
        answer = eval(task)
        result_que.put(answer)
        print "Worker %d completed task %s: answer=%d" % (id, task, answer)
```

Code:
producer_consumer_1.py

Simulation: 2 producers, 3 workers

```
def simulation2():
    task_que = Queue()
    result_que = Queue()

    # Two producers
    p1 = Thread(target=task_producer, args=(1, task_que))
    p2 = Thread(target=task_producer, args=(2, task_que))

    # Three workers
    w1 = Thread(target=worker, args=(1, task_que, result_que))
    w2 = Thread(target=worker, args=(2, task_que, result_que))
    w3 = Thread(target=worker, args=(3, task_que, result_que))

    p1.start()
    p2.start()
    w1.start()
    w2.start()
    w3.start()
    # producers and workers run forever ...
```

Code: producer_consumer_1.py

Performance Test

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

- Threaded execution

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
```

- Now, you might expect two threads to run twice as fast on multiple CPU cores

Code: threads_perf.py

Performance Test

Bizarre Results

- Performance comparison (Dual-Core 2Ghz Macbook, OS-X 10.5.6)

  Sequential : 24.6s
  Threaded : 45.5s (1.8X slower!)

- If you disable one of the CPU cores...

  Threaded : 38.0s

- Insanely horrible performance. Better performance with fewer CPU cores? It makes no sense.

Code: threads_perf.py

Threads Summary (1)

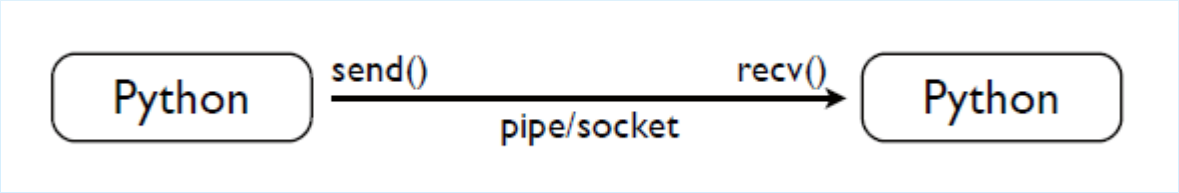
- To understand why this is so and how to make better use of threads, keep reading David Beazley Paper at: http://www.dabeaz.com/usenix2009/concurrent/
- Threads are still useful for I/O-bound apps, and do save time in these situations (which are more common than CPU-bound apps)
- For example : A network server that needs to maintain several thousand long-lived TCP connections, but is not doing tons of heavy CPU processing
- Most systems don't have much of a problem -- even with **thousands of threads**

Threads Summary (2)

- If everything is I/O-bound, you will get a very quick response time to any I/O activity
- Python isn't doing the scheduling
- So, Python is going to have a similar response behavior as a C program with a lot of I/O bound threads
- Python threads are a useful tool, but you have to know how and when to use them
- I/O bound processing only
- Limit CPU-bound processing to C extensions (that release the GIL)
- To parallel CPU bound applications use Python’s multiprocessing module … our next topic

Multi Processing

- An alternative to threads is to run multiple independent copies of the Python interpreter
- In separate processes
- Possibly on different machines
- Get the different interpreters to cooperate by having them send messages to each other



- Each instance of Python is independent
- Programs just send and receive messages

Message Passing

- Two main issues:
- What is a message?
- What is the transport mechanism?

- A Message is just a bunch of bytes (buffer)
- A "serialized" representation of some data
- Could be done via files, but it’s very slow and volatile

Message Transport

- Pipes
- Sockets
- FIFOs
- MPI (Message Passing Interface)
- XML-RPC (and many others)

Pipe Example 1

- The bc.exe (**Berkeley Calculator**) performs Math much faster than Python (think of it as a simple Matlab)
- **bc.exe** reads from **stdin** and writes to **stdout**
- It is included in the parallel programming code bundle
- Here is a **bc** program to calculate PI from term m to term n:

```
# This is not Python! This is a bc code to
# for the Gregory-Leibnitz series for of pi:
#     pi  =  4/1 - 4/3 + 4/5 - 4/7 + ...

define psum(m,n) {
    auto i
    s=0
    for (i=m; i < n; ++i)
        s = s + (-1)^i * 4.0/(2*i+1.0)
    return (s)
}
```

