# EEE3009 REPORT

## Multithreading, Concurrency & ACMs

### 1. Abstract

This coursework achieves multiple tasks. The first being the implementation of a simple console based game, that uses threads to read the users input and update the character's positions on the screen. The game is then piped through a delay, to demonstrate latency. The latency is then overcome with the use of an ACM.

Conor Newton
130327088

# 2. Introduction

With single core CPU systems, there becomes a need to share this pivotal resource. This is often performed by rapidly switching between active tasks and processes, allowing the processor to multitask between different operations. Creating the impression of *parallel computation* within the machine. Operating systems like Unix or Windows, must select processes and schedule them, allocating their time on the CPU in stages.

The term *concurrent* processing is normally used to express how multi-tasking can be achieved by serially sharing a single CPU. If multiple CPUs are available to work on a task simultaneously, this is known as *parallel* processing. Concurrent programming is only made possible by the speed at which single core CPUs are able to operate. Deciding when what and how tasks are swapped with the strategy being used to choose the next active task to run on the CPU, this is the responsibility of the system scheduler.

The method of sharing a single CPU resource among competing tasks started in the 1960's, in the days of large, expensive mainframes. The cost and wastefulness of CPU time for a single process would be extortionate if multi-tasking had not been developed to rapidly share the resources and running costs amongst many users.

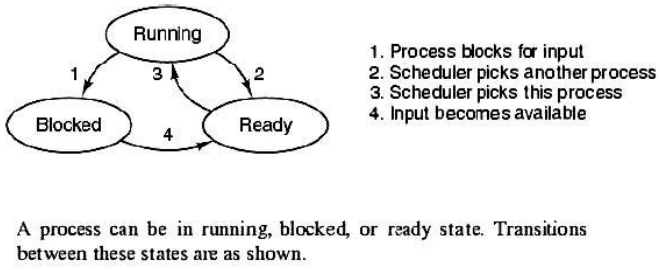


*Figure 2-1 Process lifecycle*

When a process is created it is in a ready state and stored in memory, waiting for its turn on the processor. A context switch occurs when a process is terminated or blocked, either by the system or when an I/O or clock interrupt occurs, this happens when a higher priority process needs the CPU or a process uses all of its allocated time on the CPU (known as quantum). Blocked processes will eventually be resolved and moved to memory with the rest of the ready and waiting for processes. This process lifecycle can be seen in figure 2.1.

Threads take this principle one step further and allow a program to further divide its time as a process on the CPU. Threads originally came about as an alternative to having multiple processes for one program. Processes are capable of containing multiple threads, all accessing the processes resources.

Threads may provide a solution to one problem, but they inadvertently create more. Programmers have to be wary of race conditions and deadlocking within their code. Methods have been created to aid with this, such as mutual exclusion and semaphores.

ACMs, or Asynchronous data Communication Mechanisms, researched and developed as part of the COMFORT project, by Newcastle University and Kings College London. Allow for asynchronous data to be exchanged concurrently between reading and writing processes. ACMs act as a buffer between an in and out, allowing them to operate freely at different rates to each other while attempting to remain synchronised.

## 2.1 Aims & Objectives

Aims:

- Mini-project on embedded software design
- Multithreading and concurrent programming
- Design of a system composed of several communicating processes
- ACMs and their application to dynamic systems
- Experiments with ACMs
- Portable code and cross compilation

Objectives:

- To implement the game as specified
- To run the game displaying the screen on a remote terminal connected through a slow interface with FIFO buffer; observations, conclusions.
- To implement an ACM and include it in the pipeline between the game process and the slow interface; observations, conclusions.

## 2.2 Specification

A game is created, to be played on an alphanumerical terminal, using the arrow keys to control the movements of the players' character. Only two characters are displayed: the moving target (rabbit) and the chaser (fox). Both are represented as single characters and move at the speed of 1 position every 0.25 seconds. The original specification was for 1 position every 0.5 seconds, however, this made the functionality of the game appear sluggish.

The player controls the fox and attempts to catch the rabbit, whose movements are randomly generated; both characters are unable to leave the screen. The game ends when the fox catches the rabbit, meaning the positions of their coordinates are equal. The reading of the input from the keyboard is to be done concurrently with the use of a thread, in order to make it independent from the animation. The output of the game is produced on the standard output stream, using escape characters to position the symbols correctly on the screen.

A Petri net model is to be derived to describe the communication between the keyboard and the animation parts of the game. A delay or slow interface emulation program is to be implemented, as well as a 3-slot signal type ACM program, to overcome the latency between the game and the output.

