REAL TIME AND EMBEDDED SYSTEMS

Mitigation of a Delay Using an ACM

Newcastle University

Reuben Atherton

Contents

I. Introduction

This paper will look at how Asynchronous Communication Mechanisms (ACMs) can be used to mitigate the bottleneck effect in a 'First In First Out' (FIFO) data pipeline. A game will be created whereby the user controls a character, in this case a fox, displayed by the symbol '>' with the intention of catching a second character, a rabbit displayed by the symbol '@'. The game is introduced into the FIFO pipeline, a queue like process that executes programs in a consecutive order. Next a delay is added to the pipeline after the game, in order to cause a deliberate data accumulation resulting in a game lag. These effects are observed and discussed. Finally, an ACM is then implemented into the pipeline, between the game and delay in such a way that reduces the effects of data accumulation, making the game playable at the expense of minor frame loss.

1. Aims (education and technical)

- To understand how threads can be used to create concurrency within asynchronous processes.
- To understand the bottleneck effect involved with pipelines and demonstrate how data accumulation can create a delay.
- To understand how ACM's can be used to mitigate the effects of data accumulation.

2. Objectives

- To create the terminal game in which the user controls a fox (symbol '>') attempting to catch a rabbit (symbol '@').
- Add multithreading using the 'pthread.h' C programming library to create concurrency.
- Create a delay program that causes a data accumulation in a slow-interfacing FIFO buffer.
- Calculate the rates of the game, the delay and the data accumulation, as well as the time delay and ACM delay associated with them.
- Using the calculations, design and implement an ACM into the pipeline between the game process and the slow interface buffer to counteract the latency effect caused by the pipeline delay.
- Demonstrate these effects by comparing terminal outputs using various terminal commands.
- Create Petri Net models of the different systems.

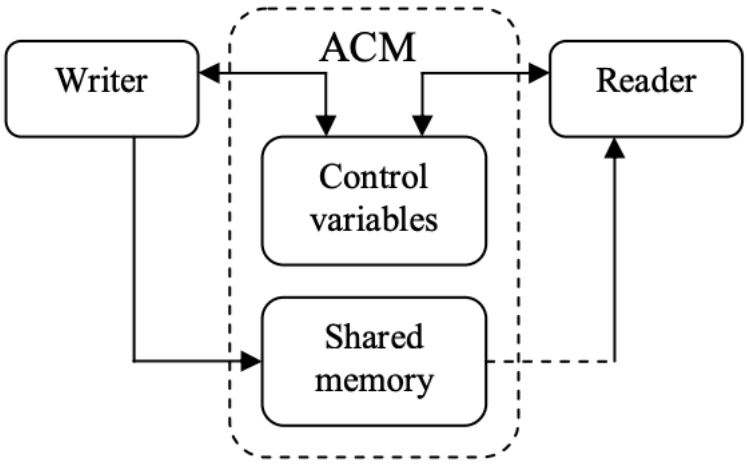
3. Background Theory

a) Bottleneck Effect

This term refers to the instance where one component with the lowest throughput struggles to keep up with rest of the system, resulting in a deteriorating system performance [1]. Due to the large impact bottlenecks have, identifying and reducing/removing them is a huge focus in systems engineering.

b) ACM Theory

The Asynchronous Data Communications Mechanism (ACM) boasts the ability to manage the transfer of data between two processes, regardless of their synchronicity. The "writer" provides the data whilst the "reader" uses the data. The third variable is the "last write" variable which will be assigned the last data item written to the ACM. The following diagram depicts the general workings of the ACM [2].



H. R. Simpson initially proposed a Pool using 4 data slots however recently, when assuming certain conditions, this number of slots can be reduced to 3. The ACM used in this project is that of the reduced, 3 slot signal type ACM. This was chosen due to its capacity for overwriting and non-re-reading attributes. There are different ACMs with different capabilities and these are chosen based on the task at hand. A more specific in-depth explanation of the ACM and its internal dynamics is given in Section II Part 4 ACM.

II. Experiment

1. Creating the Game

a) General

The user will control the fox, chasing the rabbit. The rabbit will move randomly and independent of the fox due to threads included within the 'pthread.h' library. The fox will move toward to the right (East) as soon as the program begins (due to certain variable initialisations) and will continue to move in the last entered direction until the user enters a new direction. Both characters will now continuously move until they collide. Each character will move one coordinate position every 0.5 second. On-screen limits have been set, creating an area of which the characters cannot escape, explained below. The printing to the terminal is a standard output stream and escape characters are used to separate the characters into their individually updating coordinates.

b) User Control

The fox will be controlled using the arrow keys, recognised by the program through the 'getchar()' function. The arrow keys require this function to be called three times in order to

return their respective ASCII number. To achieve this, a simple $for()$ loop was used, iterating three times. Each return value of the getchar() is assigned to a globally defined character value which is then fed into a switch case to update the coordinates of the fox. An additional global counter has also been added with the intention of displaying the number of directional changes at the end of the game made by the user; this simply adds to the quality of the game. To prevent any data inconsistencies, a mutex lock has been added which protects this .c file's segment of memory from other threads.

c) Screen Limits

Although there already existed screen limits within the template, they did not prevent the coordinates from being updated outside of such limits, only the position of the character. Consequently, the characters would sometimes become stuck on the screen boundaries whilst the coordinates would update to 'restricted' values making it impossible for the user to know exactly where the character was. To ensure not only the characters do not leave the screen but also their respective coordinates are not updating outside the limits, the following was added:

```
if(1 <= *y_ptr <= height)
    if(*y_ptr == 0) *y_ptr = 1;
    else if(*y_ptr == (height+1)) *y_ptr = height;
    (*y_ptr)--;
    break;
```

Fig 1.0 Screen Limits

This is the Y coordinate application but the same was also added for the X coordinates. Whenever the Y coordinates are updated to a value of 0, they will be immediately reset back to the minimum value of 1. Similarly, when the coordinates breach the upper Y coordinate limit, they are reset to the maximum height. The same concept applies in the X direction. This added functionality proved to work well.

d) Rabbit direction

The random movement of the rabbit was simply accomplished by using a random number generator between 0 and 3. These numbers were then assigned to local variable '$rabbit\_dir$' to then be fed into the switch case again in order to updated the rabbit coordinates.

```
//Generate random direction for rabbit
srand(time(NULL));
int rabbit_rand = rand() % 4;

if(rabbit_rand == 0) rabbit_dir = 'u';
if(rabbit_rand == 1) rabbit_dir = 'd';
if(rabbit_rand == 2) rabbit_dir = 'l';
if(rabbit_rand == 3) rabbit_dir = 'r';

update_coord(&rabbit_x, &rabbit_y, rabbit_dir);
```

Fig 1.1 Rabbit Direction

e) Catching the Rabbit

Originally, for the rabbit to be caught, the coordinates of both the fox and the rabbit needed to be the same, hence the fox had caught the rabbit. This however presented the problem in the possibility of the two characters swapping coordinates. For example, the fox would be at (4,7) and the rabbit at (5,7) travelling in the same direction such that the new coordinates of the fox becomes (5,7) and the rabbit (4,7). This would not be recognised as a 'catch' even though the fox has 'overlapped' the rabbit and essential caught it. To prevent this bug, ranges were created around each character in the X and Y direction, where if the fox enters the coordinates included within this range, the rabbit is recognised as caught and the game is over. This also makes the game easier and allows testing of the latency etc. to be completed quicker. Again, the novel idea worked perfectly and proved to be a very useful addition.

```
// ****** CREATES A RANGE FOR THE CAPTURE OF THE RABBIT ****** -- this removes problem when coordinates swap instead of equalling
int y_range_low_rabbit = (rabbit_y-1);
int y_range_high_rabbit = (rabbit_y+1);

int x_range_low_rabbit = (rabbit_x-1);
int x_range_high_rabbit = (rabbit_x+1);

if((y_range_low_rabbit <= fox_y && fox_y <= y_range_high_rabbit) && (x_range_low_rabbit <= fox_x && fox_x <= x_range_high_rabbit)) {
    break;
}
```

Fig 1.2 Catching the Rabbit

2. System Design and Calculations

The relevant variables to consider in this project are the following:

The rate of the game _____ r_g
The rate of the delay, per character _____ r_d
The rate of the data accumulation _____ r_b
The number of bytes being printed to terminal _____ n
Time, delay _____ t_{delay}
Time, acm _____ t_{acm}
Latency _____ L

Firstly, we must find r_g using the equation $r_g = \frac{n}{t}$ where n is the number of characters printed in the terminal; this however, raises an issue. When printing characters into the top left of the display, they will take on the coordinates where the X and Y values are each single digit, i.e. (2,4). As the game progresses however, the characters may move into the lower right of the display and so inherit larger coordinates, possibly taking 2 characters to print each of the X and Y values, i.e. (11,16). Taking this into consideration, we must allow for both a minimum and a maximum n meaning we will also be left with an $r_{g\,MIN}$ and $r_{g\,MAX}$. With this in mind, we count the minimum printed characters as 18, and the maximum number of characters as 22 (not including the escape character). We also know that the game has a delay of 500 000µs = 0.5 seconds, where this is the value used for t. Using the equation, we therefore get:

$$r_{g\,MIN} = \frac{18}{0.5} = 36\ characters\ per\ seconds$$

$$r_{g\,MAX} = \frac{22}{0.5} = 44\ characters\ per\ seconds$$

Next, a value is chosen for r_d such that we see a noticeable lag in the game. In this experiment, r_d was chosen to be 27 characters per second, 25% less than the original minimum of 36 characters per second.

The first t value to calculate is the t_{delay}, simply given by:

$$t_{delay} = \frac{1}{r_d} = \frac{1}{27} = 0.037037\mu s$$

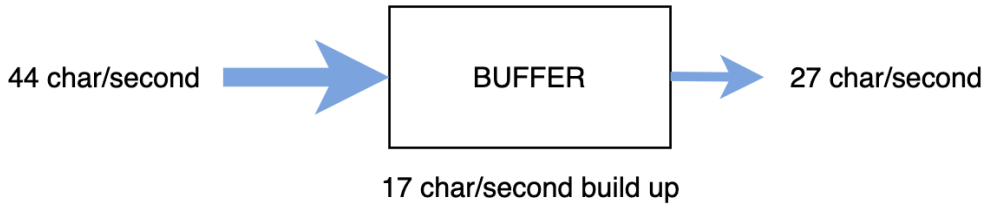
This value will be used inside 'usleep' function within the delay.c program. The second t value is t_{acm} which will be included in the ACM program. The maximum t_{acm} can then be calculated, using the following equation:

$$t_{acm} = \frac{n}{r_d} = \frac{22}{27} = 0.814s = 814\,000\mu s$$

Implementing t_{delay} will now cause data accumulation in the buffer because $r_g > r_d$ where there is a greater number of characters entering the buffer per second, than there are leaving. The rate of data accumulation, r_b, is simply the difference between the two rates, as follows:

$$r_b = r_g - r_d = 44 - 27 = 17\ characters\ per\ second$$

This value is the number of characters left in the buffer each second, resulting in a bottleneck effect.



17 char/second build up

Finally, to calculate the latency we use $L = n * \frac{1}{r_d}$, where n is the number of data items inside the buffer after time t, given by $n = r_b * t$. The variable t has been chosen as 5 seconds giving the equation:

$$n = r_b * t = 17 * 5 = 85\ characters\ after\ 5\ seconds$$

Latency is therefore given by:

$$L = n * \frac{1}{r_d} = 85 * \frac{1}{27} = 3.148 \approx 3.1\ seconds$$

This latency can be seen in the following sections using various terminal command methods.

3. Implementing Delay

As can be seen from the screenshots below, the positioning of the fox and the rabbit get progressively more and more dissimilar. The un-delayed version of the game can be seen on the left side of the screenshots, whilst the right side includes the delay. The images were taken in consecutive order meaning Fig 1.3 shows the program after only a few seconds of execution, meaning there will be fewer data items in the buffer and therefore less difference in positioning when comparing the two terminals. Fig 1.4 was taken 5 seconds time after the program started, demonstrating an expected latency of roughly 3.1 seconds between terminals.

It should be noted to achieve this parallel execution of the game, a fifo was created using the command $mkfifo\ myfifo$ and then added to the pipeline using the command $|\ tee\ myfifo\ |$ in between the game and the ACM terminal execution commands. The second terminal window must also call $cat\ myfifo$ to achieve these results.

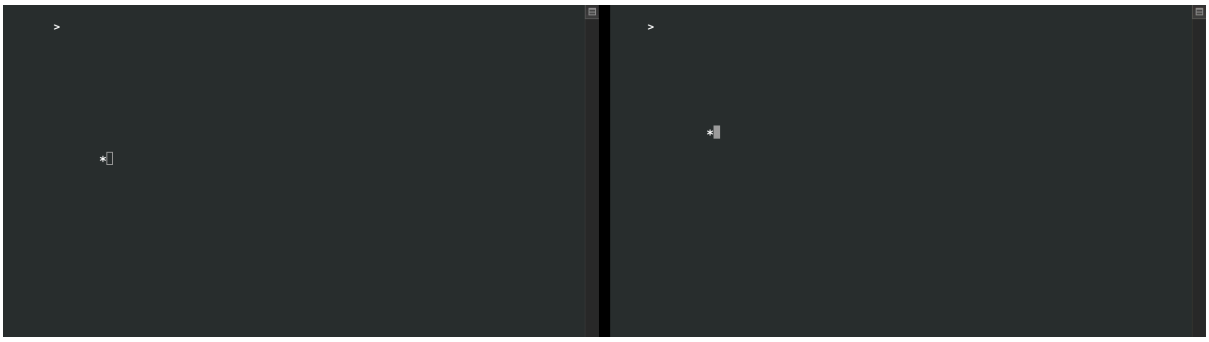


Fig 1.3 Screenshot of the delay after a short time

Fig 1.4 Screenshot of the delay after 5 seconds

This whole premise for this project is to mitigate this delay using an ACM, as explained below.

4. ACM

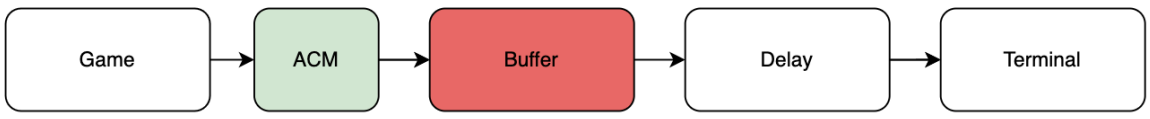


Fig 1.5 Pipeline Diagram

The ACM will be added just after the game to reduce the effect of the buffer. When discussing the ACM, there are three variables necessary to talk about, as well as their initialisations:

write,	$w = 0$
read,	$r = 1$
last write,	$l = 2$

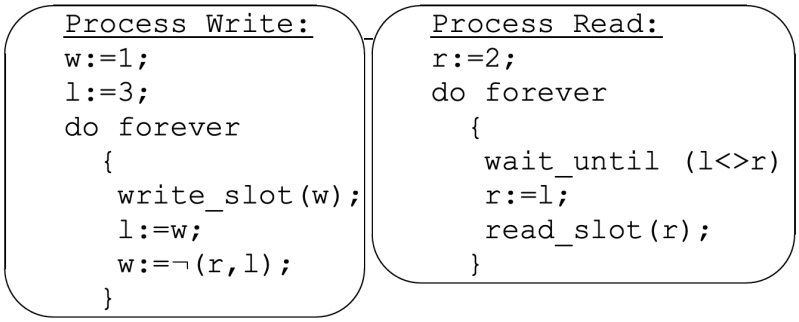


Fig 1.6 Pipeline Diagram

Fig 1.6 gives the pseudo code of the 3 slot signal type ACM. The code segments below are based on this pseudo code, as is the Petri Net model that follows in Part 5.

a) Reader Thread

```
int w=0, r=1, l=2; // define and initialise the variables as in the handout

void reader_thread () {

  while(1) { // reader loop; includes a delay to reduce the output data rate

      while(l == r);   // wait until l != r condition is true

      r = l;

      printf("%s", slots[r]); // access slot; slots[i] is a pointer to slots[i][0] READ
                              // (slots[i][0],slots[i][1],... is a 0-terminated string)
      fflush(stdout);

      usleep(814000);  // limit output rate; calculate your own value (>500000) talking about frames, should be similar to game usleep value
                       // should be a smaller that the game by the same percentage that rd is < rg
                       // should be a trade off
                       // do the maths for a lag of 7 seconds
      //printf("%d", r);

    }

}
```

Fig 1.7 Reader Thread

In the pseudo code, the initialisations are different for simplicity of understanding whereby they reference the human-indexing of slots and not the standard-computer-indexing of the slots. Therefore, 1 is subtracted from each value to give the actual initialisations required i.e. w=1 is changed to w=0. To maintain fresh inputs, the read process runs in a continuous loop whilst l and r are the same, achieved using '$while(l == r)$'. When they become different, the loop is by-passed, r is set to equal the value of l and the read slot is executed. As shown above in Fig 1.7, the t_{acm} value is implemented in the $usleep()$ function. This delays the program for 0.814 seconds, mitigating the effects of the pipeline effect as previously discussed.

b) Writing Thread

```
int main () {
  //...; // variable declarations/initialisation, if needed

  pthread_t read_thread;

  pthread_create(&read_thread, NULL, (void *) reader_thread, NULL);   //expecting a NULL pointer. Very dangerous, can point to any type

  while (1) {  // writer loop

// access slot; modify this according to the output format of your game
// in this example I keep reading until '*' appears,
// then the symbol 0 is added, which is the sting terminator.
// The terminator is needed for printf("%s",...) of the reader.

      printf("%d", l); // for testing the ACM on its own

      int j = 0;
      while ((slots[w][j++] = inp ()) != '*'); // the actual computation takes place inside the condition WRITE
      slots[w][j] = 0; // append the terminating symbol to the string

      l = w;

      w = LUT[r][l]; // Implement the look up table

    }

  pthread_cancel(read_thread);

  return 0;
}
```

Fig 1.8 Writing Thread

The writing process is similar in that it also runs in a continuous $while()$ loop. The l value is then set to equal w. Next is the implementation of the Look Up Table (LUT). This is the programming equivalent of a basic NOT operation in that the two inputs are NOT the output. The global array is designed as such:

```
const int LUT[3][3] =  {{2, 2, 1},
                        {2, 0, 0},
                        {1, 0, 1}};
```

Fig 1.9 Look up table

The value at each coordinate inside the 2D, 3x3 array, is chosen such that it is not equal to either the X or Y coordinate value. For example, inside the array coordinate (0,1) must be 2 because 0 and 1 are included in the coordinates. There is the exception of the positions (0,0), (1,1) and (2,2) where the value here can be one of two numbers. In each of these cases, one of the two possible numbers were chosen and assigned to that coordinate. Another way to achieve this would be to create a random number generator between the two values however it was decided, for this project, to use the simpler method previously explained. To implement the LUT, w is given the value of LUT array in the following way:

```
w = LUT[r][l];
```

Fig 2.0 Calling the look up table

Upon testing of the ACM, the delay was drastically reduced shown by Fig 2.1 below. The left side is again the un-delayed, original game whilst the right-side pipeline includes game, ACM and delay. As can be seen the difference in positions of all respective characters is very similar, confirming that the ACM successfully mitigated the delay, making the game playable once more.

