Threading in Python

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

1. Understand the concept of threading in Python
2. Demonstrate their knowledge on the use of threading in Python

Threading

Threading (1)

- A *thread* refers to a basic unit of CPU utilisation
  - a separate process that has its own instructions and data
  - it may also represent a process that is part of a parallel program
    - although it may also represent an independent program
- They share their code, data and other operating system resources with other threads belonging to the same process
- A traditional process will have a single thread of control
  - if a process has multiple threads of control, then it has the ability to perform more than one task at a time

Threading (2)

Benefits of Multithreading

1. **Responsiveness**
   - Interactive applications can continue to run, even if part of its blocked; increasing the responsiveness to the user
   - Multithreaded web browsers enable you to continue browsing the internet on one tab, whilst another tab has become unresponsive
2. **Resource Sharing**
   - Threads share memory and resources of the process they belong to
   - Benefits of sharing code and data
   - allows an application to have several threads of activity in the same address space
3. **Economy**
   - Allocation of memory and other resources for process creation is costly
   - Threads share resources of a process they belong to
     - therefore, providing a cost-effective resource
4. **Multiprocessor Architecture**
   - Threads may run in parallel on different processors, dependent upon the multiprocessor architecture
   - Single threaded processes may only run on one CPU, no matter how many CPUs may be available
   - Multithreaded processes on a multi-CPU machine can increase concurrency

Threading (3)

Difference Between Process and Thread i

- In multithreading, a process and thread are two closely related terms
  - they have the same goal to make a computer run tasks simultaneously
- A *process* can contain one or more threads, whilst a *thread* cannot contain a process

Threading (4)

Difference Between Process and Thread ii

Process

- An execution of a script/program to perform a task
- The operating system will assist in the creation, scheduling and termination of the processes
- Spawned processes from the main process are known as **child** processes
- The properties of a process are:
  - creating each process requires separate system calls for each process
  - an isolated execution entity and does not share data or information
  - requires more system calls to manage

Threading (5)

Difference Between Process and Thread iii

Thread

- An execution of a segment that is part of the process
  - a process can consist of **multiple** threads
  - all threads will be executed at the same time
- Considered to be lightweight and managed by a scheduler
- The properties of a thread are:
  - a single system call can create multiple threads
  - threads can share data and information between themselves
  - management of threads consumes fewer (or none) system calls

Threading (6)

Advantages and Disadvantages of Threading i

Advantages

- **Speed**
  - Multithreading can improve the speed of computation
  - Each core (or processor) can handle separate threads concurrently.
- **Responsiveness**
  - Applications can remain responsive as one thread waits for the input
  - Another thread can run the GUI at the same time
- **Variable Accessibility**
  - All threads of a particular process can access global variables
  - If a change is made to a global variable, then it is visible in the other threads too
- **Resource Utilisation**
  - Running several threads in each application utilises the resources of a CPU better
  - Idle time of a CPU decreases
- **Data Sharing**
  - No requirement for extra space to be created for each thread
  - Threads within an application can share the same data

Threading (7)

Advantages and Disadvantages of Threading ii

Disadvantages

- **Suitability**
  - Multithreading is not suitable for single processor systems
  - Difficult to achieve performance gains compared to a multiprocessor system
- **Security**
  - As threads can share the same data, there is an issue with security
  - Any unknown thread may make changes to the data
- **Complex**
  - Multithreading can increase the complexity of the application and debugging
- **Possible Deadlock**
  - There is a possibility of leading to a deadlock state
  - *Deadlock* is a situation where a set of processes are blocked
    - this is due to each process is holding a resource and is awaiting another acquired by a different process
- **Synchronisation**
  - Avoiding mutual exclusion is achieved by synchronisation
  - Leads to more memory and CPU utilisation

Multithreading Models

Multithreading Models (1)

- There are two types of threads:
  1. User Level
  2. Kernel Level

Multithreading Models (2)

User Level Threads

- These are threads managed by the **user**
- The thread management kernel is not aware of the existence of these threads
- The library used for managing threads can be used to:
  - create and delete threads
  - pass messages and data between threads
  - schedule thread execution
  - save and restore thread contexts
- **Advantages**:
  - Switching threads does not require kernel mode privileges
  - User level threads can run on any operating system
  - These types of threads are fast to create and manage
- **Disadvantages**:
  - For a typical operating system, most system calls are blocked
  - A multithreaded application cannot take advantage of multiprocessing

Multithreading Models (3)

Kernel Level Threads

- These are threads managed by the operating system
  - there is no thread management code in the application
- Any application can be programmed to be multithreaded
- All the threads in the application are supported within a single process
- The kernel maintains context information for the process as a whole
  - along with the individual threads within the process
- Scheduling by the kernel is done on a thread basis
  - performs the thread creation, scheduling and management in the kernel space
- **Advantages**:
  - Simultaneous scheduling of multiple threads from the same process on multiple processes
  - If a single thread is blocked, the kernel is able to schedule another thread for the same process
- **Disadvantages**:
  - Generally slower to create and manage compared to user-level threads
  - Transfer of control from one thread to another within the same process requires a mode switch

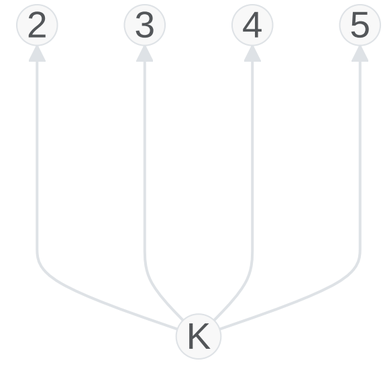
Multithreading Models (4)

- There are three methods of modeling a multithreaded application:
  1. Many-to-One
  2. Many-to-Many
  3. One-to-One