# 3. Results

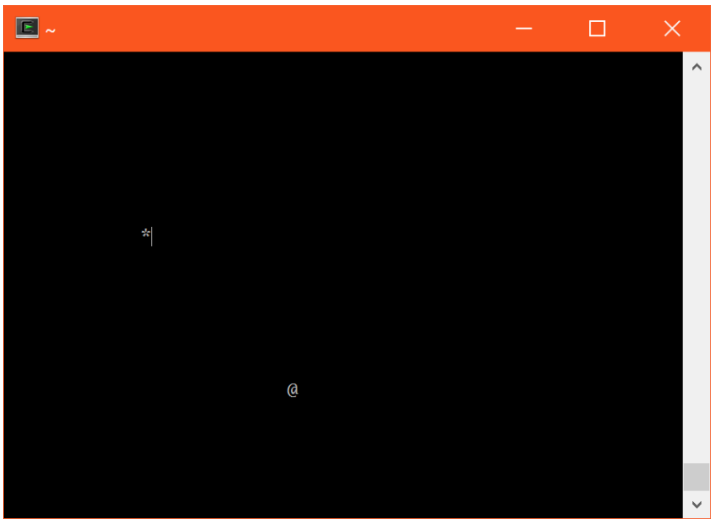


*Figure 3-1 Fox & Rabbit game output*

The correct implementation of the main program results in a simple console game played on a 40x18 sized grid. Where the movements of the rabbit (represented in this case by the use of the "*" symbol) are randomly generated. The user or fox (represented with the "@" symbol) attempts to catch the sporadically moving rabbit, using the arrow keys. The output from the game can be seen in figure 3.1. The game ends when the coordinates of the fox and rabbit are the same.

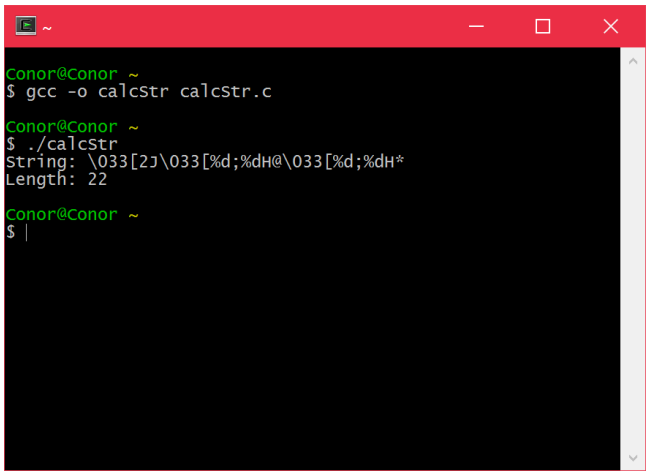


*Figure 3-2 Calculating frame length*

The game produces a new frame every 0.25 seconds (4 frames per second) consisting of the string shown in figure 3.2. A simple program was written to count the total characters being produced every frame. Resulting in a frame length of 22 characters, 4 times a second (88 characters a second).

A delay program is then added to the game; it slows the output of characters to 66 per second. Causing a build-up in the delay programs buffer, at a rate of 22 characters per second; as data is flowing into the delay program faster than it can escape it. This is known as *latency* and it makes the game unplayable. In figure 3.3 the right-hand side shows the game as it should be, you can see that the game on the left is lagging behind. This lag increases the longer the game runs for, as the buffer grows.

The program is run on a POSIX (Portable Operating System Interface), that emulates a UNIX system for windows.

