Tutorials (4-6) for Assignment 2

Ziyu XIE 121090642@link.cuhk.edu.cn

Office Hour: Thursday 20:00-21:00 (02.29-3.14)

Tutorial Arrangement:

- 02.28 - 03.07: Two weeks tutorials, normal schedule
- 03.13 (18:00 - 19:00) Q&A; 03.14 (20:00-21:00) Q&A

In this tutorial, we will first learn **Pthread** programming using c/c++.

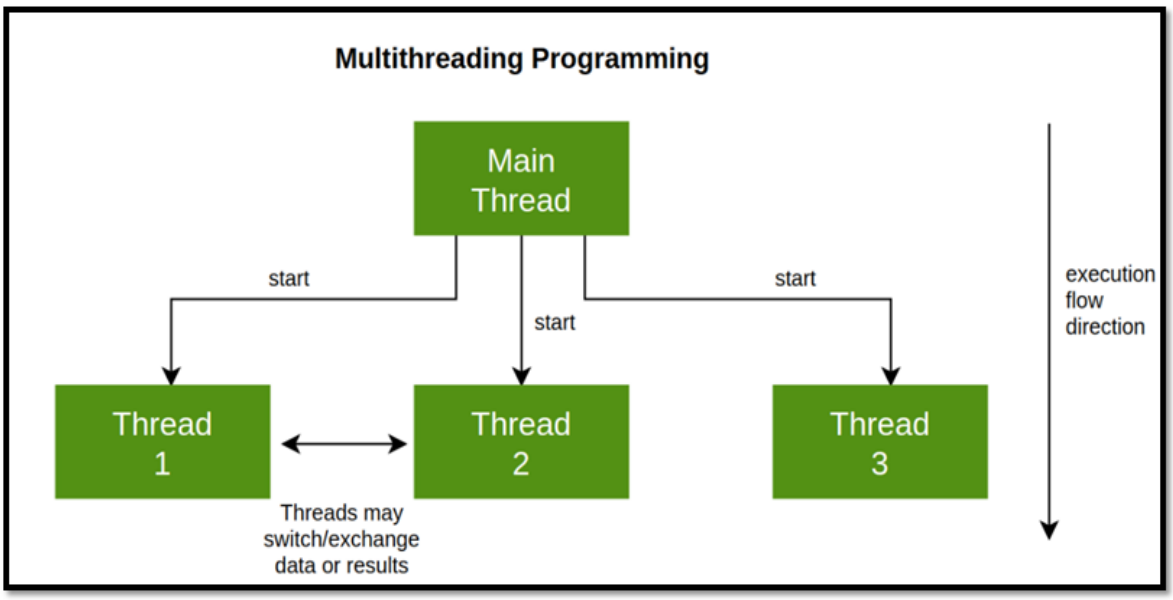
- Multithreading
- Pthread Creation & Termination
- Pthread Join
- Pthread Mutex
- Pthread Condition (optional)

Then we will study some related functions for Assignment 2.

- Keyboard Hit
- Terminal Control
- Suspend the executing thread
- Generate random number

1. Multithreading

A thread is a single sequence stream within a process. **A process can have multiple threads**, all of which share the resources within a process and all of which execute within the same address space. Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads.



1.1. Write multithreading programs in C

The **POSIX thread libraries** are a C/C++ thread API based on standards. You can write `#include <pthread.h>` in your code to use Pthreads.

- When compiling Pthread using gcc/g++, we should add option `-lpthread`.
  - Compile: `gcc test.c -lpthread` or `g++ test.cpp -lpthread`
  - Execution: `./a.out`
- All the test examples in this tutorial are executed using the commands above.
- You may get further detailed information in:
  - http://www.cs.unibo.it/~ghini/didattica/sistop/pthreads_tutorial/POSIX_Threads_Programming.htm

2. Pthread Creation & Termination

2.1. Pthread declaration

- Pthread is declared with type: `pthread_t`

```
pthread_t threads[NUM_THREAD];
```

2.2. Pthread creation

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

- **thread:** a pointer to a `pthread_t` object.
- **attr:** parameter used to set thread attributes. You can specify a thread attributes object like scheduling policy, detached state, etc. Set NULL by default.
- **start_routine:** the C routine that the thread will execute once it is created.
- **arg:** pointer to void that contains the arguments to the function.
- On success, `pthread_create()` returns 0; On error, it returns an error number, and the contents of `thread` are undefined.

