Solutions

Instructions for Homework 2:

- This is the second "k out of n" homeworks CS 4410.
- The homework may be done in pairs, or individually. If doing in pairs, one of you should upload to gradescope and add your partner to the group assignment in the upper right corner of the screen.
- The deadline is Wednesday, the 28th of September at 11:59AM.
- No late submissions will be accepted.
- You must attribute every source used to complete this homework.

1 Performance implications of task sharing

In this question you will explore the performance impact of spreading tasks across multiple threads and processes. Download the code located at `http://www.cs.cornell.edu/Courses/cs4410/2016fa/hw/hw2/timing.py`

This program always executes N independent tasks, but it spreads the work among varying numbers of threads and processes.

You can run this program by first choosing between a CPU or I/O bound task. Then you decide whether you want the task to be run sequentially, across multiple threads, or across multiple processes.

To run this code, you will also need the 4410 Synchronization Library in `rvr.py` which you will find on the git repo under P2.

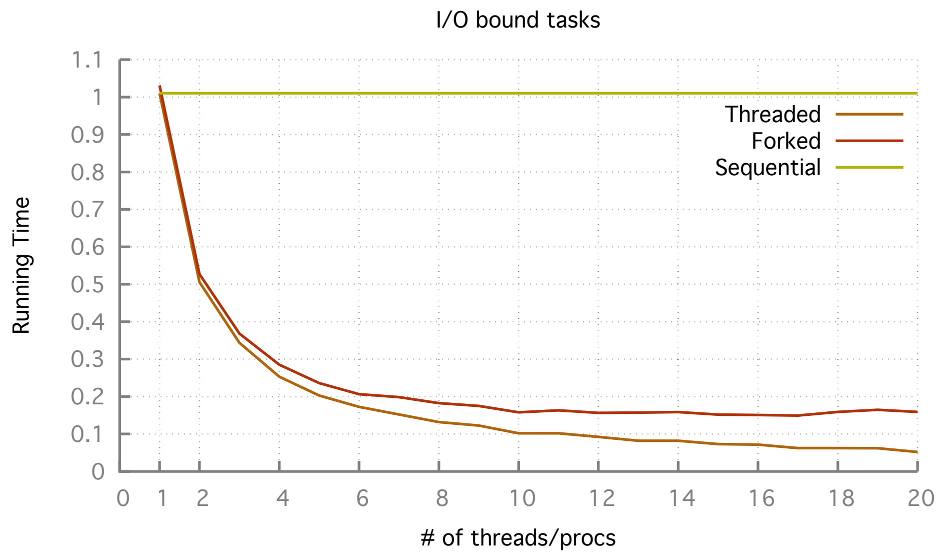
Here are some examples:

- `python timing.py cpu sequential`
  runs N cpu-bound jobs sequentially
- `python timing.py cpu threaded k`
  runs N cpu-bound jobs using k threads
- `python timing.py cpu forked k`
  runs N cpu-bound jobs using k subprocesses
- `python timing.py io (sequential | threaded | forked) ...`
  as above, but execute I/O bound jobs

