Thread and processes

2.1 Thread vs. process

Processes

- When a program on a computer is running an instance of an application that is referred to a process.
- Process consit of code, data and state information.
- Each process is independent and has its own address space.
- Computers can have hundreds of active processes running at once, and the OS of a computer will be handlding all of them.

Threads

- Within a process, there are sub elements referred to as a thread. A thread can only exist within a process.
- Each thread is an independent path of execution in a program.
- Threads are the basic units that the OS manages, and it allocates time on the processor to actually execute them.
- Each thread is typically used to complete indivudal tasks, and if we have more tasks, we can create more threads to complete the tasks, and then then destory the threads when we are done using them.
- Threads share memory space, data, and code.
- If threads do not corrdinate with one another it can cause problems, depending on the program.
- Sharing resources between processes is not easy, but between multiple threads are very easy to share. To communicate between, you must use a system provided IPC (Inter-Process Communcation).
- examples of IPCs includes:
    - sockets and pipes
    - shared memory
    - remote procdure calls.

Parallel Programs Using Threads/Processes

- Can use multiple processes working towards a common goal OR can have multiple threads under one process working on the common goal. Depending on the siutation, one option may be better than the other.
- implementation between how threads and processes work diff between OSs and programming languages.
- Foe example, If your application is going to be distributed across multiple computers, you most likely need separate processes for that.
- But, as a rule of thumb, if you can use multiple threads rather than processes, use threads insteead. Threads are considered light-weight compared to processes, which are more resource-intensive. A thread requires less overhead to create and terminate than a process, OS can tyopically switch between threads faster than it can switch between processes.

2.2: Concurrent vs. parallel execution

- A programm may be designed to have many threads, BUT that does not mean that the threads will be running in parrallel.

Concurrency

- Related to parrallel programming, but not the same at all.
- Concurrency - ability of an algorithm or program to be broken into different parts that can ran independently of each other. It can be executed out of order, or partially out of order, without affecting the end result.
- Example, making a salad can you run them in a random order does not matter.
- If we only have one processor Threads will typically exectute for a period of time, and then switch with another thread to complete its task until each thread is done its task. The threads will switch freqently, making it seem like the two tasks are running at the same time.
- True parrallel execution need parrallel processors so that two or more threads can run at the exact same time.
- Concurrenty is about program structure, so that we can deal with multiple things at once. Parrallel Programs have simultateuosly execution, acutally doing things at once.
- A concurrent program is not nessecaril;y parrallel, but given the hardware if will be.
- Parrallel Execution is not always the answers to all problems. For example, IO devices are used infreuently compared to the speed of a computer. Therefore, it is much more of a burden on the computer to run all of the IO devices in parrallel, rather than just having them spread accorss the
- Concurrent tasks are useful for GUIs. Lets says the user clicks on a somehting, then we can run the program in concurrent threads so that the user can still intreact with the GUI while the oprtation is taking place. The thread running the UI is still free to accept new operations.
- Parrellel exectuion is good for tasks that are computationally expensive, such as matrix multiplication. Executing each of those tasks in parrallel on seperate processors can speed things up a lot.

2.3: Global interpreter lock: Python demo

- Concurrent threads work in python, but the interpreter does not allow the threads to run in parrallel due to the global interpreter lock. Unique to python.
- Cpython is the most common python interpreter, and it has GIL. GIL has more advantages than disadvantages, hence it is still used.
- Other python interpreters exisits, which do not have the GIL.
- For IO dependent tasks, GIL does not affect running concurent threads. You can use python threads library for this.
- For CPU dependent tasks, than GIL will likely affect multiple threaded programs. There are ways to get around. Typically any external libraries are written in a compiled language like C++, and then are added into Python via libraries.
- If you want to use python only, then you can use pythons multiprocessing library. Communication between processes is much more complictaed, and it is much more system use heavy.

Execution Scheduling

- The OS will assign processes and threads different times to run on the CPU.
  Processes/ Threads are placed in the ready queue when they are ready to be ran, then the OS will cycle through the processor(s).
- Sometimes the OS will decide that a thread/process has spent its fair share of time using the processors, so it will be switched out by another thread. This is referred to as a Context Switch.
- During a context Switch the OS must store the last state of a thread/process so that it can resume

later. The new process that is now scheduled must load its current state information, and then run.
- The context switch is not instanteous (think hardware, so the scheduler must have streqy for how freuqnetly it changes the process that it is working on.

Scheduling Algorithims

- First come, first serve
- Shortest task first
- Priority
- Shortest Remaining Time
- Round Robin
- Multiple Level queue.

The algorithim that the scheduler will chose depends on the goals:

- maximize throughput
- maximixe fairness
- min waittime.
- minimize latency

Often times you will not have control over the scheduler, the OS handles this, and this is predetermined. Hence, you should avoid writing code, expecting different threads to execute in a specfic order or in specfic amount of time, since the OS will chose to run it differently each time you run it.

- From the file execution_scheduling.py we can see that two threads do not get the same amount of time. The output in two different runs is the following:

```
Bob chopped 17964 vegs
Sid chopped 20123 vegs
```

```
Sid chopped 11815 vegs
Bob chopped 11474 vegs
```

Both attempts are not the same, and we can see that Sid was given more time in the first, but the difference in the first run was much greater.

Thread LifeCycle

- When a program starts, it will have one thread which is referred to as the main thread. Then, it can create children threads. It is part of the same process, but each thread can run independently. Children threads, can also have children.
- When threads are done running, they notify their parent thread, and then terminate. The main thread is usually the last thread running.
- Threads typically consist of four different states.
    - New - this means that the thread has been created.
    - Runnable - this means that the OS has scheduled the thread to be run. Some languages require you to explict start a thread when you create it
    - Blocked - this state occurs when a thread is blocked by an external input (needs to wait for something to happen/finishing running), then it will typically go into the Runnable state.
    - Terminale - thread completes its tasks or it is abnormally aborted.

There are cases where a thread may need to wait for its children to finish executing before it can move on. In order to notify the parent that child is done, the child thread can call the join method. (The join method is called from the parent thread). The parent will be in a blocked state until child thread is done.

Daemon (Background) thread

- This is a type of thread that can be detached from the parent thread. An example of this type of thread is a grabage collector.
- It does not prevent the program from exiting if it is still running.
- Threads are typically spawed as non-daemon threads, and you must explicitly state that they are daemon threads.
- For daemon threads such as garbage collectors, it does not matter if the thread end aburtly, however if there is athread that is performing an IO operation (writing to a file), this can corrupt the data.
- Ensure that daemon threads do not have any negative side effects if it prematurely exits.