CSCI3180 – Principles of Programming Languages – Spring 2011

Assignment 2 — Concurrent Programming (Administrator-and-Worker)

Deadline: **(Part A) February 13, 2011 (Sunday) 23:59 pm**
(Part B and Bonus part) February 27, 2011 (Sunday) 23:59 pm

1 Introduction

In this assignment, you have to implement a popular arcade game *Light Cycle* in *Tron*. 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 RTOS, QNX) provided by the library, called Synchronous Interprocess Messaging Project for LINUX (SIMPL) version 3.2.1. You should use the `ncurses` library to implement the textual user interface. To facilitate checking, your program should be able to compile with `gcc` on `linux6.cse.cuhk.edu.hk` (Ubuntu 9.04 Server).

2 *Light Cycle* in *Tron*

Tron is an action science fiction film first released in 1982. The film tells the story of a man being transported into the digital world in a mainframe computer where he has to defeat the master control program ruling the digital world. Since video games play an important role in the film, many video games based on *Tron* have been produced over the years. One of the most successful arcade games is *Light cycle*. In the game, a light cycle is a computer or human controlled fictional motorcycle running in a bounded arena. The cycles in the arena are racing in such a high speed that their traces create a wall of light behind them as they move. If a light cycle hits the wall, it is out of the game and the last player in the game wins. In this assignment you are going to implement a simplified version of the game, which begins with two light cycles initialized in fixed positions facing each other. During the course of the game, both cycles move forward at a constant speed in four possible directions: north (N), east (E), south (S), and west (W). There is also a limited number of turbo boosts which can greatly increase the speed of a light cycle for a limited time. We use four buttons to control the directions of a light cycle: $\uparrow$ (N), $\downarrow$ (S), $\leftarrow$ (W), and $\rightarrow$ (E) for the first player and w (N), s (S), a (W), and d (E) for the second player. We use a button p for the first player and q for the second player to control the usage of turbo boosts. The rules of this game are as follows:

1. Both light cycles should move within the arena, creating a wall of light behind them as they move.
2. A light cycle fails if it collides with the light wall or the boundary of the arena.
3. Under the circumstance of simultaneous head to head collisions, the game ends in draw.
4. Both light cycles advance one unit forward every 0.05 second.
5. A light cycle has three turbo boosts during a game. If a turbo boost is used, a light cycle can move forward every 0.01 second for five consecutive steps.
6. The arena is a 200×100 rectangle. Since the size of the arena can be much larger than the size of the window, only a fraction of the arena is visible in the window as shown in Figure 2. The center of the display window should be the center between the two light cycles. For example, let (x_1, y_1) and (x_2, y_2) be the position of the two light cycles, the center of the display window should then be $((x_1+x_2)/2, (y_1+y_2)/2)$. If $(x_1+x_2)/2$ or $(y_1+y_2)/2$, results in a fractional number, it should be *rounded* to the nearest integer. In order to keep both light cycles inside the window, the distance between the cycles in the x-axis cannot exceed the width of the window minus 6. Similarly, the distance between the cycles in the y-axis cannot exceed the height of the window minus 6. If any cycle is trying to move to a new position violating these constraints, it should be forced to stop.

3 Assignment Details

The assignment consists of **two parts and a bonus part**. In **part A**, write **a simple program** to animate a cutdown version of the game. You have to first initialize the screen, and detect the size of the window. Then, initialize a light cycle in a random location and a random orientation inside the arena. The animation starts and the light cycle should move randomly within the arena. **The minimum requirement is a light cycle which is able to avoid collisions.** Note that you are going to reuse some code of the first part in the second part of the assignment. In part A, the center of the window should be the same as the position of the light cycle. In **part B**, write **programs** to model the game using the **Administrator-and-Worker paradigm**. 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. In the **bonus part**, you need to implement the processes for the human control part.

3.1 Processes Design

There are *six* major types of processes in our design.

- Game_Admin maintains the rules of the game, and the positions of the light cycles.
- Display_Admin controls the screen output sequence.
- Cycle controls the direction of a computer controlled light cycle.
- Timer sleeps for a certain amount of time.
- Courier relays messages between administration processes.
- Painter formats and paints the output to the screen.

