Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2020-2021

Homework 1 - Rogue Pipes

Due date: 18/04/2020, Sunday, 23:59
(Ver. 1.0)

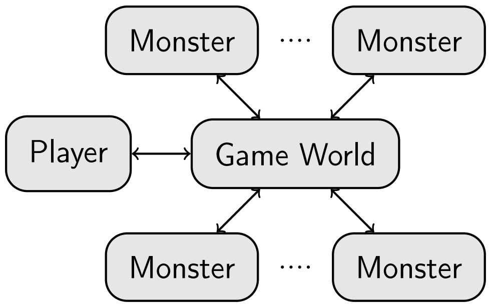
1 Introduction

The objective of this assignment is to learn the basics of IPC (inter-process communication) through the implementation of a game similar to the Rogue[1]. The Rogue game was released in 1980 and was popular in Unix-based mainframe systems. The game is built on a randomly generated map, consisting of multiple layers with each layer being composed of rooms inter-connected via doors. The rooms may contain monsters and/or rewards, and the objective of the game is to retrieve a treasure at the bottom layer. In the original version of the game, the graphics were text-based with all entities (player/monster/reward/door etc.) being represented with different ASCII characters, such as '@' and '+' denoting the player and the door respectively.

In this assignment, you will be implementing a simplified version of the game. Unlike Rogue, however, there will be only one room containing the player and an arbitrary number of monsters. The objective of the game is to survive by either leaving the room without dying, or killing every monster in the room. No treasure/reward is included. The game continues until either the player dies, leaves the room or kills all the monsters in the room.

2 Components and Communication

The system consists of three types of components: *game world*,*player* and *monsters*, all running as separate processes.



[1]https://en.wikipedia.org/wiki/Rogue_%28video_game%29

The game is turn-based. At the beginning of each turn, the game world informs the player about its current position, the accumulated damage inflicted on the player by the monsters (which will be zero in the first turn), the number of alive monsters, the positions of the alive monsters and whether the game is over or not.

In return, the player responds to the game world by either moving, attacking one or more monsters, acknowledging its death or leaving the game. Then the game world sends each of the monsters, their new position, damage inflicted on them by the player, the player's new position, and whether the game is over or not. In return, every monster sends a response to the game world which is either a movement, an attack to the player, or its death announcement. This loop continues until the game is over.

Both the player and the monsters should interact with the game world using fixed sized binary messages. They should receive these messages from their standard input and send their responses via the standard output.

The *game world* should store the state of the monsters and the player, and update them according to the messages it receives. At every turn, the game world should first communicate with the player, then the alive monsters in sequence. The actions of the monsters are carried out in an order based on their coordinates (so the game state will be deterministic).

None of the game world, the player and the monsters have the complete information about the game. The game world only stores the information about the positions and the types of the monsters and the position of the player. It is oblivious to (does not remember) their internal states like health. On the other hand, the player and the monsters only receive the external information like the position of other entities via game world messages. So each of the game world, the player and the monsters should treat each other as black boxes and only rely on the information they get from the messages/responses.

In this assignment you are expected to implement the programs for the game world and the monsters. The player will be provided to you as a separate program. You should provide separate programs for the game world and the sample monster. Your game world program should `fork`() and `exec`() the player and the monsters as separate processes before starting the game. The IPC between the game world - the player and the game world - the monsters should be coordinated properly.

2.1 Game World

The game world should first read the width and height of the room, position of the door, position of the player, the command line for the player, number of monsters, monster command line from the standard input during the initialization (see Input & Output for details). Afterwards, it should fork and execute (see `fork` and `exec` function family) the player and the monster processes.

Note that before forking them, actions necessary to establish communication via bidirectional pipes (see Communication) and standard input and output redirection using `dup2` system call should be carried out. Each player and monster processes should use separate pipes to communicate with the game world. After the launch of the player and the monster processes, the game world should wait until it receives the ready response from all child processes and print the initial state of the map.

After initialization, the game world should run in a loop until a game over condition is met.

