

# ESTR 3106 – Principles of Programming Languages – Spring 2019

## Assignment 1 Specification

### Concurrent Programming (Administrator-and-Worker)

Deadline: Mar 22, 2019 (Friday) 23:59

## 1 Introduction

In this assignment, you need to implement the real-time strategy game *Tower Defense and Attack*, which is a simplified version of *Dominance* (<https://github.com/YangVincent/Domination>). You need to implement the game using the C programming language. For concurrency control, you should use the Send/Receive/Reply (SRR) messaging primitives (first popularized in the commercial real-time operating system, QNX) provided by the library, called Synchronous Interprocess Messaging Project for LINUX (SIMPL) version 3.3.8. You should use the ncurses library to implement the textual user interface.

## 2 Tower Defense and Attack

*Tower Defense and Attack* is a two-player real-time strategy game played on a 20 by 100 arena. Each of the two players, representing two forces **RED** and **BLUE**, occupies half (20 by 50) of the arena as **territory**. The players need to place/build **units** at strategic spots in their respective territories to defend their respective bases and attack each other. Each player has to manage his resources named **Kyanite**, use resources to build units in the arena, and strategically send robots to attack the opponent.

In the beginning, each player has 1000 **health points (HP)**, 500 Kyanites and no units in the arena. The player will lose the game once his HP is reduced to zero, and the winner of the game is the last survivor. A player's HP is reduced when the player's **baseline**, being the left edge or the right edge of the arena, is reached by the opponent's units. Placing units onto the arena costs a player Kyanites. Different units have different costs, maximum HP, and their own functionality. **Buildings** and **robots** are two types of units in this game. Buildings are unmoveable units. Each player can either build a buildings on the empty territories or remove an existing buildings from the territories. The types of buildings include:

- A **Mine** generates resources. After being placed in the arena, a mine can generate 10 Kyanites for the player every second. The cost to build a mine is 200 Kyanites, and the maximum HP of mine is 200.
- A **Wall** can block friendly (in the same team) or hostile (in the opposite team) robots. The cost to build a wall is 20 Kyanites, and its maximum HP is 100.

Robots are moveable units which can cause damage to both hostile units and the opponent's HP. After being placed, it will take action (either move forward towards the opponent's baseline or attack hostile units) after a certain time interval, which is called **movement interval**. Different types of robots have different movement interval. The robots can **attack** hostile units, causing at most its **melee damage (MD)** to the unit under attack. When the robot reaches the opponent's baseline, it will cause fixed **base damage (BD)** to the opponent's HP. The rule of the robots' movement will be explained later. Here we state the parameters for two types of robots as follows.

- A **Lancer** is a robot costing 20 Kyanites, and its maximum HP is 50. The movement interval of a lancer is 0.2 seconds, and its MD and BD are 15 and 15 respectively.
- A **Hoplite** is yet another type of robot. Compared with a lancer, a hoplite has higher HP and BD but lower MD. The cost of building a hoplite is 30 Kyanites, and its maximum HP is 100. The movement interval is 0.3 seconds, and its MD and BD are 10 and 20 respectively.

All units belong to either RED or BLUE. When a unit is built, its HP is equal to its maximum HP. Once a unit's HP is reduced to zero, it will be eliminated from the arena. Each player could at most place *MAX\_UNIT* units on the arena.

Throughout the game, each player will have a **marker** indicating the current desired point for the player to build units. The RED (BLUE) player can use  $w, s, a$  and  $d$  ( $\uparrow, \downarrow, \leftarrow$  and  $\rightarrow$ ) to change the position of the marker by one grid cell in the up, down, left and right directions respectively. At the own marker position, a player can press appropriate keys to place units or remove the player's own units, if possible. The RED (BLUE) player can press  $f$  or  $g$  ( $j$  and  $k$ ) to place a lancer and a hoplite respectively in the desired point. As for buildings, pressing  $r$  or  $t$  ( $u$  or  $i$ ) allows the RED (BLUE) player to place a mine or a wall respectively. Otherwise, the mine or the wall occupying the position of the marker will be removed. There are several rules that we should obey throughout this game:

1. All units should be placed within the arena, and the RED (BLUE) can only place units on their own territories. Some units may be represented by multiple characters (as specified in 3.4), and every character should be within the arena.
2. All units in the arena cannot overlap with each other.
3. The buildings can be **removed** by a player, and half of its costs will be returned to the player.
4. For every movement interval, a robot can move forward if the cell in front is empty.
5. If a robot is blocked by a hostile unit, it will cause random damage ranging from 1 to its MD to the blocking unit.
6. If a robot is blocked by a friendly unit, the robot will wait until the friendly unit moves forward or is removed.
7. Once the robot reaches the opponent's baseline, it disappears and the opponent's HP is reduced by the robot's base damage.