Pipe Example 1

```
import subprocess

code = """
    define psum(m,n) {
        auto i
        s=0
        for (i=m; i < n; ++i)
            s = s + (-1)^i * 4.0/(2*i+1.0)
        return (s)
    }
"""     # This is code in a totally different language !

# Starting a pipe to the bc.exe program
p = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

# Sending code to bc by writing to the Python side of the Pipe
p.stdin.write(code)
p.stdin.write("scale=60\n")           # 60 digits precision
p.stdin.write("psum(0,1000000)\n")    # Now we do the calculation!
result = p.stdout.readline()          # Now we read the result!
p.terminate()
print result
```

IPC/pipe_to_bc_1.py

Pipe Example 2

- If one sub-process gets us a lot of speed, how about opening two sub-processes in parallel?

```
# Starting two pipes to the bc.exe program!
p1 = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
p2 = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

# Sending code to bc by writing to the Python side of the Pipe
p1.stdin.write(code)
p2.stdin.write(code)
p1.stdin.write("scale=60\n")   # 60 digits precision
p2.stdin.write("scale=60\n")

# Now we do the calculation!
# Both processes run in parallel in the background !
p1.stdin.write("psum(0,500000)\n")          # We divide the task to two parts !
p2.stdin.write("psum(500000, 1000000)\n")   # Part 2

result1 = p1.stdout.readline()
result2 = p2.stdout.readline()
p1.terminate()
p2.terminate()
print Decimal(result1) + Decimal(result2)
```

IPC/pipe_to_bc_2.py

Pipe Example 3

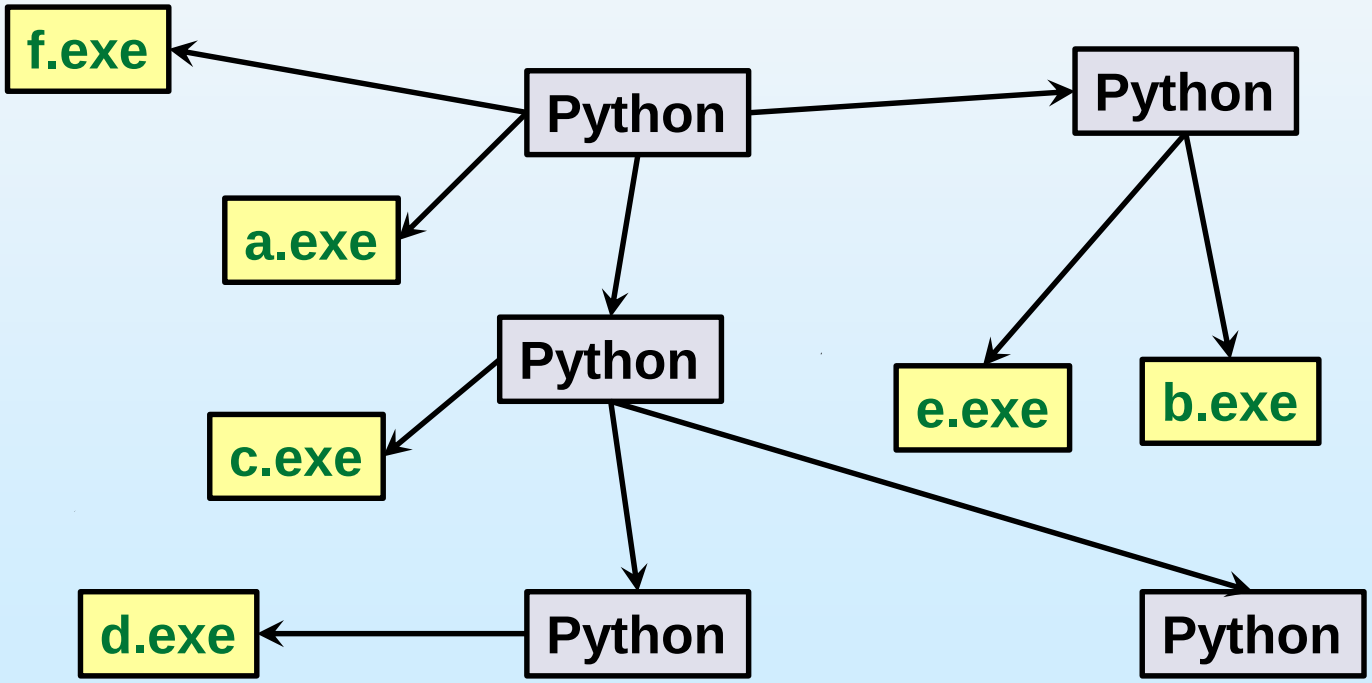
- That worked all right, but if we want to use our 8 CPU cores, we need to be more prudent!

```
def bc_worker(a,b):
    p = subprocess.Popen(["bc.exe"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
    p.stdin.write(code)
    p.stdin.write("scale=60\n")    # 60 digits precision
    p.stdin.write("psum(%d,%d)\n" % (a,b))
    return p

# 8 parallel sums of 500K terms chunks ... (total 4M terms)
procs = []
chunk = 500000
for i in range(8):
    a = i * chunk
    b = (i+1) * chunk
    p = bc_worker(a,b)
    procs.append(p)

getcontext().prec = 60
result = Decimal("0.0")
for p in procs:
    r = p.stdout.readline()
    p.terminate()
    result += Decimal(r)
```