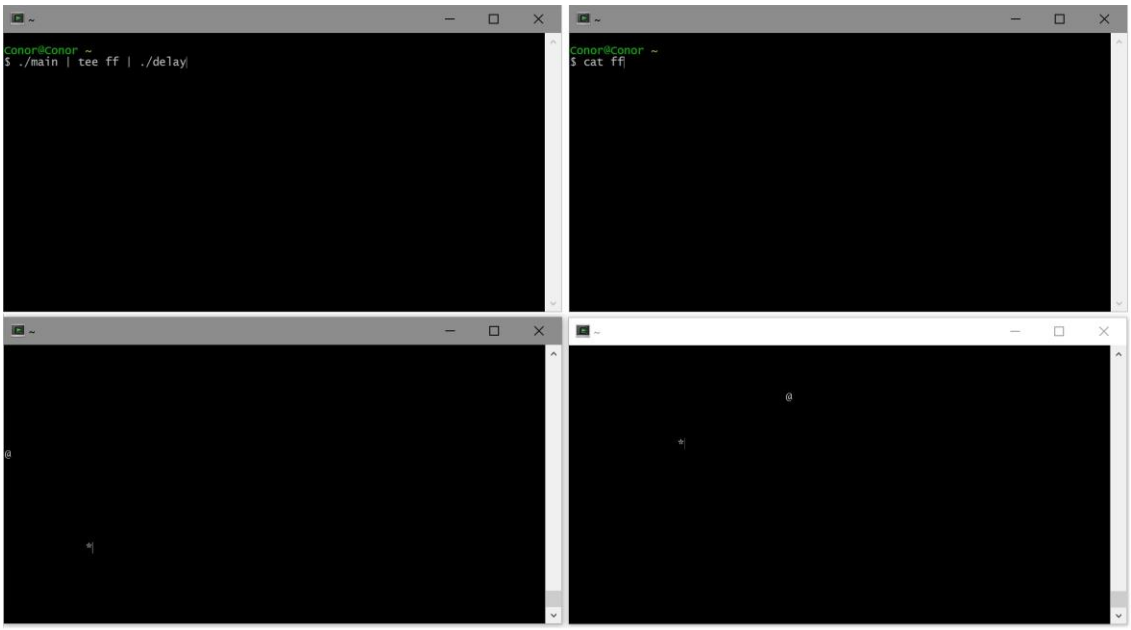


*Figure 3-3 Normal game function compared to latency effects*

A 3-slot signal type ACM then placed before the delay program to remove the latency it causes. The ACM removes the latency by reading in data at a higher rate and emitting the same data at a slower rate than the delay program, removing the buffer build-up in the delay program. The ACM removes the buffer build-up by discarding and overwriting data as it needs to. This results in a loss of information, so the game does not run as smoothly as it should, skipping over information; but without the latency, it is still very much playable and responsive.


*Figure 3-4 Normal game function compared to post ACM & delay*

Figure 3.4 shows the game running as it should be, beside the same game being piped through the delay and the ACM. As well as the commands used to execute the same game in both consoles. The ACM has now successfully removed the latency previously seen in figure 3.3, allowing the game to run smoothly and remain playable.

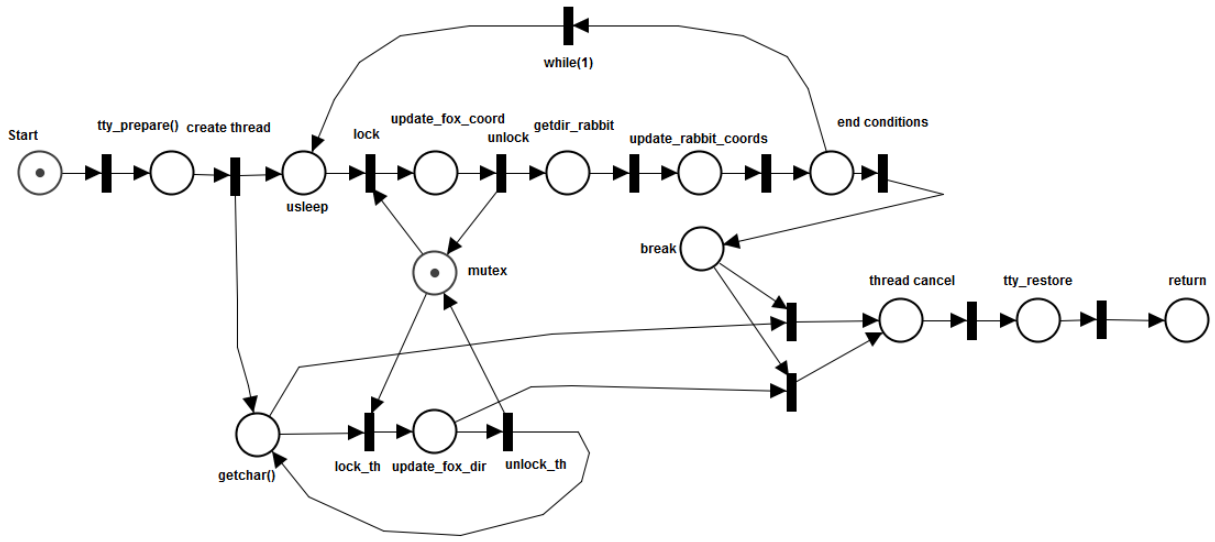


*Figure 3-5 Petri Net of the games functionality*

Figure 3.5 shows a Petri net model, used to visualise the concurrency of the games threads. It depicts the mutex that prevents both threads from accessing the mutual global variables at the same time. The Petri net shows how the keyboard thread and the main program are able to run together, as well as how they are initiated and how both threads are terminated when the game is won.

