Middle East Technical University   Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2022-2023

Homework 1 - Bomberman Game

Due date: 16 04 2023, Sunday, 23:59

1 Overview

In this homework you are going to implement a simplified version of the Bomberman game, called *bgame*, which supports the concurrent execution of multiple processes corresponding to the game entities. Your implementation will the controller of the game and game entities will communicate with your controller using inter process communication (IPC).
Keywords: *bomber, game, ipc, pipe*

2 Introduction

Bomberman[1] is a vintage game currently owned by Konami. The latest version is a mobile game released in Apple Arcade. The objective of the game is to be the last bomber standing in a 2D maze. The bombers can drop bombs to kill other bombers and to destroy obstacles. The obstacles can be both destructible or indestructible. The bombs explode in a cross pattern where the explosion goes to a certain radius along both x and y axis away from the center in both positive and negative directions. Upon meeting an obstacle, it will stop along that direction and destroy the obstacle if it is destructible. The bombers can pickup power-ups or penalties boxes during the game. It can be played in single player or multiplayer.

In our simplified version there is going to be three main entities. They are bombers, bombs and obstacles. You will create a controller program that coordinates these entities and control the maze they are operated on. The movement of bombers and their locations will integer coordinates. The bombers and bombs will be separate executables and will be executed by your program. The entities are briefly explained below:

- **Bomber:** They are the active entities that move in the maze and place bombs. They can see certain distance away from their location and will receive this information from the controller. They can place bombs and these bombs will explode after a certain period of time. The size of their explosion and period of time when they will explode will differ. If an explosion reaches them, the bomber will die. The game continues until there is only a single bomber left. There can be multiple types of bombers depending on their executable.
- **Bomb:** They are stationary entities that explode away from center after a certain period of time and to a certain distance. The time of their explosion and distance the explosion will travel, are

[1]`https://en.wikipedia.org/wiki/Bomberman`

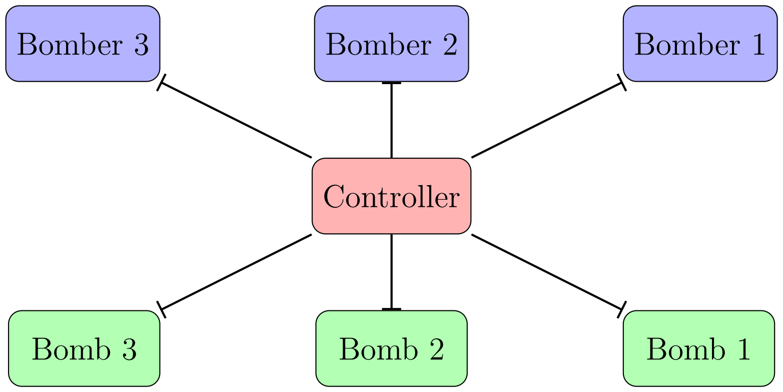

Figure 1: Controller communicating with bombers and bombs

their properties. These properties are determined at their creation. They are destroyed during explosion. There is only one type of bombs but explosion size and interval will be provided to the executable.

- **Obstacles:** These are stationary entities in the maze similar to bombs. They block bombers, their vision and explosion of the bomb along their direction. They are the only entities directly controlled by the controller. They only have one property. It is the number of explosions they can withstand. If this number is given as -1, then that means that that obstacle cannot be destroyed.

The controller is the main program that controls the game. It will constantly communicate with bombers and bombs for the game to progress. This communication will be done using bidirectional pipes. After the game starts and entities will send messages to the controller with semi random intervals for their moves. They will wait for a response and will not make any moves before receiving it. The game continues until only one bomber is left.

3 Game and Implementation Details

The controller will receive the size of the maze, the obstacle and the bomber information at the start from standard input. The bombers and bombs communicate to the controller using their standard input and output. For this reason, the controller need to create necessary pipes and redirect the bombers inputs and outputs to the pipes before executing them. This also applies to the bombs when they are created. After starting the bombers, it waits for their messages and act according to those messages. After any bombs have been planted, the controller should also wait for messages from them as well. The bombs have higher priority than bombers, therefore their messages should be served first. For example, if a bomb explosion reaches a bomber, even if they have made a move at the same turn, they will die since bombs have priority. It should continue serving the messages until there is only a single bomber left. After that point, it should wait for remaining bombs to explode and quit. The controller should reap all bomber and bomb processes to not leave any zombie process.