1.1 Plotting I/O bound tasks

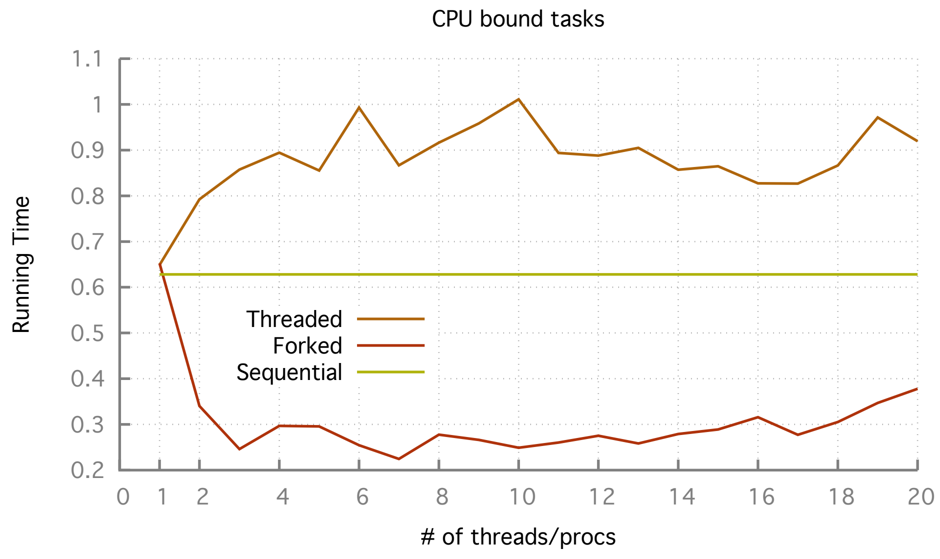
Create a graph with three curves for the I/O bound task, plotting run time vs. number of threads/processes:

- The total run time of the threaded implementation against number of threads (from 1-20 threads).
- The total run time of the multi-process implementation against the number of processes (from 1-20 processes).
- The total run time of a sequential implementation that runs in a single thread (just plot this as a straight line since the number of threads/processes will be held constant).



1.2 Plotting CPU-bound tasks

Create a second graph which is like the first, but which has the three curves for the CPU-bound task.
Do not hand-draw your graphs. Hand-drawn graphs are for middle school.



1.3 Discussion

Briefly explain why the curves have the shapes they do. A few sentences for each plot should suffice.

For I/O bound tasks, the sequential case has longest running time because every job waits for the previous one to finish. Multi-process and multi-thread both have reduced runtimes because some jobs can run on CPU while others are waiting on I/O tasks. Because these tasks spend the vast minority of their time on the CPU itself, runtimes decrease inversely with the number of processes/threads in both cases. Since *thread* creation is more light-weight from an overhead perspective, the performance of multi-thread is slightly better than that of multi-process.

For CPU bound tasks, multi-process performs better than sequential, and multi-thread performs worse than sequential case. In multi-process case, several jobs can run in parallel, making use of multiple cores, which reduces the total runtime. As it turns out, the multi-threading module used by rvr.py is actually single-threaded (running on a single core) due to the global interpreter lock in the Python interpreter. Because these tasks spend the vast majority of their time computing on the CPU, the cost of switching between threads (which share a single CPU) adds overhead that makes the runtime for the threaded case always worse than sequential case.

2 Semaphores vs. Condition Variables

You love the hungry kid example with condition variables from class so much that you decide to convert the solution into one that uses semaphores. You replace the monitor lock with a mutex semaphore. Then you replace all instances of `wait` with `P` and all instances of `signal` with `V`. The semantics of `P` and `V` remain unchanged. Your code looks like this:

```
1   int numburgers = 0;
2   Semaphore hungrykid(0);
3   Semaphore mutex(1);
4
5   void kid_eat() {
6           mutex.P()
7           while (numburgers==0)
8                   hungrykid.P()
9           numburgers--
10          mutex.V()
11  }
12
13  void makeburger() {
14          mutex.P()
15          ++numburger;
16          hungrykid.V();
17          mutex.V()
18  }
```

Safety guarantees that only one thread is in the critical section at a time. In this case, the body of functions `makeburger` and `kid_eat` are the critical section.

Liveness guarantees that a thread seeking to enter the critical section will eventually succeed. You may assume that there is a cook attempting to make an infinite number of burgers. (In other words, liveness will not be limited by the lack of burgers.)

Answer **True** or **False** for each of the following:

- This code provides safety and liveness. False. Liveness is not guaranteed. If `kid_eat` is called before any burgers have been made the thread will block on `hungrykid.P()` on line 8 while holding the mutex, preventing anyone from ever entering `makeburger`.

- This code provides safety and liveness, as long as threads can be interrupted/pre-empted. False. Preemption will not solve the problem because it will not release the mutex.

