CMPSCI 677 Operating Systems **Spring 2024**

Lecture 5: February 16

Professor: Prashant Shenoy
Scribes: Harshavardhan Rajanala(2024), Swathi Natarajan(2023), Madhuvanthi Ramesh Kumar (2022), Josh Sennett (2019)

5.1 Overview

This lecture covers the following topics:

Part 1: Threads

Part 2: Concurrency Models

Part 3: Thread Scheduling

5.2 Lecture Notes

5.2.1 Part 1: Threads and Concurrency

5.2.1.1 Review of Processes and Threads

A **process** is a program that is executing on a system. Each process has its own address space with its corresponding code, global and local variables, stack, and resources. A single process runs on a single processor or core at a time.

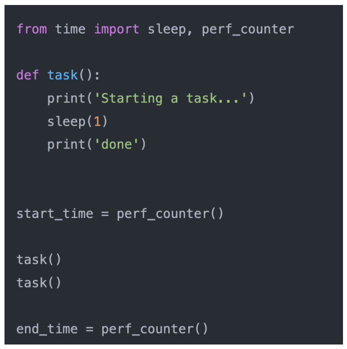
A **thread** is a light weight process that has multiple concurrent flows of control. Each flow of control executes a sequence of instructions. Multiple threads can be part of the same process and can be executed concurrently. All threads share the same address space, but have their own stack, own program counter, own copy of registers and control flow as they execute different parts of the process. Threads can run on different cores at the same time. Synchronization might be required if threads are accessing shared data structures and it can be achieved by locks, semaphores, etc...

Question: When there are multiple threads, in what order will they execute?
Ans: Order of execution depends on order of scheduling, which is done by OS, using various scheduling algorithms.
Concurrency enables handling of multiple requests. Multi-threading can be used to achieve concurrency if we are using a machine with a single core. The different threads are time multiplexed onto the processor, and different parts of the program can be executed by switching between threads.
Parallelism enable simultaneous processing of requests. Multi-threading can be used to achieve **parallelism** on a multi-core machine, as two threads can simultaneously execute in parallel on different cores.

Question: Does the process decide how much memory is allocated to each thread?
Ans: The OS decides how much memory to give each process. Thread run time decides how much stack space to allocate to each thread. The heap is shared across threads.

5.2.1.2 Threads example:

Single threaded program:



```python
from time import sleep, perf_counter

def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

task()
task()

end_time = perf_counter()
```

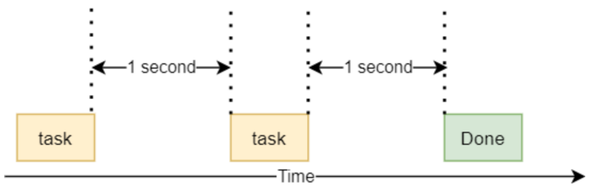
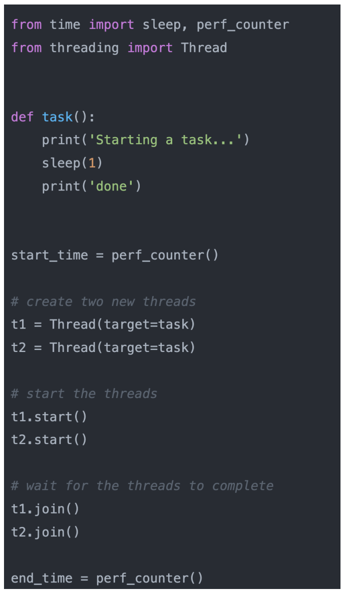


Figure 1. Single thread program and its execution timeline

The above python program exhibits basically a sequential execution and there is only one thread. The *task()* function is executed first and it sleeps for 1 second. After 'done' is printed by the *print()* statement, another *task()* function is executed with subsequent sleep.

Multi threaded program:



```python
from time import sleep, perf_counter
from threading import Thread


def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)

# start the threads
t1.start()
t2.start()

# wait for the threads to complete
t1.join()
t2.join()

end_time = perf_counter()
```

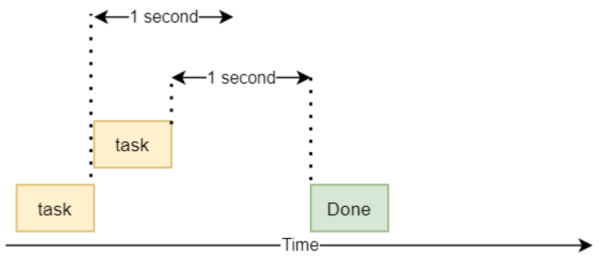


Figure 2. Multi thread program and its execution timeline

The above python code shows multi-threaded program where two threads *t1* and *t2* are created by the main thread. Then start the threads by using *start()* function. Depending on the CPU scheduling, when *t1* is in idle state, *t2's task()* function will be executed concurrently. *join()* function enables main() function to wait for the threads to complete.

