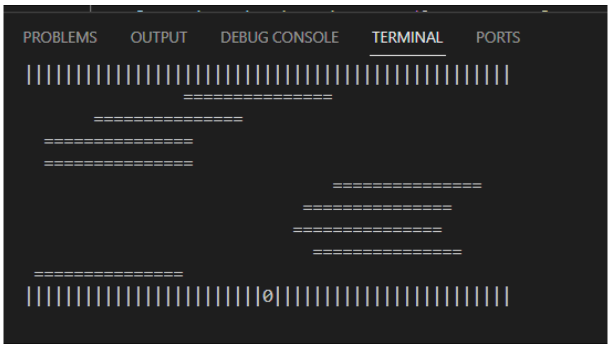
CSC3150 Assignment 2

120090211 Tao Chujun

Overview:

Project purpose:

This program is a "Frog Crosses River" game. It uses the knowledge of pthread. To win the game, player should move the frog by typing 'W', 'A', 'S', 'D' (moving up, left, down, right, respectively), from the lower bank to the higher bank. The frog will die if it hits the boundary of the lack. The users can type "Q" to exit the game.

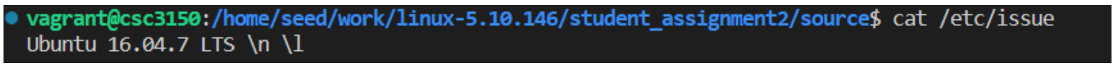


```
|||||||||||||||||||||||||||||||||||||||||||||||||
                ===============
       ===============
  ===============
  ===============
                              ===============
                           ===============
                          ===============
                            ===============
 ===============
||||||||||||||||||||||||0||||||||||||||||||||||||
```

(Beginning)

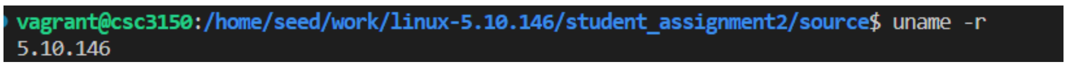
Environment:

OS:



```
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/source$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

Kernel version:



```
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/source$ uname -r
5.10.146
```

Design Logic:

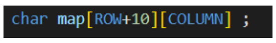
Object:

Logs: =============== (15 "=")

Frog: "0"

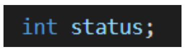
River bank: |||||||||||||||||||||||||||||||||||||||||||||||||| (50 "|")

All these objects are implemented by setting different character to a map[][].



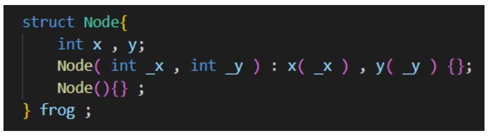
```
char map[ROW+10][COLUMN] ;
```

Status of the game: 0: Run normally 1: Interrupt (Getting input) 2: Quit



```
int status;
```

Special for Frog:

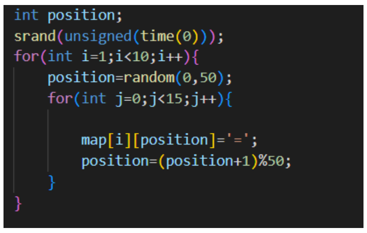


```
struct Node{
    int x , y;
    Node( int _x , int _y ) : x( _x ) , y( _y ) {};
    Node(){} ;
} frog ;
```

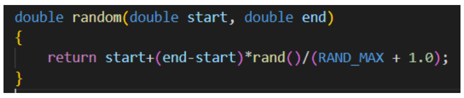
A Node struct will be used to record the location of the frog for the examination of the game status.

3 phases of the game.

- Phase 1: (Initialization)
  To start the game, first, the location of logs in each line will be randomly chosen.



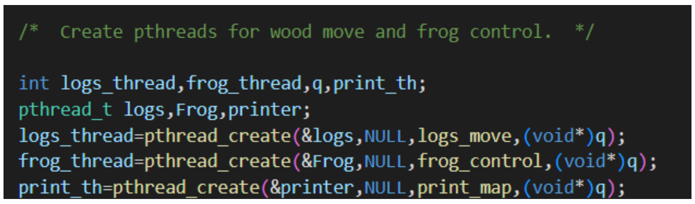
```
int position;
srand(unsigned(time(0)));
for(int i=1;i<10;i++){
    position=random(0,50);
    for(int j=0;j<15;j++){

        map[i][position]='=';
        position=(position+1)%50;
    }
}
```



```
double random(double start, double end)
{
    return start+(end-start)*rand()/(RAND_MAX + 1.0);
}
```

By using the srand function with current time (Time(0)), we can get pseudo random number with rand(). Using the above double random (double start, double end) function, a random number ranging from “start” to “end” will be generated. This helps us to randomly choose the beginning index of logs in each line. Setting the value of this location in map[][] as “=”, then it is said to set a piece of log here. After initialization, the map will be printed by the print_map thread.

Next, three threads will be created.

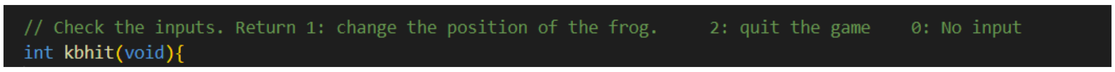