- It is possible to write a semaphore-based version of this code without the `numburgers` variable that provides safety and liveness. True.

- This code allows multiple concurrent threads to interleave their assembly-level execution of the increment/decrement of the shared variable `numburgers`. False. The mutex prevents this.

- The `makeburger` thread might never return. True. It could block forever on the `mutex.P()` on line 14.

- The `kid_eat` thread might never return. True. A first thread could block forever on `hungrykid.P()` on line 8. Subsequent ones could then block on the `mutex.P()` on line 6.

- The variable `numburgers` might go negative. False. `hungrykid.P()` on line 8 prevents this from happening.

3 Readers Writer Redux

This is a solution to the readers/writers problem that was introduced in class. The Monitor class implements a monitor lock for the programmer which is implicitly acquired/released at the beginning/end of each procedure. As usual, calling `wait` releases the monitor lock and returning from wait implies that it has been re-acquired.

```
Monitor ReadersNWriters {

   int waitingWriters=0, waitingReaders=0, nReaders=0, nWriters=0;
   Condition canRead, canWrite;

   void beginWrite() {
      if(nWriters == 1 || nReaders > 0) {
         ++waitingWriters;
         wait(canWrite);
         --waitingWriters;
      }
      nWriters = 1;
   }

   void endWrite() {
      nWriters = 0;
      if(waitingReaders)
         Signal(canRead);
      else
         Signal(canWrite);
     }

   void beginRead(){
      if(nWriters == 1 || waitingWriters > 0) {
         ++waitingReaders;
         wait(canRead);
         --waitingReaders;
       }
       ++nReaders;
       Signal(canRead);
    }

   void endRead() {
      if(--nReaders == 0)
         Signal(canWrite);
   }
}
```

3.1 Safety

In this scenario, saftey means not only that only one thread is in the monitor at a time, but also that the fundamental invariant of having only 1 writer writing *or* n readers reading at any given time is maintained. Does this code provide safety? For each error you find in the code (if there are any), identify the error in the code as a safety violation, describe a specific scenario which would violate safety, and *fix the problem*.

Answer: No, the code does not provide safety. By using an `if` statement instead of a `while` statement in line 7, it is possible for a writer to wake up, not check whether there is another active writer or reader, and return from `beginWrite`, resulting in either multiple writers or a writer and a reader accessing the shared data at the same time. Line 24 shows another case of this exact problem. A reader wakes up, doesn't check for active writers, and allows reading to take place on a shared stucture that might be being written to at the same time. The fix is to replace the `if` statements with `while` statements.

3.2 Liveness

Progress is a property of liveness stating that if no thread holds a particular lock and any thread attempts to acquire that lock, then eventually some thread succeeds in acquiring the lock. Begin with code that guarantees safety (either the original code or the code as you fixed it in the previous section). Does the code also guarantee progress? For each problem you find in the code (if there are any), identify it in the code as a progress violation and describe a specific scenario which would prevent progress.

Answer: There is a specific case where this code breaks down and no longer guarantees progress. Suppose there is an active writer and waiting readers and waiting writers. When the writer finishes, it signals a waiting reader in line 18. However, the woken up reader checks to see if there are any waiting writers in line 24. Because there *are* waiting writers, the reader goes back to waiting. (*Aside:* The only thing that would possibly wake anyone up at this point would be a spurious wakeup. In fact, there would need to be a spurious wake up for each waiting writer. Once all of the waiting writers have had a turn, the readers could finally have their turn.)

3.3 Anything Else?

Is there anything else that is either incorrect or inefficent about this cdoe that you haven't already addressed? If so, identify the problem in the code and breifly describe it here.

Answer: Lines 20, 30, and 35 are inefficient becuase they `signal` without checking to see if any threads are currently waiting on these condition variables. Lines 20 and 35 should first check `if (waitingWriters > 0)` and line 30 should first check `if (waitingReaders > 0)` before calling signal on their respective condition variables.