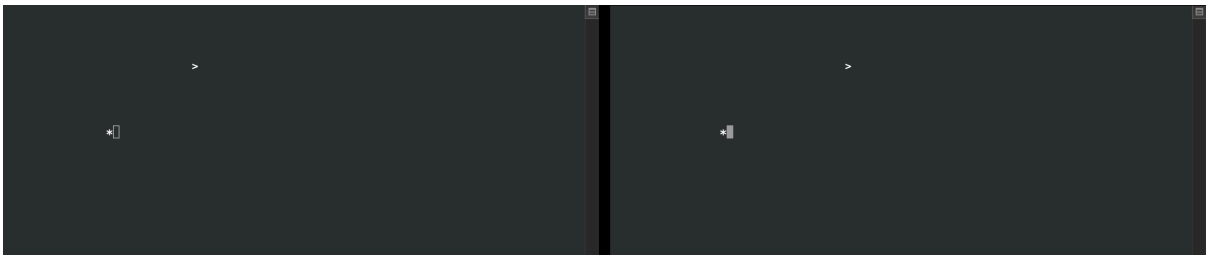


Fig 2.1 Reduced delay due to ACM

5. Petri Net Models

The models found below are examples of Petri Net models that describe the processes involved with a system. The completely white circles show the different states of the system, the arrows represent transitions between states and finally the black token is the initialisation of a state. When a token moves between states it is said to have "fired" a transition. Where two arrows meet at a single point (not at the state but just before), this is called a "join" transition, performing synchronisation between threads. This transition can only be fired after all input tokens are present in the previous state and can be compared to an AND operation. A similar event is the "merge", occurring where one or more transitions are connected to a state. In this instance, only one of the transitions are required to initialise the state and fire the next transition, comparable to that of an OR operation. Finally, a "fork" occurs when one transition splits into two separate paths connected to two different states [3].

Fig 2.2 Petri Net model for the game

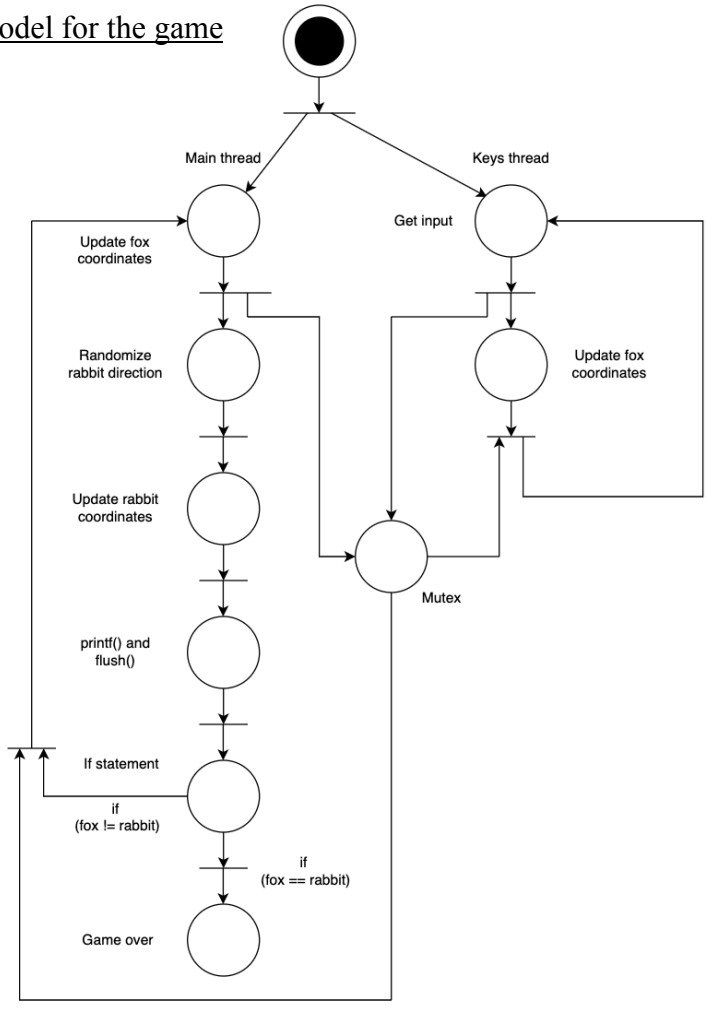
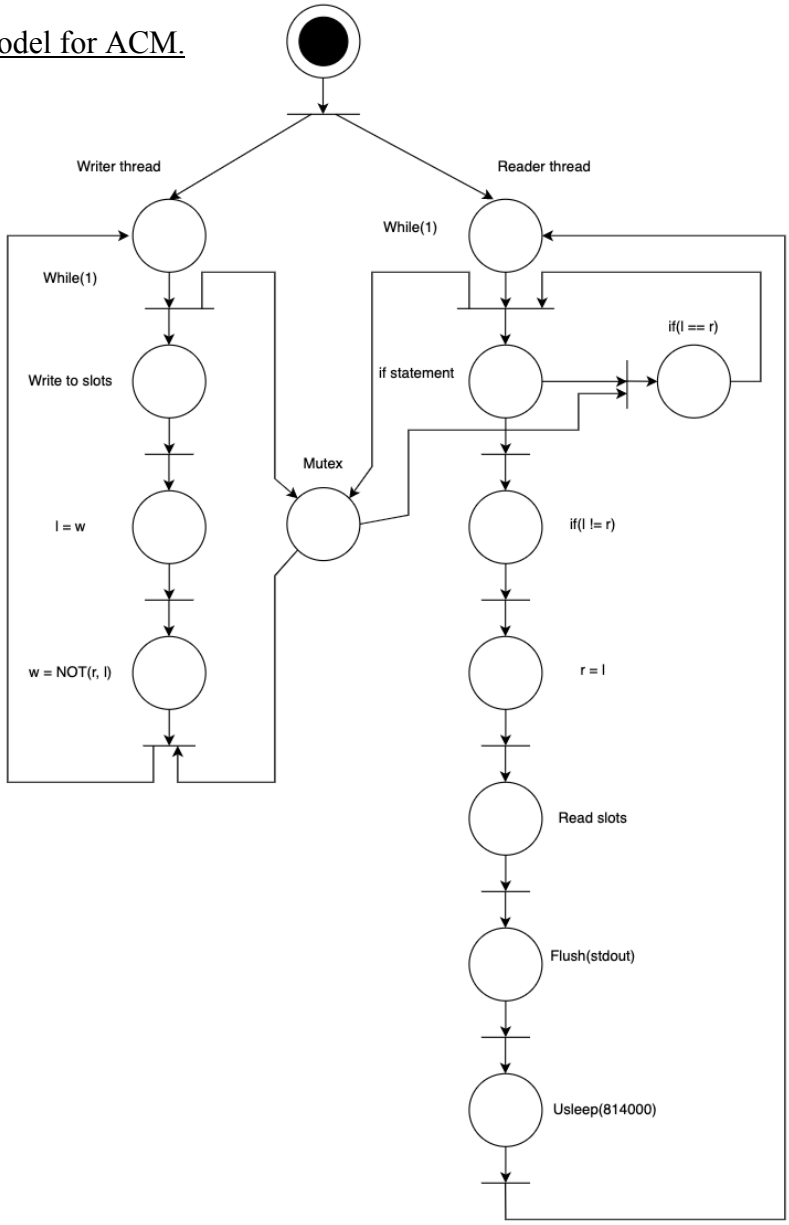


Fig 2.3 Petri Net model for ACM.



