LECTURE 5 PARALLEL PROGRAMMING WITH PYTHON

PYTHON THREADS

Flynn's classification

- SISD: single instruction single data
- SIMD: single instruction multiple data
- MISD: multiple instruction single data
- MIMD: multiple instruction multiple data

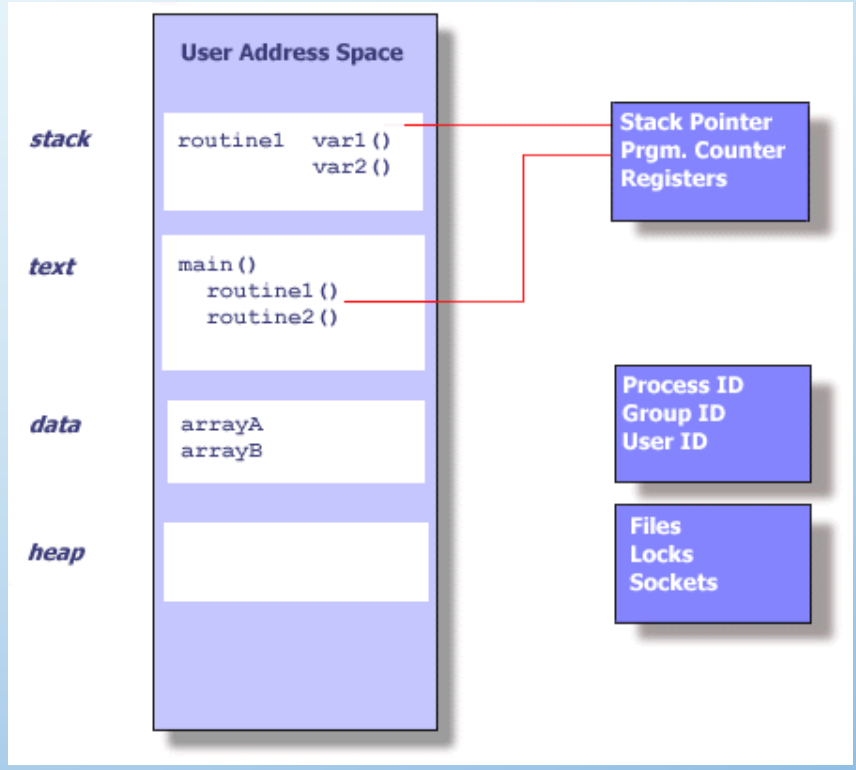
- MIMD is the most generic form of parallel programming:
  - The program creates multiple streams of instructions that operate simultaneously on different data.
  - Multithreading and multiprocessing are common mechanisms to create multiple streams (threads) of instructions.

Processes vs Threads

- Multithreading: program executes with multiple threads
- Multiprocessing: program executes with multiple processes.
- Process: A program in execution is called a process.
  - When you type ‘python3 myprog.py’, you basically create a process to run ‘python3 myprog.py’.
  - The main goal of a process is isolation: it creates an illusion that a process owns the whole computer even though many users can run programs at the same time.
  - Since many processes share the physical resources (e,g. CPU, hardware registers), to ensure correct program execution, a process should always be executed within its **process context**.