# 4. Discussion

## 4.1 Threads & Concurrency

The Petri net diagram depicted in figure 3.5 shows the workings of the game. How it is initiated as a single thread and how it initiates the second thread to handle the keyboard input concurrently. It is able to achieve this concurrency by implementing a mutex between the two threads, in order to mutually exclude one thread at a time from accessing the commonly used global variables (the direction of the fox), this prevents race conditions and hazards as this can be considered the game's critical section (region).

The Petri net describes how if the end conditions of the game are satisfied, then the main loop is broken. Ceasing the keyboard thread from transitioning any further, removing its token and canceling the thread. Ultimately terminating the entire program.

Multi-tasking and multi-threading programs are readily implemented in high-level languages. C/C++ provides the *pthread* library, that allows users to create, cancel and share resources amongst threads. A

mutex or mutual exclusion is included within the game program, to lock and unlock access to a shared variable between the threads, this is known as the critical region. Without the mutex within this region, there is a risk of partial reading and writing taking place with a race between the threads to access the variable. This results in corrupted data being loaded and causing errors within the game.

However, the use of threads and mutex provide a number of new issues and problems to be considered. One of the most common issues with mutual exclusion is known as *deadlock*. Deadlock can occur when one thread is waiting to access a mutual exclusive variable that hasn't been released by another thread. Or when a thread tries to acquire a lock it is already holding. This results in a total collapse of the program, as it cannot move forward. Precautions have been taking to avoid deadlocking within the ACM and game programs.

The use of threads within the programs can also be questioned as most modern computers, even portable microprocessors (nowadays) perform an operation on 32 bits or 64 bits at a time. As a char is a primitive data type, it is only 8 bits long, meaning it can be read and written automatically within a clock cycle, without the need for synchronisation. If a variable is larger than what the computer can operate on at a time, then it poses the risk of potentially being interrupted and corrupted, requiring exclusion and synchronisation.

By implementing a delay in the main threads of the game and the ACM, they now run significantly slower than their child threads. Meaning that when the values calculated by the child threads are needed, they have already been calculated long before the parent threads are trying to access the shared resource (critical section). Regardless of what order the threads are created, ordered or scheduled in. The mutex is simply a precaution at that point.

If the program was to be imported onto a smaller portable device, that may use a smaller word architecture, the use of mutual exclusion and threads may prove necessary.

## 4.2 Buffers & Latency

The character output rate of the game is set using the method *'usleep()'*, which suspends the execution of the program for microsecond intervals. In this case, the game is producing 22 characters every 250000 microseconds. So the characters per second (cps) can be found using the calculation below.

$$\frac{22}{250000 \times 10^{-6}} = 88\ cps$$

The delay program outputs a single character at a time, therefore the characters per second can be found for the delay program as such.

$$\frac{1}{15152 \times 10^{-6}} = 66\ cps$$

Given that the game is emitting 88 characters per second and the delay 66.

$$\frac{22}{250000 \times 10^{-6}} - \frac{1}{15152 \times 10^{-6}} = 88 - 66 = 22\ cps$$

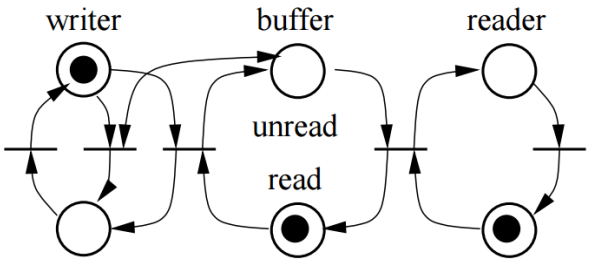


*Figure 4-1 Petri net of a signal type ACM*

We can work out that 22 characters per second are accumulating and building up within the delay programs buffer. This is the cause of the latency present in the bottom left of figure 3.3, the game becomes unplayable as the input from the user takes an increasing amount of time before the response is displayed on the screen.

To deal with this build-up a 3-slot signal type ACM (asynchronous data communication mechanism) is implemented between the game and the delay. The workings of the signal type ACM in Petri net form can be seen in figure 4.1.