Multithreading Models (5)

Many-to-One

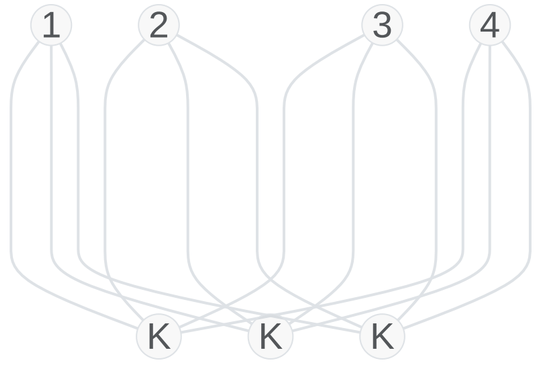
- Maps many user-level threads to one kernel thread
- Management of the thread is done by the thread library in the user-space
- Only a single thread can access the kernel at a time
  - therefore, multiple threads are unable to run in parallel on multiprocessors



Multithreading Models (6)

Many-to-Many

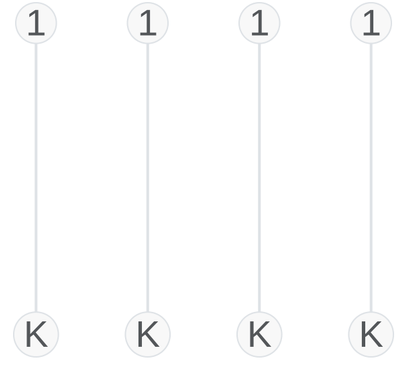
- Joins many user-level threads to a smaller or equal number of kernel threads
- The number of kernel threads may be specific to a particular application or machine
- Developers are able to create as many user threads as necessary
  - the corresponding kernel threads can run in parallel on multiprocessor machines



Multithreading Models (7)

One-to-One

- Maps each user-level thread to a kernel thread
- Provides more concurrency than the many-to-one model
  - allows multiple threads to run in parallel on multiprocessors
- Creating a user thread requires creating the corresponding kernel thread
- The overhead required for creating a kernel thread can be a burden on the performance of the application



Multithreading in Python

Multithreading in Python (1)

- Python has two libraries available for multithreading applications:
  1. `_thread`: each thread is a function
  2. `threading`: each thread is an object
- For the purpose of this module, we shall be focusing upon the `threading` module
  - `_thread` is considered to be deprecated

Multithreading in Python (2)

Creating a Threaded Application

- Implement threads in an object-oriented methodology, as such providing high-level support
- To implement a thread, the `Thread` class is used
  - e.g. `from theading import Thread`
- We can then create an instance of the `Thread` class
- Specify the function to run in the `target` argument
- Execute the thread using the `start` function

```
from threading import Thread
def hello():
    print("Hello 5062CEM!")
threadExample1 = Thread(target=hello)
```

```
threadExample1.start() -> Hello 5062CEM!
```

Multithreading in Python (3)

Running a Function with an Argument

- Create an instance of the `Thread` class
- Specify the function to run in the `target` argument
- Specify the arguments to pass th`rough in theargs` argument
  - provide the arguments as **list**
- Execute the thread using the `start` function

```
from threading import Thread
def hello(name):
    print(f"Hello {name}, and welcome to 5062CEM!")
threadExample1  = Thread(target=hello, args=(['Ian Cornelius']))
```

```
threadExample1.start() -> Hello Ian Cornelius, and welcome to 5062CEM!
```

Multithreading in Python (4)

Running a Function with Multiple Arguments

- Create an instance of the `Thread` class
- Specify the function to run in the `target` argument
- Specify the arguments to pass through in the `args` argument
  - provide the arguments as **list**
- Execute the thread using the `start` function

```
from threading import Thread
def hello(name, module):
    print(f"Hello {name}, and welcome to {module}!")
threadExample1 = Thread(target=hello, args=['Ian Cornelius', '5062CEM'])
```

```
threadExample1.start() -> Hello Ian Cornelius, and welcome to 5062CEM!
```

Multithreading in Python (5)

Creating a Custom Thread Class

- Extend an instance of the `Thread` class
- Override the `run` function
- Provide variable names with the `self` keyword
- Return the string to the `self.data` variable

```
from threading import Thread
class MyThread(Thread):
    def __init__(self, name, module):
        Thread.__init__(self)
        self.name = name
        self.module = module
        self.data = None
    def run(self):
        self.data = f"Hello {self.name}, and welcome to {self.modu
customThreadExample1 = MyThread("Ian Cornelius ", "5062CEM")
customThreadExample2 = MyThread("Terry Richards", "5069CEM")
```

```
customThreadExample1.data -> Hello Ian Cornelius , and welcome to 5062CEM!
customThreadExample2.data -> Hello Terry Richards, and welcome to 5069CEM!
```

Multithreading in Python (6)

Extending the Custom Thread Class

- Extend an instance of the `Thread` class
- Override the `run` function
- Provide variable names with the `self` keyword
  - add a new variable called `sleep`
- Return the string to the `self.data` variable

```
from threading import Thread
class MyThread(Thread):
    def __init__(self, name, module, sleep):
        Thread.__init__(self)
        self.name = name
        self.module = module
        self.sleep = sleep
        self.data = None
    def run(self):
        from time import sleep
        sleep(self.sleep)
        self.data = f"Hello {self.name}, and welcome to {self.modu
customThreadExample1 = MyThread("Ian Cornelius ", "5062CEM", 10)
```

```
customThreadExample1.data -> Hello Ian Cornelius , and welcome to 5062CEM!
customThreadExample2.data -> Hello Terry Richards, and welcome to 5069CEM!
```

Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk