The Beauty and Joy of Computing[1]

Lab Exercise 7: Concurrency and parallelism

Objectives

By completing this lab exercise, you should learn to

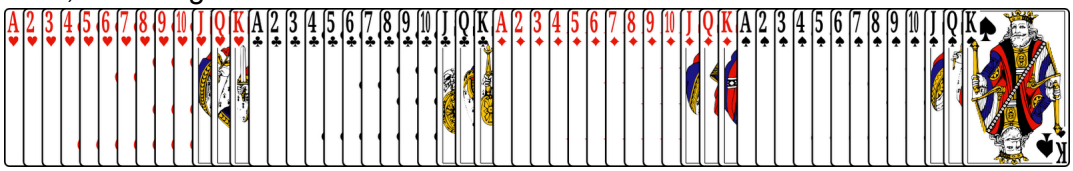
- Perform simple reasoning about concurrent tasks and parallel algorithms;
- Recognize and describe race conditions and the problems this poses for parallel programming;
- Recognize and describe deadlock and the problems this poses for parallel programming.

Background (Pre-Lab Reading)

There is no pre-lab reading this week. The activities are straightforward and focused, although some non-trivial concepts must be considered and thought about after the lab in order to answer the post-lab quiz questions.

Activities (In-Lab Work)

Activity 1: Get ready for some parallel programming - unplugged! In this activity you are to explore parallel processes performed by people, rather than electronic computers. The task is simple: Sort a deck of cards so that the cards are ordered first by suit and then by number. In other words, the end goal is a deck of cards that is ordered like this:



You should get in groups of 4 for this activity - each group will get a deck of cards to work with. You will also need to time various things, so you'll need a stopwatch. Some of you might have a watch with a stopwatch function, and others might have a phone with a stopwatch app. If you don't have either of these, visit the web site http://www.online-stopwatch.com/ and use the stopwatch from there (you bring it up with the big green arrow icon).

The first thing to do is to time each of the people in your group sorting the deck of cards. Have one person shuffle and give the cards to another person in the group, face down. Then, starting with a countdown to "Go!", the card-sorter will turn the deck over and put the cards in sorted order as fast as they can while being timed. Record the time each person in your group took.

Next, brainstorm strategies on how you can work together so that all four of you work **concurrently** (i.e., at the same time) on sorting a single deck of cards. How can you do this so that the work is spread out and you can work faster as a team? Come up with a strategy,

---

[1] *Lab Exercises for "The Beauty and Joy of Computing" [This is the final version for the Fall 2012 class]*

practice it a few times (and refine your strategy if you think of improvements), and then time your team on a sorted deck.

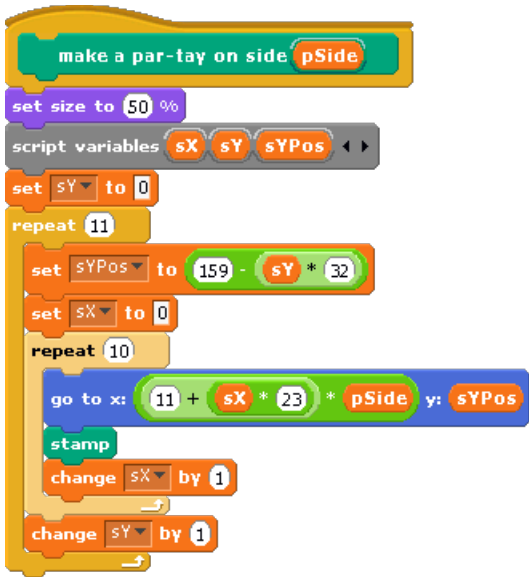
To finish up this activity, each group should collaborate on a single report that will be submitted for the group. Use Microsoft Word, put all of your names at the top, and then report the following: individual times for each group member, a good and clear description of what your strategy is and why you think it will work well for a team working together (be as technical and precise as you can here - don't just say "we thought it sounded like a good idea"), and then report the time for your team's solution on a sorted deck. Save this report as Lab7-Activity1. One person per team will submit this file, and the other team members can say in their submission "Our Activity1 report is being submitted by xxx," where xxx is the person who is responsible for submitting that report (make sure it's a responsible person -- if it doesn't get submitted, no one gets credit!).

Non-graded activity: After you've all had plenty of time to come up with a strategy and practice a few times, we'll have a race between teams. Make sure you shuffle your deck of cards good, and we'll redistribute the decks between the teams. Then we'll race and see who is the fastest!

