Program vs Process vs Thread

This lesson discusses the differences between a program, a process, and a thread. Also included is a pseudocode-example of a thread-unsafe program.

Program vs Process vs Thread

Program

A program is a set of instructions and associated data that resides on the disk and is loaded by the operating system to perform a task. An executable file or a Ruby script file are examples of programs. In order to run a program, the operating system's kernel is first asked to create a new process, which is an environment in which a program is executed.

Process

A process is a program in execution. A process is an execution environment that consists of instructions, user-data, and system-data segments, as well as lots of other resources such as CPU, memory, address-space, disk and network I/O acquired at runtime. A program can have several copies of it running at the same time, but a process necessarily belongs to only one program.

Thread

A thread is the smallest unit of execution in a process that simply executes instructions serially. A process can have multiple threads running as part of it. Usually, there would be some state associated with the process that is shared among all the threads and in turn, each thread would have some state private to itself. The globally shared state amongst the threads of a process is visible and accessible to all the threads, and special attention needs to be paid when any thread tries to read or write to this global shared state. There are several constructs offered by various programming languages to guard and discipline the access to this global

state, which we will go into further detail in upcoming lessons.

Notes

Note that a "program" and a "process" are often used interchangeably, though most of the time the intent is to refer to a process.

There's also the concept of "multiprocessing" systems, where multiple processes get scheduled on more than one CPU. Usually, this requires hardware support, where a single system comes with multiple cores or the execution takes place in a cluster of machines. **Processes don't share any resources amongst themselves, whereas threads of a process can share the resources allocated to that particular process, including memory address space.** However, languages do provide facilities to enable inter-process communication.

Counter Program

Below is an example highlighting how multi-threading necessitates caution when accessing shared data amongst threads. Incorrect synchronization between threads can lead to wildly varying program outputs depending on the order in which threads get executed.

Consider the snippet of code below:

```
1. counter = 0;
2.
3. def incrementCounter():
4.     counter += 1
```

The increment on **line 4** is likely to be decompiled into the following steps on a computer:

- Read the value of the variable counter from the register where it is stored
- Add one to the value just read
- Store the newly computed value back to the register

The innocuous-looking statement on **line 4** is really a three-step process!

Now imagine if we have two threads trying to execute the same function `increment_counter()`. One of the ways the execution of the two threads can take place is as follows:

Let's call one thread as **T1** and the other as **T2**. Say the counter value is equal to 7.

1. **T1** is currently scheduled on the CPU and enters the function. It performs step A, i.e. reads the value of the variable from the register, which is 7.
2. The operating system decides to context switch **T1** and brings in **T2**.
3. **T2** gets scheduled and luckily gets to complete all the three steps **A**, **B** and **C**, before getting switched out for **T1**. It reads the value 7, adds one to it and stores 8 back.
4. **T1** comes back, and since the operating system saved its state, it still has the stale value of 7 that it read before being context-switched. It doesn't know that the value of the variable has been updated behind its back and, unfortunately, thinking the value is still 7, adds one to it and overwrites the existing 8 with its own computed 8. If the threads executed serially the final value would have been 9.

These problems should be apparent to the astute reader. Without properly guarding access to mutable variables or data-structures, threads can cause hard-to-find to bugs.

Since the execution of the threads can't be predicted and is entirely up to the operating system, we can't make any assumptions about the order in which threads get scheduled and executed.