# CS 450: Operating Systems
# Lecture 6: Concurrent Programming

*Spring 2014, J. Sasaki*
*Dept of Computer Science*
*Illinois Institute of Technology*

# *Threads and Processes in Python*



2

## ***Lec06_procs1.py – List of Processes***

```
from multiprocessing import Process
import time
import random

def say_hello(id, seconds):
    print('Child {} is running'.format(id))
    time.sleep(seconds)
    print('Child {} is done'.format(id))
```

```
def main(nbr_procs = 5):
    # ps is a list of process objects
    ps = [Process(target=say_hello,
              args=([i, random.randint(1,9)]))
          for i in range(nbr_procs)]

    for p in ps:
        p.start()
        print('Started process {}'.format(p.pid))

    for p in ps:
        p.join()
        print('Joined process {}'.format(p.pid))

main()
```



4

```
> python3 Lec06_procs1.py
Started process 4074
Started process 4075
Child 0 is running
Started process 4076
Child 1 is running
Started process 4077
Started process 4078
Child 2 is running
Child 3 is running
Child 4 is running
Child 3 is done
Child 4 is done
Child 1 is done
Child 2 is done
Child 0 is done
Joined process 4074
Joined process 4075
Joined process 4076
Joined process 4077
Joined process 4078
```

***Lec06_thds2.py – List of threads***

```
# Create a list of threads, start them all,
# then join them all
#
from threading import Thread
import time  # for sleep
import random

# say_hello prints its id and sleeps for
# a given nbr of seconds
#
def say_hello(id, seconds):
   print('Child {} is running'.format(id))
   time.sleep(seconds)
   print('Child {} is done'.format(id))
```

```
# main(nbr_thds) creates a number of threads (default 5)
# then it starts all the threads, and then it waits until
# they all finish. Each thread sleeps for a random number
# of seconds >= 1 and <= 9),
#
def main(nbr_thds = 5):
    # ts is a list of thread objects
    ts = [Thread(target=say_hello,
              args=([i, random.randint(1,9)]))
        for i in range(nbr_thds)]

    for t in ts:
        t.start()
        print('Started thread {}'.format(t.ident))

    for t in ts:
        t.join()
        print('Joined thread {}'.format(t.ident))

main()
```

```
>python3 Lec06_thds2.py
Child 0 is running
Started thread 4318040064
Child 1 is running
Started thread 4327477248
Child 2 is running
Started thread 4332732416
Child 3 is running
Started thread 4337987584
Child 4 is running
Started thread 4343242752
Child 2 is done
Child 4 is done
Child 3 is done
Child 1 is done
Child 0 is done
Joined thread 4318040064
Joined thread 4327477248
Joined thread 4332732416
Joined thread 4337987584
Joined thread 4343242752
```

