CS162 Fall 2011 - Project 1

Concurrency and Multiprogramming

Instructors: Anthony D. Joseph and Ion Stoica

1 Project 1

Concurrency is a fundamental part of operating systems. Although underlying hardware often supports only one point of execution, all modern operating systems allow this resource to be multiplexed between multiple threads of program control. The goal of this project is for you to use the synchronization primitives available to you to build correct, parallel programs. This assignment is separated into two sections: an individual portion and a group portion. You must complete the individual portion on your own, and hand it in along with your design document. The group portion will be submitted later, and only once per group.

1.1 Individual Portion

1.1.1 Definitions

Give a concise answer (maximum two sentences) for each of the questions:

- What is the difference between a *Heavyweight* and a *Lightweight* Process?
- Give a concise definition for a semaphore.
- As discussed in lecture, what is the *Readers-Writers* problem?
- Give a concise explanation of the differences between using *Shared Memory* for sharing versus using *Message Passing* for sharing.

1.1.2 Threads

Suppose you have a variable x initialized to 0, and then start two threads running the following pieces of code:

Thread A	Thread B

`for (int i = 0; i < 3; i++) {`
`  x -= 1;`
`}`
	`for (int j = 0; j < 3; j++) {`
	`  x *= 2;`
	`}`

Assume that load and store instructions are atomic, and that x must be loaded into a register before being increased (and stored back to memory afterwards).

1. What are the possible values of x after both threads finish? Explain at least one way in which each value can be reached.
2. Suppose that in thread B, you replace the line x *= 2 with a special, atomic hardware instruction that shifts x left (this instruction cannot be preempted). What are the possible values of x after the threads finish now?

1.1.3 Readers and Writers

Some versions of UNIX support a generalized notion of semaphores: in addition to the standard **P** and **V** operations there is an additional **Z** operation in which the caller waits until the value of the semaphore reaches zero, then proceeds (without changing the value of the semaphore). Furthermore, one may perform multiple **P**, **V**, and **Z** operations on a set of semaphores as a single, indivisible operation. For example, the operation

(**P**(S1), **P**(S2), **V**(S3), **Z**(S4))

means that the caller does not proceed unless S1 and S2 are positive and S4 is zero. When the caller does proceed, no other threads may operate on the semaphores until the caller subtracts one from the values of S1 and S2 and adds one to the value of S3.

1. Give a solution to the Readers-Writers problem in which writers have priority over readers. Your solution should have a Readers function and a Writers function and should consist entirely of calls to **P**, **V**, and **Z** -- there should not be any C code.
2. Using your solution, suppose that all of the following read and write requests arrive in very short order (while R1 and R2 are still executing):
   Incoming stream: *R1, R2, W1, R3, W2, R4, W3, R5, R6, W4, R7, W5, R8, W6, R9*
   In what order would the database process these requests? If you have a group of requests that are equivalent (unordered), indicate this by surrounding them with brackets. You can assume that the wait queues of the semaphores are FIFO (i.e., **V**() wakes up the oldest thread on the queue).
3. How can you redesign the code in (1) to run requests in an order that guarantees that a read always returns the results of writes that have arrived before it but not after it? (Another way to say this is that the reads and writes occur in the order in which they arrive, while still allowing groups of reads that arrive together to occur simultaneously.)

1.1.4 Deadlocks

A restaurant would like to serve four dinner parties, P1 through P4. The restaurant has a total of 8 plates and 12 bowls. Assume that each group of diners will stop eating and wait for the waiter to bring a requested item (plate or bowl) to the table when it is required. Assume that the diners don't mind waiting. The maximum request and current allocation tables are shown as follows:

Maximum Request	Plates	Bowls
P1	7	7
P2	6	10
P3	1	2
P4	2	4

Current Allocation	Plates	Bowls
P1	2	3
P2	3	5
P3	0	1
P4	1	2

1. Determine the Need Matrix for plates and bowls.

Need	Plates	Bowls
P1		
P2		
P3		
P4		

2. Use the Bankers' Algorithm to determine if the restaurant be able to feed all four parties successfully. Explain your reason.

3. Assume a new dinner party, P5, comes to the restaurant at this time. Their maximum needs are 5 plates and 3 bowls. Initially, the waiter brings 2 plates to them. In order to be able to feed all five parties successfully, the restaurant needs more plates. What is the minimum number of plates the restaurant would need to add? Show a safe serving sequence.

1.2.1 Fork, Exec, and Wait

Read the man pages for `fork`, `exec`, and `wait` on a linux machine (i.e., type '`man fork`'). Next, examine the code below and then run the code (`forkexercise.c`). Once you familiarize yourself with the code, please answer the following questions.

`forexercise.c`

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv) {
  FILE *fp;
  int tid;
  int waitint;

  fp = fopen("test.txt", "a");
  if (!fp) {
     perror("fopen");
     return 1;
  }

  printf("[%i] parent starting up\n", getpid());
  tid = fork();
  if(tid > 0) {
     printf("[%i] parent about to wait\n", getpid());
     fprintf(fp, "[%i] parent wrote to file\n", getpid());
     fclose(fp);

     wait(&waitint);
     printf("[%i] wait returned: wait_int: %d\n", getpid(), waitint);
  } else {
     char *exec_argv[3] = {"external", "hello from exec!", NULL};

     printf("[%i] child running\n", getpid());
     fprintf(fp, "[%i] child wrote to file\n", getpid());
     fclose(fp);

     execv("external", exec_argv);
  }
  return 0;
}
```

external.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  if (argc > 1) {
     printf("[%i] [%s] %s\n", getpid(), argv[0], argv[1]);
  } else {
```

```
        printf("[%i] [%s]\n", getpid(), argv[0]);
    }
    return 0;
}
```

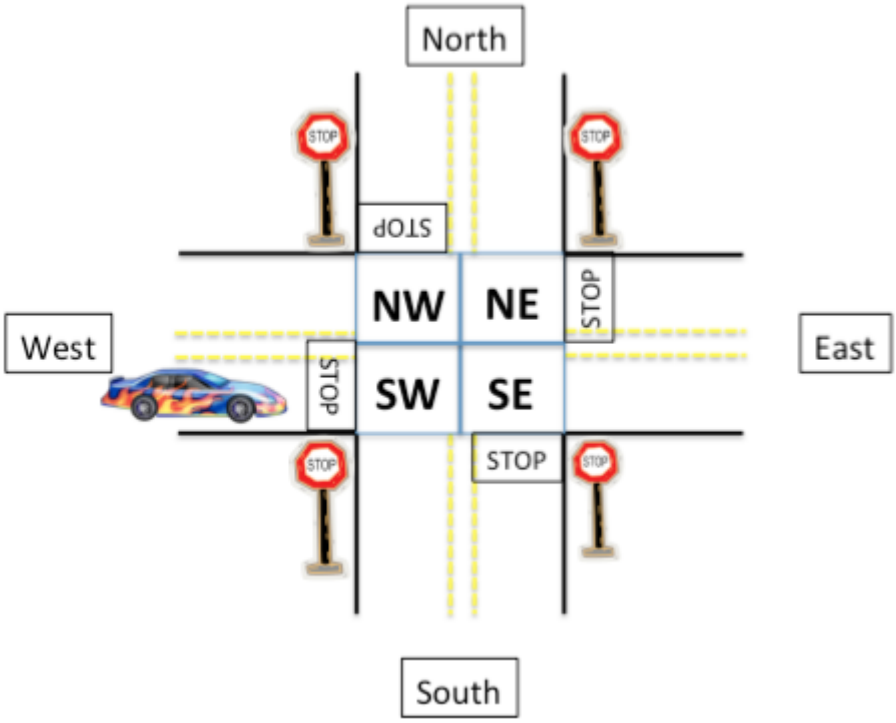
1. Write out the contents of what is printed to the screen, and explain what you see on the screen.
2. Write the contents of the text file, and explain what you see in the file.
3. If we were to use pthread `create()` instead of `fork()` would the file output change? Why or why not?

The code can be downloaded from the course web site:
`http://inst.eecs.berkeley.edu/~cs162/fa11/projects/p1/indiv.v1.tar`

1.2.2 Synchronization Primitives

Demonstrate that monitors and semaphores are all equivalent (i.e., solutions to the same types of synchronization problems can be implemented with each of them) by writing the pseudocode to implement semaphores using monitors and also implementing monitors using semaphores.

