Homework 1

Instructions

You may work in groups of up to 4 and submit a single assignment for the group. For computational problems, please show your work; for conceptual questions, please explain your reasoning. Solutions may be neatly hand-written and scanned or typeset. Please submit your solution to Moodle **in PDF format**.

Due: Friday, March 5th, 23:59 AoE

Exercises

Exercise 1 (scheduling with different processor speeds). Suppose a task T is composed of four (purely sequential) sub-tasks T_1, T_2, T_3, T_4 which require 360, 210, 210, and 180 elementary operations (respectively) to complete. We have three processors at our disposal P_1, P_2, P_3, which can perform 480, 320, and 320 operations per second (respectively).

- Find an optimal schedule for the subtasks on the three processors. What is the latency of the computation?
- Suppose the speed of the three processors are still 480, 320, 320, but it is not known *which* of the processors is faster. What are the maximum and minimum latencies achieved by greedy scheduling?
- Again in the case where it is not known which of the processors is fastest, show that there is a greedy schedule in which a non-greedy schedule would have lower latency. (A schedule is *non-greedy* if at some time, there is an idle process as well as a task that is neither completed nor in progess. That is, the task *could* be assigned to the process, but is not.)

Exercise 2 (a naive lock). In class, we saw that unpredictable behavior could occur when multiple threads attempt to access a shared `Counter` object. The following `LockedCounter` class is an attempt fix the issue by adding a `locked` state to the object:

```
public class LockedCounter {
    long count = 0;
    boolean locked = false;
```

```
    public long getCount () { return count; }

    public void increment () { ++count; }

    public void reset () { count = 0; }

    public void lock () { locked = true; }

    public void unlock () { locked = false; }

    public boolean isLocked () { return locked; }
}
```

That is, a thread should only increment the counter if it holds the lock for the counter: if the `LockedCounter` is is already locked, the thread should (1) wait until it is unlocked before (2) lock the counter (so that others cannot simultaneously increment the counter) (3) increment the counter, and finally (4) unlock the counter. For example, to increment the counter a thread should execute:

```
void incrementCounter (LockedCounter counter) {
    while (counter.isLocked()) {
        //wait
    }

    // counter is unlocked

    counter.lock();
    counter.increment();
    counter.unlock();
}
```

Does this code ensure that multiple threads cannot simultaneously increment a `LockedCounter` (which led to the anomalous behavior we saw in class)? Why or why not? (Hint: if you aren't sure, write a program to test the code!)

Exercise 3 (Amdahl's law). Suppose you are given a program with a method `M` that executes sequentially. Use Amdahl's law to answer the following questions, assuming that the remaining operations in the program can be computed in parallel.

- If `M` accounts for 20% of the program's total execution time, what is the maximum possible overall speedup on a 2 processor machine? A 3 processor machine? An 8 processor machine?
- What is the maximum possible speedup for *any* number of parallel processors (i.e., as the number n of processors becomes arbitrarily large)?

- Now suppose `M` is replaced by a method `N` that accomplishes the same task but in half the time. How much faster is the new program than the old program on a 2 processor machine? A 3 processor machine? An 8 processor machine? An n processor machine (as a function of n)?

Exercise 4 (mutual exclusion). Consider the following modified `lock()` method for the Peterson lock (cf. Section 2.3.3 in *The Art of Multiprocessor Programming*):

```
public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
    victim = i;
    flag[i] = true;
    while (flag[j] && victim == i) {} // wait
}
```

The only difference with the original method is that the statements `victim = i` and `flag[i] = true` are reversed in the modified version. Does the modified Peterson satisfy mutual exclusion? Why or why not?

Exercise 5 (a challenge). A computer science professor offers the following challenge to their class: The students are lined up in a single file line, so that each student can see all of the students in front of them and none of the students behind them. The professor places a hat on each student's head; some hats are red while others are blue. Each student can see the colors of the hats of the students in front of them in line, but not their own hat color nor the colors of hats behind them. Starting from the back of the line, the professor asks each student in turn to shout a guess of their hat color (which they cannot themselves see). All other students can hear the guesses. If a student guesses their hat color correctly, they get an 'A' for the course.

The students are informed of this procedure and given a chance to dicuss their strategy to maximize the possible number of 'A' grades awarded. In a class of N students, what is the largest number of students that can be guaranteed to get an 'A' by correctly guessing their hat color? What procedure should they follow to achieve this result?