Team Name: TADA (Team: Andrew Mendelsohn, Dennis Chen, Alison Tai)

Problem Statement: Since we are all relatively new to concurrency, we feel it would have been useful to also have some sort of education game that bolsters concurrency knowledge. We are hoping to put the puzzles we did during class in a more engaging, yet still education, context.

Project Summary

The general idea of the project is to create an amusement park themed educational game (like VIM Adventures) that goes over the basic topics of concurrency. The player will go through different concurrency puzzles abstracted to appear as typical issues to solve at an amusement park. For example, the turnstile puzzle can be equated to building an actual turnstile at an amusement park which allows people into the park's capacity when the park opens.

The game will be represented by a series of puzzles grouped into sections by topic and/or difficulty. The game will first provide basic information about pieces of concurrency vocabulary (such as semaphore, mutex, threads) and then offer a related puzzle. Players will then find solutions to the puzzles by trial and error, such as figuring out where to put a semaphore. Concurrent python code will then be run based on their responses, and simulated to the player in order to provide a view of the progress of threads as they run. The simulator will allow the player to watch their solution run in real (or slowed) time to aid in debugging their understanding of threading properties.

This game will target students/learners who know the fundamentals of code, but are complete beginners to concurrency concepts. In addition, the code involved will be in a mix of python and pseudocode so there will be a low barrier to entry, which is important to put the focus on the concepts.

Deliverables

- Minimum: text-based game
    - A playthrough should provide a simple introduction to the concepts of threads, and guide the player to find solutions to many of the classic threading problems discussed in class.
    - Game can be run from the command line (**python interface.py**) and is represented through ASCII on the command line.
    - Fully playable with at least 3 puzzle levels available.
    - Simulator without error checking to demonstrate correct solutions.
- Maximum: graphic-based game
    - Game has full GUI including visualizer for observing thread progress in the solution simulator.
    - Simulator with error checking to demonstrate all solutions correct or incorrect.
    - Large database of related topics and vocabulary for user to interact with

Original Development Plan Timeline

11/10 - Built Game I/O and Text Interface modules
11/14 - Planned out puzzles and began Puzzle module
11/20 - Built Puzzle module and 3 text puzzles
11/24 - Built text Simulator module (**Minimum Deliverable**) & look into tKinter for GUI
11/26 - Built GUI module
11/30 - Extended Simulator module to display erroneous solutions. (**Maximum Deliverable**)
12/3 - Team presentation
12/5 - Written project final report due

Design Decisions

Python

Our choice of implementation technology to use relates strongly to how the user will actually learn about concurrency. We considered Erlang, but it doesn't have the same notion of simple threads as Python does. As a result, for implementation technology, we have chosen to use Python because of a couple of different reasons. We feel it would be easier to teach the basic topics of concurrency with the paradigm that Python provides. Python also has tKinter, and since our maximum deliverables involve having a graphical interface to visualize thread movement, we want to optimize for that.

Amusement Park Theme

Another design decision we addressed was how to abstract the idea of concurrency to the user, so that they would be able to relate to it. We thought of a couple of different ideas that would allow for this such as simply doing puzzles or having an adventure type game, but we settled on using regulating an amusement park. What this means is that the user will be put in change of different events that typically occur in an amusement park, and has to make decisions that would make the park run most optimally.

For example, there are concession stands in an amusement park and we would want to keep track of how many resources we have of a specific good, like cotton candy. We want to know how many are sold in a day and we want to restock it if necessary. This would require locks to make sure the data is only being updated at one point in time. Another example is when customers come to the amusement park, and a semaphore can be used to figure out how many people are in the park at one time. A semaphore can also be used to identify how many people are on a ride so that there are not too many people on the ride at once.

We also decided that we would combine the idea of puzzles into the amusement park because we can then more easily separate ideas from each other. For example, a roller coaster puzzle would deal with semaphores and isolate that idea, while concessions could deal with locks. We chose to represent the puzzles this way because it seems like it would be a lot more relatable to an individual instead of just using generic concurrency puzzles and random sections of code.