```
/*  Create pthreads for wood move and frog control.  */

int logs_thread,frog_thread,q,print_th;
pthread_t logs,Frog,printer;
logs_thread=pthread_create(&logs,NULL,logs_move,(void*)q);
frog_thread=pthread_create(&Frog,NULL,frog_control,(void*)q);
print_th=pthread_create(&printer,NULL,print_map,(void*)q);
```

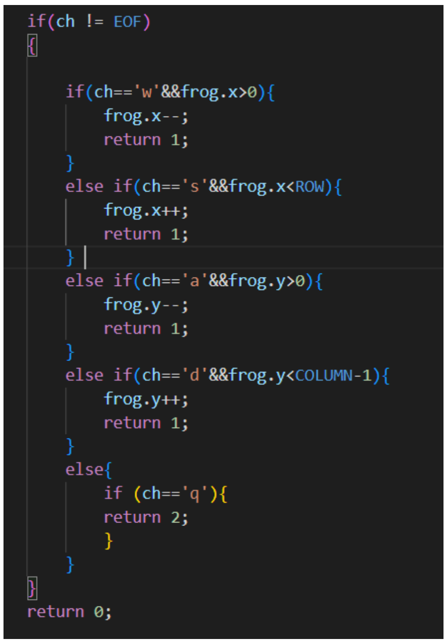
Logs:
The thread which can continuously change the position of each log.
Frog:
The thread which changes the position of frog as soon as the users input from the keyboard.
Printer:
The thread which can continuously print out all the objects.

➢ Phase 2: (Interact with keyboard input)

To achieve this function, a kbhit function will examine the input char.



```
// Check the inputs. Return 1: change the position of the frog.     2: quit the game     0: No input
int kbhit(void){
```

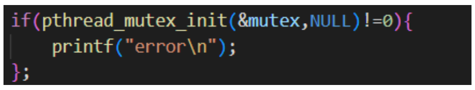


```
if(ch != EOF)
{

    if(ch=='w'&&frog.x>0){
        frog.x--;
        return 1;
    }
    else if(ch=='s'&&frog.x<ROW){
        frog.x++;
        return 1;
    }
    else if(ch=='a'&&frog.y>0){
        frog.y--;
        return 1;
    }
    else if(ch=='d'&&frog.y<COLUMN-1){
        frog.y++;
        return 1;
    }
    else{
        if (ch=='q'){
        return 2;
        }
    }
}
return 0;
```

It checks the input and change the index of frog (frog.x and frog.y) correspondingly.

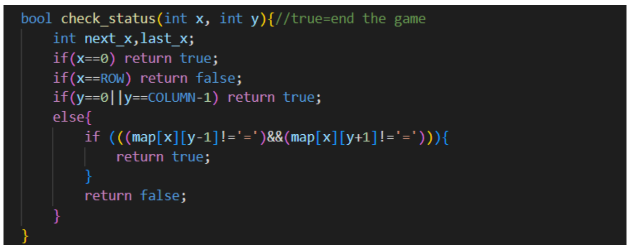
When the game starts to run, three threads will be switched and run concurrently. To ensure the safety of changing a global variable (i.e. map), using the mutex lock is a must.

The mutex lock is initialized at the beginning.



```
if(pthread_mutex_init(&mutex,NULL)!=0){
    printf("error\n");
};
```

Since we want three threads to exist during the game time, a while loop builds up the base structure of those threads. Every loop, the thread will examine the frog is alive or not using check_status function and will try to get the lock.

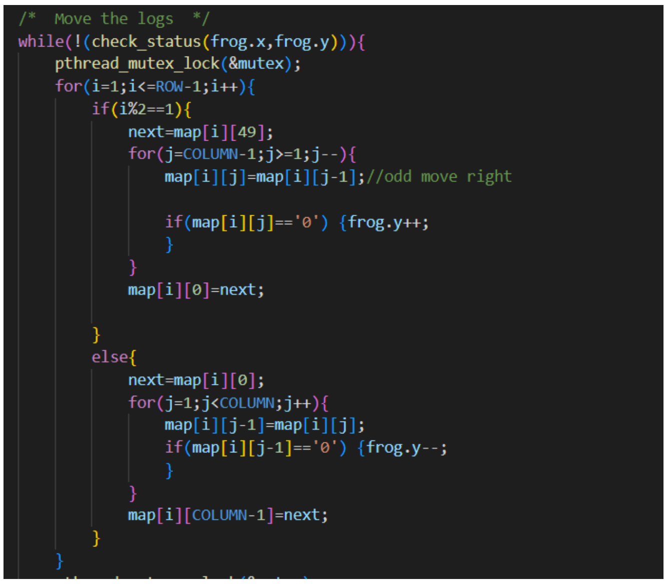


