Administrivia

• HW1 due Oct 4.

• Lectures now being recorded.
  • I’ll post URLs when available.

• Discussing Readings on Monday.
  • Keep posting discussion on Piazza
Python Multiprocessing
Topics today:

Multiprocessing vs. Threads

Multiprocessing module in Python
A brief overview of Processes and Threads
<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>“Heavy”</td>
<td>“Lightweight”</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
# Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Different OS processes</strong></td>
<td><strong>Same OS process</strong></td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>&quot;Heavy&quot;</td>
<td>&quot;Lightweight&quot;</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
OS support

• Underneath the hood, Processes and Threads are not that different.

• Hardware treats them identically.

• Serial process == single thread.

• They can run on any processor, any core.
# Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>&quot;Heavy&quot;</td>
<td>&quot;Lightweight&quot;</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
Sharing

• Multiprocessing = separate processes
  • same code, but…
  • separate memory spaces
  • separate file handles
  • separate everything
• Threading = same process
  • same code, memory, variables, files…
## Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>&quot;Heavy&quot;</td>
<td>&quot;Lightweight&quot;</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
Heavy vs. Lightweight

- Starting a process vs. starting a thread
  - More state = more work
  - Separate memory = more resources
    - ... and some caching costs
- Linux / OSX: process \( \approx \) thread
- Windows: process \( \gg \) thread
## Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td><strong>&quot;Heavy&quot;</strong> (Linux)</td>
<td><strong>&quot;Lightweight&quot;</strong></td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
## Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>&quot;Heavy&quot; (Linux)</td>
<td>&quot;Lightweight&quot;</td>
</tr>
</tbody>
</table>

- Message Passing
- Shared variables & Locks
Communication

• Multiprocessing - separate processes
  • Communication via “pipes”.
  • Data has to be converted to bytestrings before being sent between processes.

• Threads - same memory
  • Communicate using shared variables.
  • Use Locks to ensure safe access.
Locks

• A Lock is just a special type of variable.

  • Can be acquired.

  • Can be released.

  • Can’t be acquired twice at the same time.

Another thread that tries to acquire it will wait until it is released.
The GIL: Global Interpreter Lock

• In Python, only one thread can be running at a time, by default.

• Enforced by the GIL.

• So all variables have an implicit lock.

• … but between statements, that can go away. *

* technically, between instructions in the bytecode interpreter.
Lock example

- Three threads want to write to the same variable...
  - Driver: put work on the list of things to be done.
  - Worker 1: take tasks off of the list.
  - Worker 2: take tasks off of the list.
# worklist.py

work_list = [1, 2, 3, 4]

def get_work():
    global work_list
    job = work_list[0]
    work_list = work_list[1:]  
    return job
Question

What if both workers access the list at the same time?
Critical regions

• Any code where only one thread should be at a time is a “Critical Region”, and needs something like a lock.

```python
# worklist.py

work_list = [1, 2, 3, 4]

def get_work():
    global work_list
    job = work_list[0]
    work_list = work_list[1:]
    return job
```
Critical regions

• Any code where only one thread should be at a time is a “Critical Region”, and needs something like a lock.

```python
# worklist.py

work_list = [1, 2, 3, 4]

def get_work():
    global work_list
    job = work_list[0]
    work_list = work_list[1:]
    return job
```
Critical regions

• Any code where only one thread should be at a time is a “Critical Region”, and needs something like a lock.

```python
# worklist.py

work_list = [1, 2, 3, 4]
lock = Lock()

def get_work():
    global work_list
    with lock:
        job = work_list[0]
        work_list = work_list[1:]
    return job
```
## Multiprocessing vs. Threading

<table>
<thead>
<tr>
<th>Multiprocessing</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different OS processes</td>
<td>Same OS process</td>
</tr>
<tr>
<td>Separate memory (&quot;share nothing&quot;)</td>
<td>Same memory space (&quot;share everything&quot;)</td>
</tr>
<tr>
<td>&quot;Heavy&quot; (Linux)</td>
<td>&quot;Lightweight&quot;</td>
</tr>
<tr>
<td>Message Passing</td>
<td>Shared variables &amp; Locks</td>
</tr>
</tbody>
</table>
Python Multiprocessing module
Creating processes

```python
import multiprocessing
import time

def some_function():
    time.sleep(5)
    print("Hello World")

p = multiprocessing.Process(target=some_function)
p.start()
```
start() / join()

- **start()** - starts the process, calls its run() method.
  - If created with Process(target=function), run() just calls that function.
- **join()** - collect dead processes
  (happens automatically in some cases).
Process Pools

- Create N workers at once.
- Serve them jobs using map() or apply()
  (or imap, or the *_async variants)
map example
map / map_async

- map()
  - wait for result

- map_async()
  - returns immediately.
  - return a MapResult type, which get() method.
  - (also ready() and wait() methods)
map_async example
Queues & Pipes

• Interprocess, first-in, first-out.

• Pipe
  • One input, one output.

• Queue
  • Multiple writers, multiple readers
Queue example
Communication costs

- Serialization (or marshaling)
  - Must convert Python objects to byte strings (and back)
- Transport
- Locking
  - Make sure Queue is accessed safely.
Shared memory

• It actually is possible to share memory between processes.

• multiprocessing.Value, multiprocessing.Array

• (...and now you need locks... )
Monday

- GIL Reading discussion
- more Multiprocessing.