To enable human players, two additional types of processes are needed.

- Input_Admin controls the human controlled light cycles based on keyboard inputs.
- Keyboard gets the human player's input from the keyboard.

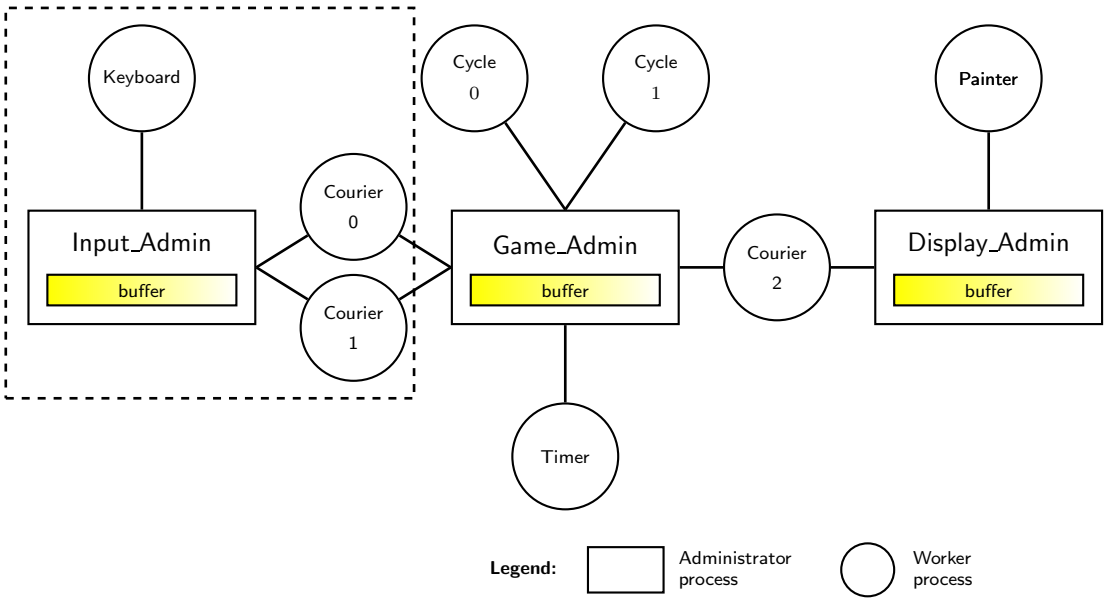

Figure 1: The Administrator-and-Worker Architecture for the Game

In part B of this assignment, you need to write a program for each of the six major types of processes: Game_Admin, Display_Admin, Cycle, Timer, Painter, and Courier 2

(the part outside dash-box as shown in Figure 1). In the bonus part, you need to implement Input_Admin, Keyboard, Courier 0, and Courier 1 (the part inside dash-box as shown in Figure 1) for human control.

In the implementation, there are two light cycles in the game. Each of them can be either computer controlled or human controlled. We use Cycle 0 to denote the first computer controlled cycle if there exists a computer controlled cycle in the game. We use Cycle 1 to denote the second computer controlled cycle if both cycles are computer controlled. If there is at least one human player in the game, we need Keyboard to get all the keys pressed and send to Input_Admin. We use Courier 0 to relay the messages from the Input_Admin to Game_Admin for the first human player. We use Courier 1 to relay the messages for the second human player if both cycles are human controlled.

A timer worker denoted by Timer is used to maintain the time interval for advancing light cycles forward. All the screen output is handled by the Painter process, which keeps on getting paint jobs from 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, Cycle 0 (if there exist a computer controlled cycle), Cycle 1 (if both cycles are computer controlled), 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. Every successfully registered Cycle should be given a unique ID. If human players participate in the game, Courier 0 and Courier 1 (if there are two players) 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 the Timer worker to start the sleep job, and replies the game start signal to human players and all registered computer controlled cycles.