Question: Does running a code count as its own thread?
Ans: When a program is executed, a main thread is created by interpreter to execute main() function. As per the program in Figure.2, there are two other threads are created by the main thread. So totally there are three threads.

Question: What is the use of join() function at the end in the program (Figure.2)?
Ans: the join() method is used to wait for a thread to complete its execution before moving on to the next operation. the join() method is used to synchronize the execution of multiple threads in a Python program, ensuring that all threads complete their tasks before the program terminates.

Question: What happens if the stack of the thread is full?
Ans: Usually, the stack space is huge. But there are chances for it to be full when there is an infinite for loop or infinite recursion. The thread then throws memory full error and it stops executing.

Question: If the thread terminates due to an error, will the whole process halts too?
Ans: It depends on the type of the error. For example: If there is an illegal memory access, then the whole process will be terminated, whereas for any other normal errors, thread simply exits and the process continues to run.

Question: Is the join() method a blocking call? Will any code after join() be executed after the thread completes execution?
Ans: Yes, it is a blocking call that is waiting for the thread to complete. The main() function will continue and the statements after join() will execute once the thread execution is complete.

Question: Is the above example(Figure 2), is the code running in a single core?
Ans: Yes, it is assumed to run in a single core so parallelism is not observed whereas only concurrency is observed. When there are multiple cores, then two threads can run the program at the same time observing parallelism.

5.2.1.3 Why use threads?

Multi-threading allows for concurrency within a single process. On a a machine with a single processor, threads achieve concurrency through time-sharing of the CPU. While one thread is doing I/O, another thread can execute on the CPU, thereby improving performance. On a multiprocessor machine, we can achieve true parallelism by running threads simultaneously on different cores.

Switching between threads of the same process or different processes is more efficient than switching between processes. Thread operations such as thread creation, deletion, switching are much more efficient and lightweight than the corresponding process operations.

Threads have access to the entire address space of the process, which gives programmers greater flexibility, allowing for shared data rather than message passing.

If a single threaded process makes a blocking call, the entire process is blocked till the blocking operation is completed. If you have multiple threads within that process, each thread is a synchronous sequential stream of execution. If one thread makes a blocking call, other threads of the process can continue executing.

To make a single-threaded process concurrent, we need to make system calls non-blocking. In between single and multi-threaded programming, there is **finite-state machine** (event-based) programming. Event-based programming attempts to achieve concurrency with a single threaded process, using non-blocking calls and asynchronous communication (which is more complex to program).

Question: What is the difference between each thread sharing address space and yet having its own stack?
Ans: Threads have their own stack and also it can access any address space in the process since it is shared.

5.2.1.4 Use cases - Client and server

An example for a multi-threaded client would be a **Web Browser**. Each task that a browser performs could be assigned to a different thread. If the browser was single threaded, upon clicking on a web link, images would have to be sequentially downloaded from the server and then sent to be parsed and rendered. In a multi-threaded browser, images can be downloaded in parallel while the page is parsed and parts of the page can be rendered as they are ready. The browser does not have to wait for everything to be downloaded and parsed before rendering. It can connect to different servers and exchange data using different threads. As a result, the user will see parts of the page more quickly.

Multi-threading is also used within **servers**. If a server only runs a single thread, it might have a queue of requests from clients which are blocked until the first request is completed. Using a pool of threads allows the server to respond to multiple requests simultaneously, thereby reducing latency. The idea is for the server to have a dispatcher and a few worker threads (dispatcher-worker architecture). When a client request comes in, the dispatcher assigns this request to one of the idle worker threads. This model is efficient because, while some of the worker threads are I/O bound, others can continue doing computation.

Question: If there's a long sequential piece of code, is there any benefit to dividing it into smaller pieces?
Ans: If the smaller pieces are independent tasks, you can put them in threads and they will execute concurrently (single processor) or even in parallel (multiprocessor). If the tasks are dependent, you can't parallelize it as there is a dependency.

Question: What is concurrency?
Ans: If there are multiple threads within a process, and a single core, one thread will execute for a small time slice, and then switch to another thread for the next time slice, and so on. This is known as concurrent execution.