In this assignment, you need to write programs to implement the game with **keyboard input**. You should use the **Administrator-and-Worker** model. In the following, we provide the process design in Figure 1. You need to implement these processes in the design using the SRR messaging primitives provided by the SIMPL library. Note that in **assignment 2**, you need to write an interpreter of a **specification language** to implement an AI to play this game.

## 3 Assignment Details

### 3.1 Process Design

There are *seven* types of processes in our design.

- Game\_Admin maintains the rules of the game and the current state of the game.
- Display\_Admin controls the screen output sequence.
- Timer sleeps for a certain amount of time.
- Courier relays the message between administration processes.
- Painter formats and paints the output to the screen.
- Input\_Admin controls the human controlled RED or BLUE players.
- Keyboard gets the human players' input from the keyboard.

**You need to write a program for each of the seven major types of process: Game\_Admin, Display\_Admin, Timer, Painter, Input\_Admin, Keyboard and Courier.**

In the implementation, players RED and BLUE are human controlled. We use Keyboard to get all the keys pressed and send to Input\_Admin. We use Courier 0 and Courier 1 to relay the messages from Input\_Admin to Game\_Admin for the first and the second human player respectively.

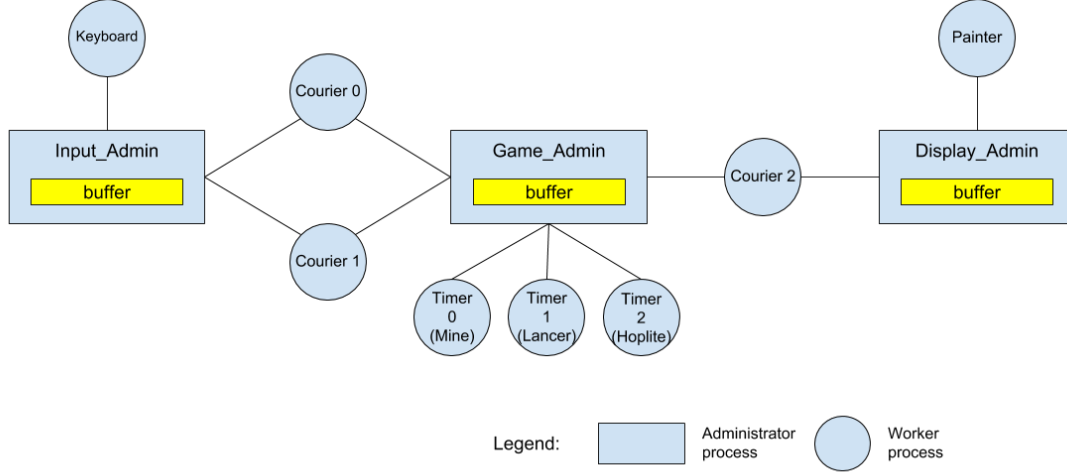


Figure 1: The Administrator-and-Worker Architecture for the Game

There are three timer workers denoted by Timer 0 to Timer 2. Timer 0 is used to maintain the time interval for Kyanite generation. Timer 1 and Timer 2 are used to maintain the time interval for moving all Lancers and Hoplites respectively in the arena. Timer processes should take an argument to distinguish their roles in the game.

All the screen output is handled by the Painter process, which keeps on getting paint jobs for Display\_Admin. The responsibility of Courier 2 is to relay the paint requests from Game\_Admin to Display\_Admin.

Before the game starts, we have to carry out the registration procedure. The first process to start is Game\_Admin. Next, Timer and Courier 2 send registration requests to Game\_Admin. Then, Game\_Admin replies immediately whether the registration is successful or not. Registration fails if the Game\_Admin already accepts enough number of processes. Courier 0 and Courier 1 have to register with Input\_Admin first. Then Courier 0 and Courier 1 starts to relay the registration messages for human players from Input\_Admin to Game\_Admin. On starting the game, Game\_Admin asks Timer workers to start the sleep job and replies the game start signal to human players.