```
bool check_status(int x, int y){//true=end the game
    int next_x,last_x;
    if(x==0) return true;
    if(x==ROW) return false;
    if(y==0||y==COLUMN-1) return true;
    else{
        if (((map[x][y-1]!='=')&&(map[x][y+1]!='='))){
            return true;
        }
        return false;
    }
}
```

(if the check status returns true, it means the game should stop)

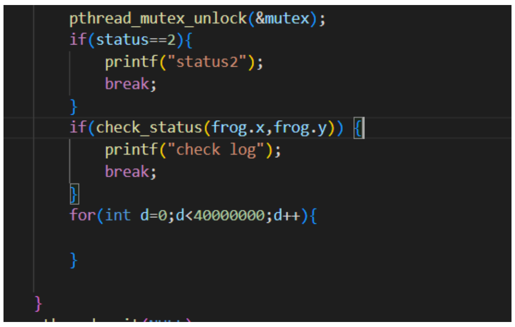
If one of the threads gets that lock, then it means it has access to the map while other threads cannot change anything to the map.

The log thread and the frog thread change the map differently.



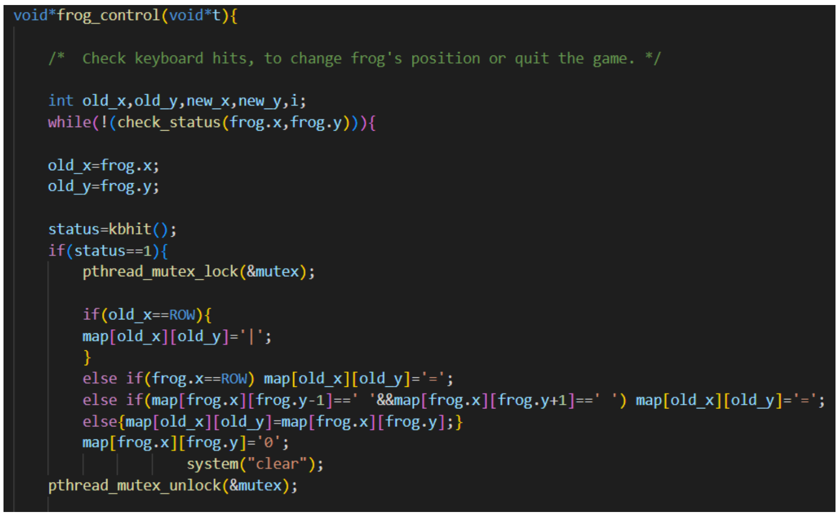
```c
/*  Move the logs  */
while(!(check_status(frog.x,frog.y))){
    pthread_mutex_lock(&mutex);
    for(i=1;i<=ROW-1;i++){
        if(i%2==1){
            next=map[i][49];
            for(j=COLUMN-1;j>=1;j--){
                map[i][j]=map[i][j-1];//odd move right

                if(map[i][j]=='0') {frog.y++;
                }
            }
            map[i][0]=next;

        }
        else{
            next=map[i][0];
            for(j=1;j<COLUMN;j++){
                map[i][j-1]=map[i][j];
                if(map[i][j-1]=='0') {frog.y--;
                }
            }
            map[i][COLUMN-1]=next;
        }
    }
```

The log thread will keep moving the logs in the map. That is, in every row with odd row number, it will move right. In every row with even row number, it will move left. After moving all the logs, the lock will be released. Then other threads can have access to the map.



```c
        pthread_mutex_unlock(&mutex);
        if(status==2){
            printf("status2");
            break;
        }
        if(check_status(frog.x,frog.y)) {
            printf("check log");
            break;
        }
        for(int d=0;d<40000000;d++){

        }

    }
```

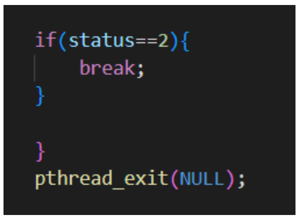
The rest of the codes are to examine the status of the game. If the frog is still alive, then nothing happens. Otherwise, it will break from the while loop and the thread will exit.



```c
void*frog_control(void*t){

    /*  Check keyboard hits, to change frog's position or quit the game. */

    int old_x,old_y,new_x,new_y,i;
    while(!(check_status(frog.x,frog.y))){

    old_x=frog.x;
    old_y=frog.y;

    status=kbhit();
    if(status==1){
        pthread_mutex_lock(&mutex);

        if(old_x==ROW){
        map[old_x][old_y]='|';
        }
        else if(frog.x==ROW) map[old_x][old_y]='=';
        else if(map[frog.x][frog.y-1]==' '&&map[frog.x][frog.y+1]==' ') map[old_x][old_y]='=';
        else{map[old_x][old_y]=map[frog.x][frog.y];}
        map[frog.x][frog.y]='0';
                    system("clear");
    pthread_mutex_unlock(&mutex);
```

The frog thread will change the map only when the keyboard hits. Once the value of status becomes 1, then it will set the map value to move the frog.

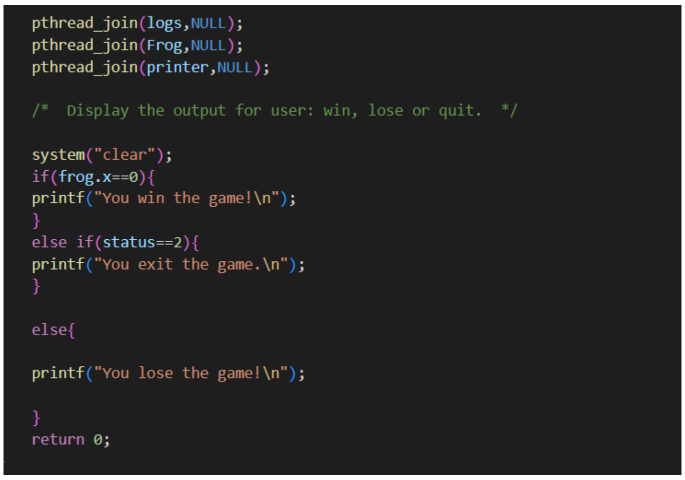
Having completed this action, the lock will be released.