Question: If there is a program that has no blocking calls, running on a single core machine, will having multiple threads reduce the time needed for execution?
Ans: Multi-threaded program does not change the time required to execute instructions. It only interleaves the instructions.

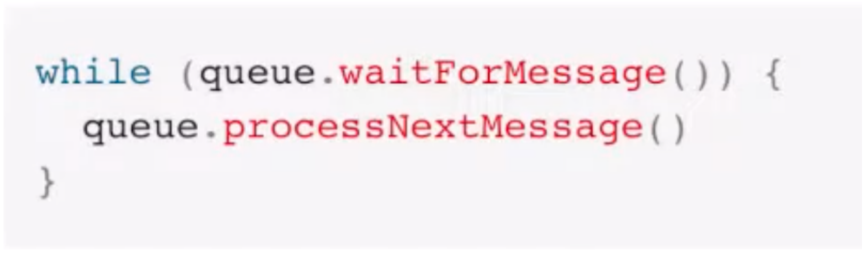
Question: Is there a limit on no. of threads in a thread pool? **Ans**: There is no real limit on the no of threads, the only real limit is the memory.

5.2.2 Concurrency Models

The type of threading system we use in the server becomes our Concurrency Model. All concurrency models use the same abstract model - Server is an infinite loop waiting for requests. All server applications involve a loop, called the event loop, to process incoming requests. Inside this loop, the server waits for an incoming request from the client over a network and then processes the request.

5.2.2.1 Sequential Server

It is a single threaded server that processes incoming requests sequentially. The server will wait for the request and process it one by one.

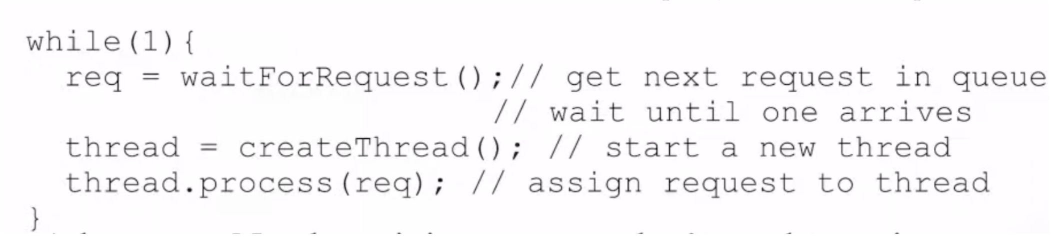


```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

Advantage:
It is very simple to implement.
Disadvantage:
If a burst of requests arrive, they queue up at the server. As the server processes requests sequentially, requests will queue up while one request is being processed. This increases the waiting time and response time.

5.2.2.2 Multi-threaded Server

- **Thread per request**
  In this model, every request is assigned to a new thread. The thread processes the request, and is then terminated. Lifetime of a thread is the lifetime of the request, which is short-span.

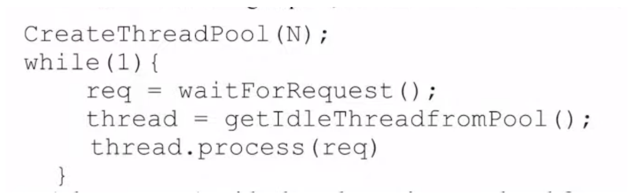
  ```
  while(1){
    req = waitForRequest();// get next request in queue
                           // wait until one arrives
    thread = createThread(); // start a new thread
    thread.process(req); // assign request to thread
  }
  ```

  Advantage:
  Newly arriving requests don't need to wait for older requests to be processed.
  Disadvantage:
  Frequent creation and deletion of threads causes overhead.

  Question: Is the time to finish a request longer because you need to share time with other requests?
  Ans: The response time depends on what kind of processing the request involves. If the processing involves I/O, you can process another request if you have multiple threads. If the request involves CPU processing, by interleaving multiple requests, the request processing time will increase due to switching. However, it is still better overall as wait times could be very high in a sequential model.

  Question: How many threads can an application have? Is it limited by number of cores? **Ans**: There is no restriction on number of threads. However, there is an overhead as each thread involves maintaining some state, which requires memory.

- **Thread pool**
  In this model, a fixed number (N) of threads are created when the server is started. This is known as a thread pool. One thread acts as the dispatcher and the other threads are the workers. The dispatcher pulls a request from the queue and assigns it to an idle thread to process.