The Petri net for the ACM illustrates how the writer can continue writing independently from the buffer and the reader. Until the reader is ready to read again, the buffers token will remain in the unread state, holding the next consecutive piece of information from what was the last read. While in this unread position, the writer is shifted into an internal loop using a controlled choice, where the information goes nowhere. The information is simply written over until the reader shifts the buffer into the read state and the writer is able to write into the buffer. This is how the latency is dealt with and removed.

However, in order for this ACM to be effective, it must have a slower output rate than that of the delays input. Otherwise, the ACM is made redundant and the latency isn't resolved, it has just been shifted. This was the original problem, that information was being fed too quickly into the delay program. A suitable output rate can be calculated in the same manner the games output rate was calculated.

$$\frac{22}{450000 \times 10^{-6}} = 49\ cps$$

The lagging effect produced from the ACM can be compensated for by further increasing the frame rate of the original game. The game specification recommends setting the game to work at a rate of 2 frames per second. However, by the time the game has gone through the ACM and the delay, the game appears sluggish. As the end goal of the implementation of the ACM is to allow it to be playable again, the framerate was therefore increased, so that the speed of the game would remain playable. Most modern games operate between 30-60 frames per second, however with such a simple consoles screen based game, anything over 20 fps becomes a bit too fast, making it harder for the user to play.

The side by side comparisons of the game running before and after the ACM & delay can be achieved by using a FIFO (first in first out) pipe. The output of the game is simultaneously output to ACM program as well as a file called *ff*. This *ff* file is then read and displayed on a second Cygwin terminal.

## 5.Conclusion

Overall the coursework achieved all the mandatory tasks that were required of it. Demonstrating the uses of multi-tasks and multi-threads within concurrent programming within a simple shell consoles game, that updates positions and reads inputs simultaneously. The coursework shows what the effects of latency are and how they can occur as well as be overcome with the utilisation of ACM's. The implementation, method, and calculations of the workings of the programs have been thoroughly discussed and explained. With thought given to the need for threads and mutexes, as well as how the programs would behave on a smaller word length and portable processor.

## 6. References


[1] Oracle Corporation and/or its affiliates (2010) *Multithreaded Programming Guide,* Available at: *https://docs.oracle.com/cd/E19455-01/806-5257/6je9h0347/index.html*

[2] A. Bystrov (2016) *EEE3009: Coursework,* Available at: *https://blackboard.ncl.ac.uk/bbcswebdav/pid-2738387-dt-content-rid-8053956_1/courses/M1617-EEE3009/Module%20Documents/Coursework/cwrk.pdf*

[3] Newcastle University & Kings College London (2001) *Synchronous COmmunication Mechanisms FOr Real-Time systems (COMFORT),* Available at: *https://www.staff.ncl.ac.uk/i.g.clark/publications/comfort-final-epsrc-rp.pdf.*

[4] Rob Williams (2006) *Real-time systems development*, 1 edn., Oxford: Butterworth-Heinemann .

# 7. Appendicies

## 7.1 Game Code

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

#define width  40
#define height 18
#define buf_length (width*height)
#define fox_init_x (width/3*2)
#define fox_init_y (height/3*2)
#define fox_init_dir 'u'                        // fox starts going up
#define rabbit_init_x (width/9*2)
#define rabbit_init_y (height/9*2)
#define rabbit_init_dir 'd'                     // rabbit starts going down

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // mutual exclusion for
shared resources between threads

//---- fox direction
char fox_dir = fox_init_dir;

//---- set keyboard mode --
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;
}

//---- restore keyboard mode --
void tty_restore (struct termios tty_attr)
{
    tcsetattr (0, TCSAFLUSH, &tty_attr);
}

//---- keyboard thread function
void keys_thread ()
{
    unsigned char a;
    while(1)
    {
        a=getchar();
        if (a == 0x1b)
        {
```

```
            a = getchar();
            if (a == 0x5b)
            {
                pthread_mutex_lock(&mutex);
                a = getchar();
                switch (a)
                {
                case 0x41:
                    fox_dir = 'u';
                    break;
                case 0x42:
                    fox_dir = 'd';
                    break;
                case 0x43:
                    fox_dir = 'r';
                    break;
                case 0x44:
                    fox_dir = 'l';
                    break;
                }
                pthread_mutex_unlock(&mutex);
            }
        }
    }