```
if(status==2){
    break;
}

}
pthread_exit(NULL);
```

Similarly, if the game is stopped by the user, it will break from the while loop and the frog thread will exit.

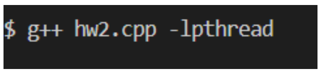
➢ Phase 3 (End of the game)
In this phase we will join the threads and examine the status of game.



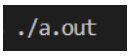
```
pthread_join(logs,NULL);
pthread_join(Frog,NULL);
pthread_join(printer,NULL);

/*  Display the output for user: win, lose or quit.  */

system("clear");
if(frog.x==0){
printf("You win the game!\n");
}
else if(status==2){
printf("You exit the game.\n");
}

else{

printf("You lose the game!\n");

}
return 0;
```

If the frog moves to the upper bank, then the user wins. If the frog hits the boundary, then the user loses. If the user stops the game, then the game will exit with hint message.

Run the program:



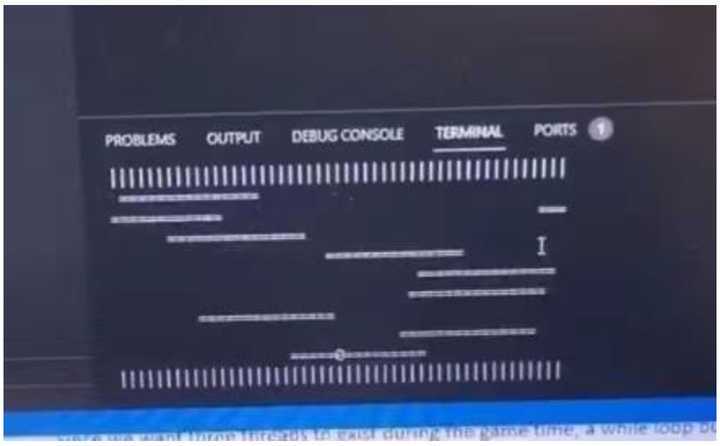
```
$ g++ hw2.cpp -lpthread
```

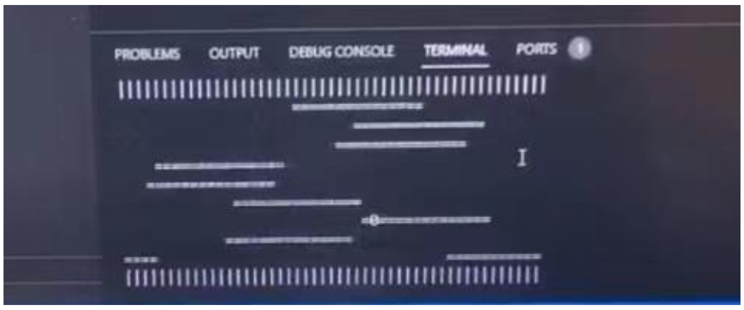


```
./a.out
```

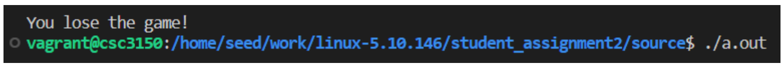
Sample Output:

Moving the frog:

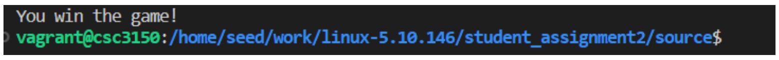




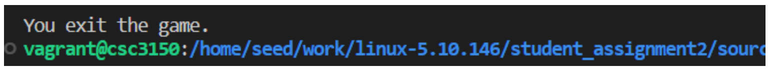
End the game:



```
You lose the game!
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/source$ ./a.out
```



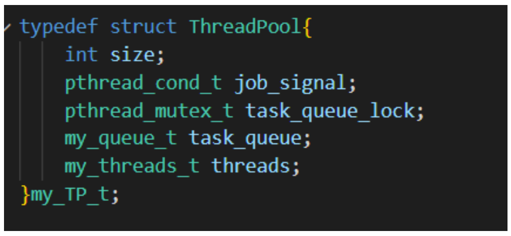
```
You win the game!
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/source$
```



```
You exit the game.
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/sourc
```

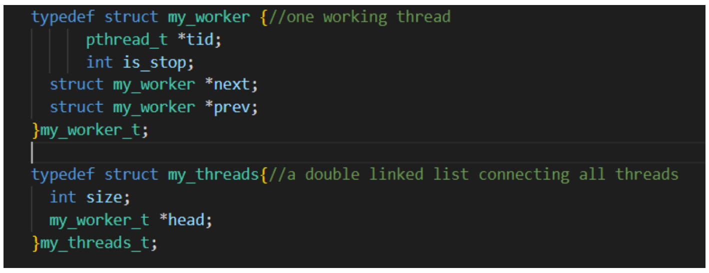
Bonus HW:

The bonus part is to implement a thread poll. In the thread poll, the number of threads is pre-determined. The thread poll struct is implemented as below.



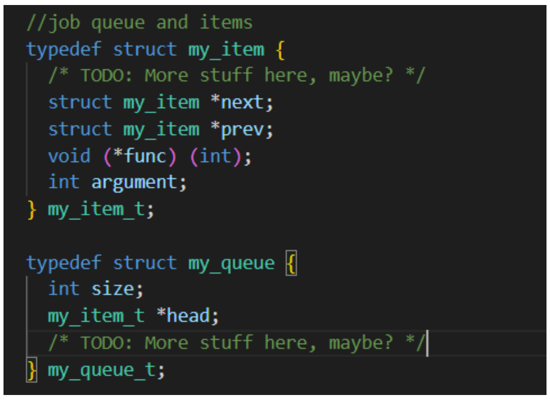
```
typedef struct ThreadPool{
    int size;
    pthread_cond_t job_signal;
    pthread_mutex_t task_queue_lock;
    my_queue_t task_queue;
    my_threads_t threads;
}my_TP_t;
```

Job_signal: a condition variable
Task_queue_lock: a lock for manipulating the task queue.
Threads: a linked list of threads. They will be reused again and again.
Task_queue: a linked list of tasks. Tasks will be put at the run time.



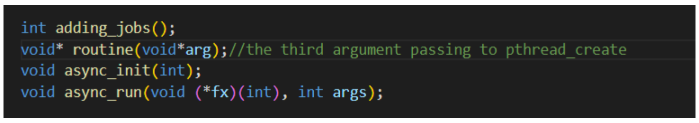
```c
typedef struct my_worker {//one working thread
        pthread_t *tid;
        int is_stop;
    struct my_worker *next;
    struct my_worker *prev;
}my_worker_t;

typedef struct my_threads{//a double linked list connecting all threads
    int size;
    my_worker_t *head;
}my_threads_t;
```

Worker struct helps us to manage the threads.