The game world should complete the following tasks in the given order in a while loop:

1. Send turn message to and receive response from the player,
2. Process the response

3. Send turn message to the every monster,
4. Collect monster responses in a loop, checking for any of the pipe ends for a new responses and reading them.
5. After all monster responses are received, sort them based on processing order, their current coordinates. The coordinates on the map will be ordered left to right (first dimension) and top to bottom (second dimension) in increasing order.
6. For every response:
   - Determine the type of the response. The response types and contents are explained in the Communication subsection.
   - Process the response and update the world information.
7. After all responses are processed, print the map to the standard output using the provided library as explained in the Input & Output section.

When an entity tries to move to a cell which is a wall, the door (with the exception of the player, since moving to the door is an objective for the player), occupied by another entity, or out of the rooms boundaries the game world should cancel the movement action and send its old position in the message. When two monsters try to move to the same cell, the game world should order their actions with monster with smaller current x coordinate first. If they have the same x coordinates, action of the monster with the smaller y coordinate will be taken first. For example, if a monster at coordinates (1,3) and another at coordinates (3,3) tries to move to the coordinates (2,3), the first monster should move to (2,3) and the second one should stay at (3,3) since (2,3) is occupied.

When any game over condition is met, it should print the map in its current state and the appropriate game over message, inform all alive child processes and wait for them to terminate before exiting.

If a monster sends the `mr_dead` message, the game world should wait for that monster process to exit before starting a new turn. Dead monsters will not be printed and will not occupy a cell in the map until the end of the game.

When a process exited, it should be handled properly (no zombie processes should be left) and the relevant pipe connections must be closed.

The communication messages between game world-player and game world-monster processes is explained in detail in the Communication section.

2.2 Player

A sample player program is provided as an executable and you should only focus on the communication between it and the game world by sending messages and processing the responses properly using the message/response structure explained in the Communication section. In order to test different scenarios, you are also welcome to write your own players.

There will be always five command-line arguments given to player with the first two being x and y positions of the door. The other three arguments may change with different players.

In a turn, the player can attack one or more monsters simultaneously. These attack values will be sent to the monsters as damage value in the messages. Note that different players can use different strategies e.g. a player can be over-aggressive and try to kill every monster without seeking for the door while another one can try to avoid them completely and only try to reach to the door.

2.2.1 Sample Player

The sample player takes x and y positions of the door, maximum attacks it can do in one turn, range of its attacks and in which turn to leave after receiving a message from the game world:

```
> ./player 6 1 4 3 2
```

where door is at (6,1) , 4 is the maximum number of monsters the player can attack in one turn, 3 is maximum the distance between it and the monsters it can attack, and 2 is the turn number the player will leave after receiving a message.

Turn number starts from 1. When the last argument is 0, the player will play the game without leaving. So in the example above the player will leave after receiving the second message from the game world.

The health of the sample player is 10, its defence is 5 and its attack is 7. It uses the same formula as the sample monster below to calculate its health.

The strategy of the sample player is relatively simple. If there are monsters in its range, it will attack as many monsters it can (with the maximum of given number) starting from the one having lowest coordinates. Otherwise it will try to move closer to the door. If there are multiple directions which may or may not be optimal but still gets the player closer to the door, it tries to choose different directions as much as it can. Note that this is only to explain the behaviour of the sample player. You are not required to implement a player for your homework.

2.3 Monsters

Monsters' main objective is to kill the player before reaching the door. Beside making sure the monsters and the game world communicating properly with the correct message and responses, you will implement a sample monster program. Monsters will always take four command-line arguments. An example execution is:

```
./monster 10 1 4 3
```

- 10 is the health of the monster
- 1 is the damage induced to the player when the monster attacks
- 4 is the defence of the monster. When the player attacks the monster, this value will be reduced from the damage before removing from the health
- 3 is the range of the attack

When the player is in the range of the monster, if the distance (no diagonal moves) between two is less than or equal to the range, the monster will attack. If not, the monster will try to get closer to the player by moving one of the adjacent cells: up,upper-right,right,bottom-right,down,bottom-left,left,upper-left. When there are more than one directions getting monster to the best distance, the monster should choose its direction according the list above with up being the first choice and upper-left being the last. Both range check and distance measurements are based on Manhattan distance: If coordinate of the player is (x, y) and the coordinate of the monster is (x', y') the Manhattan distance between two is $|x - x'| + |y - y'|$.