### 3.2 Message types and Format

Twenty message types are available for process communications:

```
REGISTER_CYCLE, INIT, FAIL, CYCLE_READY, START, MOVE, UPDATE, END,
REGISTER_TIMER, TIMER_READY, SLEEP,
REGISTER_HUMAN, HUMAN_READY, HUMAN_MOVE,
REGISTER_COURIER, COURIER_READY,
DISPLAY_ARENA, OKAY, PAINTER_READY, PAINT.
```

For simplicity, we use a single message format for all the communications.

```
/* Type of Units */
typedef enum {
    MINE, HOPLITE, LANCER, WALL,
} UNIT_TYPE;

/* Type of Timers */
typedef enum {
    LANCER_TIMER, HOPLITE_TIMER, MINE_TIMER
} TIMER_TYPE;

/* Force */
typedef enum {
    RED, BLUE
```

```

} FORCE;

/* Action */
typedef enum {
    NOACTION, MOVEEAST, MOVESOUTH, MOVEWEST, MOVENORTH,
    PLACELANCER, PLACEHOPLITE, UPDATEMINE, UPDATEWALL
} ACTION;

/* Coordinate */
typedef struct {
    int x,y;
} COORDINATE;

/* Player */
typedef struct {
    COORDINATE pos;
    FORCE force;
    int health;
    int resource;
    int unit_no;
} PLAYER;

/* Unit */
typedef struct {
    int active;
    int highlight;
    COORDINATE pos;
    FORCE force;
    UNIT_TYPE unit_type;
    int health;
} UNIT;

/* Arena */
typedef struct {
    UNIT units[2*MAX_UNIT];
    PLAYER players[2];
} ARENA;

/* Message Format */
typedef struct {
    MESSAGE_TYPE type;
    TIMER_TYPE timer_type;
    int interval;
    ACTION act;
    ARENA arena;
    int key;
    int humanId;
} MESSAGE;

```

The message types and message formats are given in the header file “message.h”, which must be included in your code. **You are not allowed to have any additional message type or make any modification to the message format.**

The administrators and workers can use the MESSAGE.type field to differentiate messages, and can store/retrieve data in the necessary fields in the MESSAGE structure with respect to the type of the messages. For example, when Game\_Admin sends an INIT message to a Timer, Game\_Admin only needs to specify the type, timer\_type and interval fields in the MESSAGE structure. Thus, when a Timer receives a message, it can immediately sleep for interval microseconds. We list in

appendix A the possible messages between the different processes and the fields used for each message. **Message field usage must be strictly followed.**

### 3.3 Program Libraries

The concept of Administrator-and-Worker can be realized by the following primitives in the SIMPL library:

```
int name_attach(char *processName, void (*exitFunc)());
int name_detach(void);
int name_locate(char *protocolName:hostName:processName);

int Send(int fd, void *out, void *in, unsigned outSize, unsigned inSize);
int Receive(char **ptr, void *inArea, unsigned maxBytes);
int Reply(char *ptr, void *outArea, unsigned size);
```

The API details can be found in the file `$$SIMPL_HOME/docs/simpl-function.synopsis` where `$$SIMPL_HOME` is the installation path of the `SIMPL` library. You can download the pre-compiled binaries of the `SIMPL` library as a gzipped tar package from the course webpage. The details of `SIMPL` project and the sources of the `SIMPL` library are available at:

<http://www.icanprogram.com/simpl/>

Description and sample programs on using the NCURSES library are available at:

<http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/>

### 3.4 Output

For simplicity, we use the textual user interface based on the reference output format (using the `ncurses` library) as depicted in Figure 2.