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.

```
/* Direction */
typedef enum {
    EAST = 0, SOUTH = 1, WEST = 2, NORTH = 3
} DIRECTION;

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

/* Cycle */
typedef struct {
    COORDINATE pos;
    DIRECTION dir;
}CYCLE;

/* Wall */
typedef enum {
    NONE = 0, REDCYCLE = 1, GREENCYCLE = 2, BOUNDARY = 3
}WALL;
```

```
/* Arena */
typedef struct {
    WALL wall[MAX_WIDTH][MAX_HEIGHT];
    CYCLE cycle[MAX_CYCLE];
} ARENA;

/* Booster */
typedef enum {
    YES = 0, NO = 1
} BOOST;

/* Message Format */
typedef struct {
    MESSAGE_TYPE type;  /* Message Type */
    int cycleId;        /* Cycle ID */
    int interval;       /* Sleep Interval */
    DIRECTION dir;      /* Cycle move direction */
    BOOST boost;        /* Cycle use booster? */
    ARENA arena;        /* Current Game State */
    /* Message field for the bonus part */
    int humanId;        /* Courier ID*/
    int key[MAX_KEYS];  /* List of key ASCIIs*/
} 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 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 Cycle, Game_Admin only needs to specify the `type` and `cycleId` fields in the `MESSAGE` structure. Thus, when a Cycle receives a message, it can retrieve its ID from the `cycleId` field of the structure. We list below the possible messages between the different processes and the fields used for each message. **Message field usage must be strictly followed.**

Messages between Game_Admin and Cycle 0, Cycle 1
`REGISTER_Cycle` - Cycle registers to Game_Admin. (Field: `type`)
`INIT` - Game_Admin confirms Cycle registration. (Fields: `type, cycleId`)
`FAIL` - Game_Admin fails Cycle registration. (Field: `type`)
`CYCLE_READY` - Cycle tells Game_Admin that it is ready. (Fields: `type, cycleId`)
`START` - Game_Admin declares the game start to Cycle. (Fields: `type, arena`)
`MOVE` - Cycle requests Game_Admin to change direction. (Fields: `type, cycleId, dir, boost`)
`UPDATE` - Game_Admin sends the current game status to Cycle. (Fields: `type, arena`)
`END` - Game_Admin sends an ending message to Cycle. (Fields: `type`)

Messages between Game_Admin and Timer
`REGISTER_TIMER` - Timer registers to Game_Admin. (Field: `type`)
`INIT` - Game_Admin confirms Timer registration. (Fields: `type`)
`FAIL` - Game_Admin fails Timer registration. (Field: `type`)
`TIMER_READY` - Timer tells Game_Admin that it is ready. (Fields: `type`)
`SLEEP` - Game_Admin asks Timer to sleep for certain amount of time. (Fields: `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, cycleId, arena) (Here cycleId denotes the winner of the game, cycleId = -1 if the game ends in draw.)

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 - Acknowledgment from Display_Admin. (Field: type)
END - Courier 2 delivers an ending message to Display_Admin. (Fields: type, cycleId, arena)

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 result, and terminate itself. (Fields: type, cycleId, arena)

Messages between Game_Admin and Courier 0, 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, humanId)
HUMAN_MOVE - Courier notifies Game_Admin of human player's request to change direction. (Fields: type, humanId, dir, boost)
UPDATE - Game_Admin asks Courier to deliver a message that confirms the change in direction. (Fields: type, humanId)
END - Game_Admin asks Courier to deliver an ending message and terminate itself after delivering the message. (Fields: type)

3.3 Messages for the Bonus Part

For the bonus part, three additional message types are needed:
REGISTER_KEYBOARD, KEYBOARD_READY, KEYBOARD_INPUT.
You also need the following additional messages.

Messages between Input_Admin and Courier 0 and Courier 1
REGISTER_COURIER - Courier registers to the Input_Admin. (Field: type)
INIT - Input_Admin confirms the registration. (Fields: type)
FAIL - Input_Admin fails the courier registration. (Field: type)
COURIER_READY - Courier notifies Input_Admin that it is ready. (Fields: type)
REGISTER_HUMAN - Input_Admin asks courier to deliver 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)
HUMAN_READY - Input_Admin asks Courier to deliver a ready signal for the human player. (Fields: type, humanId)