1.2.3 Directing Traffic



Synchronization problem: We will use locks to direct traffic. Cars come to the intersection, stop at the stop sign, and either continue in the direction they are driving, make a left turn, or make a right turn. Each quadrant of the intersection has a lock associated with it (NW, SW, NE, and SE).

Correctness constraints:

- No two cars should ever be heading in directions that cross. Keep in mind that a car making a left turn will be moving across the intersection (e.g., if the car in the above figure turns left, it will pass through the SW, SE, and NE quadrants of the intersection).
- A car's path should also never intersect with another car's path (i.e., a car moving EAST-to-SOUTH intersecting with a car travelling SOUTH-to-NORTH or NORTH-to-SOUTH).

We have provided you with **incomplete** code (`intersection.c`, `intersection.h`) that uses a pthreads lock for each quadrant and prints out the actions taken by a car. Before the action is taken (i.e., printed), *you must first acquire the necessary locks*. Read the man page documentation on POSIX pthreads locks and add the necessary code to correctly manage the locks so that there is correct operation and no deadlock. If the code exits successfully every time, the locking strategy was successful.

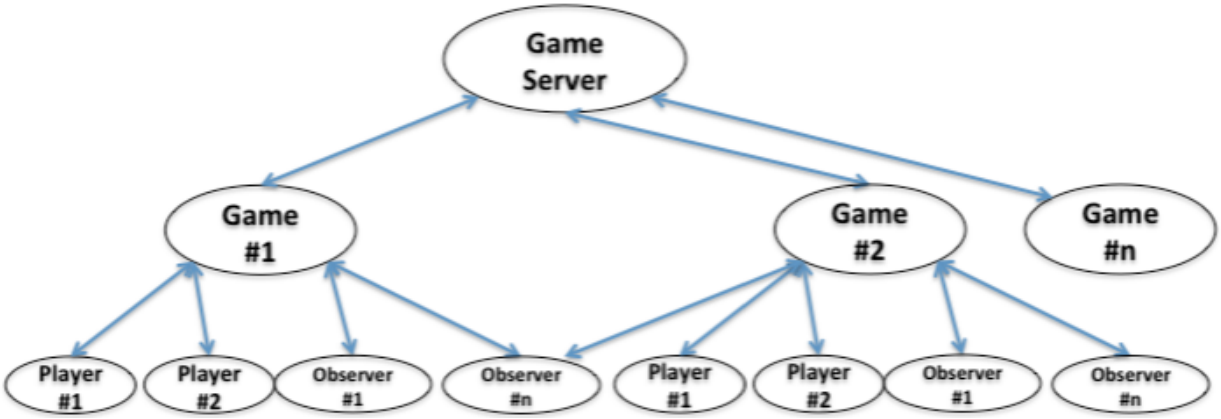
2 Group Portion: Thread-Safe Go Server

Now that you have gotten your feet wet with using some multi-programming primitives, we will start applying those concepts in the implementation of a concurrent thread-safe Go server. Thread safety means that the system should provide each concurrent user a consistent view of the system. The code written in this assignment will serve as the core logic for the machinery behind the distributed Go server that you will build in future projects.

2.1 Specification Overview

The game of Go is played by two players, taking turns placing stones on a board that is typically 19 by 19 squares in size. For a background on the game and its rules, we recommend you read the wikipedia Go page. In this project, you'll implement a single-process Go application. The key concepts to be modeled are Games, Players, Observers and the GameServer, but you'll also need to model data types such as the Board. One of the key challenges is to ensure concurrent safe access to the data. You are expected to use the synchronization primitives learned in class to ensure correct, consistent operation. We will test your synchronization mechanisms and how your

system responds under stress-testing with a heavy load: a large number of games and with many observers and players performing concurrent operations.



There will be two main parts to the project: implementing synchronization structures, and implementing the Go game server. The core functionality of the Go server is subject to various constraints, highlighted in this specification.

At a high level, the Go server should satisfy the following constraints:

1. There is a single game server.
2. There are multiple games, all managed by the game server.
3. Each game has two players, and any number of observers. Observers can enter and leave at any time.
4. The game is played by each player making moves in turn. A move consists of either placing a stone or passing. Moves have to be made within a specified time. The game instructs the players to make moves when it is their turn.
5. When a player places a stone, sometime other stones are captured. Players and Observers should be notified when this happens. The Game is responsible for determining which stones were captured.
6. It is illegal to place a stone on another stone. If this happens, or the player makes any other type of error (e.g., exceeding the per-turn time limit), the Game should notify the players and observers and the offending player loses the game.
7. When both players pass in a row, the game is over. The game should notify players and observers about this, along with the final score for each player.

