Ceng334

Homework 1 Recitation

Çağrı Utku Akpak

March, 2023

Outline

Introduction

- Implement bgame, a simplified version of the Bomberman Game
- Implement the controller that run game entities as processes and communicate with them using IPC

Introduction

Objectives

- Learn to use process execution and file descriptors.
- Hands on knowledge with fork, exec, dup2, wait and similar functions.
- Gain understanding of programs are executed.

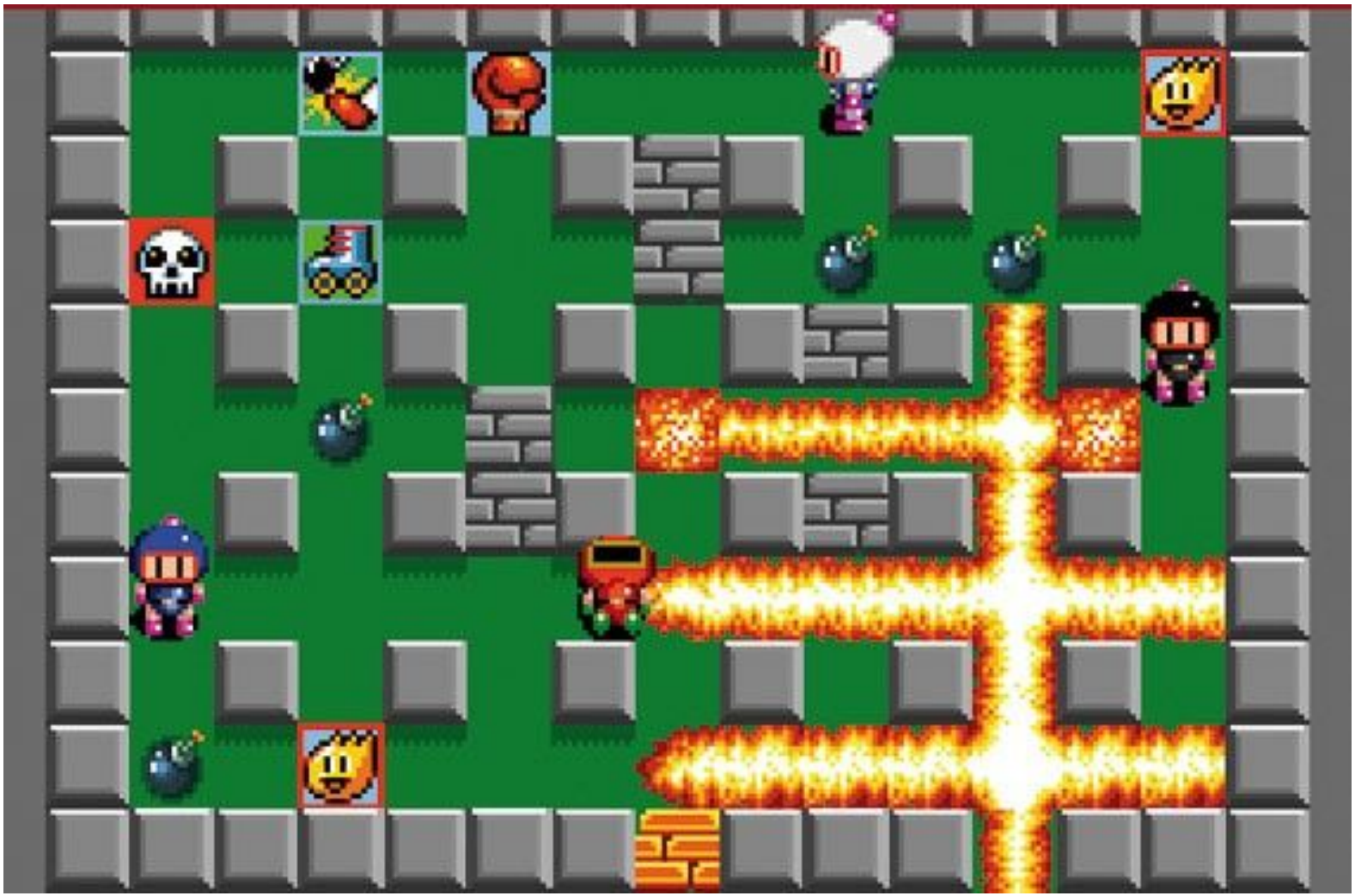
Background

Bomberman Game

- Originally released by Konami
- Bombers, bombs, obstacles and various other entities
- Last bomber standing wins
- In our version, bombers and bombs are separate executables

