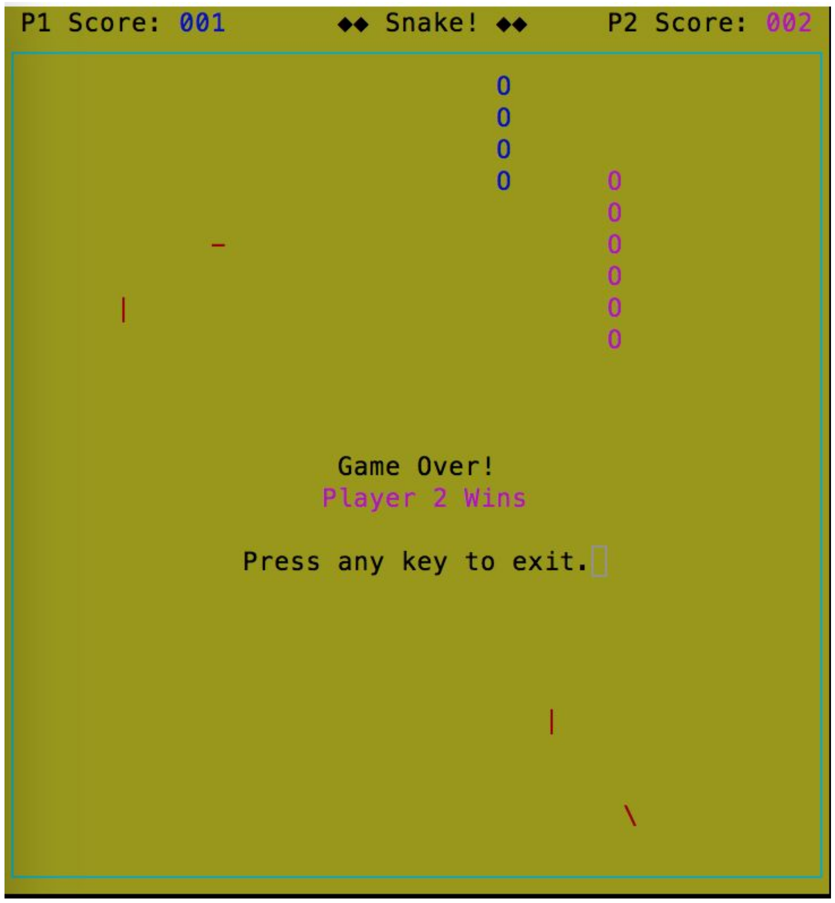
Project Report: Multiplayer Snake Game



```
P1 Score: 001          ◆◆ Snake! ◆◆          P2 Score: 002

                     Game Over!
                   Player 2 Wins

               Press any key to exit.
```

Jemuel Santos and Ana Segebre
Professor Charlie Miskimen Curtsinger
CSC 213: Operating Systems & Parallel Algorithms

Project Overview

Snake has traditionally been a single player game. However, with the ever evolving gaming industry in the 21st century, the demand for multiplayer games has risen. Grinnell College students have the moral obligation to incorporate a multiplayer feature into old beloved games. Additionally, with Grinnell's remote location, entertainment is scarce which incentivizes students to revolutionize old passions such as snake. As a result, we created a multiplayer snake game which will promote community through healthy competition. The game involve two snakes on a single board which will revolutionize the way the game has been traditionally played. This will force players to create innovative strategies in order to become the biggest snake in Grinnell history. The rules are simple. Each player has to eat apples to have a higher score. If a snake crashes against the board borders, itself, or the other snake, then the game is over. The player with the highest score wins!

The game has a central server design, where player 2 connects to player 1. Player 1's server constantly sends the board and snake movements to player 2, and player 2 sends their pressed keys to player 1 for them to be processed. We send these game updates through sockets, and with the help of threads, we constantly check for new updates to read or write to each player. We also use a round-robin scheduler to determine which tasks need to happen depending on the updates. For example, the scheduler makes sure to render apples at random locations, and after a snake eats it, the apples is scheduled to disappear and then the snake increases size and moves locations. With the use of scheduling, parallelization and communication across networks, we can successfully render the game for both player 1 and player 2 in different machines.

Design & Implementation

Scheduler:

The “Multiplayer Snake” has some basic components needed to function. We need to update the two snakes (their position, direction, and length), we need to draw the board, we need to read input from the players, and we need to generate the apples and update them once they are eaten or have disappeared. These are the necessary tasks for the game to run smoothly. We use a round-robin scheduler in order to handle these different tasks. We implemented this scheduler for the “Worm!” lab and it works without preemption. A task runs until it is either done or performs a blocking operation. The task waits until it can run, then after running, it exits, and before running again, it either waits for other tasks to be completed, waits for user input, or sleeps for an arbitrary amount of time. When the task is waiting, our scheduler finds the next appropriate task to run and runs it. After that task is done, any other task that was blocked by this now finished task, runs. Since our scheduler follows the round-robin policy, it is sequential in the sense that if the first task is blocked, it moves on the the second one, and so on. We implemented the scheduler by using a struct that contains the information of the task, including its context or state, exit information, and what the task is waiting for. We use the `makecontext` function to create a task and `swapcontext` to make it run, where it switches from one task to the next one (determined by the first task’s waiting information). Our scheduler initializes a task, creates it, sets up its wait information, set up its sleep state, handles exits and we also individually handle the reading input from user case where we wait until a key is pressed. With our scheduler, we can handle the game’s tasks after establishing a connection between player 1 and player 2.