Process context

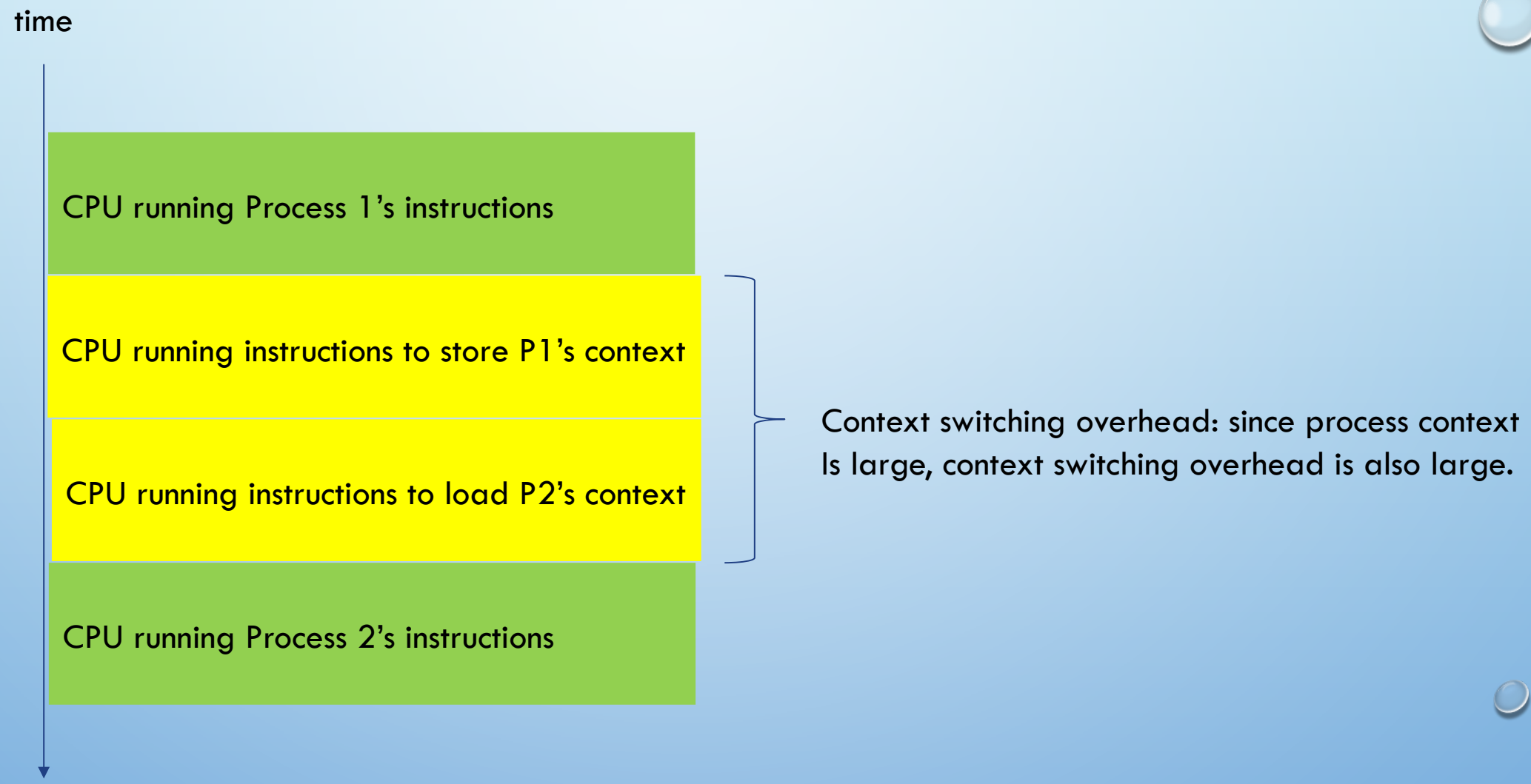
- Process context include all information/resources that are needed to run program correctly.
- Some of the items in process context:
  - Process ID
  - Environment
  - Program instructions
  - Registers (including PC)
  - Stack
  - Heap
  - Global memory
  - Shared libraries
  - ……
- The process context must be initialized before the system can run a program.
  - Process creation is an expensive operation.



Context Switching

- Many processes share computing resources such as CPU, registers.
- In order for a process to run correctly, it must execute within its own context.
- Before the computer starts the execution of a program, it must establish the process context for the program.
- Programs usually do not run to their completion. A program is often switched out to another program. In this case, the context for the other program (that gets the CPU) must be first re-estimated its context. This is called **context switching**.
- Context switching: Switching the context in the computer to run the program.
  - Example: all processes share the registers in CPU, so to switch from one program to another, all register values must be changed.