2.3 Pthread termination

```
void pthread_exit (void *value_ptr);
```

- This method accepts a mandatory parameter `value_ptr` which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be **global** so that any thread waiting to join this thread may read the return status.
- This routine is used to **explicitly** exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work.

2.4 Example 1

```
#include <pthread.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("\n%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Possible output:

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
4: Hello World!
3: Hello World!
0: Hello World!
1: Hello World!
2: Hello World!
```

2.5. Example 2

- If `main()` finishes before the **threads it has created, and exits with** `pthread_exit()`,the other threads will continue to execute. Otherwise, they will **be automatically terminated** when `main()` finishes.
- Recommendation: Use `pthread_exit()` to exit from all threads, especially `main()`.

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

void *PrintHello(void *threadid)
{
    sleep(2);
    printf("Hello World!\n");
```

```
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t thread;
    int rc;
    void *i;

    printf("In main: create thread\n");
    rc = pthread_create(&thread, NULL, PrintHello, i);

    if (rc)
    {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(1);
    }

    printf("Main thread exits!\n");
    // pthread_exit(NULL);

    return 0;
}
```

output:

```
In main: create thread
Main thread exits!
```

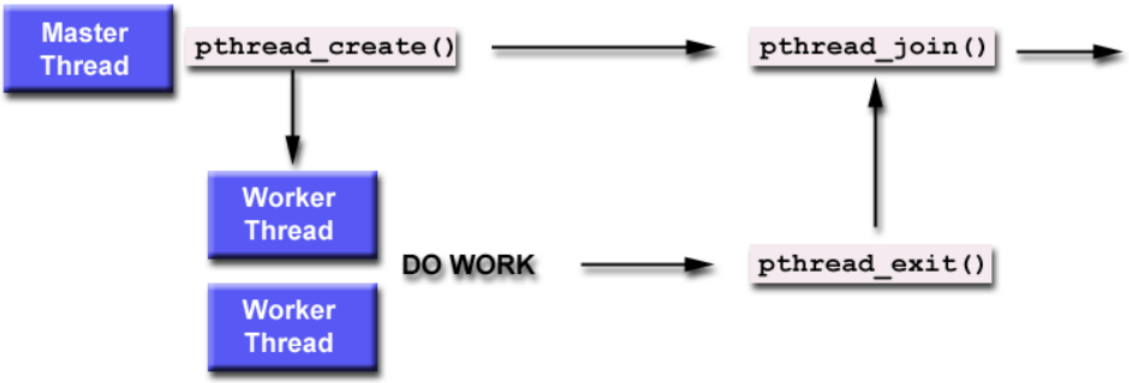
if the line `pthread_exit(NULL)` is NOT commented, the output:

```
In main: create thread
Main thread exits!
Hello World!
```

3. Pthread Join

```
int pthread_join (pthread_t thread, void *value_ptr);
```

- "Joining" is one way to accomplish **synchronization** between threads.
- The `pthread_join()` subroutine blocks the calling thread until the **specified thread terminates**.
- The programmer is able to obtain the target thread's termination return status if specified through `pthread_exit()`, in the status parameter.
- On success, `pthread_join()` returns **0**; On error, it returns an **error number**.
- When a thread is created, one of its attributes defines whether it is joinable or detached. Detached means it can never be joined. (`PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`)



3.1. Example

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

int sum;

void *add1(void *cnt)
{
    for (int i = 0; i < 5; i++)
        sum += i;
    pthread_exit(NULL);
    return 0;
}

void *add2(void *cnt)
{
    for (int i = 5; i < 10; i++)
        sum += i;
    pthread_exit(NULL);
    return 0;
}

int main(int argc, char *argv[])
{
    pthread_t ptid1, ptid2;
    sum = 0;
    pthread_create(&ptid1, NULL, add1, &sum);
    pthread_create(&ptid2, NULL, add2, &sum);
    // pthread_join(ptid1, NULL);
    // pthread_join(ptid2, NULL);
    printf("sum = %d\n", sum);
    pthread_exit(NULL);
}
```