Activity 2: Let's have a par-tay! As everyone knows, the key to having an awesome party is plenty of party hats, right? In this activity we're going to explore sequential versus parallel performance by filling the stage full of party hats, once with a **sequential algorithm** (an algorithm that uses only one **thread**, or sequence of operations), and once by filling the two sides of the stage concurrently. First, we need a party hat. Bring up BYOB, go to the sprites pane and create a new sprite using "Choose new sprite from file" and selecting "partyhat3" under "Things" - partyhat3 should look like this:

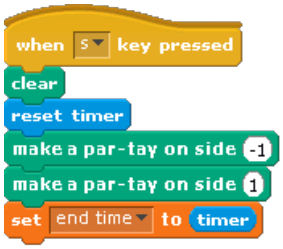
Even if you're tempted to pick a different party hat, use this one. Our block that fills the screen has some values hard-coded which depend on the size of the hat costume, and all of the hats are slightly different sizes - the code provided here is specifically for "partyhat3".

Now we want to define a block that fills either the right side or the left side of the screen with party hats. The following block definition does that, where the argument determines the side of the screen to fill: if the argument is -1 it fills the left side, and if the argument is 1 it fills the right side. Build this block using BYOB and test it with both -1 and 1 to make sure it works properly.



While this drawing script isn't the point of this lab, it's worth your time to figure out how this works - it's a good test of how well you understand BYOB!

We want to fill both sides of the stage with party hats, so one reasonable approach is to stack one "make a par-tay" block on top of another, and use argument -1 for one and 1 for the other. Let's time how long this takes - we'll start when the user presses the 's' key (here 's' is for "sequential"), so you want a script that looks like this:



Run it, make sure it fills the screen with party hats, and record the time.

Next, what if we filled the two sides of the stage concurrently, using a **parallel algorithm** (i.e., an algorithm that uses multiple threads to solve a single problem)? To try this, first duplicate the script that was triggered by the 's' key and make the following changes: change it so that it starts when the 'p' key is pressed ('p' is for "parallel"), and remove one of the "make a par-tay" blocks so that you fill only the left or the right side of the stage. Does that work for half the stage when you press the 'p' key? Now duplicate that script, and change the argument to "make a par-tay" so that it fills the other side of the stage. If you followed those directions, you should have two scripts that are both triggered by pressing the 'p' key and each fills a different side of the stage. Since both scripts are triggered by the 'p' key, if you press 'p' both will run concurrently. Here's what the stage looks a little after pressing 'p':

If that mostly worked ("mostly" because note that there's a gap on the top row where a hat should be - we'll talk about that below), then record the amount of time that this takes.

Finally, think about why there's a hat missing in the top row (that wasn't missing when we did the same operations sequentially!). In the quiz I'll ask you to explain what went wrong, but if you want to explore this issue in the lab, this is the time to do it.

Extra credit (5 points) opportunity: Can you correct this code so that all hats are drawn with two concurrent threads? If you do this extra credit, do not change/destroy the scripts you just wrote that were triggered by the 'p' key - instead, make new scripts that are triggered by a different key (like 'f' for "fixed"). Hint: if there is an initialization operation that is happening in both threads, but should really just happen once before either thread starts running, you can do something like this: have the keypress event trigger a single sequential script that does the initialization once, and then broadcasts a signal to start the two concurrent threads. In doing this, you have divided your algorithm into a sequential part and a parallel part, which is done quite often when writing parallel programs.

When you are finished you should have at least two sets of scripts in your project: the one that is triggered by the 's' key and fills the stage sequentially, and the ones that were triggered by 'p' and fill the stage in parallel. If you did the extra credit you'll have additional scripts as well. When you complete this activity, save your project in a file named Lab7-Activity2.

Activity 3: *This activity does not build on the previous ones - after you're sure the previous activity has been saved, start a new project in BYOB to clear out your previous work.*

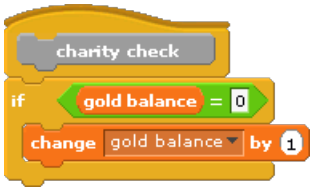
Race conditions are situations in which the interleaving of operations between different concurrent threads violates conditions that should be maintained in a program. Race conditions most often arise when some operation should only be performed when a particular condition is satisfied, but there is a gap in time between when the condition is checked and when the operation is performed. The time gap can be very short - microseconds even - but in that time

gap, some other thread may access the data and make it so the condition no longer holds. Since the first thread had already checked the condition and made the decision to perform the operation, it ends up performing the operation when it shouldn't.

Here's an example of a race condition: We are writing a game (SuperSpender™) in which a player has a certain amount of gold, and certain spending actions will decrease the amount of gold the player has - but the amount should never be allowed to be negative! We have a particular action in the game where, if the player has some gold then some processing will occur (using the gold) and then the amount of gold in his inventory will be decreased by one. We don't have a complicated game to use, so in this lab we'll just put a random "wait" in place of this processing, and we define the following block:

Notice that it should be impossible for any use of this block to result in a negative gold balance.

Next, our game has a "charity" function: if the player ever gets down to a zero balance, then the game gives him one gold piece. Here's the "charity check" block:



The main loop of the game can be described this way: we iterate forever, spending gold if possible and doing the charity check. The first thing to do for this activity is to build these block definitions as shown above, and then create the "forever" loop that repeats these two block operations. You will have to create the "gold balance" variable, and should initialize it to 1. Set "gold balance" as a watch variable, and run your loop - pay attention to what you see in the "gold balance" value. Since this is in a "forever" loop, you'll have to click on the stop sign to make it stop executing.