IPC/pipe_to_bc_3.py

The Big Picture

- Can easily have 10s-100s-1000s of communicating Python interpreters and external programs through pipes and sockets
- However, always keep the “golden rules” in mind …



Golden rule: He who has the gold makes the rules

The Multiprocessing Module

- This is a module for writing concurrent programs based on communicating processes
- A module that is especially useful for concurrent CPU-bound processing
- Here's the cool part:
  You already know how to us multiprocessing!
- It is exactly as using Threads, just replace “Thread” with “Process”
- Instead of "**Thread**" objects, you now work with "**Process**" objects
- But! One small difference: you need to use Queue’s for process communication (or else you have independent processes with no shared data at all)

Multiprocessing Example 1

- Define tasks using a Process class
- You inherit from Process and redefine run()

```
import time, os
from multiprocessing import Process

print "Parent Process id:", os.getpid()

class CountdownProcess(Process):
    def __init__(self, name, count):
        Process.__init__(self)
        self.name = name
        self.count = count

    def run(self):
        print "Child Process id:", os.getpid()
        while self.count > 0:
            print "%s:%d" % (self.name, self.count)
            self.count -= 1
            time.sleep(2)

        return
```

countdownp1.py

Multiprocessing Example 1

- To launch, same idea as with threads
- You inherit from Process and redefine run()

```
if __name__ == '__main__':
    p1 = CountdownProcess("A", 10)  # Create the process object
    p1.start()                      # Launch the process

    p2 = CountdownProcess("B", 20)  # Create another process
    p2.start()                      # Launch
```

countdownp1.py

- Processes execute until run() stops
- critical detail: Always launch in main as shown (or else your Windows will crash)

Multiprocessing Example 2

- Alternative method of launching processes is by using simple functions instead of classes

countdownp2.py

```
def countdown(name, count):
    print "Process id:", os.getpid()
    while count > 0:
        print "%s:%d" % (name, count)
        count -= 1
        time.sleep(2)
    return

# Sample execution
if __name__ == '__main__':
    p1 = Process(target=countdown, args=("A", 10))
    p2 = Process(target=countdown, args=("B", 20))

    p1.start()
    p2.start()
```

- Creates two Process objects, but their run() method just calls the countdown function

Does it Work ?

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential Execution:

```
count(100000000)
count(100000000)
```

→ 24.6s

- Multiprocessing Execution

```
p1 = Process(target=count,args=(100000000,))
p1.start()
p2 = Process(target=count,args=(100000000,))
p2.start()
```

→ 12.5s

- Yes, it seems to work

Other Process Features

- Joining a process (waits for termination)

```
p = Process(target=somefunc)
p.start()
...
p.join()
```

- Making a daemonic process

```
p = Process(target=somefunc)
p.daemon = True
p.start()
```