output:

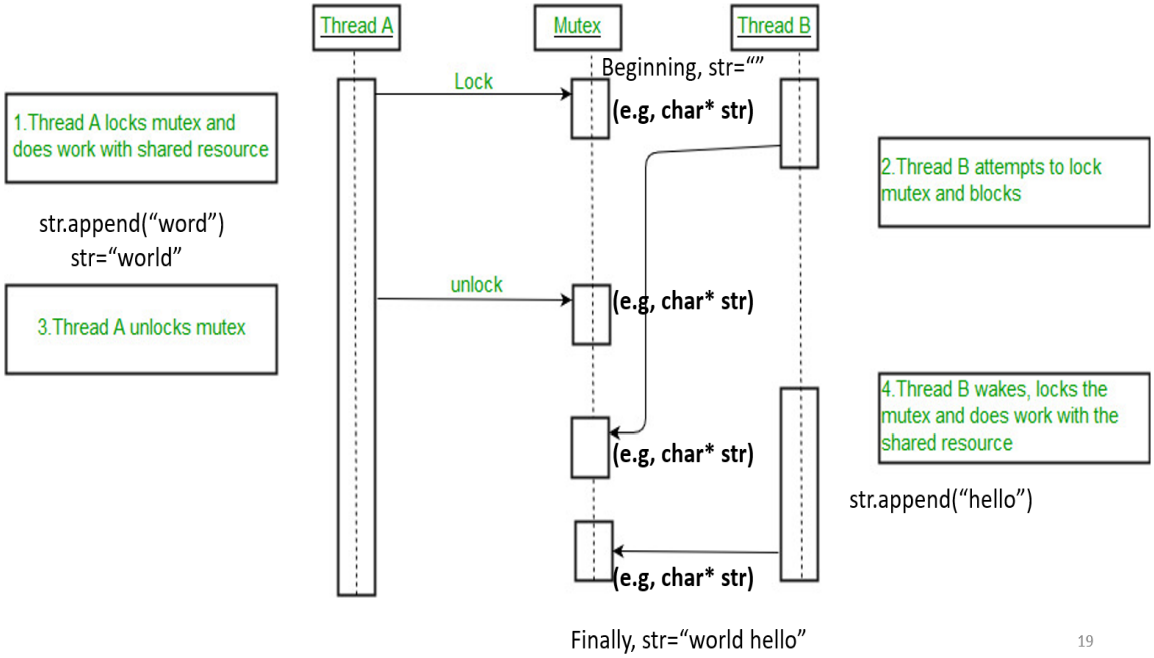
```
sum = 0
```

if the two lines `pthread_join(ptidx, NULL)` is NOT commented, the output:

```
sum = 45
```

4. Pthread Mutex

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- **A mutex variable acts like a "lock" protecting access to a shared data resource.**
- You can go to https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/ for further information.