And now one last simple addition: The player can cause an action to occur, which we will simulate by the player pressing the spacebar, that causes the "spend gold if possible" block to execute. Create the script that does this - it's just two blocks: the appropriate hat block followed by the call to "spend gold if possible."

Think about this a little before proceeding: the "spend gold if possible" block is the only place the balance is ever decreased, and it always checks to make sure there's at least one gold piece available before decrementing by a gold piece. Obviously we should never have a negative balance, right?

Start your "forever" loop going, watching the "gold balance" variable, and start pressing the spacebar. Before too long you should see the "balance" decrease to -1. How can that be

possible? Make sure you can answer that question clearly, because it will be a question on the post-lab quiz!

Finally, let's fix this problem. Race conditions are typically avoided by using **locks** that give a thread exclusive use of a variable like the "gold balance". We will define a new variable named "gold locked" - a thread can "get" the gold lock if it is false (so no other thread is holding the lock), in which case it will immediately set the gold lock variable to true so that no other thread can get the lock. If this thread couldn't get the gold lock because it is true (i.e., some other thread is holding the lock), then we wait until the lock becomes false before we proceed. When that thread is finished with the variable, it will "free" the lock by setting the "gold locked" variable to false. We define these two blocks to get and free the gold lock:

Once these are defined, modify the "spend gold if possible" definition so that it starts with a call to "get gold lock" and ends with a "free gold lock". Now, making sure you start with "gold locked" initialized to false, try the experiment again with the "forever" loop running while you press the spacebar. Can you make the balance go to -1 now? Make sure you can clearly explain why this fixes the problem for the quiz questions!

When you're finished with this activity, save your project as Lab7-Activity3.

Activity 4: Deadlock is another big danger in parallel programming, which comes up precisely because of the locks we created to avoid race conditions! A system is in **deadlock** if a thread is waiting on a lock that will never be freed because some other thread holds the lock and can't make progress that is needed to release the lock. We were very successful with our first game, so now we're ready to produce SuperSpender™ v2.0. The exciting change? Now we have silver as well as gold! This will surely be a best-seller.

We need to be able to convert between gold and silver, so we create blocks to convert gold to silver and silver to gold. Since we learned our lesson about race conditions so well, we use a lock on the gold balance and a lock on the silver balance. When converting gold to silver, we lock the gold balance and work with the gold balance a little bit, then lock the silver balance, work with it a little bit, and finally free both locks when we're done. Converting silver to gold works the same way, but with the order reversed, giving:

To define these blocks you'll need to first define a "silver lock" and the get/free blocks for that lock (they look just like the blocks for the gold lock). Once that's done, create the gold/silver conversion blocks -- now we want to test how well this works. Our testing script will do this: make sure both locks are free, and then we'll start a counter going and repeatedly convert gold to silver and silver to gold, saying the counter variable each iteration. This is the testing script:

Build that script, start it running, and then the counter will start steadily increasing. So far so good! Finally, we want a user-initiated silver-to-gold conversion, so start with a hat block that is triggered when the 'z' key is pressed and just calls "silver to gold". While the counter is increasing, start pressing the 'z' key - eventually the counter will stop increasing and everything will freeze up. That's a deadlock!

Examine these scripts - experiment with them a little if you need some more insight - and make sure you understand what sequence of events causes the deadlock. Once you're finished with this activity, save the project that contains all these scripts as Lab7-Activity4.

Discussion (Post-Lab Follow-up)

No specific post-lab discussion topics are included in this lab.

Terminology

The following new words and phrases were used in this lab:

- *concurrently*: two or more tasks or threads that are executing at the same time
- *deadlock*: the situation that arises when a set of threads are all waiting on a lock held by some other thread in the set, so that no progress can be made by any thread
- *lock*: a special kind of variable that controls exclusive access to some section of code - only one thread can be executing in this section of code at a time, and we say that thread "holds" the lock, while all threads must wait until the lock is released (or freed)
- *parallel algorithm*: an algorithm that uses multiple concurrent threads to solve a single problem
- *race condition*: a situation in which two threads interfere with one another, so that thread A has determined that a necessary condition is met, and then thread B violates that condition before thread A gets a chance to act on that condition
- *sequential algorithm*: an algorithm with a single thread
- *thread*: a "flow of control" or sequence of operations that occur in an algorithm

Submission

In this lab, you should have saved the following BYOB files: Lab7-Activity2, Lab7-Activity3, and Lab7-Activity4. In addition, one member of each group will have an MS Word file named Lab7-Activity1. Turn your files in using whatever submission mechanism your school has set up for you. If you are not the person in your group submitting the Activity1 write-up then just include a note saying which of your group members is submitting that report.