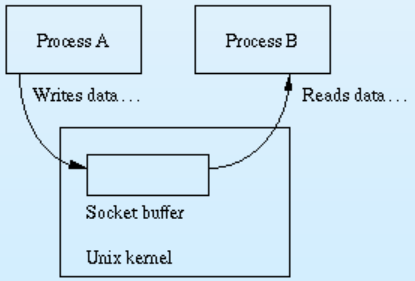
- Terminating a process

```
p = Process(target=somefunc)
...
p.terminate()
```

- These mirror similar thread functions

Distributed Memory

- Unlike Threads, with multiprocessing, there are no shared data structures, in fact no sharing at all !
- Every process is completely isolated!
- Since there are no shared structures, forget about all of that locking business
- Everything is focused on messaging



http://fxa.noaa.gov/kelly/ipc/

Pipes

- A channel for sending/receiving objects

```
(c1, c2) = multiprocessing.Pipe()
```

- Returns a pair of connection objects (one for each end-point of the pipe)
- Here are methods for communication

```
c.send(obj)             # Send an object
c.recv()                # Receive an object

c.send_bytes(buffer)    # Send a buffer of bytes
c.recv_bytes([max])     # Receive a buffer of bytes

c.poll([timeout])       # Check for data
```

Pipe Example 1

- A simple data consumer

```
From multiprocessing import Process, Pipe

def consumer(p1, p2):
    p1.close()   # Close producer's end (not used)
    while True:
        try:
            item = p2.recv()
        except EOFError:
            break
        print "Consumer got:", item
```

- A simple data producer

```
def producer(outp):
    print "Process id:", os.getpid()
    for i in range(10):
        item = "item" + str(i)   # make an item
        print "Producer produced:", item
        outp.send(item)
```

pipe_for_producer_consumer.py

Pipe Example 1

- **Launching Consumer and Producer**
- The consumer runs in a child process
- But the producer runs in the parent process
- Communication is from parent to child

```
if __name__ == '__main__':
    p1, p2 = Pipe()

    c = Process(target=consumer, args=(p1, p2))
    c.start()

    # Close the input end in the producer
    p2.close()

    run_producer(p1)

    # Close the pipe
    p1.close()
```

pipe_for_producer_consumer.py

Message Queues

- multiprocessing also provides a queue
- The programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)       # Put an item on the queue
item = q.get()    # Get an item from the queue
```

- There is also a joinable Queue