START - Courier notifies Input_Admin that the game starts. (Fields: type, humanId)
HUMAN_MOVES - Input_Admin asks Courier to deliver a human player request to change direction. (Fields: type, humanId, dir, boost)
UPDATE - Courier notifies Input_Admin that the change in direction is done. (Fields: type, humanId)
OKAY - Acknowledgment from Input_Admin. (Field: type)
END - Courier notifies Input_Admin that the game ended. (Fields: type)

Messages between Input_Admin and Keyboard
REGISTER_KEYBOARD - Keyboard registers to Input_Admin. (Field: type)
INIT - Input_Admin confirms the registration. (Fields: type)
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_INPUT - Keyboard notifies Input_Admin for list of keys pressed. (Fields: type, key)
OKAY - Acknowledgment from Input_Admin. (Field: type)
END - Input_Admin notifies Keyboard that the game ended. (Fields: type)

3.4 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/`

3.5 Output

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

The reference output format is the MINIMUM requirement, text log file will not be accepted! Each light cycle is denoted by a `@`. The wall created by the light cycles and the boundary of the arena are denoted by s. Two different colors (red and green) are used to differentiate the light cycles and the walls created by them. We use the white color to denote the boundary of the arena.

You are free to improve the reference output format. **Such improvement must be done only in the Painter process and must obey strictly the messaging interface.** Also, the following basic components must be included in the screen shot: the locations of the cycles, the wall and the boundary of the arena if inside the display window, and the winner on the final screen shot. In addition, the screen should be updated after every change of the positions of the light cycles. Of course, the output has to reflect the real situation of the arena by respecting the order of the events.

3.6 Other Requirements

1. **Although the implementation of the bonus part is optional, your Game_Admin should still anticipate for messages coming from Courier 0 and Courier 1.**

2. **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.**

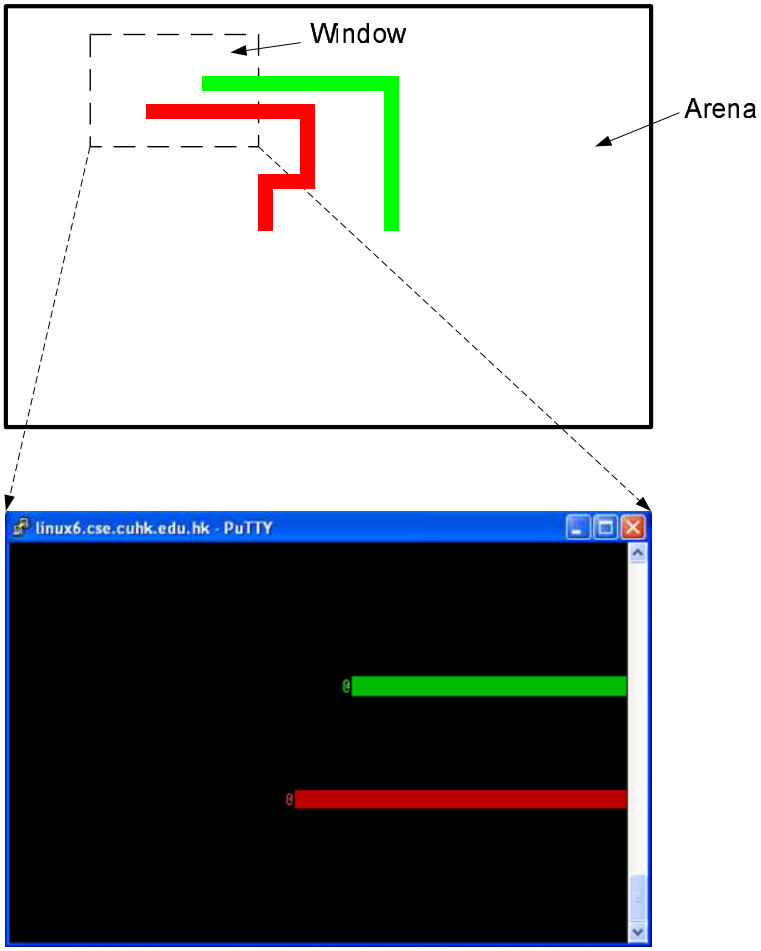

Figure 2: Sample Screen Output

3. The process starting sequence is critical. For example, the Cycle processes cannot be started earlier than Game_Admin. One possible process starting sequence is: Game_Admin → Cycle 0 (for the first computer controlled cycle, if exists), Cycle 1 (for the second computer controlled cycle, if exists) → Display_Admin → Painter → Courier 2 → Input_Admin → Courier 0 (for the first human player, if exists), Courier 1 (for the second human player, if exists) → Timer → Keyboard (If there is a human player). You should control the starting sequence by using shell script which accepts the following game parameter: the number of human players in the game. Hint, you can reuse the shell script provided.

4. 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.

5. 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 2 is not available yet. Analogously, the same requirement applies to Input_Admin, Courier 0 and Courier 1.

6. Upon game end, an ending message should be sent to each active process to notify them of termination so that you do not need to kill them manually after the game.

7. You have to design some AI for the computer controlled cycles. The minimum requirement is a randomly moving cycle which is able to avoid collisions.

8. **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 Important Notice

For both parts in this assignment, you can use any of the following four Linux Workstation (Ubuntu 9.04 Server): `linux6`, `linux7`, `linux8` and `linux9`. Please be considerate to the other users, **remember to kill all the useless processes**, especially when you press `Ctrl-C` to terminate a process which communicates with another process.

5 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 *Chan Tai Man*,
   your student ID is *1009234567*,
   your username is *tmchan*, and
   your email address is *tmchan@cse.cuhk.edu.hk*.

2. In your source files, insert the following header.

