Zürcher Hochschule für Angewandte Wissenschaften
Studiengang Informatik
Seminararbeit ZHAW / FS 2015

Zürcher Hochschule
für Angewandte Wissenschaften

Concurrent C Programming

Student: Alexander Gustafson

Lecturer: N. Schottelius

ZHAW 2015

Concurrent C Programming

Introduction

The purpose of this seminar project is to become familiar with writing concurrent programs in c. The main challenge is to manage shared resources that multiple clients can access simultaneously. During the period when one client is using a specific resource it must be guaranteed that another client cannot manipulate the resource. However, any attempt to protect an individual resource should not block the system as a whole.

Requirements

- There should be no global lock
- The server will save the names of the players
- Communication via TCP/IP
- Concurrency can be implemented through forking and shared memory

  or optionally with threads

  one thread or process per client connection

  the main thread can bind/listen/accept
- The focus is on the Server Implementation

  The Client is primarily for testing and "having fun"

  The Server will be tested by a Script from the Lecturer
- Locking, and simultaneous access is implemented by the Server
- Debugging messages from the Client/Server to stderr

Goals

The assigned task is to program a multi-player game. When at least two players have connected to the game server the game will begin. More players can join or leave the game at any time. Players compete to obtain cells in the games n x n grid, where n is equal to the number of players. The goal is for a player to obtain all the cell in the playing field.

Gameplay

- The Server starts and waits until more than 2 players have joined the game.
- As soon as 2 players have joined, the players may begin trying to obtain the field.
- New players may join or leave the game at any time
- The Server tests the status of the game field every y seconds, where 1 <= y <= 30

  If a player, at the time the status is checked, has obtained all the cells of a field, that player has won the game and the game is ended.

Communication Protokol

- Every command is ended with \n
- No command is longer than 256 characters, including \n
- Every player can only send 1 command and must wait for the response

Successful Registration

```
Client: HELLO\n
Server: SIZE n\n
```

Unsuccessful registration:

```
Client: HELLO\n
Server: NACK\n
-> Terminate Connection
```

Game Start

```
Server: START\n
Client: - (no response expected)
```

Obtain cell successful

A cell can be taken when no other player is currently in the process of take the same cell.

```
Client: TAKE X Y\n
Server: TAKEN\n
```

Obtain cell unsuccessful

If one of more other players are currently in the process of taking the same cell, all attempts except the first one will be unsuccessful

```
Client: TAKE X Y\n
Server: INUSE\n
```

Show Cell Status

```
Client: STATUS X Y\n
Server: Name-of-Player\n
```

End of Game

As soon as a player has obtained all the cells of the playing field he or she has won the game The server will respond to any other client's command by sending the win message.

```
Server: END Name-of-Player\n
Client: - ( client ends game )
```

Server Installation

Typing > make in the root directory of the project will start the makefile build process. The server and client console applications will be compiled and built in their respective folders

sem_concurrent_c/server/Builds/Linux/build/server
sem_concurrent_c/client/Builds/LinuxMakefile/build/client

To run the server start the server in the console. Type any character followed by enter to end the server. The default parameters that the server uses when starting are port = 65002 and size = 4. When starting the server you can pass these parameters as arguments to the server.

```
./server 65123 16
```

This will start the server listening on port 65123 and a playing field of 16 x 16 cells.

The default settings of the client and servers are matched so they will automatically connect if run from the same computer.

Test Installation

There are 3 test applications, one for testing TCP Handling, one for testing the game resources, and one that tests both parts together. In order to run the test applications you must download the JUCE C++ framework. The project build settings expect to find the JUCE library files locationed in the same directory where the project is located. If you have downloaded the project to you Documents folder, the JUCE directory should also be located in your Documents folder.

The test applications can be build by running make from within their respective /Builds/Linux/ subdirectories.

Project Implementation

Field Manager and a **TCP Server**. The Field Manager manages handling and access to the field resources. The necessary locks to prevent race conditions are implemented in the field manager. The TCP Server manages the connections to the clients, starts threads for each client. The TCP Server's thread handler function passes commands to the Field Manager via a set of function pointers which are registered before the TCP Server starts.

The decision to split the program into two components was made in oder to makes testing of each component easier. Each component can be tested independently of the other component. The Field Manager can be tested without the extra complexity of the socket handling. The TCP Server can be tested without the extra complexity of the lock and mutex handling.

The use of function pointers as the connection between the Field Manager and the TCP Server makes testing easier as well because other functions can be registered that have a predefined behavior that "mocks" the function of the other component in a controlled manner.

Field Manager

The requirements of the game state that the game field is n*n in size, and for the game to start a minimum of n/2 players must join the game. That means that a minimum of 4 players must join a game that is 8 x 8 in size. The requirements also state that players can join or leave the game at any time. The Field Manager module is responsible for managing the "game logic".

