Multithreading

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Review

Remind PCB

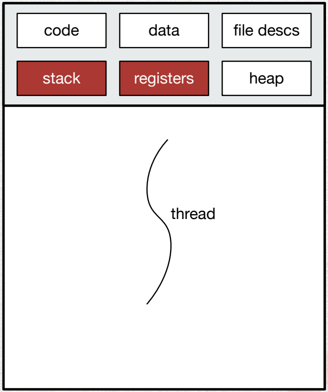
- Process Control Block

Remind PCB

- Process Control Block
- Contains
  - Process ID
  - Process state (new/ready/running/waiting/terminated)
  - **Processor state (program counter, registers)**
  - File descriptors
  - Scheduling information (next section)
  - Accounting information (limits)

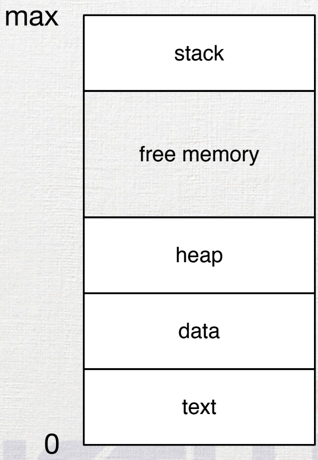
Thread & Single-threaded process

- Thread
  - a single flow of execution
  - belongs to a process
  - can be considered as lightweight process
- Single-threaded process
  - Default
  - Only one thread per process



Single-threaded process

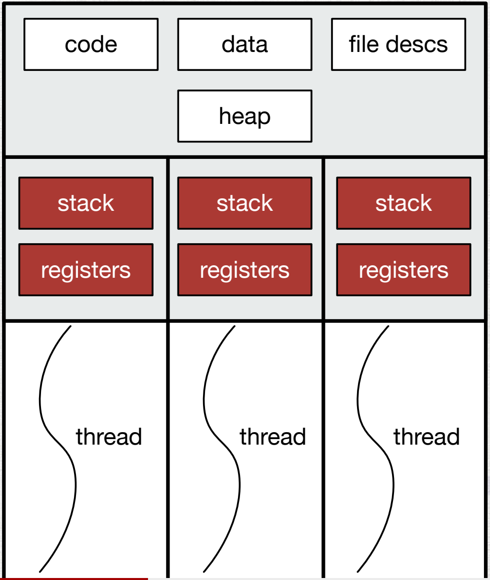
- Single stack
- Single text section (code)
- Single data section (global data)
- Single heap (dynamic allocation)



Multi-threaded process

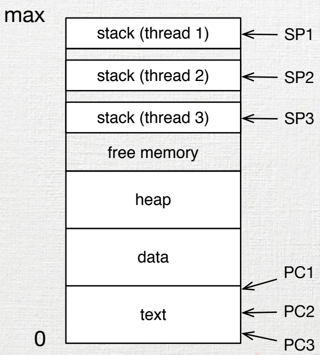
- More than one thread per process
- Share the same PCB among threads
    - Process state
    - Memory allocation (heap, global data)
    - File descriptors (files, sockets, etc.)
    - Scheduling information
    - Accounting information
- **Different** processor state (program counter, registers)
- **Different** stack

Multi-threaded process



Multi-threaded process

- Each thread has:
  - Private stack
  - Private stack pointer
  - Private program counter
  - Private register values
  - Private scheduling policies
- Share:
  - Common text section (code)
  - Common data section (global data)
  - Common heap (dynamic allocation)
  - File descriptors (opened files)
  - Signals...



Process memory space

Multi-threaded process vs Multi process

- Same goals

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization
  - Increase responsiveness

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What is the principal difference between these two types of process?

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What is the principal difference between these two types of process?
  - Multi-process with `fork()`: «resource cloning»

Multi-threaded process vs Multi process

- Same goals
  - Do several things at the same time
  - Increase CPU utilization
  - Increase responsiveness
- What is the principal difference between these two types of process?
  - Multi-process with `fork()`: «resource cloning»
  - Multi-thread process: «resource sharing»

Why?