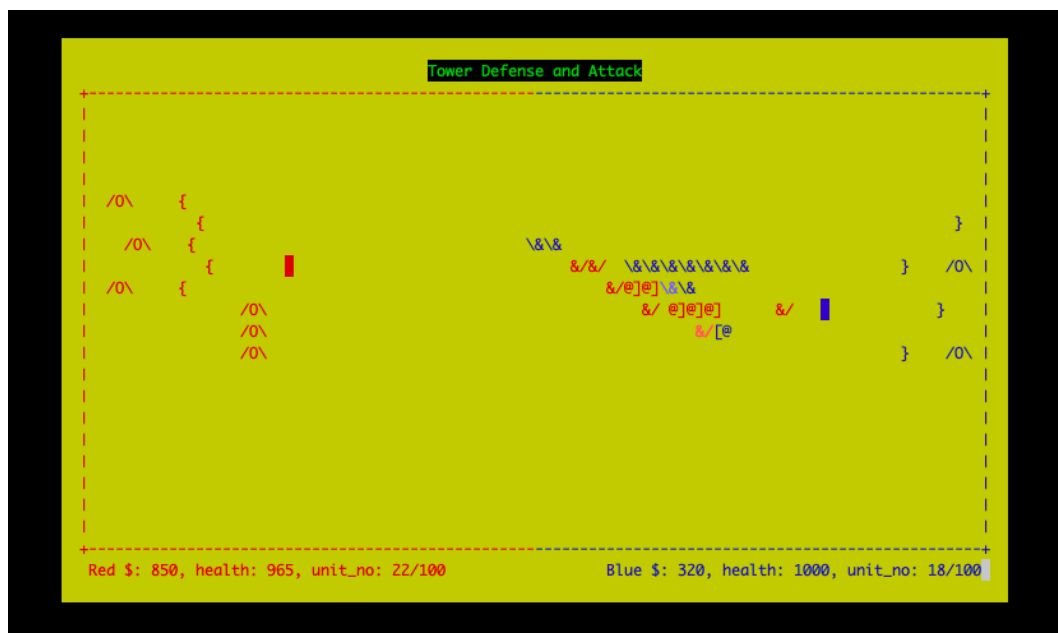




Figure 2: Sample Screen Output

The reference output format is the MINIMUM requirement. The arena is a rectangle whose size is 100 columns by 20 rows. The marker of each player is represented by a solid block of either  or . The mine is denoted by three characters “/0\” forming a 1 by 3 rectangle. Other units

will have different representations due to their forces. For example, the wall of red is “{”, while it is “}” for blue. Similarly, lancers are denoted as “&/” and “\&”, and hoplites are denoted as “@]” and “[@”. The upper part of the screen shows the name of this game, “Tower Defense and Attack”. The lower part of the screen displays the state of players including their HP, Kyanites and unit number. When the game ends, it will display the winner on the final screen shot. Note that some units may be represented by multiple characters. The position of mine, lancer and hoplite is the position of the character “0”, “&” and “@” respectively.

You are free to improve the reference output format (like using different colors). **Such improvement must be only in the Painter process and must obey strictly the messaging interface.** Also, the following basic components must be included in each screen shot: the boundary of the arena, the locations of different units, the state of players and the winner on the final screen shot. In addition, the screen should be updated after every change of the mouse and cheese. Of course, the output has to reflect the real situation of the arena by respecting the order of the events.

### 3.5 Other Requirements

1. You should test the correctness of your implementation by checking its compatibility with our components available on the course homepage. Your submission will be graded component by component by plugging your components into our implementation.
2. The process starting sequence is critical. For example, the Timer process cannot be started earlier than Game\_Admin. One possible process starting sequence is: Game\_Admin → Display\_Admin → Painter → Courier 2 → Timer 0 to Timer 2 → Input\_Admin → Courier 0 to Courier 1 → Keyboard. You should control the starting sequence using the shell script.
3. The processing time of each painting task for the Painter process can be relatively long. The Display\_Admin process should maintain a FIFO buffer to store the states of the arena that cannot be handled in time by the Painter process.
4. Also, Game\_Admin should maintain another FIFO buffer to store the states of the arena, since it is possible that when Game\_Admin needs to send a display message, Courier \* is not available yet. Analogously, the same requirement applies to Input\_Admin and Courier 0 to Courier 2.
5. Upon the game end, an ending message should be sent to each active process so that you do not need to kill them manually after the game.
6. **NO PLAGIARISM!** You are free to design your own algorithm and code your own implementation, but you should not “steal” or “borrow” code from your classmates. If you use an algorithm or code snippet that is publicly available or use codes from your classmates or friends, be sure to cite it in the comments of your program. Failure to comply will be considered as plagiarism.