Field Manager Lifecycle

- Field Manager is initialized with the desired field dimension. All variables are set to their initial state. The memory for the field is allocated.

  ```
  initialize_field_manager(int size)
  ```

- Players can join the game at any time by calling join_game()

  ```
  int join_game()
  ```

- Players can request the status of a cell or request to take a cell.

  ```
  int get_cell_player(int x, int y);
  int take_cell(int x, int y, int player_id);
  ```

- A "referee" thread can check if the field has been completely acquired by any single player.

```
int is_there_a_winner();
```

- The field manager can be shut down.

```
void release_field_manager();
```

Field Initializsation

When the Field Manager is initialized, the size parameter is used to determine the size of the field and the minimum number of players required to start the game. The field is n x n, and the minimum number of players is n/2, where n is the given size.

The field itself is initialized as a `struct field`, with a pointer an array of nxn `struct cell` elements. Each `struct cell` holds a variable `int player_id` which stores the id of the current owner as well an a mutex which prevents the cell from being manipulated by two threads at the same time.

```
struct cell {
 int player_id;
 pthread_mutex_t cell_lock;
};

struct field {
 pthread_rwlock_t lock;
 struct cell* cells;
};
```

Cell and Field Data Structure

```
int initialize_field_manager(int size)
{
 join_countdown = size/2;
 field = malloc(sizeof(struct field));
 field->cells = NULL;
 if(pthread_rwlock_init(&field->lock, NULL) != 0)
 {
   perror("could not initialize field lock");
   return NULL;
 }

 dim = 0;
 delay = 0;
 set_size(size);
 player_count = 0;
 return 1;
}
```

Initialize Field

```
void set_size(int n)
{
  int new_count = n * n;
  int current_count = dim * dim;
  int diff = new_count - current_count;
  if (diff > 0) // increase field size
  {
    struct cell* new_cells = create_new_cells(diff);
    if(dim > 0){
      field->cells = realloc( \
                    field->cells, \
                    sizeof(struct cell) * new_count);
      memcpy(&field->cells[current_count], \
             new_cells, \
             sizeof(struct cell) * diff);
      free(new_cells);
    }else
    {
      field->cells = new_cells;
    }
    new_cells = NULL;
  }
  else if (diff < 0) // remove cells
  {
    int i;
    for (i = 0; i < diff; i++)
    {
      realloc(field->cells, \
              sizeof(struct cell) * new_count);
    }
  }
  dim = n;
  min_players = dim / 2;
}
```

Initialize Cells

Clients can join a game

The game is initialized with size n. The playing field is created as an nxn "grid" of cells. Then players can register to join the game. The game will not start until n/2 players have joined, at which point the server will send the "START\n" message to all players that are waiting to join the game.

Each client is managed in the server by a specific thread. The client threads (Implemented in the TCP Manager) can register for a game concurrently. The mechanism that counts the number of registered clients and allows the game to begin is a critical section that must be managed carefully in order to not have race conditions or deadlocks.

Originally I tried to implement the joining mechanism using pthread_barrier_wait. The idea was that the pthread_barrier_t mutex would be initialized with count n/2. Every attempt to join decrements the count until it reaches 0. At that moment all waiting threads are able to proceed. The problem is that the pthread_barrier_t resets to the initial count value after reaching 0. New players attempting to join the game would have to wait until

an additional n/2 new players attempt to join. Also pthread_barrier_t is not available natively on OSX.

Instead the joining process is managed with a conditional mutex.

```
pthread_cond_t join_ready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t join_lock = PTHREAD_MUTEX_INITIALIZER;

int join_game()
{
    pthread_mutex_lock(&join_lock);
    if (--join_countdown > 0)
    {
        pthread_cond_wait(&join_ready, &join_lock);
    }
    pthread_mutex_unlock(&join_lock);
    pthread_cond_signal(&join_ready);
    return 0;
}
```

Join Game Function

The mutex_lock make the if-test, decrement, and cond_wait an atomic process. Only one single thread can be active after the first lock, until it reaches and waits at the pthread_cond_wait() function, or until it reaches the pthread_mutex_unlock() function. After the unlock it will signal any other threads waiting at pthread_cond_wait() that they may resume.