***Lec06_pool3.py – Use pool of processes***

```
from multiprocessing import Pool
import os, random, time

# This time, say hello prints out the id and returns
# its process id. To make it easier (?) to read the
# output, the messages about the child running /
# finishing have >'s prepended when we start and <'s
# when we finish.
#
def say_hello(id):
    print(id*'>' + ' Child {} is running'.format(id))
    time.sleep(1/random.randint(1,9))
    print(id*'<' + ' Child {} is finished'.format(id))
    return os.getpid()
```

```
# Create a number of say hello processes and run them
# using the pool of available processes.  We print
# out a list of the process ids used. Note the number
# of distinct process ids = poolsize.
#
def main(poolsize=2, nbr_procs=8):
   pool = Pool(processes = poolsize);
   print(pool.map(say_hello, range(nbr_procs)))
   pool.close()    # Start cleanup
   pool.join()     # Wait for cleanup to finish

main()
```

```
 Child 0 is running
> Child 1 is running
 Child 0 is finished
< Child 1 is finished
>> Child 2 is running
>>> Child 3 is running
<< Child 2 is finished
>>>> Child 4 is running
<<< Child 3 is finished
>>>>> Child 5 is running
<<<< Child 4 is finished
>>>>>> Child 6 is running
<<<<< Child 5 is finished
>>>>>>> Child 7 is running
<<<<<<< Child 7 is finished
<<<<<< Child 6 is finished
[4207, 4208, 4207, 4208, 4207, 4208, 4207, 4208]
```

```
>>> main(poolsize=5) # More concurrency
 Child 0 is running
> Child 1 is running
>> Child 2 is running
>>> Child 3 is running
>>>> Child 4 is running
<< Child 2 is finished
>>>>> Child 5 is running
< Child 1 is finished
>>>>>> Child 6 is running
 Child 0 is finished
>>>>>>> Child 7 is running
<<<< Child 4 is finished
<<< Child 3 is finished
<<<<<< Child 6 is finished
<<<<<<< Child 7 is finished
<<<<< Child 5 is finished
[4222, 4223, 4224, 4225, 4226, 4224, 4223, 4222]
```

```
>>> main(poolsize=5, nbr_procs=20) # More procs
 Child 0 is running
> Child 1 is running
>> Child 2 is running
>>> Child 3 is running
>>>> Child 4 is running
<<< Child 3 is finished
>>>>> Child 5 is running
<< Child 2 is finished
>>>>>> Child 6 is running
<<<<< Child 5 is finished
>>>>>>> Child 7 is running
 Child 0 is finished
< Child 1 is finished
>>>>>>>> Child 8 is running
>>>>>>>>> Child 9 is running
<<<<<<< Child 7 is finished
>>>>>>>>>> Child 10 is running
<<<<<<<< Child 8 is finished
>>>>>>>>>>> Child 11 is running
```

```
<<<<<< Child 6 is finished
<<<<<<<<< Child 9 is finished
>>>>>>>>>>>> Child 12 is running
>>>>>>>>>>>>> Child 13 is running
<<<<<<<<<<<< Child 12 is finished
>>>>>>>>>>>>>> Child 14 is running
<<<< Child 4 is finished
>>>>>>>>>>>>>>> Child 15 is running
<<<<<<<<<<< Child 11 is finished
>>>>>>>>>>>>>>>> Child 16 is running
<<<<<<<<<<<<< Child 13 is finished
>>>>>>>>>>>>>>>>> Child 17 is running
<<<<<<<<<<<<<<< Child 15 is finished
>>>>>>>>>>>>>>>>>> Child 18 is running
<<<<<<<<<<<<<<<< Child 16 is finished
>>>>>>>>>>>>>>>>>>> Child 19 is running
<<<<<<<<<<<<<<<<<< Child 18 is finished
<<<<<<<<<<<<<<<<<<< Child 19 is finished
<<<<<<<<<<<<<<<<< Child 17 is finished
<<<<<<<<<< Child 10 is finished
<<<<<<<<<<<<<< Child 14 is finished
[4371, 4372, 4373, 4374, 4375, 4374, 4373, 4374, 4371, 4372,
4374, 4371, 4373, 4372, 4373, 4375, 4371, 4372, 4375, 4371]
>>>
```

```
>>> main(poolsize=20, nbr_procs=20) # More concurrency?
 Child 0 is running
> Child 1 is running
>> Child 2 is running
>>> Child 3 is running
>>>> Child 4 is running
>>>>> Child 5 is running
>>>>>> Child 6 is running
>>>>>>> Child 7 is running
>>>>>>>> Child 8 is running
>>>>>>>>> Child 9 is running
>>>>>>>>>> Child 10 is running
>>>>>>>>>>> Child 11 is running
>>>>>>>>>>>> Child 12 is running
>>>>>>>>>>>>> Child 13 is running
>>>>>>>>>>>>>> Child 14 is running
>>>>>>>>>>>>>>> Child 15 is running
>>>>>>>>>>>>>>>> Child 16 is running
>>>>>>>>>>>>>>>>> Child 17 is running
>>>>>>>>>>>>>>>>>> Child 18 is running
>>>>>>>>>>>>>>>>>>> Child 19 is running
```



15

```
<<<<<<< Child 7 is finished
<<<< Child 4 is finished
<<<<<<<<<<<<<<<<<<< Child 19 is finished
<<<<<<<<<<<<<< Child 14 is finished
<<<<< Child 5 is finished
<<<<<<<<<<< Child 11 is finished
<<<<<<<<<<<<<<<< Child 16 is finished
<<<<<<<<< Child 9 is finished
<<<<<<<<<<<<<<<<< Child 17 is finished
< Child 1 is finished
<<<<<< Child 6 is finished
<<< Child 3 is finished
<<<<<<<< Child 8 is finished
<<<<<<<<<< Child 10 is finished
<<<<<<<<<<<<< Child 13 is finished
<<<<<<<<<<<<<<< Child 15 is finished
<< Child 2 is finished
<<<<<<<<<<<<<<<<<< Child 18 is finished
 Child 0 is finished
<<<<<<<<<<<< Child 12 is finished
[4339, 4340, 4341, 4342, 4343, 4344, 4345, 4346, 4347, 4348,
4349, 4350, 4351, 4352, 4353, 4354, 4355, 4356, 4357, 4358]
>>>
```

# *Concurrent Programming*

# ***Why Concurrent Programming?***

- Break up program to understand it better
- Avoid blocking whole program …
  to improve resource utilization
- To speed up our programs?
  - Run different threads on different CPUs

# *Improving Performance via Concurrency*

- With 1 processor we still might improve performance using concurrency.
- Run I/O- and CPU-bound parts of our program concurrently (less time waste).
- Waiting for different I/O devices might be done concurrently.
- Note concurrency might *degrade* performance due to overhead.