```c
//job queue and items
typedef struct my_item {
  /* TODO: More stuff here, maybe? */
  struct my_item *next;
  struct my_item *prev;
  void (*func) (int);
  int argument;
} my_item_t;

typedef struct my_queue {
  int size;
  my_item_t *head;
  /* TODO: More stuff here, maybe? */
} my_queue_t;
```

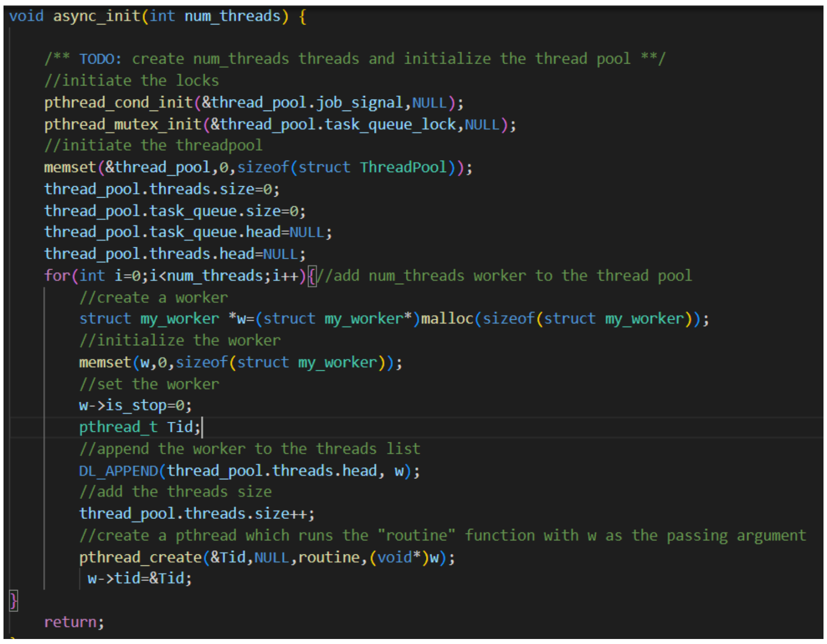
My_item is a task. My_queue is the task queue.



```c
int adding_jobs();
void* routine(void*arg);//the third argument passing to pthread_create
void async_init(int);
void async_run(void (*fx)(int), int args);
```

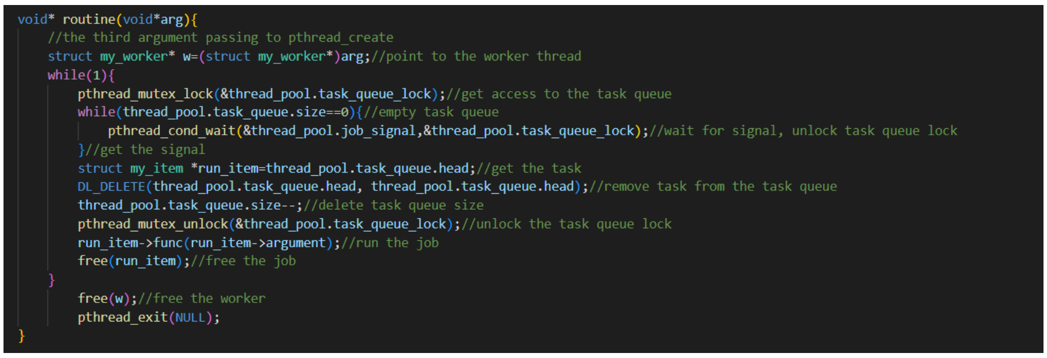
These are four methods we use to create and reuse the threads.
Adding_jobs is called when we put a new task to the task queue.
Routine is the function which every thread will execute.
Async_init is called when we initialized the thread.
Async_run is called when there is a new task available.

Desing Logic: (Next page)



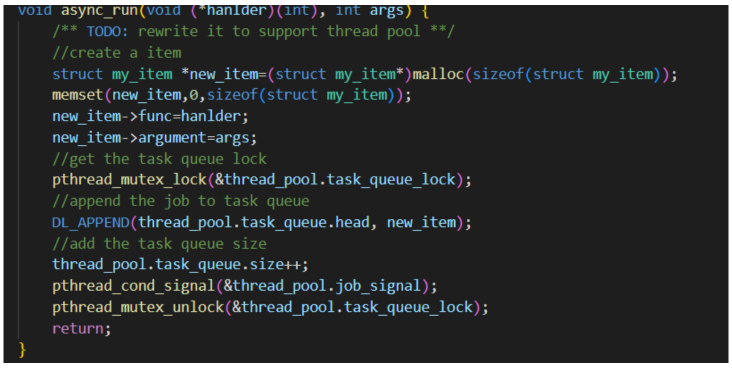
```c
void async_init(int num_threads) {