Requesting to "Take" a cell

By calling take_cell(int x, int y, int player_id) a player makes a request to take a cell. The field_manager converts the xy-coordinated to the respective array index. With the index it retreives the cell and tests the cell's mutex to see if it is currently in use by another thread. If it's free then the mutex is attained, the cell's player_id variable is set to the thread's client's id, then the mutex is released. If the cell's mutex is held by another cell then the function returns EBUSY.

```
int coords_to_index(int x, int y)
{
    if (x > dim || y > dim || x < 0 || y < 0)
    {
        return - 2; //out of bounds
    }
    int max = MAX(x, y) + 1;
    int min = MIN(x, y) + 1;
    int diff = max - min;
    int cell_index = (max * max) - (2*diff);
    if (y > x)
        cell_index++;
    return cell_index - 1;
}

struct cell* get_cell(int x, int y)
{
    if (coords_to_index(x, y) < 0)
    {
        return NULL;
    }
    return &field->cells[coords_to_index(x, y)];
}

int request_cell_lock(int x, int y)
{
    if (get_cell(x, y) == NULL) {
        return -1;
    }
    return pthread_mutex_trylock(
            &get_cell(x, y)->lock
        );
}

void release_cell_lock(int x, int y)
{
    pthread_mutex_unlock(
            &get_cell(x, y)->lock
        );
}

int take_cell(int x, int y, int player_id)
{
    int respsonse;
    request_global_read();

    int result = request_cell_lock(x, y);
    if (result == 0)
    {
        get_cell(x, y)->player_id = player_id;
        sleep(delay);
        release_cell_lock(x, y);
        respsonse = 1;
    }else{
        respsonse= result;
    }
    release_global_read();
    return respsonse;
}
```

Take Cell Process

Checking for a Winner

The Field Manager has a function which can be called by a "referee" thread that will test if there is currently a winner. This is the only function that will temporarily block all activity on a global level. While the "referee" is checking all the cells, no other threads can take a cell. In addition,

the "referee" must wait for all pending take requests to finish before it can block all cells. This is implemented using a pthread_rwlock_t in the field struct.

When player threads request the read permissions of the field's pthread_rwlock_t variable. Only if the "referee" is currently holding the write key of the pthread_rwlock_t will the take request be blocked.

```
int is_there_a_winner()
{
    request_global_lock();
    if(dim < 2)
    {
        release_global_lock();
        return -1;
    }
    struct cell cell = field->cells[0];
    int winning_player = cell.player_id;
    int i;
    for (i = 1; i < (dim * dim); i++) {
        struct cell next_cell = field->cells[i];
        if (next_cell.player_id == winning_player) {

        }else{
            release_global_lock();
            return -1;
        }
    }
    release_global_lock();
    return winning_player;
}
```

Test for Winner Process

TCP Server

The TCP Server manages the communication with the client. It is multi-threaded, with one thread per client, one referee thread, and one main server thread. It interacts with the Field Manager via function pointers which are registered at startup.

TCP Server Lifecycle

- Before starting the TCP Server the Field Manager functions should be registered with the TCP Server.

```
void register_callbacks(
    join_cb jcb,
    leave_cb lcb,
    take_cb tcb,
    size_cb scb,
    status_cb stcb,
    winner_cb wcb
);
```

- The server is started with a port number.

```
int startserver(int port);
```

- In the startserver() funtion a referee thread is started which periodically calls the winner_cb callback.

```
void *run_referee(void *arg);
```

- For each connecting client a thread is started which continually listens for client commands on the client socket.

```
void *handle_tcp_client(void *arg)
```

- When the referee thread has found that a single client has obtained all the cells on a field it sets the winner_established variable to 1, indicating that no further commands should be processed.

```
winner_established = 1;
```

Registering Field Manager Function Pointers

All communication between the TCP Server and the Field Manager is handled through function pointers. The TCP Server calls functions on the Field Manager, the return values indicate whether the method succeeded.

The decision to use function pointers was made in order to make it possible to easily swap out the Field Manager's functions with other mocked functions for testing. Without this possibility it would be more difficult to do real multi-threaded testing. By swapping the real functions with mock functions it is possible to test the TCP Manager in completely entcoupled form the Field Manager.

```
typedef int (*join_cb)(void);
typedef int (*leave_cb)(void);
typedef int (*take_cb)(int, int, int);
typedef int (*size_cb)();
typedef int (*status_cb)(int, int);
typedef int (*winner_cb)();
```

Callback Definition

```
join_cb join_callback;
leave_cb leave_callback;
take_cb take_callback;
size_cb size_callback;
status_cb status_callback;
winner_cb winner_callback;

void register_callbacks(join_cb jcb, leave_cb lcb,
                        take_cb tcb, size_cb scb,
                        status_cb stcb, winner_cb wcb
                        )
{
    join_callback = jcb;
    leave_callback = lcb;
    take_callback = tcb;
    size_callback = scb;
    status_callback = stcb;
    winner_callback = wcb;
}
```

Callback Registration

Testing

While developing the modules, tests were developed at the same time in a pseudo test driven development method. Pseudo, because the tests were not strictly written first and not every function was explicitly tested. Instead, threadable client objects were created which could be given a playbook of instructions.