```
/*
 * CSCI3180 Principles of Programming Languages
 *
 * --- Declaration ---
 *
 * I declare that the assignment here submitted is original except for source
 * material explicitly acknowledged.  I also acknowledge that I am aware of
 * University policy and regulations on honesty in academic work, and of the
 * disciplinary guidelines and procedures applicable to breaches of such policy
 * and regulations, as contained in the website
 * http://www.cuhk.edu.hk/policy/academichonesty/
 *
 * Assignment 2
 * Name : Chan Tai Man
 * Student ID : 1009234567
 * Email Addr : tmchan@cse.cuhk.edu.hk
 */
```

   The sample file header is available at

   `http://www.cse.cuhk.edu.hk/~csci3180/resource/header.txt`

3. **The following file naming convention must be followed strictly.** For the first part, the C source file should have the filename `animate.c`. For the second part, use the filenames `game_admin.c`, `display_admin.c`, `courier.c`, `painter.c`, `timer.c`, and `cycle.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. Make sure you can compile and run the program without any problem.

5. `Tar` your source files to `username.tar` by

   `tar cvf tmchan.tar [source files] [shell script] [makefile]`

6. `Gzip` the `tarred` file to `username.tar.gz` by

```
gzip tmchan.tar
```

7. `Uuencode` the `gzipped` file and send it to the course account with the email title "HW2A *studentID yourName*" by

```
uuencode tmchan.tar.gz tmchan.tar.gz \
| mailx -s "HW2A 1009234567 Chan Tai Man" csci3180@cse.cuhk.edu.hk
```

   This is for the first part of the assignment. **For the second part, follow the same procedure but remember to use HW2B instead.**

8. Please submit your assignment using your Unix accounts.

9. An acknowledgement email will be sent to you if your assignment is received. **DO NOT** delete or modify the acknowledgement email. You should contact your TAs for help if you do not receive the acknowledgement email within 5 minutes after your submission. **DO NOT** re-submit just because you do not receive the acknowledgement email.

10. You can check your submission status at

    `http://www.cse.cuhk.edu.hk/~csci3180/submit/hw2a.html`.

    and

    `http://www.cse.cuhk.edu.hk/~csci3180/submit/hw2b.html`.

11. You can re-submit your assignment, but we will only grade the latest submission.

12. The following late penalty scheme for assignments submission will be adopted in this assignment:

    - 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

13. Enjoy your work :>