Puzzle Representation

We have chosen to implement puzzles as a given outline of a correct solution with the key concurrency logic stripped away. This allows us to simplify the puzzles and avoid taking

arbitrary code as input from the user, reducing the burden on our simulator and isolating the most important piece of understanding from extraneous logic in the puzzle. A solution to the puzzle will be constructed by inserting calls to concurrency logic functions in the given code. In the text interface, this will be done using line numbers. Ideally the GUI implementation would allow for drag-and-drop input.

Data Structures and Algorithms

We will use Python to simulate correct answers that will then be displayed to the user in a slowed down visualization of the state of threads and semaphores throughout the code being run. Otherwise, the data structures that we are using are simple enough that we can use the ones that Python provides for us, like lists. There are not really any algorithms that we are currently planning to use, besides those implemented for the puzzle solutions. We are simply using the idea of puzzles to propagate concurrency knowledge.

Puzzles

The puzzles we will be using are all puzzles from the book (*The Little Book of Semaphores* by Allen B. Downey). The ones we will be implementing as levels of the game are as follows:

1. Rendezvous (two people meet before entering the park)
2. Turnstile (let everyone into the park when it opens)
3. Reusable Barrier (maintaining the line for the rollercoaster)
4. Producer-Consumer (cotton candy stand making cotton candy for parkgoers)

The puzzles will be very closely based on the ones from the book, though are making a few tweaks to ensure they run well in our threading simulator. An example might look like this:

```
Mutex puzzle 01
Global:
mutex=Semaphore(1)

Thread:
1
2 #criticalsection
3

Choose a line to put mutex.wait(): ___
Choose a line to put mutex.signal(): ___
```

Results

Outcome

The minimum deliverables were achieved. While one or two deadlines were missed, planning for finishing the minimum deliverables before break (and then having break to continue working on the project) gave plenty of cushion to complete them. As can be seen

from the development timeline, maximum deliverables were left for Thanksgiving Break to allow for the possibility of having enough free time to work on the project's extra potential. We ran into some scheduling problems with work from other classes, and ended up extending work on the simulator into Thanksgiving Break.

We then ran into the realization that we had failed to account for the time required to link the pieces together, such as fitting the puzzles module in with the simulator so that everything was accurate, and ended up unable to reach our maximum deliverables. All in all, though, we are happy with the project that we produced and are happy with what we've produced. Upcoming improvements would be to create more puzzles to fit into the amusement park.

Design

The best design decision we made was to modularize the interface, puzzles, and simulator. This made for an easier development process since we could each work on pieces individually. This also made debugging easier since each piece could be individually tested before being fitted in with the others. It also meant that the concurrent part of the project was encapsulated solely within the simulator, and then guaranteed that failure or bugs that were from a concurrency issue could solely come from one location. This was helpful in debugging heisenbugs or anything that pertained to the concurrent piece of the project, which could involve the most frustrating bugs. The way this was structured also allowed for easy building in of a GUI interface, though that is something we did not get to.

Something we never specifically designed, in order to jump in and get started coding as soon as possible, is what exactly the interface would look like. We left that up to whatever worked while coding, but it could have been more effective to have a thoroughly thought through interface for the user. While the current interface is useful, and gets across the game, it definitely could be improved through tweaks to the language or visualization.

Division of Labor

Since we didn't keep in constant communication on progress with each piece, we often let an interface slip a little here and there. That being said, we had divided the project in a way that easily allowed the members of the team to work independently. The class diagram we designed helped us establish what exact data structures and information needed to be passed in and out of the interface in order for the project to work. It was easy to then put the pieces together when we had built each of our pieces once they were complete.

Bug Report

The most difficult bug we had to find was within the simulator code. There, we translate the abstracted puzzle responses into code, and then use exec() which is a Python function that takes a string of code and runs it. We ran into an issue where python will not allow use of exec in a nested or dynamically created function. This was quite worrying as we were pretty far along in our project at this point, and this bug threatened to break our entire approach to thread simulation. Fortunately, this bug was fixed by executing code in a

temporary context of globals(), locals() as this will not violate the scoping rules. In retrospect, this solution is quite a hack and it would be nice to find a cleaner solution.