```
from multiprocessing import JoinableQueue

q = JoinableQueue()
q.task_done()     # Signal task completion
q.join()          # Wait for completion
```

Queue Implementation

- Queues are implemented on top of pipes
- A subtle feature of queues is that they have a "feeder thread" behind the scenes
- Putting an item on a queue returns immediately
  - Allowing the producer to keep working
- The feeder thread works on its own to transmit data to consumers

Deadlocks

- Assume Alice wants to transfer money to Bob and at the same time Bob wants to transfers money to Alice
- Alice's bank grabs a lock on Alice's account, then asks Bob's bank for a lock on Bob's account
- Bob's bank locked Bob's account and is now asking for a lock on Alice's account
- Bang! you have a deadlock!

http://www.eveninghour.com/images/online_transfer2.jpg

Code:
DEADLOCK/bank_account_1.py
DEADLOCK/bank_account_2.py

Dining Philosophers

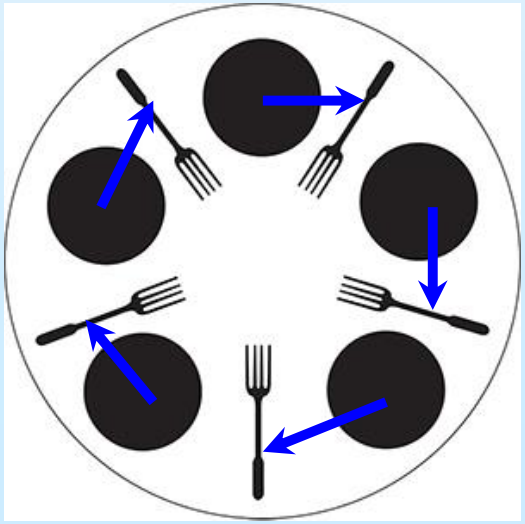
- 5 philosopher with 5 forks sit around a circular table
- The forks are placed between philosophers
- Each philosopher can be in one of three states:
  - **Thinking** (job is waiting)
  - **Hungry** (job ready to run)
  - **Eating** (job is running)
- To eat, a philosopher must have two forks
  - He must first obtain the **first fork** (left or right)
  - After obtaining the **first fork** he proceeds to obtain the **second fork**
  - Only after having two forks he is allowed to eat
  - (The two forks cannot be obtained simultaneously!)
- Analogy: a process that needs to access two resources: a disk and printer for example

Code: DEADLOCK/dining_philosophers.py

Dining Philosophers: Deadlock

- A Deadlock is a situation in which all 5 philosophers are hungry but none can eat forever since each philosopher is waiting for a fork to be released
- Sometimes this situation is called: **full starvation**
- In operating systems, a philosopher represents a thread or a process that need access to two resources (like two files or a disc and printer) in order to proceed
- Operating system puts every process into a device Queue each time it needs to access a device (disc, memory, or CPU)

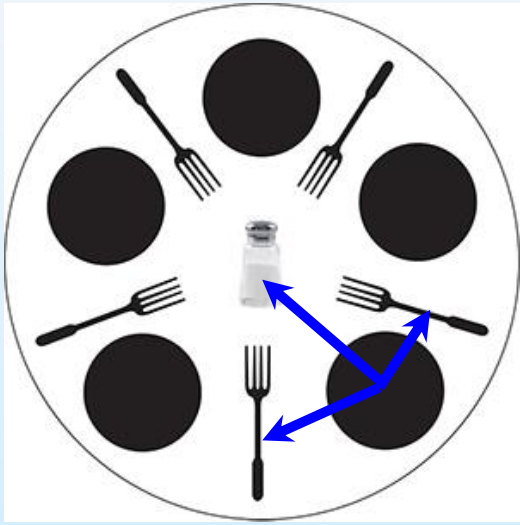
Typical deadlock situation: Each Philosopher grabbed the left fork and waits for the right fork



Code: DEADLOCK/dining_philosophers.py

Dining Philosophers: Solution 1

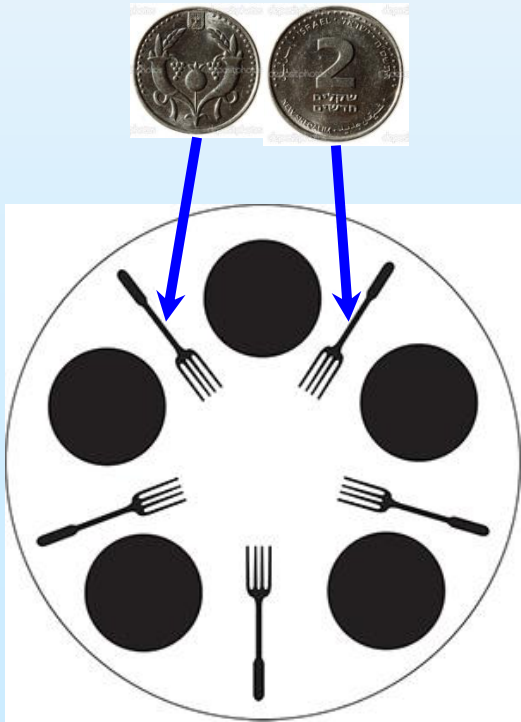
- A philosopher who wants to eat first picks up the salt shaker on the table
- Assume only one salt shaker exists!
- All other philosophers that do not have the salt shaker must release their forks
- The philosopher that got the salt shaker picks up his forks, eats and when finishes must put the salt shaker back at the table center
- This solution works but is not optimal: only one philosopher can eat at any given time