# *Improving Performance via Simultaneous Execution*

- Our intuition says the more computations we do truly in parallel, we sooner we should finish.
- But performance doesn't increase linearly with the number of processors/cores.
- Also need kernel-supported threads (for threaded programs)

# *Parallelizing Code*

- *Parallelizing* code = Breaking up code into parts that can be run simultaneously.
- Usually can't break up all the code — there's some *serial part* that can't be parallelized.
- Classic example of perfectly parallelizable code: Matrix Multiplication

# *Matrix Multiplication*

# *Matrix Multiplication*

- First implementation: plain sequential (not parallel); triply-nested loop
  - (m×n matrix) × (n×p matrix) = (m×p matrix)
  - C[i][j] = $\Sigma_{k=0 \ldots n-1}$ A[i][k] * B[k][j]
    - where i ($0 \le i < m$) is a row number for A and j ($0 \le j < p$) is a column number for B

```
     [[7, 7],           [[7, 5,      6],
A =          ,     B =  [4, 9,       5]]
      [9, 8]]

               [[77,  98, 77, 77]
A×B = C = [27,  50,      31],
               [95, 117, 96, 94]]
```

$1 \times 8 + 5 \times 3 = 23$



24

***Lec06_mmu4.py -- Matrix Multiplication***

```
import random
random.seed(0) # for repeatable results

(m, n, p) = (30, 50, 70)

A = [[random.randint(1, 9) for _ in range(n)] \
     for _ in range(m)]
B = [[random.randint(1, 9) for _ in range(p)] \
     for _ in range(n)]
```

```python
# Sequentially multiply matrix A x B; return
# result
#
def seq_mat_mult():
    C = [[0 for col in range(p)] \
        for row in range(m)]

    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```



26

```
# Run sequential multiplications and return time
# to completion in ms
#
from time import time
def seq():
    start = time()
    C_seq = seq_mat_mult()
    end = time()
    seq_delta = 1000*(end-start)
    print('(SEQ) Elapsed: {:0.1f} ms'.\
        format(seq_delta))
    return seq_delta

# Run sequential multiplication nbr_trials times
# and print average of runtimes
#
def go_seq(nbr_trials = 5):
    times = [seq() for i in range(1, nbr_trials)]
    average = sum(times)/len(times)
    print('(SEQ) Average of {} runs is {:0.1f} ms'.\
        format(nbr_trials, average))
    return average
```

***Run sequential multiplication:***

```
> python3 -i Lec06_mm4.py
>>> C = seq_mat_mult()
>>> C
[[1168, 1411, 1306, ... omitted ....
>> go_seq()
(SEQ) Elapsed: 40.6 ms
(SEQ) Elapsed: 36.8 ms
(SEQ) Elapsed: 37.2 ms
(SEQ) Elapsed: 38.7 ms
(SEQ) Average of 5 runs is 38.3 ms
38.33878040313721
>>>
```

# *Parallel Execution*

- For parallel execution, we'll use a pool of processes; each process calculates a row of the result.
- The function `mat_mult_row(r)` calculates row `r` of the result (`0` ≤ `r` < `m`).
- The `par_mat_mult()` function will use `pool.map` to run `mat_mult_row(0)`, …, `mat_mult_row(m-1)` and collect the result.
- Size of process pool will affect speed.

***More of Lec06_mm4.py:***

```
# Row r of A (m x n) times B (n x p) = C (m x p)
#
def mat_mult_row(r): # 0 <= r < m
    result = [0 for col in range(p)]
    for j in range(p):
        for k in range(n):
            result[j] += A[r][k] * B[k][j]
    return result

from multiprocessing import Pool

# Calculate A times B with the rows of the
# result calculated in parallel
#
def par_mat_mult(poolsize = 2):
    pool = Pool(processes = poolsize)
    C = pool.map(mat_mult_row, range(m))
    pool.close()
    return C
```

```
# Run parallel multiplication and return time
# to completion in ms
#
def par(poolsize = 2):
    start = time()
    C_par = par_mat_mult(poolsize)
    end = time()
    par_delta = 1000*(end-start)
    print('(MAP) Elapsed: {:0.1f} ms'.format(par_delta))
    return par_delta

# Run parallel multiplications nbr_trials times
# and print average of runtimes for this pool size
#
def go_par(poolsize = 2, nbr_trials = 5):
    print('(MAP) With {} processes'.format(poolsize))
    times = [par(poolsize) for i in range(1, nbr_trials)]
    average = sum(times)/len(times)
    print('(MAP) Average of {} runs is {:0.1f} ms'.\
        format(nbr_trials, average))
    return average
```


31

***Run parallel multiplication:***

```
> python3 -i Lec06_mm4.py
>>> C = par_mat_mult()
>>> C == seq_mat_mult()
True
>>> go_par()
(MAP) With 2 processes
(MAP) Elapsed: 31.6 ms
(MAP) Elapsed: 25.1 ms
(MAP) Elapsed: 35.7 ms
(MAP) Elapsed: 31.7 ms
(MAP) Average of 5 runs is 31.0 ms
31.02630376815796
```



32

# Try Different Pool Sizes

- `>>> [_ for _ in map(go_par, range(1,10))`
  (output omitted)
- Results are: 38.3, 28.8, 27.2, 29.4, 31.8, 34.2, 42.5, 42.1, 42.8 ms
  - Pool size 3 is fastest
  - Compare with sequential version: 38.5 ms