The clients are started simultaneously, at which point they begin to follow their instructions as defined in the playbook.

The test applications are console based applications programmed using the JUCE C++ library.n The JUCE library includes a simple Unit Testing framework. A unit test is made by creating an object that inherits from UnitTest and overriding the `void runTest();` method.

The TCPServerTests object has 3 tests that are called from the `void runTest();` method.

```
void TCPServerTests::runTest()
{
    FirstTest();
    ServerTest();
    Thread::sleep(5);
    ConnectionTest();
    Thread::sleep(15);
    ThreePlayerTest();
}
```

TCPServerTests runTest()

The sleep directives are adding so that the TCP Server has time to shutdown all it's threads properly and close it's sockets before the next test begins.

The FirstTest() function simply tests that the testing environment is running properly and does not even start the TCP Server. The other tests will always register callbacks with the TCP Server, start the server, do some tests, then shutdown the server.

```
void TCPServerTests::ConnectionTest()
{
    beginTest("Connection Test");
    register_callbacks(
        always_allow_join,
        always_allow_leave,
        always_allow_take,
        return_4,
        always_status_1,
        never_winner
    );
    startserver(65002);
    Thread::sleep(20);

    ScopedPointer client1 = new Client();
    client1->setName("Frank");
    add_instruction(client1, "sleep", 2);
    add_instruction(client1, "join", "");
    add_instruction(client1, "sleep", 2);

    NamedValueSet* instruction =
        add_instruction(client1, "take", "");
    instruction->set("x", 0);
    instruction->set("y", 1);
    instruction->set("name", "Frank");
    instruction =
        add_instruction(client1, "status", "");
    instruction->set("x", 0);
    instruction->set("y", 1);
    add_instruction(client1, "sleep", 2);

    pool.addJob(client1, false);
    while(pool.getNumJobs() > 0 )
    {
        Thread::sleep(5);
    }
    stopserver();
    while( server_running() > 0)
    {
        Thread::sleep(10);
    }
}
```

TCPServerTests ConnectionTest()

This ConnectTest() method begins by registering a set of mock functions that simulate behavior of the Field Manager. These can be swapped out quickly with other functions to test different outcomes. `always_allow_take` for instance can be replaced with `every_second_take`. The corresponding TCP Server responses to the client are written to the stderr console.

The server is started on port 65002 before the client is configured. Simple instructions like join or sleep can be passed to the client in one line. Instructions

that require more parameters like take and status can be passed to the instruction object which has already been registered with the client. In the test above, the client is given the instruction to sleep for 2 seconds, the join the game, then sleep again for 2 seconds. The next instruction given to the client is to request a cell using the take command. It will attept to take the cell with coordinates x:0, y:1, with the name "Frank".

When all the instructions are passed to a client object, the client itself is handed over to a thread pool manager. The main test thread periodically checks the thread pool to see if jobs are still running. When all the jobs are finished the server is instructed to shutdown. The test function completes when the server_running() function returns 0. After this is still might take a moment for the operating system to completely close all sockets.

The Field Manager tests are also run using client objects using instructions, managed by thread pools. They do not require any socket handling, but it must be possible to test the mutex blocking properly. The Field Manager has the option to add an artificial delay to all it's methods. By giving two client objects the same take instructions and setting the delay to over a second, it's possible carefully test the behavior or the Field Manager when take requests compete for the same resources.

The project also includes a consolidated_tests suite which tests the behavior of the TCP Manager and the Field Manager together. It includes the following tests:

```
CreateAndReleaseTests();
TwoPlayersSimple();
TwoPlayersWithDelay();
ThreePlayersWithDelay();
```

Consolidated Test Methods

Conclusion

This is a project with a lot of moving pieces; sockets, threads, mutexes, memory management. In order to really learn how the pieces fit together it's important to be able to understand the real behavior and quirks of all the pieces. By splitting responsibilities into separate modules and simultaneously developing the tests for them was very helpful. The testing part of the code grew to be as large as the actual implementation, but during development it also became playground to quickly try out different ideas. By having tests in place and knowing that previous portions of code are working properly, one can have more confidence trying out other ideas, without the fear breaking things.

I originally wanted to try the testing framework cmocka (https://cmocka.org/) but it required using CMAKE and I lost too much time just trying to figure out how to integrate cmocka into a project.

I decided to use the JUCE library unit testing features. I already had some experience using JUCE and knew it included several tools that make integrating it into a project very easy. Although the unit testing features of JUCE are very simple, JUCE comes with many other features that made building the test environment easy and fun. Especially the Thread Pool tools and the NamedValueSet made it easy to quickly put together very advanced automated client objects.