You can use the following formula to calculate the new health of the monster when attacked by the player:

$new\_health = old\_health - max(0, damage - defence)$

The game world, player and monster executables rely on each others' messages and do not make consistency assumptions. The only controls enforced by the game world are:

- Entities cannot move to wall cells.
- Entities cannot move out of boundaries.
- Monsters cannot move to the door.
- Entities cannot move to a position occupied by another entity
- If the move is invalid game control returns a different position from target (your implementation will return the current cell position)

Similarly game world does not calculate health of entities, simply trusts them.

Note that although both the sample player and the sample monster advances only to an adjacent cell in the next move, in other implementations of player/monsters may move in arbitrary steps. For example a monster/player executable can move in range of 3 cells distance or even teleport itself from one corner cell to another corner cell. Or maybe in another game world implementation, instead of canceling the invalid movement completely, it might send the nearest valid cell back. Like the movement, the health calculation formula is only for the sample monster type you will implement. Different monster types can have different health formulas like gaining some lost health over time. So your game world or monster code should not assume anything about other entities except the message/response structures and assume the information they received are correct.

2.4 Communication

The communication between the game world to player and game world to monster 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 game world should be serving a player and an arbitrary number of monsters (with the maximum number of MONSTER_LIMIT which is defined in message.h). That means, it should create **n** pipes and read requests from **n** different file descriptors.

The game world pseudo-code is given as:

```
while game is not over:
    serve message to player
    read response from player
    serve messages to all monsters
    for all alive monsters:
        read response
```

2.4.1 Messages

The message structure between the game world, player and monsters is defined in the `message.h` file.
coordinate structure which is used in player/monster messages/responses is defined as:

```
typedef struct coordinate {
  int x, y;
} coordinate;
```

Note that the map coordinates starts from (0,0).

Game World - Monster Messages
The messages between the monster and the game world are defined with the following structure:

```
typedef struct monster_message {
  coordinate new_position;
  int damage;
  coordinate player_coordinate;
  bool game_over;
} monster_message;

typedef enum monster_response_type {
  mr_ready,
  mr_move,
  mr_attack,
  mr_dead,
} monster_response_type;

typedef union monster_response_content {
  coordinate move_to;
  int attack;
} monster_response_content;

typedef struct monster_response {
  monster_response_type mr_type;
  monster_response_content mr_content;
} monster_response;
```

The game world sends `monster_message` struct to the monster process and receives `monster_response` struct in response. In the message:

- **new_position**: the current position of the monster in the map. If the monster did not move or its move is unsuccessful in previous turn, its value should not change. The first message from the game world to the monster should contain the initial position of the monster.
- **damage**: the damage inflicted by the player to the monster
- **player_coordinate**: current coordinate of the player in the map
- **game_over**: the flag to acknowledge the game is over. After this, the monster process should terminate.

There are four types of responses from the monster to the game world:

1. `Sending Ready`: When the monster process is initialized and ready to communicate with the game world, it sends a response with **mr_ready** as **mr_type**. **mr_content** may be any value but initializing to 0 is advised. It is only sent before receiving any message from the game world.

2. `Moving`: when the monster wants to move to a cell, it sends a response with **mr_move** as **mr_type** and coordinate of the cell as **mr_content.move_to** .

3. `Attacking`: When the monster wants to attack to the player it sends a response with **mr_attack** as **mr_type** and attack value as **mr_content.attack** .

4. `Dying`: When a monster is dead, it sends a response with **mr_dead** as **mr_type** and terminates. **mr_content** may be any value but initializing to 0 is advised. After the monster is dead, it should not be printed on the map and its position should not be sent to the player. Note that even if the monster sends dead signal in previous turn, until its response is processed you should assume that it is still alive. So another monster can only move to the cell of the dead monster only if its response priority is less than the dead monster. The game world should wait for the monster to terminate before starting the next turn.