4.1. Pthread mutex declaration

- Pthread mutex is declared with type: `pthread_mutex_t`

```
pthread_mutex_t *mutex;
```

4.2. Pthread mutex initialization

```
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

- It initialises the mutex referenced by **mutex** with attributes specified by **attr**.
- If **attr** is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object.
- Upon successful initialisation, the state of the mutex becomes initialized and unlocked.

4.3. Pthread mutex destroy

- Mutex should be free if it is no longer used:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

4.4. Pthread mutex lock routines

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified mutex variable. **If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked**.
- `pthread_mutex_trylock()` will **attempt to lock a mutex**. However, if the **mutex is already locked, the routine will return immediately with a "busy" error code**. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- `pthread_mutex_unlock()` will **unlock a mutex if called by the owning thread**. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An **error** will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

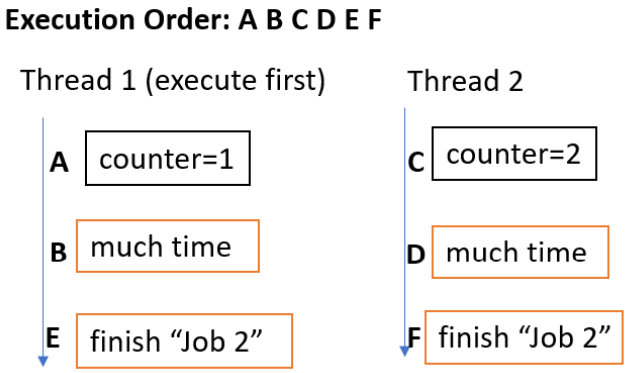
4.5. Example

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

int counter;
// pthread_mutex_t lock;

void *trythis(void *arg)
{
    // pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("Job %d has started\n", counter);

    for (i = 0; i < (0x0FFFFFFF); i++)
        ;
    printf("Job %d has finished\n", counter);
    // pthread_mutex_unlock(&lock);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid[2];
    // if (pthread_mutex_init(&lock, NULL) != 0)
    // {
    //     printf("\nMutex init has failed\n");
    //     return 1;
    // }
    int i = 0;
```

```
    int error;
    counter = 0;
    while (i < 2)
    {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
        {
            printf("\nThread can't be created\n");
        }
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    // pthread_mutex_destroy(&lock);
}
```

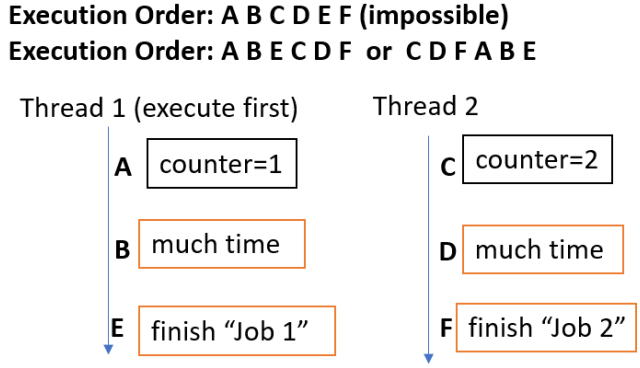
output:

```
Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished
```



if all lines are NOT commented (i.e. with mutex), the output:

```
Job 1 has started
Job 1 has finished
Job 2 has started
Job 2 has finished
```



5. Pthread Condition (optional)

Pthread condition is useful when programming, however you may not need to use this in your assignment. Hence this part is **optional**.

- **Condition variables** provide yet another way for threads to synchronize.
- While **mutexes** implement synchronization by controlling thread access to data, condition variables allow threads **to synchronize based upon the actual value of data.**
- A condition variable is always **used in conjunction with a mutex lock**.

5.1. Pthread condition declaration

- Pthread conditon is declared with type: `pthread_cond_t`

```
pthread_cond_t *condition;
```

5.2. Pthread condition initialization

```
int pthread_cond_init(
    pthread_cond_t *condition,
    const pthread_condattr_t *attr);
```

- **condition:** Specifies the condition to be created.
- **attr:** Specifies the condition attributes object to use for initializing the condition variable. If the value is NULL, the default attributes values are used.