III. Conclusion

The game was created and the program successfully terminated upon the instance the fox satisfied the state in which the rabbit was classed as caught. The thread successfully allowed the continuous waiting for user input whilst concurrently, the rabbit coordinates were updated randomly and everything was printed to the screen. Various new methods, outside the original template were also added which improved the games quality and allowed for better gameplay. The delay was then added successfully, demonstrated by the two, side by side terminals with one displaying the un-delayed game whilst the other confirmed the effects of the data accumulation due to the combination of the FIFO buffer and the delay. The calculations were also correctly completed, making sure to adhere to the size of the data rates whereby $r_g > r_d > r_{acm}$.

Using the values calculated for the t_{acm} etc. the ACM was successfully designed and included in the pipeline, executing just after the delay. The effects of data accumulation were mitigated

and the latency in the game was severely reduced, with the termination of the delayed program now being at almost the same time as the un-delayed program. This was proved again by comparing terminals side by side showing the difference in character position. Reducing the delay came at the cost of partial frame loss however this was to be expected.

Finally, models of the game and the ACM were successfully created used the Petri Net style of flow diagram. This gives an overview of the internal workings of the code and can be useful when explaining or demonstrating what is happening within the system.

To conclude, the project has been successful with all intended aims and objectives being completed as described.

IV. References

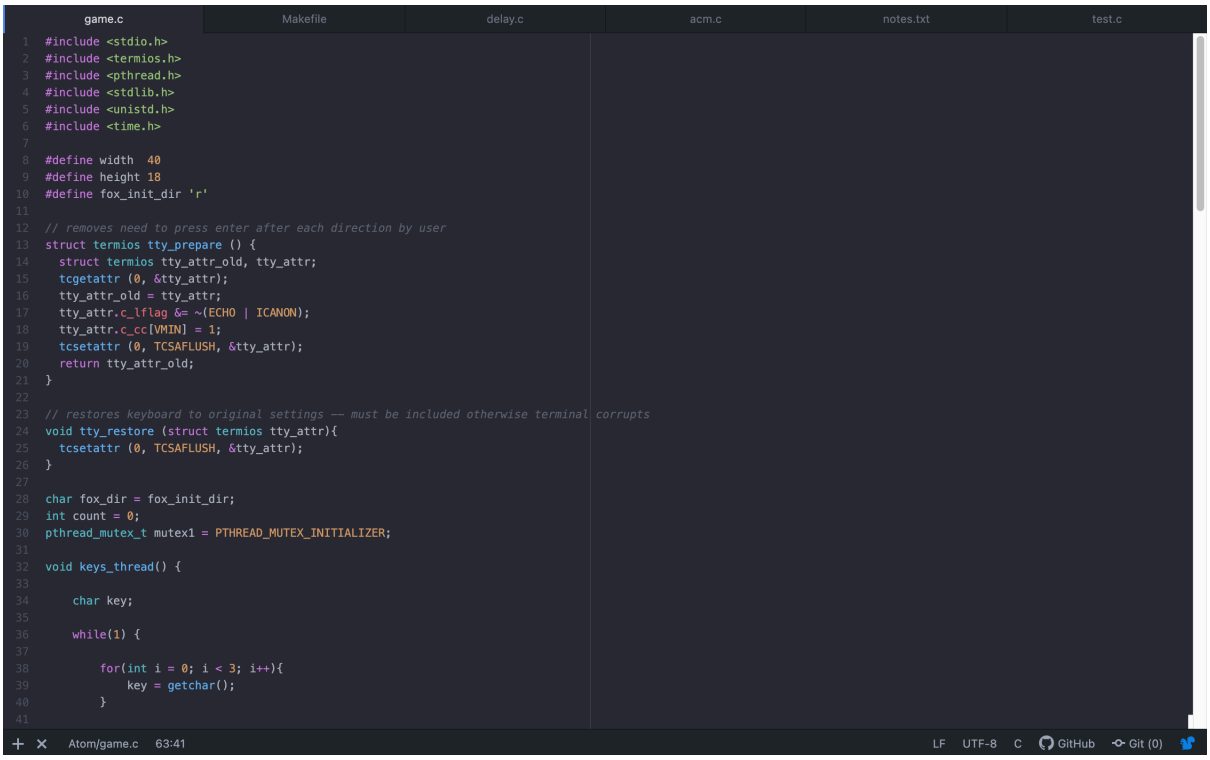
[1] Apica. “The 5 Most Common PC Bottlenecks”, [Online]https://www.apica.io/5-common-performance-bottlenecks/#:~:text=The%20term%20%E2%80%9Cbottleneck%E2%80%9D%20refers%20to,system%2C%20thus%20slowing%20overall%20performance.

[2] F.Hoa, Newcastle University, “Implementation of a Three-Slot Signal ACM”, [Online]: https://www.staff.ncl.ac.uk/i.g.clark/publications/UKForum-12-2002.pdf