- Responsiveness
- Performance
- Resource Sharing
- Scalability

Responsiveness

- Perform different tasks **at the same time**

Responsiveness

- Perform different tasks **at the same time**
  - Several operations can block (e.g. network, disk I/O)

Responsiveness

- Perform different tasks **at the same time**
  - Several operations can block (e.g. network, disk I/O)
  - UI needs responsiveness

Responsiveness

- Perform different tasks **at the same time**
  - Several operations can block (e.g. network, disk I/O)
  - UI needs responsiveness

→ one thread for UI, other threads for background tasks

Performance

- Creating (`fork()`) a new process is slower than a thread
- Terminating a process is also slower than a thread
- Switching between processes is slower than between threads

Resource Sharing

- Memory is always shared
  - Heap
  - Global data
- All file descriptors are also shared
  - Open files
  - TCP sockets
  - UNIX sockets
  - Devices
- No need to use `shm*()`

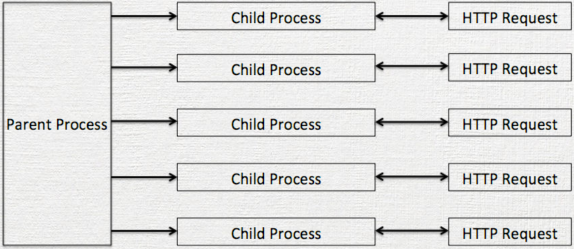
Scalability

- More CPU cores: simply increase number of threads
- Don't create too many threads
  - Overhead
  - Synchronization

Why **NOT** multi-thread?

- Threads are evil
  - Nondeterministic
  - Synchronization
  - Deadlocks
- Complication

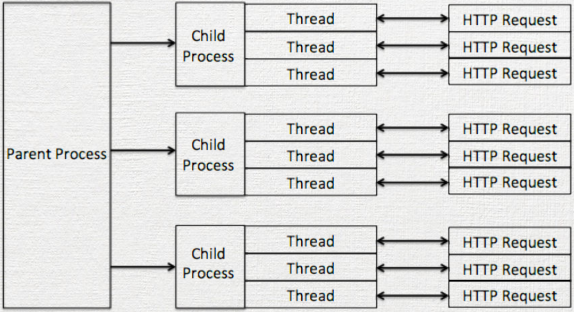
Multi-process real world app



Apache HTTPD Prefork Model[1]

[1] Image courtesy of Toni Miu's blog

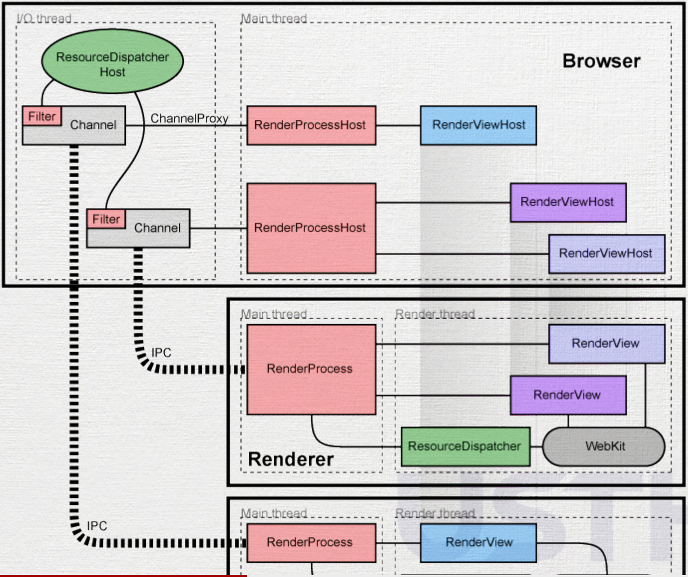
Multi-thread, multi-process, real world app



Apache HTTPD Worker Model[2]

[2] Image courtesy of Toni Miu's blog

Multi-thread, multi-process, real world app



Multithreading

Python threading

- Global Interpreter Lock
  - Implemented in CPython
  - Mutex
  - Only 1 thread can control the Python intepreter
  - Only one thread can be executed at any given time
    - Bottleneck in Python CPU-bound code
    - Not a problem in wrapper-to-native-code[3]
    - Not a problem in IO-bound programs

[3] e.g. numpy uses native libraries, so no GIL problem

Python threading

- Why GIL?
  - Memory management
    - Reference counting
    - Garbage collector
  - Simplification of thread-safety
    - Only 1 mutex on the intepreter
    - No multiple mutexes on each object
    - No deadlock
  - That's not a bug

Python threading

- Why GIL?
  - Memory management
    - Reference counting
    - Garbage collector
  - Simplification of thread-safety
    - Only 1 mutex on the intepreter
    - No multiple mutexes on each object
    - No deadlock
  - That's not a bug but a feature

Python threading

- Removing GIL?
  - Slower single-threaded performance
    - 1 mutex per object reference. . .
    - Potential deadlocks
  - Less compatbility

How?