Background

Bomberman Game



Background

BGame

- Simplified version of Bomberman game
- Implement the main controller that maintains a 2D maze
- There are bombers, bombs and obstacles in our version
- Controller connect to bombers and bombs via pipes
- The game continues until one bomber left

Background

BGame - Bomber

- See three steps away from themselves. Request from controller
- Move horizontal or vertical directions one step at a time
- Movement can be blocked by obstacles or other bombers
- Can drop bombs that explode after a certain time to a certain distance
- Bombers can die if an explosion reaches them

Background

BGame - Bombs

- Explode after a certain duration that is specified during their creation
- Explosion can only go to vertical and horizontal directions
- They quit immediately after explosion
- Obstacles block explosion along their direction but can be damaged if they are destructible

Background

BGame - Obstacles

- Block movement, explosion and vision along their direction
- Can have limited durability and after their durability drops to zero, they are removed

Implementation details I

Controller

- Read the input information about the game from standard input
- Create pipes for bombers IPC
- 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:}
  - Select or poll the bomb pipes to see there is any input (see IPC for details)
  - Read and act according to the message (see IPC)
  - Remove any obstacles if their durability is zero and mark killed bombers
  - Reap exploded bombs (see `wait` family of functions)

Implementation details II

Controller

- Select or poll the bomber pipes to see there is any input (see IPC for details)
- Read and act according to the message (see IPC) 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)

- Wait for remaining bombs to explode and reap them

Implementation details I

Bomber Messages

- **Start:** This message is sent after bomber is starting to let controller know they have started.
  The controller should respond with their coordinates.
- **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.

Implementation details II

Bomber Messages

- **Plant:** This message is sent when the bomber wants to plant a bomb at its location together with its parameters.
  The plant should be successful if there is no other bombs at the location.
  The server should create necessary pipes, fork the process, redirect the pipes and execute the bomb with its parameters.
  The response should contain whether the plant is successful or not.

Implementation details I

Bomb Message

- **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.

Implementation details II

Bomb Message

The explosion points can be calculated with a simple formula:

$$(x, y) = (c_x + i, c_y) \qquad \forall i \in [-r, r]$$
$$(x, y) = (c_x, c_y + i) \qquad \forall i \in [-r, r]$$

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

IPC I

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.
- 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.

IPC II

Bidirectional Pipe

- The controller should be serving an arbitrary number of bombers and bombs.
- It should create **n** sockets and read requests from **n** different file descriptors.
- 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.

IPC III

Bidirectional Pipe

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
```

IPC I

Messages

- The 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;
```

IPC II

Messages

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

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

IPC III

Messages

- 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;
```

IPC IV

Messages

```
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;
```

IPC V

Messages

- **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 answer to this message is `BOMBER_LOCATION` message.
- **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.
  For conditions needs to be satisfied for a move to be accepted.
  The answer is `BOMBER_LOCATION` message and the data is the new position of the bomber.

IPC VI

Messages

- **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.
  The bomb executable is named `bomb` and it only takes one argument which is the explosion interval.
  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.
  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.
  When the last bomber makes a request, `BOMBER_WIN` should be sent to that bomber to let them know, they had won the game.

IPC VII

Messages

Bombers need to be reaped after they have died or won.
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.
  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.
  Only occupied locations should be sent.

Input&Output I

Input

```
<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> ... <bomber
....
<bomberK_x> <bomberK_y> <bomberK_total_argument_count>
<bomberK_executable_path> <bomberK_arg1> <bomber1_arg2> ... <bomberl
```

- The indestructible obstacles have -1 as their durability.
- The bomber total argument count also includes the executable path as well.

Input&Output I

Use the provided function and structures for printing output

```
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);
```

Input&Output II

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.

   ```
   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.

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

   If `BOMBER_VISION`, you also need to fill the object data so that the objects you sent is printed. Do not call the function twice.,

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

Input&Output III

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.

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