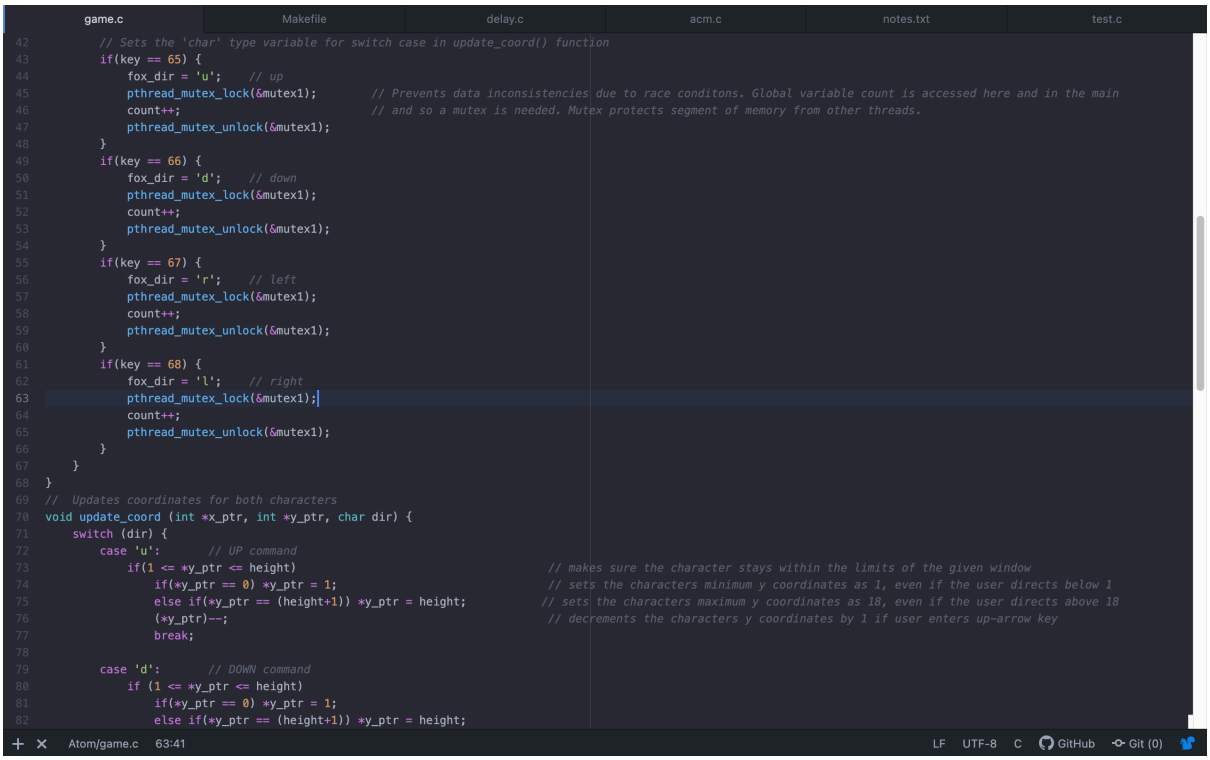
[3] A. Bystrov, Newcastle University, “Discrete Event Models Part 2: Petri Nets”, [Online] Real Time and Embedded Systems Notes.