## 4 Submission Guidelines

Please read the guidelines CAREFULLY. If you fail to meet the deadline because of submission problem on your side, marks will still be deducted. So please start your work early!

1. In the following, suppose  
your name is *Choi Chiu Wo*  
your student ID is *02345678*  
your username is *cwchoi*, and  
your email address is *cwchoi@cse.cuhk.edu.hk*
2. In your source files, insert the following header.

```

/* -----
* ESTR 3106 - Assignment 1
* Name : Choi Chiu Wo
* Student ID : 02345678
* Email : cwchoi@cse.cuhk.edu.hk
*
* Failure/Success
* -----*/

```

3. **The following file naming convention must be followed strictly.** For this assignment, use the filenames `game_admin.c`, `display_admin.c`, `courier.c`, `painter.c`, `timer.c`, and `mouse.c`, `input_admin.c` and `keyboard.c`. You should also provide a `makefile` and a `run` shell script. All filenames should be in lowercase.
4. You can submit along with the tarball a README file if you believe there is any special information we need to know about the compilation and execution of your program. In that case, you should email `cwchoi@cse.cuhk.edu.hk` (Choi Chiu Wo) about it. If we do not receive your email about it, the README file will be ignored. Do not write a README unless you have a compelling reason to do so.
5. Make sure you compile and run the program without any problem. Otherwise, mark it **Failure** in the header of the source file.
6. Tar your source files to `username.tar` by

```
tar cvf cwchoi.tar [source files] [shell script] [makefile]...
```
7. Gzip it to `username.tar.gz` by

```
gzip cwchoi.tar
```
8. Uencode the gzipped file and send it to the course account with the email title “EHW1 *student\_ID your\_name*”, by

```
uencode cwchoi.tar.gz cwchoi.tar.gz \
| mailx -s "EHW1 02345678 Choi Chiu Wo" csci3180@cse.cuhk.edu.hk
```

The backslash \ is for line breaking, i.e., if you type \, you should immediately press enter before continuing your command. Note that you can also typed it as *one* single line *without* the backslash. Ignore the warning message:

```
uencode: ISO8859-1 to 646 conversion: Invalid argument,
if present.
```
9. Please submit your assignment using your Unix accounts. For non-CSE students, please try it with your departments’ Unix account FIRST. If it fails, you can email your submission directly to `cwchoi@cse.cuhk.edu.hk`. CSE students who do not submit to `csci3180@cse.cuhk.edu.hk` will have their marks deducted.
10. An acknowledgment email will be sent to you if your assignment is received. Please check that your receipt contains your submission as an attachment. If you cannot find your submission attached in the receipt, you probably *cannot* submit successfully. DO NOT delete the acknowledgment email.

For non-CSE students who submit to `cwchoi@cse.cuhk.edu.hk` directly, we will try to provide you with an acknowledgment as soon as we check the emails. If you submit to `csci3180@cse.cuhk.edu.hk` and do not receive an acknowledgment email 10 minutes after your submission, please contact the tutors. DO NOT resubmit immediately if you do not get the email.
11. You can check your submission status at

<http://www.cse.cuhk.edu.hk/~csci3180/submit/EHW1.txt> and EHW1.txt

12. You can resubmit twice. That makes the total number of submission possible to 3. But we will only grade the latest submission. Extra resubmissions will be subject to mark penalties.
13. The following late penalty scheme for assignments submission will be adopted:
  - Submitted within 24 hrs after deadline: 70% of original marks
  - Submitted within 24-48 hrs after deadline: 40% of original marks
  - Submitted 48 hrs after deadline: 0 mark

## A Message Between Processes

### Messages between Game\_Admin and Timer 0...Timer 2

REGISTER\_TIMER - Timer registers to Game\_Admin. (Field: type)  
 INIT - Game\_Admin confirms Timer registration. (Fields: type, timer\_type)  
 FAIL - Game\_Admin fails Timer registration. (Field: type)  
 TIMER\_READY - Timer tells Game\_Admin that it is ready. (Fields: type, timer\_type)  
 SLEEP - Game\_Admin asks Timer to sleep for a certain amount of time. (Fields: type, timer\_type, interval)  
 END - Game\_Admin sends an ending message to Timer. (Field: type)