5.3. Pthread condition free

- Condition should be free if it is no longer used:

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

5.4. Pthread condition routines

```
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
int pthread_cond_signal(pthread_cond_t *);
int pthread_cond_broadcast(pthread_cond_t *);
```

- `pthread_cond_wait()` blocks the calling thread until **the specified condition** is signalled. This routine should be called **while mutex is locked**, and it will **automatically release the mutex while it waits**.
- **The** `pthread_cond_signal()` routine is used **to signal (or wake up) another thread which is waiting** on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() routine to complete.
- **The** `pthread_cond_broadcast()` routine should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

5.5. Example

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

```c
#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 10

int count = 0;
int thread_ids[3] = {0, 1, 2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
    int i = 0;
    int taskid = 0;
    int *my_id = (int *)idp;

    for (i = 0; i < TCOUNT; i++)
    {
        pthread_mutex_lock(&count_mutex);
        taskid = count;
        count++;

        if (count == COUNT_LIMIT)
        {
            pthread_cond_signal(&count_threshold_cv);
        }
        printf("inc_count(): thread %d, count = %d, unlocking mutex\n", *my_id,
count);
        pthread_mutex_unlock(&count_mutex);
        sleep(1);
    }
    printf("inc_count(): thread%d, Threshold reached.\n", *my_id);

    pthread_exit(NULL);
}

void *watch_count(void *idp)
{
    int *my_id = (int *)idp;

    printf("Starting watch_count(): thread %d\n", *my_id);

    pthread_mutex_lock(&count_mutex);
    while (count < COUNT_LIMIT)
    {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): thread %d Condition signal received.\n", *my_id);
    }

    count += 100;
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    int i, rc;
    pthread_t threads[3];
```

```
    pthread_attr_t attr;

    /* Initailize mutex and condition variable objects */
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init(&count_threshold_cv, NULL);

    /* For portability, explicitly create threads in a joinable state */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_create(&threads[0], &attr, watch_count, (void *)&thread_ids[0]);
    pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
    pthread_create(&threads[2], &attr, inc_count, (void *)&thread_ids[2]);

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
    printf("Main(): Waited on %d threads. Done.\n", NUM_THREADS);

    /* Clean up and exit */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit(NULL);

    return 0;
}
```

output:

```
Starting watch_count(): thread 0
inc_count(): thread 1, count = 1, unlocking mutex
inc_count(): thread 2, count = 2, unlocking mutex
inc_count(): thread 1, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 1, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
watch_count(): thread 0 Condition signal received.
inc_count(): thread 1, count = 111, unlocking mutex
inc_count(): thread 2, count = 112, unlocking mutex
inc_count(): thread 1, count = 113, unlocking mutex
inc_count(): thread 2, count = 114, unlocking mutex
inc_count(): thread 1, count = 115, unlocking mutex
inc_count(): thread 2, count = 116, unlocking mutex
inc_count(): thread 1, count = 117, unlocking mutex
inc_count(): thread 2, count = 118, unlocking mutex
inc_count(): thread 1, count = 119, unlocking mutex
inc_count(): thread 2, count = 120, unlocking mutex
inc_count(): thread1, Threshold reached.
inc_count(): thread2, Threshold reached.
Main(): Waited on 3 threads. Done.
```

For more details and implementations, you can go to https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/ for further study.

6. Keyboard Hit

- In Assignment 2, we've provided a similar function named `int kbhit(void)`, you could use it directly. If a key has been pressed then it returns a non zero value, otherwise it returns zero.

6.1. Example

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

int kbhit(void)
{
    struct termios oldt, newt;
    int ch;
    int oldf;
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);
    ch = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);
    if (ch != EOF)
    {
        ungetc(ch, stdin);
        return 1;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    int isQuit = 0;
    while (!isQuit)
    {
        if (kbhit())
        {
            char dir = getchar();
            if (dir == 'w' || dir == 'W')
                printf("UP Hit!\n");
            if (dir == 's' || dir == 'S')
                printf("DOWN Hit!\n");
            if (dir == 'a' || dir == 'A')
                printf("LEFT Hit!\n");
            if (dir == 'd' || dir == 'D')
```