V. Appendix

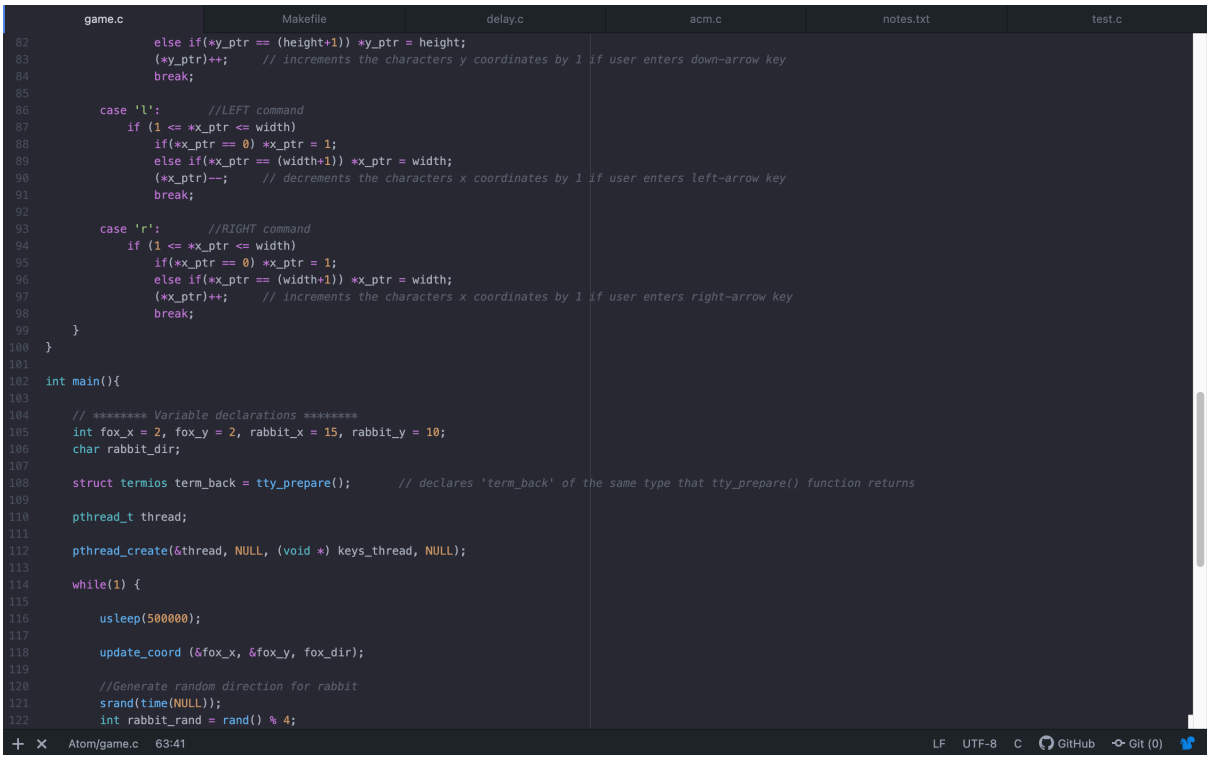
Game



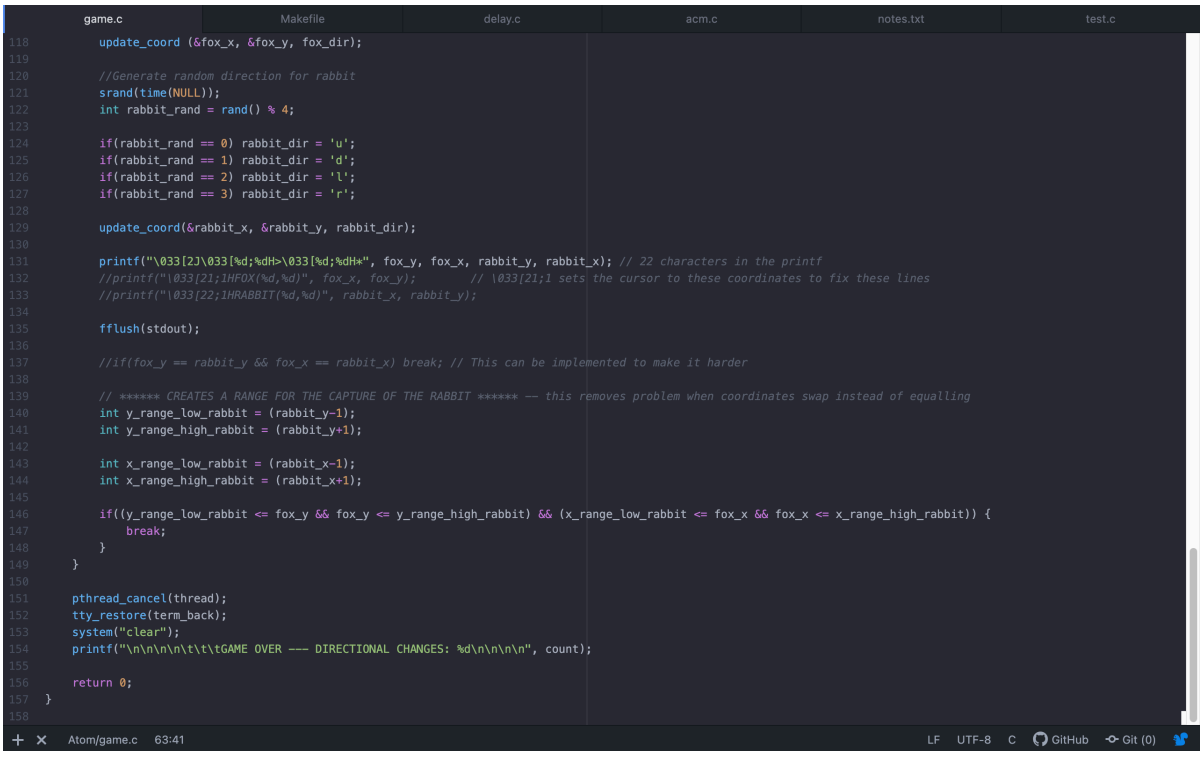
```c
#include <stdio.h>
#include <termios.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define width  40
#define height 18
#define fox_init_dir 'r'

// removes need to press enter after each direction by user
struct termios tty_prepare () {
  struct termios tty_attr_old, tty_attr;
  tcgetattr (0, &tty_attr);
  tty_attr_old = tty_attr;
  tty_attr.c_lflag &= ~(ECHO | ICANON);
  tty_attr.c_cc[VMIN] = 1;
  tcsetattr (0, TCSAFLUSH, &tty_attr);
  return tty_attr_old;
}

// restores keyboard to original settings -- must be included otherwise terminal corrupts
void tty_restore (struct termios tty_attr){
  tcsetattr (0, TCSAFLUSH, &tty_attr);
}

char fox_dir = fox_init_dir;
int count = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void keys_thread() {

    char key;

    while(1) {

        for(int i = 0; i < 3; i++){
            key = getchar();
        }
```



```c
        // Sets the 'char' type variable for switch case in update_coord() function
        if(key == 65) {
            fox_dir = 'u';    // up
            pthread_mutex_lock(&mutex1);        // Prevents data inconsistencies due to race conditons. Global variable count is accessed here and in the main
            count++;                            // and so a mutex is needed. Mutex protects segment of memory from other threads.
            pthread_mutex_unlock(&mutex1);
        }
        if(key == 66) {
            fox_dir = 'd';    // down
            pthread_mutex_lock(&mutex1);
            count++;
            pthread_mutex_unlock(&mutex1);
        }
        if(key == 67) {
            fox_dir = 'r';    // left
            pthread_mutex_lock(&mutex1);
            count++;
            pthread_mutex_unlock(&mutex1);
        }
        if(key == 68) {
            fox_dir = 'l';    // right
            pthread_mutex_lock(&mutex1);
            count++;
            pthread_mutex_unlock(&mutex1);
        }
    }
}
//  Updates coordinates for both characters
void update_coord (int *x_ptr, int *y_ptr, char dir) {
    switch (dir) {
        case 'u':       // UP command
            if(1 <= *y_ptr <= height)                       // makes sure the character stays within the limits of the given window
                if(*y_ptr == 0) *y_ptr = 1;                 // sets the characters minimum y coordinates as 1, even if the user directs below 1
                else if(*y_ptr == (height+1)) *y_ptr = height;          // sets the characters maximum y coordinates as 18, even if the user directs above 18
                (*y_ptr)--;                                 // decrements the characters y coordinates by 1 if user enters up-arrow key
                break;

        case 'd':       // DOWN command
            if (1 <= *y_ptr <= height)
                if(*y_ptr == 0) *y_ptr = 1;
                else if(*y_ptr == (height+1)) *y_ptr = height;
```