The pseudocode of the controller are below:

Algorithm 1 Controller Loop

```
Read the input information about the game from standard input
Create pipes for bombers (see IPC section)
Fork the bomber processes (see fork function)
Redirect the bomber standard input and output to the pipe (see dup2) and close the unused end.
Execute bomber executable with its arguments (see exec family of functions)
while there are more than one bomber remaining: do
    Select or poll the bomb pipes to see there is any input (see IPC for details)
    Read and act according to the message (see Messages)
    Remove any obstacles if their durability is zero and mark killed bombers
    Reap exploded bombs (see wait family of functions)
    Select or poll the bomber pipes to see there is any input (see IPC for details)
    Read and act according to the message (see Messages) unless the bomber is marked as killed
    Inform marked bombers
    Reap informed killed bombers (see wait family of functions)
    Sleep for 1 millisecond (to prevent CPU hogging)
end while
Wait for remaining bombs to explode and reap them
```

The bombers will behave based on their initiative and the bombs will explode after a certain duration (plus some random variation to make it unpredictable). You will only need to respond to their messages and update maze information you hold. The messages received from bombers are explained below:

- **Start:** This message is sent after bomber is starting to let controller know they have started. The controller should respond with their coordinates. Bombers do not know their location at the start.
- **See:** This message is sent when bomber wants learn its surroundings. The bombers can see everything around them up to 3 steps away excluding their location. The obstacles will block the vision of the along that direction.
- **Move:** This message is sent when the bomber want to move a certain location. It can only move in x or y axis and only 1 step. If there is no obstacle or bomber in the way and the move is valid, the server should sent back the new location.
- **Plant:** This message is sent when the bomber wants to plant a bomb at its location together with its parameters. The bomb plant can only be successful if there is no other bombs at that location. The server should create necessary pipes, fork the process, redirect the pipes and execute the bomb with its parameters. The response to this message contains whether the plant is successful or not.

There is only one message received from the bombs:

- **Explode:** This message is sent when the bomb has exploded. The bomb will quit after sending this message. The explosion follows a cross pattern where it goes to a certain distance away from the center. The distance is determined by the bomber and should also be known by the controller. It should only affect $-x, +x, -y, +y$ directions away from the center. The explosion should be blocked along the direction of the obstacle if there is any and the obstacle should lose one point of durability. If its durability reaches zero, it should be removed. If it reaches a bomber, the bomber should die. Bombers can die from the explosions of their own bombs. The explosion points can be calculated with a simple formula:

$$
\begin{aligned}
(x, y) &= (c_x + i, c_y) && \forall i \in [-r, r] \\
(x, y) &= (c_x, c_y + i) && \forall i \in [-r, r]
\end{aligned}
$$

where c_x, c_y are the location of the bomb and r is the explosion radius.

4 IPC

4.1 Bidirectional Pipe

The communication between the controller and game entity processes will be carried out via bidirectional pipes which can be created as follows:

```
#include <sys/socket.h>
#define PIPE(fd) socketpair(AF_UNIX, SOCK_STREAM, PF_UNIX, fd)
```

Linux pipes created with `pipe()` system call are unidirectional, data flow in one direction. Only one end can be read and the other end can be written. The `socketpair()` function above is a replacement for Linux pipes providing bidirectional communication. If you use `PIPE(fd)` above, data written on `fd[0]` will be read on `fd[1]` and data written on `fd[1]` will be read on `fd[0]`. Both file descriptors can be used to write and read data.

The controller should be serving an arbitrary number of bombers and bombs. That means, it should create **n** sockets and read requests from **n** different file descriptors. If it blocks in one of them the requests from others has to wait. Therefore, it should not block on any socket, instead should check whether there is data to be read on that socket. In order to check whether there is data on a particular socket, the `poll()` or `select()` system calls can be used.

The controller pseudo-code is given as:

```
while there more than 1 active bombers:
    select/poll on socket file descriptors of bombs
    for all file descriptors ready (have data)
        read request
        serve request
    select/poll on socket file descriptors of active bombers
    for all file descriptors ready (have data)
        read request
        serve request
    sleep 1 milliseconds (to prevent CPU hogging)
wait for bombs remaining bombs to explode and finish
```

4.2 Messages

There are five messages that can be received from the bomber and the bomb executables. These five messages uses struct data type for communicating with the server. It is called `incoming_message`. The code for the struct is given below:

```
typedef enum incoming_message_type {
    BOMBER_START,
    BOMBER_SEE,
    BOMBER_MOVE,
    BOMBER_PLANT,
    BOMB_EXPLODE,
} imt;

typedef struct bomb_data {
    long interval;
    unsigned int distance;
} bd;

typedef union incoming_message_data {
    coordinate target_position;
    bd bomb_info;
} imd;

typedef struct incoming_message {
    imt type;
    imd data;
} im;
```

The type indicate which of the five messages it received. The data is union of two structures necessary for the messages. First is a coordinate to indicate (x,y) value and the second is the properties for the bomb necessary for the plant bomb message.