    /** TODO: create num_threads threads and initialize the thread pool **/
    //initiate the locks
    pthread_cond_init(&thread_pool.job_signal,NULL);
    pthread_mutex_init(&thread_pool.task_queue_lock,NULL);
    //initiate the threadpool
    memset(&thread_pool,0,sizeof(struct ThreadPool));
    thread_pool.threads.size=0;
    thread_pool.task_queue.size=0;
    thread_pool.task_queue.head=NULL;
    thread_pool.threads.head=NULL;
    for(int i=0;i<num_threads;i++){//add num_threads worker to the thread pool
        //create a worker
        struct my_worker *w=(struct my_worker*)malloc(sizeof(struct my_worker));
        //initialize the worker
        memset(w,0,sizeof(struct my_worker));
        //set the worker
        w->is_stop=0;
        pthread_t Tid;
        //append the worker to the threads list
        DL_APPEND(thread_pool.threads.head, w);
        //add the threads size
        thread_pool.threads.size++;
        //create a pthread which runs the "routine" function with w as the passing argument
        pthread_create(&Tid,NULL,routine,(void*)w);
         w->tid=&Tid;
}
    return;
```

First, we initialize two locks and the threadpool. Next, we create some threads and linked them as a whole (thread list).



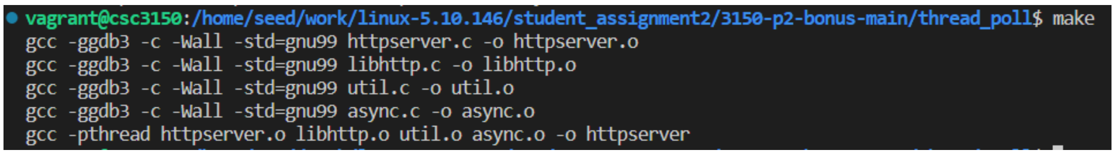
```
void* routine(void*arg){
    //the third argument passing to pthread_create
    struct my_worker* w=(struct my_worker*)arg;//point to the worker thread
    while(1){
        pthread_mutex_lock(&thread_pool.task_queue_lock);//get access to the task queue
        while(thread_pool.task_queue.size==0){//empty task queue
            pthread_cond_wait(&thread_pool.job_signal,&thread_pool.task_queue_lock);//wait for signal, unlock task queue lock
        }//get the signal
        struct my_item *run_item=thread_pool.task_queue.head;//get the task
        DL_DELETE(thread_pool.task_queue.head, thread_pool.task_queue.head);//remove task from the task queue
        thread_pool.task_queue.size--;//delete task queue size
        pthread_mutex_unlock(&thread_pool.task_queue_lock);//unlock the task queue lock
        run_item->func(run_item->argument);//run the job
        free(run_item);//free the job
    }
        free(w);//free the worker
        pthread_exit(NULL);
}
```

Every thread will run a routine function. They will wait until the task queue is not empty and they happens to get the lock luckily. Then it will dequeue the task queue, get a task and releases the task lock to other threads. The task will be executed in no time.

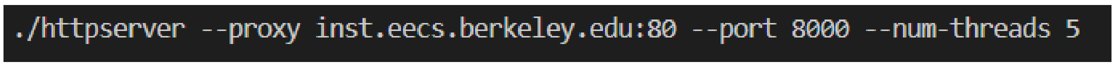


```
void async_run(void (*hanlder)(int), int args) {
    /** TODO: rewrite it to support thread pool **/
    //create a item
    struct my_item *new_item=(struct my_item*)malloc(sizeof(struct my_item));
    memset(new_item,0,sizeof(struct my_item));
    new_item->func=hanlder;
    new_item->argument=args;
    //get the task queue lock
    pthread_mutex_lock(&thread_pool.task_queue_lock);
    //append the job to task queue
    DL_APPEND(thread_pool.task_queue.head, new_item);
    //add the task queue size
    thread_pool.task_queue.size++;
    pthread_cond_signal(&thread_pool.job_signal);
    pthread_mutex_unlock(&thread_pool.task_queue_lock);
    return;
}
```

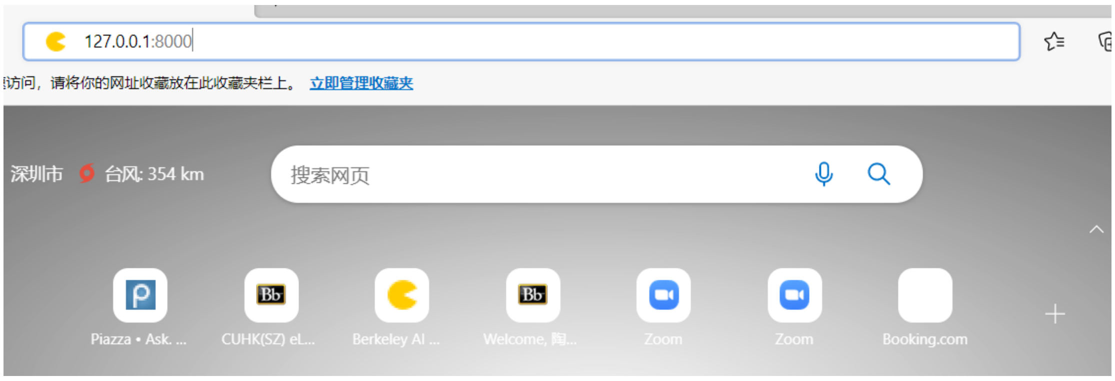
The async_run is called in the httpserver.c whenever a new task comes. Once there is a new task, this thread will try to get the task queue lock in order to safely manipulate the task queue. An item will be created and the handler function and the arguments will be stored in two fields of the my_item struct. After initialization, the new task will be added to the task queue and the size of the task queue will be increased by 1. Then it will signal the threads (wake them up) and release the task queue lock for those threads convenience.
Run the program:



```
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_assignment2/3150-p2-bonus-main/thread_poll$ make
gcc -ggdb3 -c -Wall -std=gnu99 httpserver.c -o httpserver.o
gcc -ggdb3 -c -Wall -std=gnu99 libhttp.c -o libhttp.o
gcc -ggdb3 -c -Wall -std=gnu99 util.c -o util.o
gcc -ggdb3 -c -Wall -std=gnu99 async.c -o async.o
gcc -pthread httpserver.o libhttp.o util.o async.o -o httpserver
```



```
./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000 --num-threads 5
```



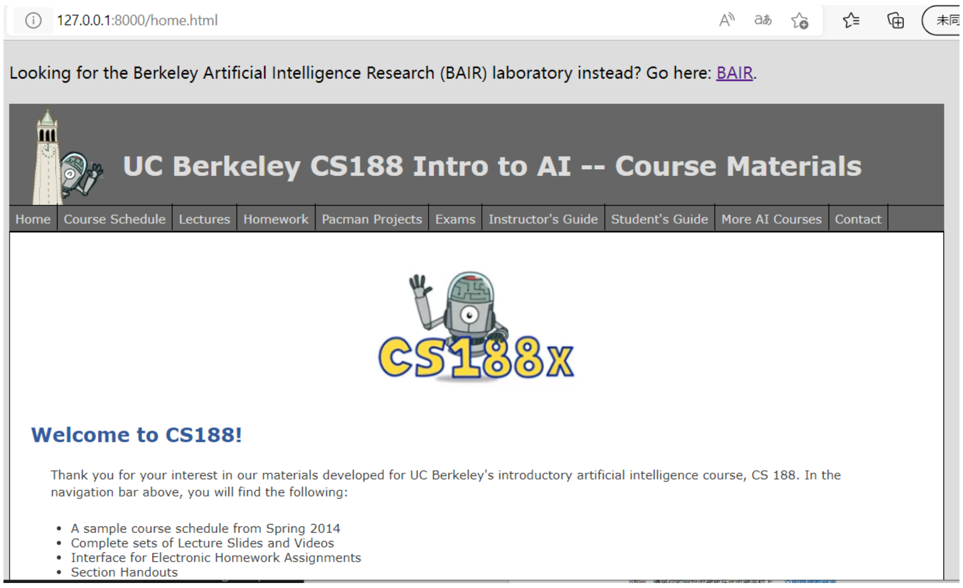
127.0.0.1:8000

访问，请将你的网址收藏放在此收藏夹栏上。 立即管理收藏夹

深圳市 台风: 354 km

搜索网页

Sample output:



127.0.0.1:8000/home.html

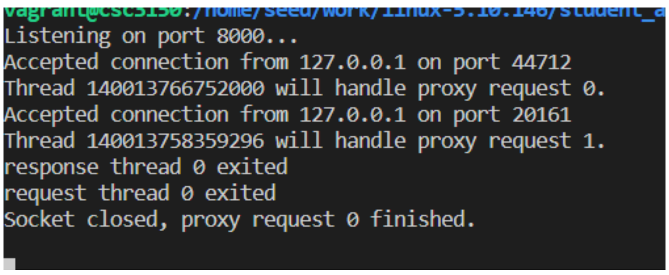
Looking for the Berkeley Artificial Intelligence Research (BAIR) laboratory instead? Go here: BAIR.

UC Berkeley CS188 Intro to AI -- Course Materials

Welcome to CS188!

Thank you for your interest in our materials developed for UC Berkeley's introductory artificial intelligence course, CS 188. In the navigation bar above, you will find the following:

- A sample course schedule from Spring 2014
- Complete sets of Lecture Slides and Videos
- Interface for Electronic Homework Assignments
- Section Handouts

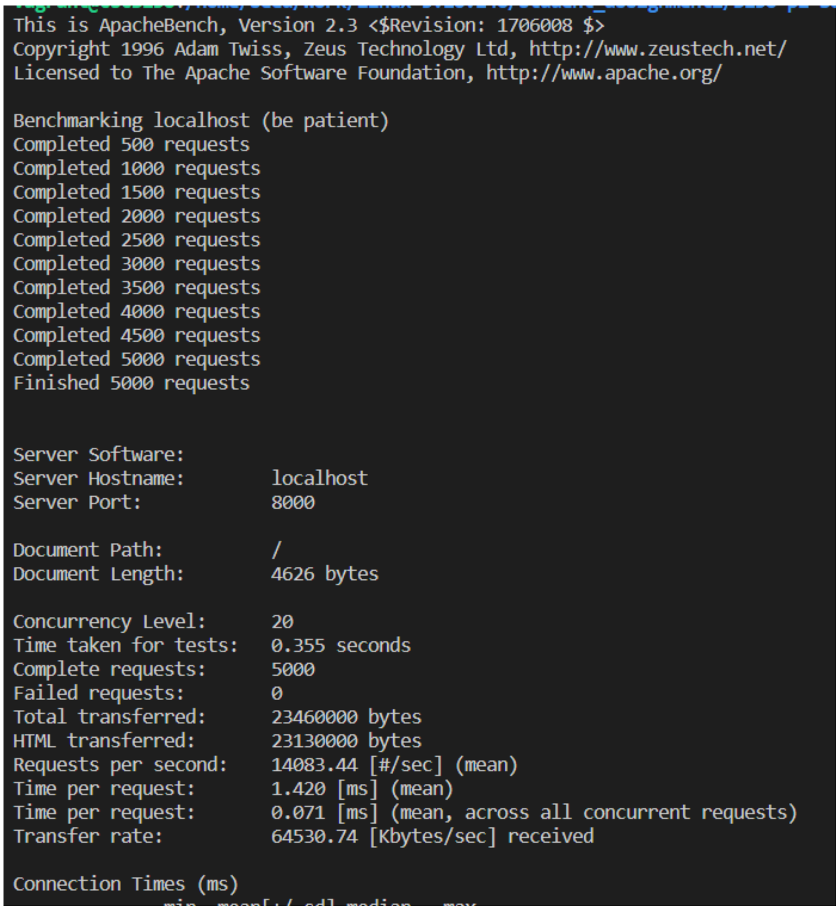