### Messages between Game\_Admin and Courier 2

REGISTER\_COURIER - Courier 2 registers to Game\_Admin. (Field: type)  
 INIT - Game\_Admin confirms Courier 2 registration. (Fields: type)  
 FAIL - Game\_Admin fails Courier 2 registration. (Field: type)  
 COURIER\_READY - Courier 2 notifies Game\_Admin that it is ready. (Fields: type)  
 DISPLAY\_ARENA - Game\_Admin asks Courier 2 to carry a display message. (Fields: type, arena)  
 OKAY - Courier 2 notifies Game\_Admin that the display message is delivered. (Fields: type)  
 END - Game\_Admin asks Courier 2 to deliver an ending message, and asks Courier 2 to terminate itself after delivering the message. (Fields: type, humanId, arena) (Here playerId denotes the winner of the game)

### Messages between Display\_Admin and Courier 2

DISPLAY\_ARENA - Courier 2 notifies Display\_Admin that there is a new display message. (Fields: type, arena)  
 OKAY - Acknowledgement from Display\_Admin. (Field: type)  
 END - Courier 2 delivers an ending message to Display\_Admin

### Messages between Display\_Admin and Painter

PAINTER\_READY - Painter notifies Display\_Admin that it is ready to paint. (Field: type)  
 PAINT - Display\_Admin asks Painter to paint the current game status. (Fields: type, arena)  
 END - Display\_Admin asks Painter to paint the final screen, announce the winner, and terminate itself. (Fields: type, humanId, arena)

### Messages between Game\_Admin and Courier 0 to Courier 1

REGISTER\_HUMAN - Courier notifies Game\_Admin of human player registration. (Field: type)  
 INIT - Game\_Admin confirms the human player registration. (Fields: type, humanId)  
 FAIL - Game\_Admin fails the registration if all human players are already registered. (Field: type)  
 HUMAN\_READY - Courier notifies Game\_Admin that the human player is ready. (Fields: type, humanId)  
 START - Game\_Admin declares the game start. (Fields: type, arena)  
 HUMAN\_MOVE - Courier notifies Game\_Admin of human player's request to take actions. (Fields: type, humanId, act)  
 UPDATE - Game\_Admin sends the current game status to Courier. (Fields: type, arena)  
 END - Game\_Admin got a winner and terminates human player. (Fields: type, humanId)

### Messages between Input\_Admin and Courier 0 to Courier 1



REGISTER\_COURIER - Courier registers to the Input\_Admin. (Field: type)  
 INIT - Input\_Admin confirms the registration. (Fields: type, humanId)  
 FAIL - Input\_Admin fails the courier registration if all couriers are already present. (Field: type)  
 COURIER\_READY - Courier notifies Input\_Admin that it is ready. (Fields: type, humanId)  
 REGISTER\_HUMAN - Input\_Admin asks the courier to carry a registration message. (Field: type)  
 INIT - Courier notifies Input\_Admin that the registration is successful. (Fields: type, humanId)  
 FAIL - Courier notifies Input\_Admin that the registration failed. (Fields: type, humanId)  
 HUMAN\_READY - Input\_Admin asks Courier to carry a ready signal for the human player. (Fields: type, humanId)  
 START - Courier notifies Input\_Admin that the game start. (Fields: type, humanId, arena)  
 HUMAN\_MOVES - Input\_Admin asks Courier to carry a human player request to take action. (Fields: type, humanId, act)  
 UPDATE - Courier notifies Input\_Admin of the current game status. (Fields: type, humanId, arena)  
 END - Courier notifies Input\_Admin that the game ended. (Fields: type, humanId)

#### Messages between Input\_Admin and Keyboard

REGISTER\_KEYBOARD - Keyboard registers to Input\_Admin. (Field: type)  
 INIT - Input\_Admin tells Keyboard the human ID for the list of human players. (Fields: type, humanId)  
 KEYBOARD\_READY - Keyboard notifies Input\_Admin that it is ready to start. (Field: type)  
 START - Input\_Admin tells Keyboard to start getting keys. (Field: type)  
 KEYBOARD\_MOVES - Keyboard notifies Input\_Admin when there is any key pressed. (Fields: type, key)  
 OKAY - Acknowledgment from Input\_Admin. (Field: type)  
 END - Input\_Admin notifies Keyboard that the game ended. (Fields: type, humanId)