

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.

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.
- This code provides safety and liveness, as long as threads can be interrupted/pre-empted.
- It is possible to write a semaphore-based version of this code without the `numburgers` variable that provides safety and liveness.
- This code allows multiple concurrent threads to interleave their assembly-level execution of the increment/decrement of the shared variable `numburgers`.
- The `makeburger` thread might never return.
- The `kid_eat` thread might never return.
- The variable `numburgers` might go negative.

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.

```
1 Monitor ReadersN Writers {
2
3     int waitingWriters=0, waitingReaders=0, nReaders=0, nWriters=0;
4     Condition canRead, canWrite;
5
6     void beginWrite() {
7         if(nWriters == 1 || nReaders > 0) {
8             ++waitingWriters;
9             wait(canWrite);
10            --waitingWriters;
11        }
12        nWriters = 1;
13    }
14
15    void endWrite() {
16        nWriters = 0;
17        if(waitingReaders)
18            Signal(canRead);
19        else
20            Signal(canWrite);
21    }
22
23    void beginRead(){
24        if(nWriters == 1 || waitingWriters > 0) {
25            ++waitingReaders;
26            wait(canRead);
27            --waitingReaders;
28        }
29        ++nReaders;
30        Signal(canRead);
31    }
32
33    void endRead() {
34        if(--nReaders == 0)
35            Signal(canWrite);
36    }
37 }
```

3.1 Safety

In this scenario, safety 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*.

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.

3.3 Anything Else?

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