Game World - Player Messages
The messages between the player and the game world are defined by the following structures:

```
typedef struct player_message {
  coordinate new_position;
  int total_damage;
  int alive_monster_count;
  coordinate monster_coordinates[MONSTER_LIMIT];
  bool game_over;
} player_message;

typedef enum player_response_type {
  pr_ready,
  pr_move,
  pr_attack,
  pr_dead,
} player_response_type;

typedef union player_response_content {
  coordinate move_to;
  int attacked[MONSTER_LIMIT];
} player_response_content;

typedef struct player_response {
  player_response_type pr_type;
  player_response_content pr_content;
} player_response;
```

The game world sends `player_message` struct to the player process and receives `player_response` struct in response. The message structure is similar to the monster message.In the message:

- **new_position**: It is same as the **new_position** in the **monster_message**. The position of the player currently in the map. If the player did not move or its move is unsuccessful, its value should be the former position of the player.The first message from the game world to the player should contain the initial position of the player.
- **total_damage**: The total damage inflicted on the player by the monsters in the previous turn. It will be 0 for the first turn.
- **alive_monster_count**: The number of alive monsters on the map.
- **monster_coordinates**: The positions of all of the alive monsters sorted by their coordinates. e.g. if the MONSTER_LIMIT is for 4 and there are only 2 alive monsters at (3,1) and (2,8) the value of **monster_coordinates** should be:
  {(2,8),(3,1),-,-}[2].
  Note that there should be no dead monsters before or between alive monsters. So if the monster limit is 5 and there are 4 alive monsters at (1,1),(1,2),(2,1),(2,3) and (1,1) and (2,1) dies at the current turn, in next turn coordinates array should be {(1,2),(2,3),-,-,-}.
- **game_over**: the flag to acknowledge the game is over. After this, the player process should terminate.

There are five types of responses from the player to the game world:

1. `Sending Ready`: When the player process is initialized and ready to communicate with the game world, it sends a response with **pr_ready** as **pr_type**. **pr_content** may be any value but initializing to 0 is advised. It is only sent before receiving any message from the game world.
2. `Moving`: When the player wants to move to a cell, it sends a response with **pr_move** as **pr_type** and coordinate of the cell as **pr_content.move_to**.
3. `Attacking`: When the player wants to attack the monsters, it sends a response with **pr_attack** as **pr_type**. Unlike the monsters, the player can attack multiple entities at once. So **pr_content.attacked** should contain the array of all attacks. Order of attacks should be same as
   player_message.monster_coordinates. For the monsters that are not being attacked, this value should be 0. So for the coordinates {(2,8),(3,1),-,-}, if the player only attacks the monster at (3,1) with 5, the value of the array should be:
   {0,5,-,-} .
4. `Dying`: When the player is dead, it sends a response with **pr_dead** as **pr_type** and terminates. **pr_content** may be any value but initializing to 0 is advised. This is a lose condition for the player.
5. `Leaving`: This response is different from others since it is done by not sending any response to the game world. Instead, when the game world waiting a response from the player after sending a message, the player just closes the pipe and terminates. This is a lose condition for the player.

When a game condition is met before sending a message/after processing a response the game world should not wait for the end of the turn. When that happens, the game world should send all entities game over signal without waiting for their order in the turn and end the game loop.

[2]- can be any value, although initializing it to the default is advised

3 Input & Output

The game world reads the map information from the standard input and prints game information to the standard output. For output, you are provided with a library that outputs information in a strict format.

3.1 Input

The input to your program should be:

```
<width_of_the_room> <height_of_the_room>
<x_position_of_the_door> <y_position_of_the_door>
<x_position_of_the_player> <y_position_of_the_player>
<executable_of_the_player> <argument_3> <argument_4> <argument_5>
<number_of_monsters>
<first_monster_executable> <symbol> <x_position> <y_position> [<arguments>]
<second_monster_executable> <symbol> <x_position> <y_position> [<arguments>]
...
...
<last_monster_executable> <symbol> <x_position> <y_position> [<arguments>]
```

The first two arguments of a player will always be the x and y positions of the door. The last three arguments will be given in the same line with the the player executable. The number of monster arguments will always be four. The monster symbol is an ASCII letter to print in map output.
A sample input is:

```
6 8
2 0
3 3
./player 3 2 0
2
./monster s 3 5 5 5 1 2
./monster J 1 1 10 6 4 2
```

With this input the game world will execute:

- ./player 2 0 3 2 0
- ./monster 5 5 1 2
- ./monster 10 6 4 2

processes.

