Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2017-2018

Homework 1 - Asynchronous Hunter Prey Modeling

Due date: 16 03 2018, Friday, 23:59

1 Introduction

The objective of this assignment is to learn the basics of process management, IPC and polling. Your task is to implement a hunter prey modeling system with asynchronous execution on a fixed sized two dimensional map with obstacles. There are three main components on the system; namely, the server, hunters and preys. The server, all the hunters and the preys should be executed in separate processes. Grid information is stored and updated on the server and each hunter and prey receive information from the server. After receiving the information, they request to move to a calculated location on the grid and inform the server of their new requested location. Hunters have predetermined energy and die after this energy is exhausted. Preys do not have predetermined energy, however they will provide some amount of energy if eaten. The system should continue until all hunters or preys are dead. Details of each component are explained in the following section.

2 Components and Communication

There are three main components of the system and your code should create three executables upon compilation. Details of these components and communication between them are explained in the following subsections.

2.1 Server

The server is the main program that creates and starts all hunter and prey processes. Upon starting, it should read the map width and height, obstacle/hunter/prey locations and their energy values from the standard input, then it should print the map according to the specifications given in section Input & Output. After printing, it should fork and execute (see `fork` and `exec` function family) all the hunter and prey processes. Hunter and prey programs should start with two arguments. These should be the width and height of the map. Before starting them, it should establish communication between them using bidirectional pipes (see Communication and redirect their standard input and output to this pipe using `dup2` function. After their creation, it should immediately send them their **state information**. **State information** is defined as the location, the location of their closest adversary (hunter for prey and prey for hunter), number of neighbouring objects (friendly objects or obstacles) and the location of these objects. Closest adversary should be calculated with Manhattan Distance (see formula 1). If there is more than one with same distance, you can use any selection criteria to select one. Hunter/prey processes only

use neighbouring cell information and their closest adversary to determine their next move. Only vertical and horizontal cells are counted as neighbours because prey and hunter can only move one tile vertically or horizontally.

After these initial steps it should run in a loop until only one type of processes remain. While in a loop, it should complete these tasks in the given order:

1. Check for any of the file descriptors (from pipe) connected to the child processes for an input to be ready. (use one of the `poll` or `select` system calls)
2. If there is data to be read on a file descriptor:[1]
   - Read the data
   - Check the requested new location of prey/hunter
   - If there is no collision with a friendly object then update the map. Hunters should also lose 1 energy at every successful move.
   - Send the new state information. In case of collision, state will include the same old location.
3. If a hunter is in the same position with a prey, kill the prey with a `SIGTERM` signal and add its energy to the hunter. Do not forget to close the pipe and reap (`(wait())` the process.
4. Check if any of the hunters have died from 0 energy. Kill the hunter process (`SIGTERM`) and reap them. (see `kill()` function and `wait` function family)
5. If the map is updated, print the updated map on stdout.

After this loop it should kill the remaining processes, reap them and terminate. There should be no zombie processes left. It should also close all the remaining socket connections.
Communication messages between server and hunter/prey processes is explained in detail in Communication section.

2.2 Hunter

Hunter process start with two arguments that are the width and height of the map. It should save these values of the map when first starting. These values are used to create valid move requests to the server. Since server redirects its stdin and stdout, it should communicate with server using these file descriptors. It should run in an infinite loop until terminated by the server. While in this loop, it should complete the following tasks:

1. Read the data (should block until data is ready)
2. Set/Update the location given in the data as the current location. At the first run this is the initial location. In the subsequent messages this will indicate whether the move request is granted.
3. For all neighbor tiles that are not occupied (obstacle or another hunter):
   - Check if the location is obstructed. If it is, move on to the next direction.
   - Calculate the Manhattan Distance from the current location to the closest prey from the data received from the server. Formula is

$$MH = |x_{prey} - x_{hunter}| + |y_{prey} - y_{hunter}| \quad (1)$$

   .

[1] Hunter or Prey. Server keeps track of the process type for each pipe end

- If the location is not obstructed, calculate the Manhattan Distance between the hunter and the closest prey from that location.
- if MH distance of neighbor is smaller than the current tile, it is a possible move.

4. If there are no possible moves, send the current location as the move request to the server.
5. If there are possible moves, select one and send it as the move request to the server using standard output. Any selection criteria is acceptable.
6. Sleep for a random time between 10 and 100 ms. (you can use `usleep(10000*(1+rand()\%9))` function call)

2.3 Prey

Prey process start with two arguments that are the width and height of the map. It should save these values of the map when first starting. These values are used to create valid move requests to the server. Since server redirects its stdin and stdout, it should communicate with server using these file descriptors. It should run in an infinite loop until terminated by the server. While in this loop, it should complete the following tasks:

1. Read the data (should block until data is ready)
2. Set/Update the location given in the data as the current location. At the first run this is the initial location. In the subsequent messages this will indicate whether the move request is granted.
3. For all neighbor tiles that are not occupied (obstacle or another prey):
   - Check if the location is obstructed. If it is, move on to the next direction.
   - Calculate the Manhattan Distance from the current location to the closest hunter from the data received from the server.
   - If the location is not obstructed, calculate the Manhattan Distance between the prey and the closest hunter from that location.
   - if MH distance of neighbor is bigger than the current tile, it is a possible move.
4. If there are no possible moves, send the current location as the move request to the server.
5. If there are possible moves, select one and send it as the move request to the server. Any selection criteria is accepted.
6. Sleep for a random time between 10 and 100 ms. (you can use `usleep(10000*(1+rand()\%9))` function call)

2.4 Communication

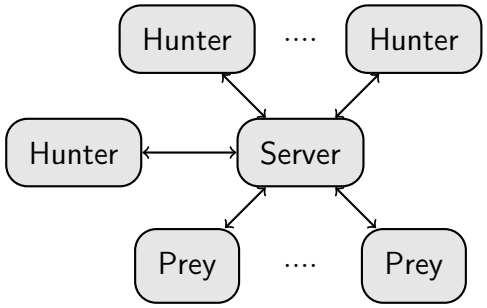
Communication between server and hunter/prey processes will be done using bidirectional pipes. Unfortunately, Linux pipes are not bidirectional. You can use the following defines to create sockets that behave very similar to pipes but bidirectional:

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

If you use PIPE(fd), data written on fd[0] will be read on fd[1] and data written on fd[1] will be read on fd[0]. Moreover, both file descriptors can be used to write and read data.