}

//---- update x and y coord-s according to direction; used in main()
void update_coord (int *x_ptr, int *y_ptr, char dir) // call by reference to
x and y
{
    switch (dir)
    {
    case 'u':
        if (*y_ptr > 1) (*y_ptr)--;
        break; // *y_ptr is called "dereference",
// which is the target pointed at by the pointer
    case 'd':
        if (*y_ptr < 18) (*y_ptr)++;
        break;
    case 'l':
        if (*x_ptr > 1) (*x_ptr)--;
        break;
    case 'r':
        if (*x_ptr < 40) (*x_ptr)++;
        break;
    }
}

//---- the program starts its execution from here
int main ()
{
    // variable declarations and initialisation
    pthread_t keyT;                         // keyboard thread
    struct termios term_back;
    term_back = tty_prepare ();             // keyboard thread defined
```

```
    pthread_create (&keyT, NULL, (void *) &keys_thread, (void *)
fox_init_dir); // create the keyboard thread

    char rabbit_dir = rabbit_init_dir;
    int fox_x = fox_init_x;
    int fox_y = fox_init_y;
    int rabbit_x = rabbit_init_x;
    int rabbit_y = rabbit_init_y;

    while (1)
    {
        usleep (250000);                                // emitting data at 4
fps or 88 cps
        pthread_mutex_lock(&mutex);                     // fox_dir is locked
        update_coord (&fox_x, &fox_y, fox_dir);         // fox dir is updated
        pthread_mutex_unlock(&mutex);                   // fox dir is
unlocked
        int r = rand() % 4; // generate the rabbit direction at random to
modulo 4
        switch (r)          // select rabbit direction relating to random
number
        {
        case 0:
            rabbit_dir = 'u';
            break;
        case 1:
            rabbit_dir = 'd';
            break;
        case 2:
            rabbit_dir = 'l';
            break;
        case 3:
            rabbit_dir = 'r';
            break;
        default: break;
        }

        update_coord (&rabbit_x, &rabbit_y, rabbit_dir);    // rabbit
coordinates
        printf ("\033[2J\033[%d;%dH@\033[%d;%dH*", fox_y, fox_x, rabbit_y,
rabbit_x); // 22 characters
        fflush (stdout);

        if ( (fox_y == rabbit_y) && (fox_x == rabbit_x) ) break; //
conditions of game termination
    }

    pthread_cancel ((void *) &keys_thread);
    tty_restore (term_back);                    // keyboard is restored

    return 0;
}
```

## 7.2 ACM code

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

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

char slots[3][22]; // 3 slots each 22 chars long
const int LUT[3][3] = { { 1, 2 , 1}, { 2, 0 , 0}, { 1, 0 , 1}};
int w= 0;   // write frame
int r= 1;   // read frame
int l= 2;   // last frame
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void
reader_thread ()
{
  while(1)  // reader loop; includes a delay to reduce the output data rate
    {
        pthread_mutex_lock(&mutex);
        if(l != r){
          r = l;
          pthread_mutex_unlock(&mutex);
          printf ("%s", slots[r]); // access slot; slots[i] is a pointer to
slots[i][0]
          fflush(stdout);
          usleep(450000);                   // outputting 49 characters per
second
          } else {
          pthread_mutex_unlock(&mutex);
      }
    }
}

char input() // getchar() wrapper which checks for End Of File EOF
{
  char c = getchar();       // get char from stdin
  if(c==EOF) exit(0);       // exit the whole process if input ends
  return c;
}

int
main ()
{
    pthread_t readerTh;
    pthread_create(&readerTh, NULL, (void*) &reader_thread, NULL);  //
creates reader thread

  while (1)  // writer loop
    {
        int j = 0;  // writes into slot j
        while ((slots[w][j++] = input()) != '*'); // the actual computation
takes place inside the condition
```

```
        slots[w][j] = 0; // append the terminating symbol to the string

        pthread_mutex_lock(&mutex);
        l = w;                       // update last write slot
        w = LUT[r][l];               // update written slot
        pthread_mutex_unlock(&mutex);
    }

  return 0;
}
```

## 7.3 Delay code

```
#include <stdio.h>
#include <unistd.h>

int main()
{
 char c;
for(;;)
   {
     c=getc(stdin);
     if (c == EOF) return 0;
     usleep(15152); // specify delay for your experiment
     printf("%c",c);
     fflush(stdout);
   }
 return 0;
}
```

## 7.4 String Length Calculation

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char str[] = "\033[2J\033[%d;%dH@\033[%d;%dH*";
    int charcount = sizeof str - 1; // -1 to exclude terminating '\0'
    printf ("String:\t\\033[2J\\033[%%d;%%dH@\\033[%%d;%%dH* \n");
    printf ("Length:\t%i\n", charcount);
    return 0;
}
```