Processes sharing CPU



From Process to Thread

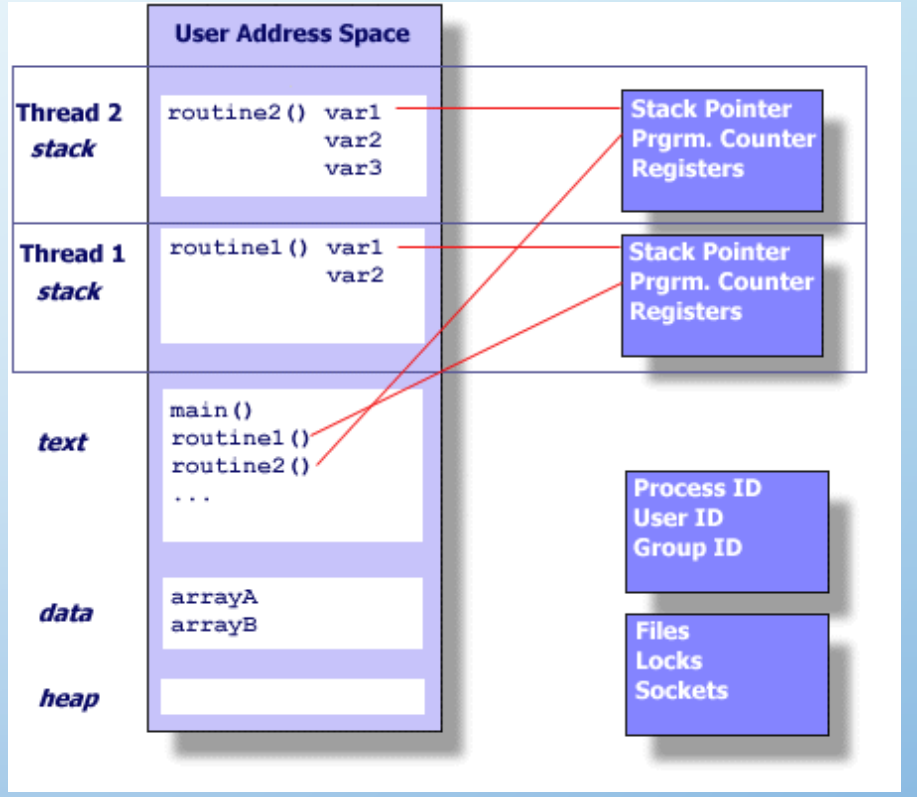
- Process is designed to achieve isolation
  - Large process context results in high process creation and context switching overheads
- In the context of parallel computing, why are we talking about processes? We want to use it to create multiple threads of execution within a program!
- If we do not care about isolation, but just want to have a way to create multiple threads of execution, can we make the context smaller? This is where thread comes into the picture.

Thread

- Threads exist within a process, and thus share (and have access to) all process context.
- Thread context is the minimum part of the process context that are absolutely necessary to support a thread of execution. Note that threads still share hardware resources.
  - Process is more for isolation than for just providing a thread of execution!
- What are the absolute necessary?
  - Process ID
  - Environment
  - Program instructions
  - Registers (including PC)
  - Stack
  - Heap
  - Global memory
  - Shared libraries
  - ……

Thread Context

- What are the absolute necessary?
  - Process ID
  - Environment
  - Program instructions
  - Registers (including PC)
  - Stack
  - Heap
  - Global memory
  - Shared libraries
  - ……
- Thread creation and switching is much cheaper than process creation and switching!
  - This is why thread is sometimes called lightweight process.



Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Multiprocessing vs Multithreading

- Multiprocessing: create multiple processes in a program and the processes work together to complete the task
  - Process creation and switching are expensive
  - Processes do not share memory by default. Communication between processes must use inter-process communication (IPC) mechanism
    - IPC is less prone to errors
- Multithreading: create multiple threads in a program and the threads work together to complete the task
  - Thread creation and switching are inexpensive
  - Threads share process memory. Communication between threads is trivial.
    - But this make multithreading prone to errors.
- In general, multithreading is preferred for parallel computing due to its performance advantages.

Concurrent Programming vs Parallel Programming

- Is multithreading/multiprocessing = parallel execution?
  - Multithreading/multiprocessing creates multiple streams of instructions
  - What about multithreading/multiprocessing on a computer with only one CPU?
- Multithreading/multiprocessing = concurrent programming
  - Multiple threads progress concurrently (but may not in parallel)
- Parallel programming
  - Multiple threads progress in parallel.
- Concurrent Programming and parallel programming share similar issues
  - When the same memory (variable) is accessed, there can be problems for both.

Thread safe/challenges in concurrent programming

- Consider using push_front() of the C++ linked list library in an environment with multiple threads.
  - What can happen with two threads run the push_front()?
  - Depending on the order of the instructions executed in the two threads, the results are different!
- Many library and system calls were developed for single thread execution.
  - They are not safe to be used in the multithreading environment by default
  - The ones that can be used in multithreading environment are marked as **thread safe**.

Thread 1

```
push_front():
    n = new node (…)
    n->next = front
    front = n
```

Thread 2

```
push_front():
    n = new node (…)
    n->next = front
    front = n
```

Python Threads

- By default, each program has at least one thread of execution, which usually called **the main thread**
  - In C++, it is the execution of the main function.
  - In Python, it is the execution of the python statements in the .py file.
- The first support needed for writing parallel programs is a way to start multiple threads of execution in a program.
  - This is a necessary condition for parallel execution, not sufficient.
- The operating system/runtime system provides support for the execution of multiple threads through an API. For a particular language like Python, we need to know at the language level how to use such functionality.
  - Some language has parallel constructs such as parallel loop
  - Others use libraries/modules - **Python thread using a built-in threading module to manage threads**