Server can serve arbitrary number of hunters and preys. That means, it should create n pipes and read requests from n different file descriptors. If it blocks in one of them the requests from others has to wait. In order to deal with such a case poll() or select() system calls can be used. The server pseudo code will be:

```
while there are children from different types:
    select/poll on pipe file descriptors of alive children
    for all file descriptors ready (have data)
        read request
        serve request
```



2.4.1 Messages

The messages between server and its children is defined and fixed. The message structure sent to the prey/hunter processes from server is defined below:

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

typedef struct server_message {
    coordinate pos;
    coordinate adv_pos;
    int object_count;
    coordinate object_pos[4];
} server_message;
```

where `pos` indicates prey/hunter position, `adv_pos` is the position of their adversary, `object_count` is the number of objects that is near the hunter/prey and final array is their locations. If there is less than 4 objects near the hunter/prey, then the remaining array should be ignored. The first position is used to indicate either the initial position at beginning or the acceptance of the move request after the request is received. The message structure sent to the server processes from the hunter/prey processes is defined below:

```
typedef struct ph_message {
    coordinate move_request;
}
```

It is a simple struct with only the coordinates for the move request should be sent to the server.

3 Input & Output

In this section inputs given to the program and the output that is being continuously printed on the screen are described.

3.1 Input

Input is given in this format:

```
<map_width> <map_height>
<# of obstacles>
<x coordinates of obstacle_1> <y coordinates of obstacle_1>
<x coordinates of obstacle_2> <y coordinates of obstacle_2>
...
...
<x coordinates of obstacle_n> <y coordinates of obstacle_n>
<# of hunters>
<x coordinates of hunter_1> <y coordinates of hunter_1> <energy of hunter_1>
<x coordinates of hunter_2> <y coordinates of hunter_2> <energy of hunter_2>
...
...
<x coordinates of hunter_n> <y coordinates of hunter_n> <energy of hunter_n>
<# of prey>
<x coordinates of prey_1> <y coordinates of prey_1> <energy of prey_1>
<x coordinates of prey_2> <y coordinates of prey_2> <energy of prey_2>
...
...
<x coordinates of prey_n> <y coordinates of prey_n> <energy of prey_n>
```

where the coordinate $(0, 0)$ is the top leftmost corner, x values indicate row and y values indicate column.

3.2 select() and poll()

select() and poll() system calls are used to block on multiple file descriptors simultaneously. In our example server should read data from N players. If it reads it one by one, first `read()` could block for a long time while another process already sent its request. These two system calls lets you block on *any* of the file descriptors. If at least one of them have data to read (or write), call will return the set of file descriptors that are ready so far. Then you can read those data without blocking.

3.3 Output

Output of the server when printing the map should be in this format:

```
+<width # of - symbols>+
|<row_0 contents>|
|<row_1 contents>|
|<row_2 contents>|
...
|<row_(height-1) contents>|
+<width # of - symbols>+
```

The representation for various objects is given below:

- Empty cell should be represented with a single space character.
- Hunter is represented with a single '`H`' character.
- Prey is represented with a single '`P`' character.
- Obstacle is represented with a single '`X`' character.

3.4 Example Input/Output

Input:

```
10 10
3
0 0
1 1
2 2
2
3 5 5
7 8 7
2
1 0 3
6 2 2
```

First output of this input:

```
+----------+
|X         |
|PX        |
|  X       |
|     H    |
|          |
|          |
|  P       |
|        H |
|          |
|          |
+----------+
```

4 Specifications

- The codes must be in C. No C++ codes are accepted.
- Your programs will be compiled with gcc and run on the department inek machines. No other platforms and/or gcc versions etc. will be accepted, Therefore make sure that your code works on inek machines before submitting it.
- Beware of timing. After printing something, `fflush()` to stdout to ensure that output is not buffered.
- Do not forget to close the socket connections and reap (see `wait` & `waitpid`) the hunter and prey processes after they have died.
- Your program must not leave any zombie processes. If you leave zombie processes in a testcase, you will lose half the points for that testcase.
- Your code will be tested both with black box inputs. Because of the unpredictability of the system, black box will only check whether you have any errors not actual output matching. Errors include leaving open sockets, leaving zombie processes, multiple moves between two prints. We will also test your hunter/prey processes without server to make sure they can be run independently and correctly.
- Using any piece of code that is not your own is strictly forbidden and constitutes as cheating. This includes friends, previous homeworks, or the internet. The violators will be punished according to the department regulations.
- Follow the course page on Piazza for any updates and clarifications. Please ask your questions on Piazza instead of e-mailing if the question does not contain code or solution.

5 Submission

Submission will be done via COW. You will submit a tar file called "hw1.tar.gz" that contain all your source code together with your makefile. Your tar files should not contain any folders. Your makefile should be able to create three executables named server, hunter and prey in the same folder and run using the following commands.

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

If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points. However, if you do not create all three executables, you will not be graded and will receive 0.