```
vagrant@csc3150:/home/seed/work/linux-5.10.146/student_a
Listening on port 8000...
Accepted connection from 127.0.0.1 on port 44712
Thread 140013766752000 will handle proxy request 0.
Accepted connection from 127.0.0.1 on port 20161
Thread 140013758359296 will handle proxy request 1.
response thread 0 exited
request thread 0 exited
Socket closed, proxy request 0 finished.
```

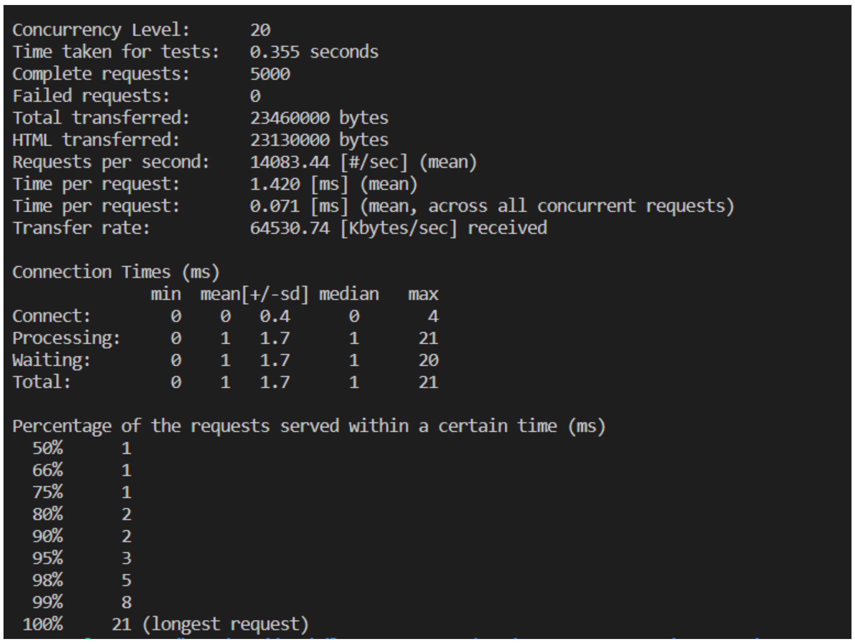
Testing with another method:

```
$ ./httpserver --files files/ --port 8000 --num-threads 20
```

```
ab -n 5000 -c 20 http://localhost:8000/
```



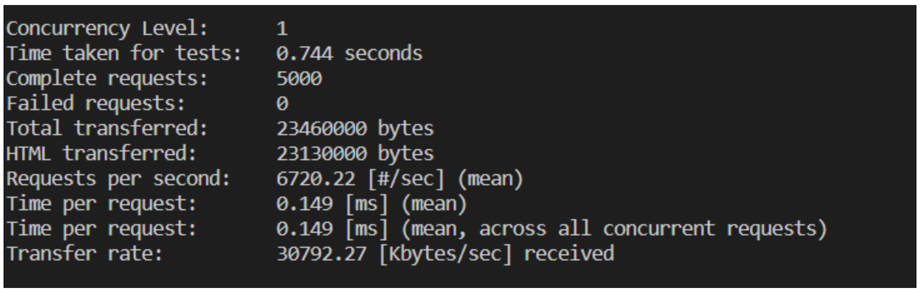
```
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests


Server Software:
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        4626 bytes

Concurrency Level:      20
Time taken for tests:   0.355 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    14083.44 [#/sec] (mean)
Time per request:       1.420 [ms] (mean)
Time per request:       0.071 [ms] (mean, across all concurrent requests)
Transfer rate:          64530.74 [Kbytes/sec] received

Connection Times (ms)
```



```
Concurrency Level:      20
Time taken for tests:   0.355 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    14083.44 [#/sec] (mean)
Time per request:       1.420 [ms] (mean)
Time per request:       0.071 [ms] (mean, across all concurrent requests)
Transfer rate:          64530.74 [Kbytes/sec] received

Connection Times (ms)
```

```
              min  mean[+/-sd] median   max
Connect:        0    0   0.4      0       4
Processing:     0    1   1.7      1      21
Waiting:        0    1   1.7      1      20
Total:          0    1   1.7      1      21

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      2
  90%      2
  95%      3
  98%      5
  99%      8
 100%     21 (longest request)
```

Compared with 1 thread:



```
Concurrency Level:      1
Time taken for tests:   0.744 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    6720.22 [#/sec] (mean)
Time per request:       0.149 [ms] (mean)
Time per request:       0.149 [ms] (mean, across all concurrent requests)
Transfer rate:          30792.27 [Kbytes/sec] received
```