Game state across network:

Setup
We first set up the game using POSIX sockets to have a central server-client system. Player 1 is the server and player 2 is the client. The server opens a socket and waits for a connection. On the other hand, the client connects to the server through its port number. Once a connection is established, we do the initial setup of the game. We initialize the screen with the `ncurses` library and verify the machine supports color. Then we initialize color pairs for the player 1 and player 2 snakes, the text, the background, the apples, and the borders. Then we initialize the keypad to support input from the user. We call `init_display` to print the “Snake” title as well as the board borders and turn on and off attributes using functions from the `ncurses` library to color these objects. We finally initialize the board to all 0’s to represent empty spaces and set two cells next to each other in the middle of the board to 1 and 625 respectively. The 1 represents the head of the snake of player 1 and the 625 represents the head of the snake of player 2. We chose the number 625 because it is half the area of the game board. We also check if the user typed the right arguments and if they request the rules,

we print out the rules of the game. We then initialize the scheduler and we can move on to the individual roles of the server and client.

Server's Role
The server uses the scheduler to run most of the tasks in the game. The server updates player 1's snake and player 2's snake. It also draws the board and reads input only from user 1, and finally it generates and updates the state of the apples. The server creates these tasks and waits for them to finish, with an exception to the generate apples task as it creates a delay when exiting. The server also has a thread that is constantly reading player 2's keys sent by the client and uses that information to update the direction of player 2's snake.

Client's Role
The client only uses the scheduler to run two tasks in the game: drawing the board and reading input from player 2. It then waits for these tasks to finish. The client has a thread that is constantly reading the board information sent by the server. We had several issues reading the board from the server, and the server reading the player 2's keys from the client. The game board would not render the same on each screen. We then call a helper function everytime we read information from sockets. This helper function has a buffer and it verifies that the package it read is the size it is supposed to be. If it is not, it keeps reading until it read all of the buffer. This helped fix the incomplete reading across networks.

Tasks:

Now that we have discussed the overall set up and individual roles of the server and client, we will explain the specifications of each task that we've mentioned. All of these tasks work as long as the `running` variable is true. When a player exits the game or collides with itself, another player, or the board border, the `running` variable is set to false and we exit these tasks.

Update Snake Task
This task is divided in two parts: one for player 1 and another for player 2. They function is the same except they are in charge of updating different snakes. While `running` is true, we traverse through each of the cell's content of the board and find the head of the snake (1 for player 1 and 625 for player 2). Then we add 1 to the following cell to increase the length. The initial length of each snake is 4. If the cell is bigger than our recorded length (meaning it is a cell we went through but we are no longer in that position), we set this cell to 0, thus removing this segment from the snake. In order to move the snake, we simply add 1 to the next cell in the direction we are moving (dictated by the user input) and then add 1 to the previous cells. If the snake collides with the game board, with itself or another snake, then `running` is set to false and we send a key as our input so our task for reading input isn't blocked and can exit properly. If the snake collides with an apple, then we add one segment to the snake's length.

Finally, we write the board to the client using their socket file descriptor so the client can update the board accordingly.

Draw Board Task
This tasks constantly renders the boards' borders, snakes, blank spaces, apples, and scoreboard. In order to make sure the client knows the appropriate scores, it calls the function `score_counter` which simply determines the snakes current score by reading the board cell's to find its length. It also determines which player is the current winner. The board colors the snakes and scores different colors using `ncurses` to differentiate between the two players.

Read Input from Players Task
This task is divided in two parts: one for player 1 and another for player 2. They function the same except they are in charge of reading input by different players. First, it reads if a key was pressed, and depending on the key, it updates the snake's directions to north, south, east or west. If a player presses `q`, then it exits. When reading keys from the client, we write the updated snake's directions to the server using its socket file descriptor.

Generate Apples Task and Update Apples Task
When generating apples, we find a random cell in the board, and if it is empty, we set the value to -1, which means it is an apple. Then when updating them, we set the -1 cell to 0, meaning it is empty again. We sleep within these tasks to make the apples disappear after an amount of time.

Exit Setup:

We had several issues exiting the game properly across networks. The client would not know the results of the game. In our proposal, we mentioned we would have a scoreboard at the end that requires saving the top scores in a file. Instead of sending a file across networks, we initially tried sending the scores across networks as we did with our board, but there was an evident lag in the game, thus having a thread sending several things at once was not efficient. We then decided to keep track of the score, both as a server and as a client, by calling the `score_counter` function mentioned earlier. Since the client knows the score when the game ends, it can determine the winner. In the end, we call the `end_game` function to print a "Game Over" message and print which player won or if it was a tie. The players can now click any key to exit the game and then we finally clear the board screen.

Evaluation

To evaluate our system, we created a baseline for how the game is supposed to work. In order for us to say that the system is working, the game must terminate properly for both the server and the client. The terminating conditions are players colliding with the other player, or colliding with the edge of the board or pressing the q key while in play. Once one of the terminating conditions have been met, the player on the server end must be presented with the Game Over screen as well as the winner of the game. Once the player on the server quits, the client will then be presented with the results of the game. The game can be played again by re-executing the program from the terminal on both the server and client end.

We have tested the game within one machine and across multiple machines. When testing within the same machine, we opened two terminals and used one as the server and the other as the client. The game worked for both terminals which means that player 1 and player 2's inputs were read properly, and the game terminated properly. When testing across two machines, one was used as the server and the other as the client. The system worked properly in this case as well.

To expand our test cases, we tested across 2 different rooms on campus in which case the system continued to work as intended. The game was also tested on a single personal machine, a Macbook Pro, in which case the system continued to work as intended. This test was performed on and off-campus to test the system's reliance on Grinnell's network but the system continued to work as designed.

Overall, the system has worked properly across all testing conditions and should be expected to work across two machines which have a network connection.

Sources

We utilized Professor Curtsinger’s “Worm!” lab as a base for our basic functions of the snake game and the scheduler.
(https://www.cs.grinnell.edu/~curtsinger/teaching/2019S/CSC213/labs/worm/)