There are only four message responses to because one message does not have a response. Similarly `outgoing_message` is also a struct that is used for communication. The code is given below:

```
typedef enum outgoing_message_type {
    BOMBER_LOCATION,
    BOMBER_VISION,
    BOMBER_PLANT_RESULT,
    BOMBER_DIE,
    BOMBER_WIN,
} omt;

typedef enum object_type {
    BOMBER,
    BOMB,
    OBSTACLE
} ot;

typedef union outgoing_message_data {
    unsigned int object_count;
    coordinate new_position;
    int planted;
} omd;

typedef struct outgoing_message {
    omt type;
    omd data;
} om;
```

```
typedef struct object_data {
    coordinate position;
    ot type;
} od;
```

Unlike incoming messages, there are two different messages that can be sent from the controller. First one is the outgoing message struct that used as an answer to queries from the bombers. In the data portion, it has three possible values. First one is object count for vision request, second one is a coordinate for a move request response and finally a boolean value to indicate if planting a bomb is successful. The second message is `object_data` struct that is sent as an additional response to the vision request. This struct is sent in object count amount to indicate the position of nearby objects. The details of incoming messages and their responses are given below:

- **BOMBER_START**: This message is the first message bombers send to the server. It indicates that they are operating and expecting their location information. The data part of the message is ignored.
  The answer to this message is `BOMBER_LOCATION` message. In the data part coordinate is filled with the location of the bomber as bombers are unaware of their location at the start of the game.
- **BOMBER_MOVE**: This message is sent when a bomber tries to move to a location. The data will be the target coordinate the bomber is trying to move. There are four conditions that needs to be met for a move to be accepted. First, the position should be only one step away from the bomber position. Second, it should be a horizontal or vertical move, a diagonal move is not accepted. Third, there should be no obstacles or bombers in the target position. There could be a bomb in the target position, a bomber and a bomb can occupy the same position. Finally, the target should not be out of bounds. The answer is `BOMBER_LOCATION` message and the data is the new position of the bomber. If the move is not possible, it should sent back the old location.
- **BOMBER_PLANT**: This message is sent when a bomber tries to plant a bomb at its own location. The data of the incoming message contains the bomb information. In the bomb data, there are two parameters. First one is how long until the bomb explodes in milliseconds. The second should be radius of the explosion. The first argument should be passed to the bomb executable as an argument. The second argument should be stored to be used when the explosion happen.
  The bomb executable is named `bomb` and it only takes one argument which is the explosion interval. The plant should be successful as long as there is no other bombs at the location. The response is `BOMBER_PLANT_RESULT` message, the data should a boolean to indicate the success or failure of the plant.
- **BOMB_EXPLODE**: This message is sent by the bomb when it explodes. The data portion is not used for this message. The controller should use the stored explosion radius information to kill the bomber caught inside it or decrease the durability of obstacles. If a bomber is affected by the explosion, the server marks the bomber. When they send their next request, `BOMBER_DIE` should be sent without any other data to indicate that the bomber has died. If that bomber is second to last bomber, the last remaining bomber should be marked as well. When the last bomber makes a request, `BOMBER_WIN` should be sent to that bomber to let them know, they had won the game. After both of these messages, bombers would quit and they need to be reaped. If the explosion affects all of the remaining bombers, the one bomber should be furthest away from the center of the explosion should win. If there are more than one of them at the same distance, you can randomly select one of them.
- **BOMBER_SEE**: This message is sent by the bomber to see get information about its surroundings. The bombers as mentioned before can see three steps into all four directions unless it is blocked by an obstacle. The data part of the message is not used. After receiving this message the controller

should send `BOMBER_VISION` message type with number of objects as the data argument. After sending this message, it should follow it up with n `object_data` structures where n is the number of objects. This number can at most be 24 due to seeing three steps into four cardinal directions and each step could be occupied by one bomber and one bomb. Only occupied locations should be sent.