Your design document should use the these requirements to guide its design. We require at least three types of threads - game server, game, and player/observer, but you may decide to introduce more. Please describe all the threads in your design, the placement of message buffers, and the synchronization mechanisms you use to allow the threads in your system to communicate safely.

Please keep in mind that all buffer sizes are finite and your design must handle the case where a message queue is full, a game has reached the maximum number of observers, etc. Your design document should include some discussion of how errors will be handled.

2.1.1 Synchronization

You will start by implementing your own synchronization primitives. You'll be using these implementations for synchronization throughout the rest of the project. Use of classes in `java.util.concurrent` or the `synchronized` keyword is not allowed. There are two exceptions to this rule: to implement your own synchronization primitives, you are allowed to use `java.util.concurrent.atomic.AtomicBoolean`. , and you're allowed to throw `java.util.concurrent.TimeoutException`. You'll notice that `AtomicBoolean` provides a method call `getAndSet()`, which is essentially equivalent to the `testAndSet()` atomic operation discussed in lecture.

Using `AtomicBoolean`, we ask you to implement four different primitives:
`SpinLock` - a simple lock using an efficient busy-wait loop implementation.
`Semaphore` - as discussed in lecture, with one addition: a P() method with a timeout, specified in milliseconds. If the timeout expires, you should throw a `TimeoutException`.
`Lock` - a lock that does not use busy-wait. It should support acquiring a lock with a timeout, throwing a `TimeoutException` if the lock cannot be acquired within the specified time.
`ThreadSafeQueue<E>` - a queue supporting `add()`, `get()`, and `getWithTimeout()`, with a fixed size specified at construction time.

You're allowed to use any of these primitives to help implement the others. We recommend that you implement them in the order listed above, but you may choose a different order. Hint: you can use `Thread.currentThread()` to get the currently running `Thread`, and you can use `Thread.interrupt()` to interrupt a `Thread` which is sleeping. You may assume that if `interrupt()` is called on a Thread while it is not sleeping, its next call to `sleep()` will be interrupted instantly.

For the operations that have timeouts, we require that you give the methods at least the provided amount of time to finish, but abort as soon as possible after the timeout has expired. You may take up to 20ms longer than the caller specified time to return.

2.1.2 Go Server

For each run, your program will be given a configuration file. Each line of the file will list either a player, an observer, or a game to be played. Lines are of the form

```
Player <PlayerClassName> <PlayerName>
```

`Observer <ObserverClassName> <ObserverName>`
or
`Game <GameName> <BlackPlayerName> <WhitePlayerName> <BoardSize> <TimeoutInMs> <Observers>`

where `<Observers>` is a list of 0 or more observers watching the game, `<BoardSize>` is a single integer in the range [3,19], and `<TimeoutInMs>` is a single unsigned integer representing the number of milliseconds each player in the game is allowed to make a move. The player lines will appear first in the file, followed by all observer lines, followed by all game lines.
Player names, Observer names and Game names will all be unique, but there may be multiple instances of any specific Player or Observer class. A sample configuration file, `sample.config`, is provided as part of the project.

Players, Observers and Games only know about each other in terms of their names. All messages that need to be passed between Games, Players and Observers will have to be routed through the GameServer. There is only one GameServer in the process, but there will be many Games, many Players, and many Observers.

All Games and Players are responsible for keeping track of the board state, independently. Games are responsible for telling all observers when the state of the board changes. Games also check when the game is over, which happens when both players pass in a row. We provide a Rules class which you may use to help with the mechanics of the game. At this time, we don't provide any players or observers, but later during the semester, we will make a number of machine players available for your use.

To facilitate automatic testing, we have provided some framework code for you to start with. It is available for download from the project page as a tar file. You are expected to implement all public methods in the provided files, as they will be called by the testing framework. You are also free to create any new classes that might be necessary or helpful for your design.

2.1.3 Game Rules and Scoring

We strongly recommend that you read through the Wikipedia page to familiarize yourself with the game and the basic rules. There are many flavors of go rules, in this project we use a simplified set which is appropriate for machine players and scoring.