```
CreateThreadPool(N);
while(1){
    req = waitForRequest();
    thread = getIdleThreadfromPool();
     thread.process(req)
  }
```

Advantage:
We do not need to create and delete threads for each request.
Disadvantage:
It is difficult to choose the right N. If more than N requests arrive concurrently, there will be blocking till a thread becomes idle. If we have a very large thread pool, resources will be wasted as many threads will be idle.

Question: Is there any interleaving between idle threads?
Ans: If a thread is waiting, it will not be scheduled for execution. Only active threads get time slices and are involved in the switching.

Question: How do we know the thread is idle?
Ans: The idle thread will be in the thread pool. The mechanism will be done while creating the program.

Question: If there are N threads, will main thread included in the N number of threads?
Ans: No, main thread will spawn N threads, so the program will have N+1 number of threads.

Question: Is N configurable? **Ans**: N is only cofigurable at the server startup time. Once the server starts, we can't change N. We need to terminate the server, change N and start the server again.

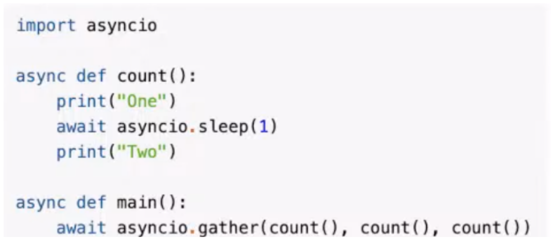
- **Dynamic Thread pool**
  As we saw in the previous model, it is often difficult to choose the optimal pool size as it depends on the rate of incoming requests. In this model, we have a dynamic pool where we start with N threads and monitor number of idle threads.

  – If the number of idle threads fall below low threshold, we increase N and start new threads and add it to the pool.

  – If the number of idle threads is greater than high threshold, we reduce the size of the thread pool and terminate some idle threads.

  We can configure the size of the pool, and even set a max and min limit. This model is pro-active as it creates/deletes threads before it reaches the min/max.
  Advantage:
  We need not worry about choosing an optimal pool size.
  Disadvantage:
  Server is more complicated, requires monitoring and adjustment of pool size dynamically.

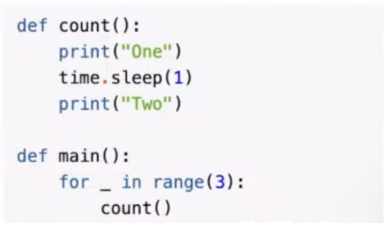
- **Asynchronous Event Loop**
  It is a single threaded server, but uses non-blocking operations. It provides concurrency similar to synchronous multi-threading, but with a single thread. If a blocking operation needs to be executed, we switch to some other operation and let the blocking operation continue working in the background.