Code Overview

The full program can be run from any terminal with Python 2.7.6 (or higher), using the following command: `python interface.py`. The included files are named as follows: interface.py, textInterface.py, puzzle.py, rendez_vous.py, and simulator.py. Descriptions of each of these files follows.

interface.py – This file calls the desired interface module that has been set. Right now, it uses functions from textInterface.py, but the interface module could be changed to a GUI interface by only changing what file is imported. It gives a quick overview of set-up and starts the program.

textInterface.py – This file is the text portion of the project that the interface relies on. It has code relating to how the code is visualized to the user such as a welcoming screen and other sorts. This is where the puzzles are first found for the user depending on how many they want to see (between 1-3). It also allows the user to view progress and view a dictionary of helpful words/phrases about concurrency, such as what deadlocking is.

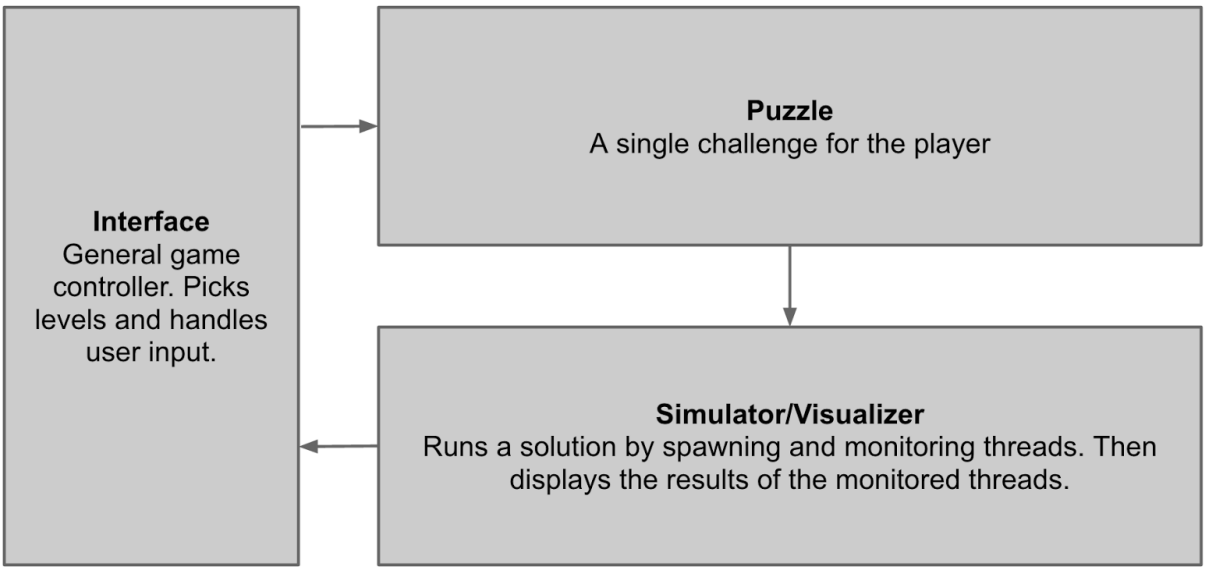
puzzle.py – This file contains the body of how a puzzle is put together. Puzzles are based off this module. For example, the rendezvous puzzle (rendez_vous.py) contains the puzzle information and initializes itself through the puzzle module. Each puzzle has sections such as title, lesson, hint, answer and space for code. The puzzle then has an important part and it is the start_puzzle() function. This function, when called, runs the necessary components to run the puzzle for the user. This is funneled into the interface.

simulator.py – This file works with the puzzle to actually simulate what the user puts as their answer. There is a visualizer function in this file that will allow the user to see what mistakes they made and why their solution did not pass. This file is also where the concurrency of the project comes into play because it will spawn threads to run the simulation. For example, in rendez_vous.py, it will actually have alice and bob wait for each other so if the user puts in incorrect code, they will see that they will both be waiting for each other.

rendez_vous.py - This contains all of the code and information required for the rendezvous, or waiting to enter the park together, puzzle.

Diagrams

Class Diagram



Object Diagram