At a high level, the game is played by two players placing stones in turn. Each location on the board is connected to its four neighbors, north, south, east, and west. A stone on the board is considered alive if it has a *liberty*, meaning at least one of its four neighbor locations is unoccupied. Blocks of stones of the same color can be *connected*, and the entire block of stones is alive if any of the stones has a liberty.

When it is a player's turn to move, he can choose to put a stone down in any unoccupied location, or to pass. If the player chooses to put a stone down, and the location is the last liberty of a block of stones belonging to the other player, those stones are *captured*. A captured stone counts as one point when scoring the game. If the location is the last liberty of a block of the player's own stones, those stones and the newly played stone are captured by the opponent. This is called *suicide*, and we'll consider it a legal move.

The game is over when both players pass in sequence. When that happens, the game is scored. Each captured stone counts as one point. In addition, each unoccupied location that is surrounded by stones from a single player, or situated between a player's stones and the edges of the board, counts as one point of *territory*. The final score is the sum of these two components. In addition, the white player gets ½ extra point, since black has the advantage of going first. As a result, no game can be a draw.
There is one final rule, called the "ko rule". This is meant to avoid repeated board positions and infinitely long games. In our version of the Ko rule, a move is illegal if it returns the board to exactly the same state as it was in after the last move the player made. You'll need to implement this rule.

The main point of this project is synchronization and threads, and you should spend the majority of your time on that. We have provided an implementation of the basic game rules in the `Rules` class. In particular, we provide two methods that will prove helpful. `Vector<Location> Rules.getCapturedStones(Board b, StoneColor c, Location l)`, which computes the set of stones that would be captured if the specified player adds a stone in the specified location, and `int countOwnedTerritory(Board b, StoneColor c)`, which counts the amount of territory surrounded by the specified player.

2.1.4 Players and Observers

In addition to your Go server, you will also need to implement some players and observers to participate in the games. You are welcome to create as many `Player` or `Observer` classes as you wish; however, we will require that you create a `HumanPlayer`, a `MachinePlayer`, and a `PrintingObserver`. Each of these classes should function as follows:

`HumanPlayer`: The `HumanPlayer` is a `Player` that prompts the user for input in order to make a move. Users should be able to choose to place a stone at a particular place on the board, or pass. Users must also be provided some visual representation of the board (a text-based representation is fine) when making a move.

`MachinePlayer`: The `MachinePlayer` will choose and make moves without any user input. You may choose to make the `MachinePlayer` as intelligent (or unintelligent) as you wish; we will not be testing the AI of your player.

`PrintingObserver`: The `PrintingObserver` should print to stdout all game events it receives in the order they were received.

2.2 Part A - Initial Design

For this part of the assignment, you will write a design document outlining your group's design decisions for the project. The specification and requirements are intentionally broad to allow your group flexibility in design choices. However, we do require a few mechanisms to be in place so that we can standardize the testing process across the code from different teams. The code we have included with the assignment consists of the following classes:

`GameServer`: A thread that manages all the running games and players. The `GameServer` should be the only object with direct access to `Game` and `Player` objects; any messages that passed between games and players must go through the server.

`Game`: Represents a single game of Go. There may be multiple games running at the same time, each with two players and any number of observers. The Game keeps track of the board state, and checks whether moves made by the players are legal. It also decides when the game is over. Each game should know the names of its players, but should not have direct access to `Player` objects.

`Board`: Represents the state of the board in a game.

`Location`: Represents a single location on the Go board.

`Observer`: A thread that can observe games. Observers may watch any number of games they wish and should be notified of all game events.

`Player`: A thread that is able to play in a Go game. The board state the player keeps must be private (i.e., not globally visible to other players, observers, or the game itself). Each player should know the name of the game it is playing on, but should not have direct access to the `Game` object.

`Launcher`: This class contains the `main()` method. It creates a single game server, loads a configuration file, creates and starts all necessary players, observers and games, and waits for the specified games to run and finish.

Please note that you should NOT modify any of the provided interfaces (method signatures, etc.). However, you are free to add additional classes, methods, and fields as you wish.

2.3 Part B - Implementation

Once your design has been approved by your TA you are ready to start implementing. Good luck!