Python Threads

- To create a Python thread:
  1. Create an instance of the **threading.Thread** class.
  2. Specify the name of the function via the "**target**" argument.
  3. Call the **start()** function.
  4. We can explicitly wait for the new thread to finish executing by calling the **join()**
- Run lect6/RunFunctionThread.py, you can see the main thread and the task thread are progressing concurrently.

```
# Example adapted from Karen Works
from time import sleep
from threading import Thread

# function to run as a thread
def task():
    print('Message') # print a message

# create a thread instance
thread = Thread(target=task)
thread.start()   #
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```

Python Threads

- To run a function in a thread with parameters:

1. Create an instance of the **threading.Thread** class
2. Specify the name of the function via the "**target**" argument
    1. The function has parameters
3. Specify the arguments in order that the function expects them via the "**args**"
4. Call the **start()** function.
5. We can explicitly wait for the new thread to finish executing by calling the **join()**

- See lect6/RunFunctionThreadWArgs.py

```
……
def task(arg1, arg2):
    # display a message
    print("arg1=",arg1)
    print("arg2=",arg2)

# create a thread
thread = Thread(target=task, args=("One",2))
```

Daemon Threads

- You can also start a thread as a daemon thread by setting the daemon to true in the thread object.
- Whether a child thread is a daemon affects the behavior of the main thread:
  - The main thread can exit without waiting for a daemon thread to finish. The exit of the main thread will also kill the daemon thread.
    - ❖ If a child thread is not a daemon, the main thread cannot exit before the child thread is finished.
- See lect6/daemon.py and lect6/nondaemon.py

```
……
def task(arg1, arg2):
    # display a message
    print("arg1=",arg1)
    print("arg2=",arg2)

# create a thread
thread = Thread(target=task, args=("One",2),
                daemon=true)
```

Thread Attributes

- Each See lect6/ThreadInstanceAttributes.py for exathread has a unique name as it attributes.
- The thread name is an attribute.
- The current thread is returned by the current_thread function

```
def worker(val):
    global num
    num+=val
    print ("No! This is Patrick!",val,
      threading.current_thread().name)
    print(num)
   return
```

Note about Python thread

- Python threads are executed concurrently, but not in parallel. Due to Python interpreter implementation, at one time, only one thread can run. If we partition computation job into multiple threads, you will not see speedups. See lect6/cpuwiththreads.py
- This highlights the distinction between concurrency and parallelism
  - **Concurrency**: Multiple threads progress concurrently
  - **Parallel**ism: Multiple threads progress in parallel
- What is the use of Python threads?

Python Threading

- The software applications can be classified into CPU-bound applications (application performance bounded by the processing speed) and IO-bound applications (application performance bounded by the IO speed).
- Python threading can benefit IO-bound applications.
- See lect6/cpuwiththreads.py and lect6/iowiththreads.py

Thread Synchronization

- When multiple threads assess the same variable with at least one write (instructions in different threads have dependence), the outcome of the execution may be non-deterministic.
  - The timing of the progress of a thread is not controlled by the programmer
- See lect6/racecondition.py
- For such cases, a synchronization mechanism must be employed to ensure the correctness of the program.
  - Lock, event, condition, semaphore, barrier, etc
  - Python synchronization mechanisms can be found in https://docs.python.org/3/library/asyncio-sync.html
  - Different design patterns use different types of synchronization mechanisms.

Critical Section

- Critical section is a section of code with shared variables and/or resources. Only one thread can enter the section at a given time. If more than one thread executes instructions in a critical section, the data may not be consistent or the outcome becomes non-deterministic.
  - An example is shown in the code to the right
  - This is a design pattern in parallel and concurrent programming. Other examples include:
    - To make most of data structures work with threads, the insert and remove functions are critical sections.
      - Insert and remove functions in tree, linked list, etc

```
import threading
count = 0

def produce():
    global count
    for x in range(10):
        x = count
        time.sleep(1)
        count = x + 1
```

Implementing critical section: lock

- Acquire a lock before entering critical section
- Release the lock after exiting critical section
- The lock ensures only one thread will be in the critical section

```
import threading
count = 0

def produce():
    global count
    for x in range(10):
        Lock()
        x = count
        time.sleep(1)
        count = x + 1
        Unlock()
```

Lock in Python

- See lect6/raceconditionfixed.py

```
import threading
count = 0

lock = threading.lock()
def produce():
    global count
    for x in range(10):
        lock.acquire()
        x = count
        time.sleep(1)
        count = x + 1
        lock.release()
```

```
import threading
count = 0

lock = threading.lock()
def produce():
    global count
    for x in range(10):
        with lock:
            x = count
            time.sleep(1)
            count = x + 1
```