```
                printf("RIGHT Hit!\n");
            if (dir == 'q' || dir == 'Q')
            {
                isQuit = 1;
                printf("Quit!\n");
            }
        }
    }
    return 0;
}
```

If clicking "WWSSAADDWSQ", the output:

```
UP Hit!
UP Hit!
DOWN Hit!
DOWN Hit!
LEFT Hit!
LEFT Hit!
RIGHT Hit!
RIGHT Hit!
UP Hit!
DOWN Hit!
Quit!
```

- `int getchar(void)` function is used to get/read a character from keyboard input.
- `int putchar(int char)` function is a file handling function which is used to write a character on standard output/screen.
- `int puts(const char *str)` writes a string to stdout up to but not including the null character. A newline character is appended to the output.
- In Assignment 2, you may use above functions to complete your keyboard read and map write. Further reading: https://www.tutorialspoint.com/c_standard_library/c_function_puts.htm

7. Terminal Control

- When printing the message, you could use "\033" to control the **cursor in terminal**. Further reading about how to use this: https://www.student.cs.uwaterloo.ca/~cs452/terminal.html

These are the most essential terminal control sequences that you will need for your train program.

Code	Effect
"\033[2J"	Clear the screen.
"\033[H"	Move the cursor to the upper-left corner of the screen.
"\033[r;cH"	Move the cursor to row **r**, column **c**. Note that both the rows and columns are indexed starting at 1.
"\033[?25l"	Hide the cursor.
"\033[K"	Delete everything from the cursor to the end of the line.

7.1. Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    int isStop;
    printf("\033[2J");
    for (isStop = 1; isStop < 11; isStop++)
    {
        printf("\033[%d;1OH", (11 - isStop));
        printf("Printing withing loop %d!\n", isStop);
        sleep(1);
    }
    printf("\033[H\033[2J");
    return 0;
}
```

Try to run this in your virtual machine and see what will happen by yourself.

8. Suspend executing thread

- `int usleep(useconds_t usec)` suspends execution for microsecond intervals.
- The `usleep()` function suspends execution of the calling thread for (at least) usec **microseconds**.
- The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.
- The `usleep()` function returns 0 on success. On error, -1 is returned, with errno set to indicate the cause of the error.
- The `sleep()` function accepts time in **seconds** while `usleep()` accepts in microseconds.
- The `nanosleep()` allows the user to specify the sleep period with **nanosecond precision**.

8.1. Example

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

int main(int argc, char *argv[])
{
    printf("Sleep program for 5 seconds\n");
    sleep(5);

    printf("Sleep program for 1000000 micro seconds (1 second)\n");
    usleep(1000000);

    printf("Program finished\n");
    return 0;
}
```

output:

```
Sleep program for 5 seconds
Sleep program for 1000000 micro seconds (1 second)
Program finished
```

9. Generate random number

What is rand() ?

- The function `rand()` is used to generate the **pseudo random number**. It returns an integer value and its range is from 0 to RAND_MAX. Like we want to generate a random number between 1-6 then we use this function： **Num = rand() % 6 + 1;**

What is srand()?

- `srand()` is used to initialise random number generators. The argument is passed as a seed for generating a pseudo-random number. Whenever a different seed value is used in srand the pseudo number generator can be expected to generate different series of results the same as `rand().`

9.1. Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void){
   srand(time(0));
   printf("Randomly generated numbers are: ");
   for(int i = 0; i<5; i++)
      printf(" %d ", rand());
   return 0;
}
```

More information can be refered to: [rand() and srand() in C/C++ (tutorialspoint.com)]

10. Hint and Tips for Assignment 2

- **You must use Pthread (multithreading) to implement this assignment**.
- You can use `khbit()` to get the input character.
- You can provide each object moving on your screen with a thread, and join them to realize multithreading.
- You can use mutex to ensure the global variables is not modifed by multiple threads simultaneously.
- You can use `usleep()` to control the speed of the game.
- You can generate random number using `rand()`.