- 2 «How» questions:

How?

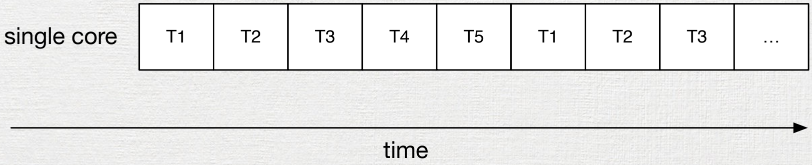
- 2 «How» questions:
  - Q1: How does thread achieve concurrency?

How?

- 2 «How» questions:
    - Q1: How does thread achieve concurrency?
    - Q2: How to use thread?

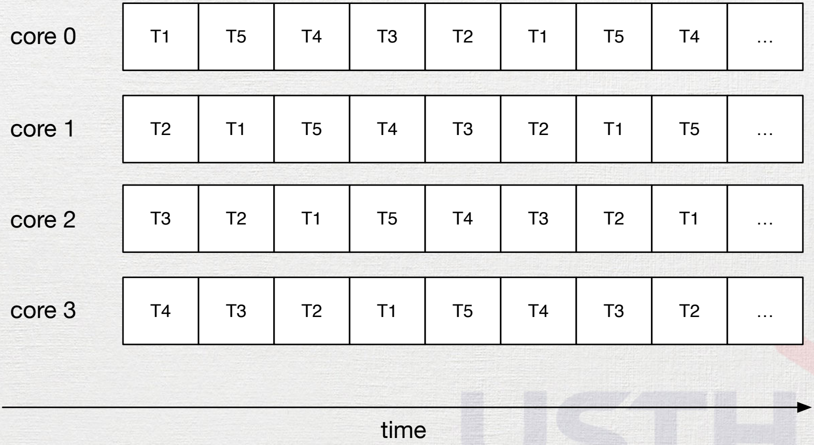
How (Q1): Concurrency on Single Core

- Q1: How does thread achieve concurrency?



How (Q1): Concurrency on Multi Cores

- Q1: How does thread achieve concurrency?



How (Q2): Using thread

- Use the module
- Subclass Thread
- Create new instance
- Launch the new thread
- [optional] Wait for thread to finish

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- The `threading` module

```
import threading
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Define a subclass of `threading.Thread`
  - Override `run()` method to run in background
  - [optional] Implement `__init__()` method for passing parameters

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

```
class BackgroundThread(threading.Thread):
    def __init__(self, sleepTime):
        threading.Thread.__init__(self)
        self.__sleepTime = sleepTime

    def run(self):
        time.sleep(self.__sleepTime)
        print(f"Finished sleeping {self.__sleepTime}s")
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Create new instance of the thread class

```
backgroundThread = BackgroundThread(10)
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- Launch the new thread with `.start()`
  - **NOT** `.run()`

```
backgroundThread.start()     # note no args here
```

How (Q2): Using thread

1. Use the module
2. Subclass Thread
3. Create new instance
4. Launch the new thread
5. Wait for thread to finish

- [optional] Wait for thread to finish with `.join()`

```
backgroundThread.join()
```

How (Q2): Using thread

```
import threading
import time

class BackgroundThread(threading.Thread):
    def __init__(self, sleepTime):
        threading.Thread.__init__(self)
        self.__sleepTime = sleepTime
    def run(self):
        time.sleep(self.__sleepTime)
        print(f"Finished sleeping {self.__sleepTime}s")

backgroundThread = BackgroundThread(10)
backgroundThread.start()    # note no args here
backgroundThread.join()
print("Finished main thread")
```

How: Extras

- Simple threading without subclassing:

```
def threadFunction(sleepTime):
    time.sleep(sleepTime)
    print(f"Finished sleeping {sleepTime}s")

t = threading.Thread(target=threadFunction, args=(10,))
t.start()
```

How: Extras

- Synchronization between threads `threading.Lock` (also called mutex)
  - `.acquire()`
  - `.release()`
  - Automatic `.acquire()` and `.release()` using `with` statement
- Be careful with race conditions while using `Lock`

How: Extras

```python
lock = threading.Lock()
lock.acquire()
# do something dangerous here
lock.release()

with lock:
    # do something dangerous here
    print("Dangerous function")

# lock is released automatically
```

Practice!

Practical work 8: multithreaded management system

- Copy your `pw6` directory into `pw8` directory
- Upgrade the persistence feature of your system to use `pickle` **in background thread**, still with compression
- Push your work to corresponding forked Github repository