3.2 Output

The output of the game world should be printed **only** using functions provided in `logging.h` and `logging.c`.

3.2.1 Output

The header file is given in below:

```
#ifndef LOGGING_H
#define LOGGING_H

#include "message.h"

typedef struct map_info {
  int map_width, map_height;
  coordinate door;
  coordinate player;
  int alive_monster_count;
  char monster_types[MONSTER_LIMIT];
  coordinate monster_coordinates[MONSTER_LIMIT];
} map_info;

void print_map(map_info *mi);

typedef enum game_over_status {
  go_reached,
  go_survived,
  go_died,
  go_left,
} game_over_status;

void print_game_over(game_over_status go);

#endif
```

`map_info` struct contains the all information needed to draw world map. Although the struct is mostly self-explanatory there are several important things to notice:

1. The walls and the door are at the boundaries of the room. So if the room size is 3×3, all tiles except the single cell in the middle (1,1) will be walls or the door.
2. The initial input should not contain erroneous cases with a monster being on the same cell with the player or another monster. No checks are needed.
3. monster_types and monster_coordinates must be ordered by the coordinates of the monster and there should be no dead monster before or between the alive monsters similar to the coordinates/attacks in the player message/responses.

The function `print_map` is used to print the map. In the output '`@`' is the player, '`#`' is the wall, '`+`' is the door, '`.`' is the empty space and all other ASCII letters are possible monsters. There may be more than one monsters to have the same type (letter), so monster symbols are not required to be unique in the input.

For the example in 3.1 section, the initial output of the function should be:

```
##+###
#J...#
#....#
#..@.#
#....#
#..s.#
#....#
######
```

At the end of the game after printing the last status of the game map using `print_map` you should call `print_game_over` before exiting. There are four different types of game over messages:

1. **go_reached**, when the player reached the door
2. **go_survived**, when the player eliminated all monsters
3. **go_left**, when the player left the game
4. **go_died**, when the player died

4 Specifications

- Your codes must be written in C or C++.
- Your programs will be compiled with gcc or g++ and run on the department inek machines. No other platforms and/or gcc/g++ versions etc. will be accepted. Therefore make sure that your code compiles and executes on inek machines before submission.
- Do not forget to close the pipe ends and reap (see `wait` & `waitpid`) the player and monster processes. Note that pipes are only freed when there is no process keeping them open. All parties that do not use a pipe end should close it. Also do not forget to close the original pipe file descriptors after duplicating them as standard input and output.
- Your program must not leave any zombie processes. If you leave zombie processes in a test case, you will lose points.
- Do not make any changes to the codes provided to you and only use the given message structures for communication between the processes.
- Your code will be tested with black box inputs.
- **Using code, even partial, that is not your own is strictly forbidden and constitutes as cheating. This includes code from your friends, previous homework, or the internet. We have a zero tolerance policy on cheating.**
- Follow the course pages on COW and ODTUClass for any updates and clarifications. Please ask your questions on COW instead of e-mailing if the question does not contain code or solution.

5 Submission

Submission will be done via ODTUClass. You will submit a tar file called “hw1.tar.gz” that contain all your source code together with your makefile. You do not need to submit the libraries we have provided. Even if you do, they will be replaced with the originals, so do not make any changes on them. Your tar file should not contain any folders. Your makefile should be able to create two executables with the name of ’world’ and ’monster’ and it should be able to run using the following command sequence.

```
> tar -xf hw1.tar.gz
> make all
> ./world
```

You can assume that all the executables and library files are present for the required compilation and execution. **If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points.**