```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())
```



```
def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()
```

Figure 3. Asynchronous and synchronous version

Switching between parts is driven by code, rather than the OS.

Note: If you make a function async, you need to make all blocking calls inside the function also async. Refer `https://python.plainenglish.io/build-your-own-event-loop-from-scratch-in-python-da77ef1e3c39` for more details.

Question: How does the program know when a blocking call has finished executing?
Ans: Standard I/O operations such as read are blocking. Asynchronous read sends the request to the OS and the control returns back to the process. When the read finishes, the OS sends a signal to the process as an event, and the process goes back to what issued the asynchronous call and continues execution from there.

Question: What happens after the sleep of first count() in Figure 3?
Ans: It has to wait for whatever it is being executed to be completed and yield control.

Question: Who is monitoring whether the function is over or not?
Ans: The asyncio library is responsible for monitoring the status of functions (i.e., asynchronous functions) and determining whether they have completed or not. It is managed by the programmer.

Question: Where is the speedup compared to dynamic thread pool?
Ans: There is only one thread involved in this model but it behaves as a multi threaded program. This means that there is no OS scheduler involved in switching threads which may lead to increase the speed. It is not to be used as a way to speed up but as a way to achieve concurrency in a single threaded program.

Process Pool Servers These are multi process servers that use a separate process to handle each request. This can be done by creating a (dynamic) process pool. As there are multiple processes involved, this model requires inter-process communication. For example, Apache web server uses process pool model which also supports multi threading.
Advantage:
If a process crashes, other processes continue to run. Crashes only impact that request, not the entire application. In a multi-threaded server, all threads are part of the same process. If one thread crashes, the entire server crashes.This model enables address space isolation. Sensitive operations can be carried out as one process doesn't have access to another process' memory.
Disadvantage:
More expensive, process switching is more heavyweight than thread switching.

5.2.2.3 Summary

Based on architecture, servers can be broadly classified into four categories:

- Pure sequential
- Event based

- Thread based
- Process based

Which architecture is more efficient?
The answer depends on what kind of machine it is being used on.
For single core machines- event-based model is more efficient than thread-based as there are no context switching involved.
For multi core machines - thread-based server yield high performance because they can achieve true parallelism.
For best performance, we need a multi-threaded event based architecture.

5.2.3 Parallelism vs Concurrency

- **Concurrency** enables us to handle multiple requests.
  - Request processing does not block other requests.
  - This can be achieved using threads or async (non-blocking) calls.
  - Concurrency can be achieved on a single core/processor.
- **Parallelism** enables simultaneous processing of requests.
  - Request processing does not block other requests. Requests are processed in parallel.
  - This can be achieved using multiples threads or processes as threads or processes can run on multiple cores/processors
  - Need the right hardware support. Multiple cores/processors are needed.

5.2.4 Part 3: Thread Scheduling

5.2.4.1 User-Level Threads

User-level threads are created and managed by user libraries. The kernel is unaware of there being multiple threads. The kernel sees the address space of the threads as a traditional single-threaded process. Two-level scheduling happens. The OS picks the process, the library scheduler picks the thread to execute.

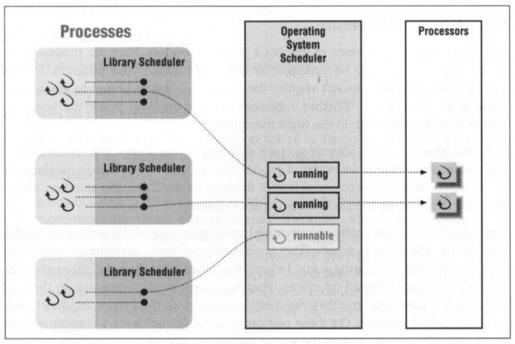


Figure 6–1: User-space thread implementations

Advantage:
Creation of threads is very lightweight because it doesn't require system calls. It offers flexibility to the

programmers as they can fix the scheduling algorithm.
Disadvantage:
Threads compete with each other. They cannot take advantage of parallelism. As the OS sees only a single thread, even if there are multiple threads, they cannot run on multiple cores.

Question: How do you write a user level thread library?
Ans: The library has to use event based programming. It needs to see threads as being blocks of code, and switch between them. Need to provide the same APIs.

5.2.4.2 Kernel-Level Threads

Kernel-level threads are created and managed by the OS. Kernel can see the presence of threads and can schedule them.

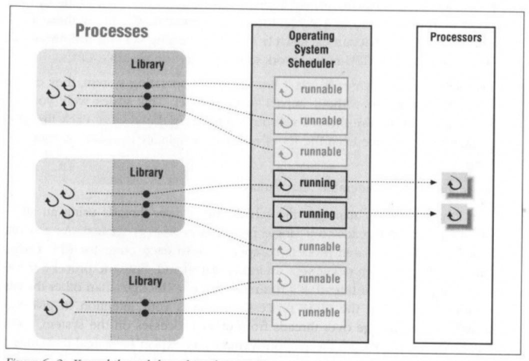


Figure 6–2: Kernel thread–based implementations

Advantage:
They allow for real parallelism between threads as the OS is aware of multiple threads and can run different threads on different cores.
Disadvantage:
They are more expensive as thread management is handled by the OS. Creation and deletion of threads is more expensive as it results in a system call to OS.

Question: Is there a particular algorithm that OS uses to pick threads ?
Ans: OS scheduler algorithms are used to pick threads. It is much similar to process scheduler.

5.2.4.3 Scheduler Activation

- User-level threads use two-level scheduling: OS scheduler and Library scheduler. The kernel might switch user-level thread during an important task. So information passing between kernel and library is needed.
- **Scheduler Activation** is an OS mechanism for user-level threads.
  - Kernel notifies user-level threads about kernel events with calls like I/O is done, CPU available etc...
  - Library informs the kernel to create/delete threads. N:M mapping - N user-level threads mapped onto M kernel entities.

5.2.4.4 Lightweight Processes

- Lightweight processes sit between threads and processes. Instead of creating threads per process, several LWPs are created per heavyweight process amd threads are mapped onto these LWPs.
- Each LWP when scheduled searches for a runnable thread (two-level scheduling).
- When a LWP thread is blocked on a system call, OS context switches to another LWP.