- if we further stipulate that the philosophers agree to go around the table and pick up the salt shaker in turn, this solution is also fair and ensures no philosopher starves.



Code: DEADLOCK/dining_philosophers.py

Dining Philosophers: Solution 2

- Each philosopher flips a coin:
  - Heads, he tries to grab the right fork
  - Tails, he tries to grab the left fork
- If the second fork is busy, release the first fork and try again
- With probability 1, he will eventually eat
- Again, this solution relies on defeating circular waiting whenever possible and then resorts to breaking 'acquiring while holding' as assurance for the case when two adjacent philosophers' coins both come up the same.
- Again, this solution is fair and ensures all philosophers can eat eventually.



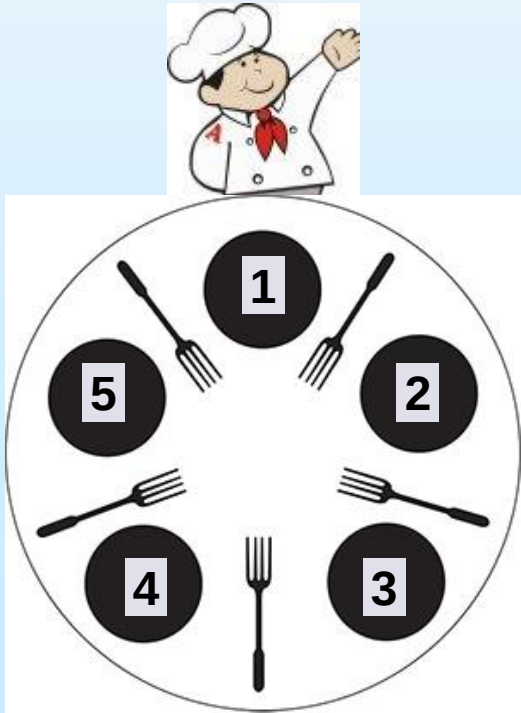
Code: DEADLOCK/dining_philosophers.py

Dining Philosophers: Solution 3

- The chef that cooked the meal dictates who should eat and when to prevent any confusion. This breaks the 'blocking shared resources' condition.
- The chef assures all philosophers that when they try to pick up their forks, they will be free!
- Effectively the chef enforces a fair “fork discipline” over the philosophers
- This is the most efficient solution (no shared resources/locking involved) but is in practice the hardest to achieve (the chef must know how to instruct the philosophers to eat in a fair, interference-free fashion).
- **For example**, the chef can assign a number to each philosopher and decide that the following pairs of philosophers eat at the following order:

  (3, 5) -> (1, 4) -> (2, 4) -> (1, 3) -> (5, 2)

- This schedule ensures that each philosopher gets to eat twice in each round and will neither deadlock nor starve

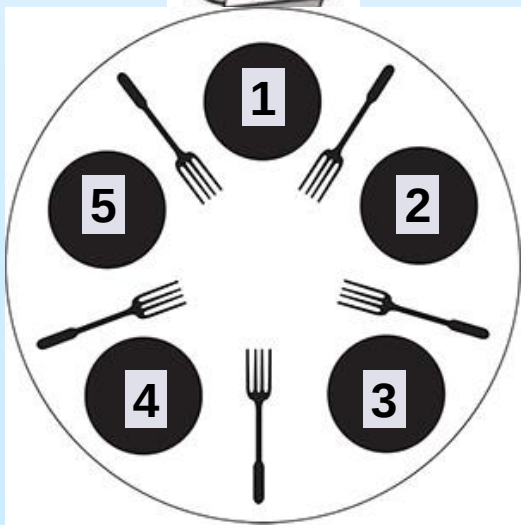


Dining Philosophers Solution 3 Implementation

- A Python program model for the dining philosophers is coded in the file:

PARALLEL_PROGRAMMING_LAB/DEADLOCK/dining_philosophers.py

- Based on this code, try to implement a Chef Thread which monitors the 5 philosophers and solves the problem as described above
- How to go about solution 2 ?



Dining Philosophers: Solution 4

- Each philosopher behaves as usual. That is whenever it gets hungry, he is trying to acquire the two forks as usual (in whatever order he wants)
- Each Philosopher is assigned a **“Hunger Index”**
- This is roughly the time that has passed since he last ate
- As soon as the highest Hunger Index rises above a fixed threshold, the neighbors of this philosopher must release the forks near the starving philosopher (or complete their food if they were eating and then release the forks)
- This guarantees that the starving philosopher will get to eat in a short time.
- Once the starving philosopher is satiated, his “Hunger Index” drops down below the next starving philosopher
- How would you implement this solution?
  Start with the file:
  PARALLEL_PROGRAMMING_LAB/DEADLOCK/dining_philosophers.py