5 Input&Output

5.1 Input

The controller takes the input from the standard input. The input is in the following format:

```
<map_width> <map_height> <obstacle_count> <bomber_count>
<obstacle1_location_x> <obstacle1_location_y> <obstacle1_durability>
<obstacle2_location_x> <obstacle2_location_y> <obstacle2_durability>
....
<obstacleN_location_x> <obstacleN_location_y> <obstacleN_durability>
<bomber1_x> <bomber1_y> <bomber1_total_argument_count>
<bomber1_executable_path> <bomber1_arg1> <bomber1_arg2> ... <bomber1_argM>
....
<bomberK_x> <bomberK_y> <bomberK_total_argument_count>
<bomberK_executable_path> <bomberK_arg1> <bomber1_arg2> ... <bomberK_argL>
```

The indestructible obstacles have -1 as their durability. The bomber total argument count also includes the executable path as well. For example if we look at the first bomber in the format the number would be $M + 1$ as there are M arguments for bomber1. The argument count can be different for every bomber.

5.2 Output

You will be using a function provided by us for your outputs. The structures required to call the function and its prototype is given below:

```
typedef struct incoming_message_print {
    pid_t process_id;
    im *in_message;
} imp;

typedef struct output_message_print {
    pid_t process_id;
    om *out_message;
} omp;

typedef struct obstacle_data {
    coordinate position;
    int remaining_durability;
} obsd;
void print_output(imp *in, omp *out, obsd *obstacle, od* objects);
```

The `incoming_message_print` is for incoming messages and `outgoing_message_print` is for outgoing messages. Finally, obstacle data is to report obstacle information after an explosion and `objects` is used when a response to vision request is sent to the bomber. There are three main cases where outputs need to be printed:

1. When there is an incoming message from a bomber or a bomb. In this case, you need to fill the `incoming_message_print` with the `incoming_message` pointer and the process id of the game entity that sent the message. After filling that information, you need to call the function with `NULL` pointer on other parameters. Demonstration:

   ```
   print_output(in, NULL, NULL, NULL);
   ```

2. After the controller has sent an outgoing message, you need to print information about it. In this case, you need fill the `outgoing_message_print` with the `outgoing_message` pointer and the process id of the game entity that the controller sent the message to. After filling that information, you need to call the function with `NULL` pointer on other parameters. Demonstration:

   ```
   print_output(NULL, out, NULL, NULL);
   ```

   One thing to note when sending the response `BOMBER_VISION`, you also need to fill the object data so that the objects you sent is printed. It will be different from the regular print output function call. Do not call the function twice. Demonstration:

   ```
   print_output(NULL, out, NULL, objects);
   ```

3. For the final case, when an obstacle is damaged from an explosion in the game, you need to call the print function to indicate event. You need to fill `obstacle_data` structure with the coordinate and the remaining durability after the explosion. After filling that information, you need to call the function with `NULL` pointer on other parameters. Demonstration:

   ```
   print_output(NULL, NULL, obstacle, NULL);
   ```

   This also includes the indestructible obstacles. In their case, keep the remaining durability as -1.

6 Specifications

- The bgame must terminate when there is only a single bomber remain.
- All processes in the bgame that are being executed should run in parallel. Therefore, order of the outputs will be non-deterministic. In other words, every execution of the exact same inputs may generate different outputs. Output order will not be important during evaluation.
- X and Y coordinates are correlated to horizontal and vertical planes respectively. This is opposite of how C and C++ access array values. This means that if you are using an array for storing objects, the access order should be reversed. Location $(0, 0)$ refers to the top leftmost corner of the maze.
- You can sleep for 1 milliseconds at each loop ends to prevent CPU hogging. All message intervals will be longer than this duration.
- The bgame should reap all its child processes. It should not leave any zombie processes. Otherwise you will lose points for each testcase that it happened in.
- Do not modify the provided "message.h", "message.c", "logging.h", "logging.c" files, they will be replace with their originals during grading. There is also no point in submitting them.
- Evaluation will be done using black box technique. So, your programs must not print any unnecessary character and outputs of the processes must not be modified.

7 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your own work. Any work used from third party sources will be considered as cheating and disciplinary action will be taken under or "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

8 Submission

Submission will be done via ODTUClass. Create a tar.gz file named `eXXXXXXX.tar.gz` that contains all your source code files (provided files excluded) along with a makefile. eXXXXXXX is your complete student ID (all seven digits). The tar file should not contain any directories! The make should create an executable called bgame. Your code should be able to be executed using the following command sequence.

```
$ tar -xf eXXXXXXX.tar.gz
$ make
$ ./bgame
```