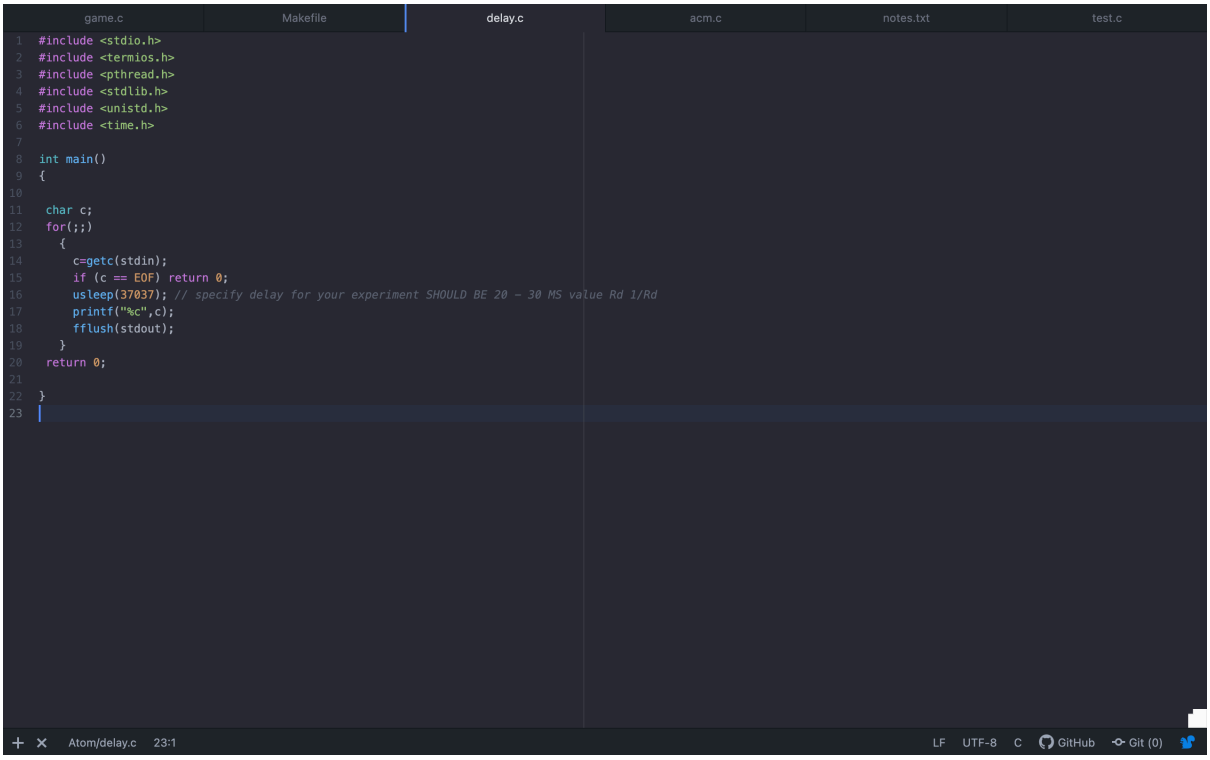


```
game.c | Makefile | delay.c | acm.c | notes.txt | test.c

                else if(*y_ptr == (height+1)) *y_ptr = height;
                (*y_ptr)++;     // increments the characters y coordinates by 1 if user enters down-arrow key
                break;

        case 'l':       //LEFT command
            if (1 <= *x_ptr <= width)
                if(*x_ptr == 0) *x_ptr = 1;
                else if(*x_ptr == (width+1)) *x_ptr = width;
                (*x_ptr)--;     // decrements the characters x coordinates by 1 if user enters left-arrow key
                break;

        case 'r':       //RIGHT command
            if (1 <= *x_ptr <= width)
                if(*x_ptr == 0) *x_ptr = 1;
                else if(*x_ptr == (width+1)) *x_ptr = width;
                (*x_ptr)++;     // increments the characters x coordinates by 1 if user enters right-arrow key
                break;
    }
}

int main(){

    // ******** Variable declarations ********
    int fox_x = 2, fox_y = 2, rabbit_x = 15, rabbit_y = 10;
    char rabbit_dir;

    struct termios term_back = tty_prepare();       // declares 'term_back' of the same type that tty_prepare() function returns

    pthread_t thread;

    pthread_create(&thread, NULL, (void *) keys_thread, NULL);

    while(1) {

        usleep(500000);

        update_coord (&fox_x, &fox_y, fox_dir);

        //Generate random direction for rabbit
        srand(time(NULL));
        int rabbit_rand = rand() % 4;

Atom/game.c  63:41   LF  UTF-8  C  GitHub  Git (0)
```



```
game.c | Makefile | delay.c | acm.c | notes.txt | test.c

        update_coord (&fox_x, &fox_y, fox_dir);

        //Generate random direction for rabbit
        srand(time(NULL));
        int rabbit_rand = rand() % 4;

        if(rabbit_rand == 0) rabbit_dir = 'u';
        if(rabbit_rand == 1) rabbit_dir = 'd';
        if(rabbit_rand == 2) rabbit_dir = 'l';
        if(rabbit_rand == 3) rabbit_dir = 'r';

        update_coord(&rabbit_x, &rabbit_y, rabbit_dir);

        printf("\033[2J\033[%d;%dH>\033[%d;%dH*", fox_y, fox_x, rabbit_y, rabbit_x); // 22 characters in the printf
        //printf("\033[21;1HFOX(%d,%d)", fox_x, fox_y);        // \033[21;1 sets the cursor to these coordinates to fix these lines
        //printf("\033[22;1HRABBIT(%d,%d)", rabbit_x, rabbit_y);

        fflush(stdout);

        //if(fox_y == rabbit_y && fox_x == rabbit_x) break; // This can be implemented to make it harder

        // ****** CREATES A RANGE FOR THE CAPTURE OF THE RABBIT ****** -- this removes problem when coordinates swap instead of equalling
        int y_range_low_rabbit = (rabbit_y-1);
        int y_range_high_rabbit = (rabbit_y+1);

        int x_range_low_rabbit = (rabbit_x-1);
        int x_range_high_rabbit = (rabbit_x+1);

        if((y_range_low_rabbit <= fox_y && fox_y <= y_range_high_rabbit) && (x_range_low_rabbit <= fox_x && fox_x <= x_range_high_rabbit)) {
            break;
        }
    }

    pthread_cancel(thread);
    tty_restore(term_back);
    system("clear");
    printf("\n\n\n\n\t\t\tGAME OVER --- DIRECTIONAL CHANGES: %d\n\n\n\n", count);

    return 0;
}

Atom/game.c  63:41   LF  UTF-8  C  GitHub  Git (0)
```

Delay


game.c
Makefile
delay.c
acm.c
notes.txt
test.c
#include <stdio.h>
#include <termios.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
int main()
{
char c;
for(;;)
{
c=getc(stdin);
if (c == EOF) return 0;
usleep(37037); // specify delay for your experiment SHOULD BE 20 – 30 MS value Rd 1/Rd
printf("%c",c);
fflush(stdout);
}
return 0;
}
Atom/delay.c 23:1
LF UTF-8 C GitHub Git (0)

ACM

game.c | Makefile | delay.c | acm.c | notes.txt | test.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

// decide what type ACM to use, look in the handout for pseudo-code -- Signal type ACM

char slots[3][23]; // 3 slots; add the size of each slot instead of ... 23 index's because 23 characters in the printf line
// Find how to use LUT
const int LUT[3][3] = {{2, 2, 1},
                       {2, 0, 0},
                       {1, 0, 1}};     // look up table = few clock cycles, very very fast

int w=0, r=1, l=2; // define and initialise the variables as in the handout

void reader_thread () {

  while(1) { // reader loop; includes a delay to reduce the output data rate

      while(l == r);   // wait until l != r condition is true

      r = l;

      printf("%s", slots[r]); // access slot; slots[i] is a pointer to slots[i][0] READ
                              // (slots[i][0],slots[i][1],... is a 0-terminated string)
      fflush(stdout);

      usleep(814000);  // limit output rate; calculate your own value (>500000) talking about frames, should be similar to game usleep value
                       // should be a smaller that the game by the same percentage that rd is < rg
                       // should be a trade off
                       // do the maths for a lag of 7 seconds
      //printf("%d", r);

    }

}

char inp() {    // getchar() wrapper which checks for End Of File EOF

  char c;
```

Atom/acm.c 68:1 LF UTF-8 C GitHub Git (0)

game.c | Makefile | delay.c | acm.c | notes.txt | test.c

```c
}

char inp() {    // getchar() wrapper which checks for End Of File EOF

  char c;
  c=getchar();
  if(c==EOF) exit(0); // exit the whole process if input ends
  return c;
}

int main () {
  //...; // variable declarations/initialisation, if needed

  pthread_t read_thread;

  pthread_create(&read_thread, NULL, (void *) reader_thread, NULL);   //expecting a NULL pointer. Very dangerous, can point to any type

  while (1) {  // writer loop

// access slot; modify this according to the output format of your game
// in this example I keep reading until '*' appears,
// then the symbol 0 is added, which is the sting terminator.
// The terminator is needed for printf("%s",...) of the reader.

      //printf("%d", l); // for testing the ACM on its own

      int j = 0;
      while ((slots[w][j++] = inp ()) != '*'); // the actual computation takes place inside the condition WRITE
      slots[w][j] = 0; // append the terminating symbol to the string

      l = w;

      w = LUT[r][l]; // Implement the look up table

    }

  pthread_cancel(read_thread);

  return 0;
}
```

Atom/acm.c 20:1 LF UTF-8 C GitHub Git (0)