```c
/*
Compile : gcc Hw4.c -o Hw4 -lpthread
To Run : ./Hw4 <number of jobs>
*/

#ifndef __QUEUE_H__
#define __QUEUE_H__
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>


typedef struct _queue
{
    int size;
    int *buffer;
    int start;
    int end;
    int count;
} queue;

struct scheduler
{
    int jobid;
    char command[100];
    char value[1000];
    char status[1000];
    char startime[1000];
    char endtime[1000];
};

struct scheduler *assignJob;

int job_counter, queue_length, argument_value;

queue *queue_init(int n);
int queue_insert(queue *q, int element);
int queue_delete(queue *q);
void queue_display(queue *q);
void queue_destroy(queue *q);
void queue_submitdisplay(queue *q);
char *subargs[1000];

queue *q;
```

```
#endif

queue *queue_init(int n)
{
    queue *q = (queue *)malloc(sizeof(queue));
    q->size = n;
    q->buffer = malloc(sizeof(int) * n);
    q->start = 0;
    q->end = 0;
    q->count = 0;

    return q;
}

int queue_insert(queue *q, int element)
{
    queue_length = queue_length + 1;
    if ((q == NULL) || (q->count == q->size))
        return -1;

    q->buffer[q->end % q->size] = element;
    q->end = (q->end + 1) % q->size;
    q->count++;

    return q->count;
}

int queue_delete(queue *q)
{
    if ((q == NULL) || (q->count == 0))
        return -1;

    int x = q->buffer[q->start];
    q->start = (q->start + 1) % q->size;
    q->count--;

    return x;
}
void queue_submitdisplay(queue *q)
{
    int i;
    if (q != NULL && q->count != 0)
    {
        printf("jobid:\tcommand\tstatus\tstarttime\tendtime\n");
        for (i = 0; i < job_counter; i++)
        {
            if (strcmp(assignJob[q->buffer[(q->start + i) % q-
>size]].status, "Done") == 0)
            {
                printf("%d", assignJob[i].jobid);
                printf("\t%s", assignJob[i].value);
                printf("\t%s", assignJob[i].status);
                printf("\t%s %s", assignJob[i].startime,
assignJob[i].endtime);
```

```
            }
        }
    }
    else
        printf("No jobs\n");
}
void queue_destroy(queue *q)
{
    free(q->buffer);
    free(q);
}

int handler(struct scheduler jobs)
{
    if (strcasecmp(jobs.command, "submit") == 0)
    {
        assignJob[jobs.jobid].jobid = jobs.jobid;
        strcpy(assignJob[jobs.jobid].command, jobs.command);

        queue_insert(q, jobs.jobid);

        printf("Job %d added to the queue \n", job_counter++);

        return 1;
    }

    else if (strcmp(jobs.command, "showjobs") == 0)
    {
        queue_display(q);
    }
    else if (strcmp(jobs.command, "submithistory") == 0)
    {
        queue_submitdisplay(q);
    }

    return 0;
}

void queue_display(queue *q)
{
    int i;

    if (q != NULL && q->count != 0 && queue_length != 0)
    {
        printf("\tjobid:\t\tcommand \t\t\t\t\tstatus \n");

        for (i = 0; i < q->count; i++)
        {
            if (strcmp(assignJob[q->buffer[(q->start + i) % q-
>size]].status, "Done") != 0)
            {
                printf("\t%d\t", q->buffer[(q->start + i) % q->size]);
                printf("\t%s\t", assignJob[q->buffer[(q->start + i) % q-
>size]].value);
```

```
                printf("\t\t%s\n", assignJob[q->buffer[(q->start + i) %
q->size]].status);
            }
        }
        printf("\n");
    }
    else
        printf("no jobs\n");
}

void *compute(void *args)
{
    pid_t pid;
    char file_pid[BUFSIZ], file_err[BUFSIZ];
    int fderr, fdout, status;

    time_t now;
    time(&now);
    int value = *(int *)args;
    pid = fork();
    if (pid == 0)
    {
        snprintf(file_pid, BUFSIZ, "%d.out", value);
        snprintf(file_err, BUFSIZ, "%d.err", value);

        fdout = open(file_pid, O_CREAT | O_TRUNC | O_WRONLY, 0755);
        fderr = open(file_err, O_CREAT | O_TRUNC | O_WRONLY, 0755);

        if (fdout == -1 || fderr == -1)
        {
            printf("Error opening file %d\n", getpid());
            exit(-1);
        }

        dup2(fdout, 1);
        dup2(fderr, 2);
        execvp(subargs[0], subargs);
    }
    else if (pid > 0)
    {

        strcpy(assignJob[value].status, "Running");
        strcpy(assignJob[value].startime, ctime(&now));

        waitpid(pid, &status, WUNTRACED);
        time(&now);
        strcpy(assignJob[value].status, "Done");
        strcpy(assignJob[value].endtime, ctime(&now));

        queue_length = queue_length - 1;
    }
    return NULL;
}
void *reCompute(void *args)
```

```
{
    pthread_t tid1[1000];
    int jobid = *(int *)args;
    int value = job_counter;

    while (1)
    {
        if (value > 0 && queue_length <= argument_value)
        {
            pthread_create(&tid1[value], NULL, compute, &(jobid));
            pthread_detach(tid1[value]);
            value = queue_length - value;
        }
        sleep(2);
    }
    return NULL;
}

int main(int argc, char **argv)
{
    char *line = NULL;
    size_t maxlen = 0;
    ssize_t n;
    job_counter = 0;
    queue_length = 0;

    pthread_t tid[1000];

    q = queue_init(100);

    struct scheduler jobs;
    argument_value = atoi(argv[1]);
    if (argument_value > 8)
        argument_value = 8;

    assignJob = malloc(sizeof(struct scheduler));

    while (argv[1])
    {
        int count = 0;
        printf("Enter Job Command > ");

        if ((n = getline(&line, &maxlen, stdin)) > 0)
        {
            if (strlen(line) == 1)
            {
                printf("InvalidCommand\n");
            }
            else
            {
                strcpy(jobs.command, strtok(line, " \n\t"));

                jobs.jobid = job_counter;
                if (handler(jobs) == 1)
                {
                    strcpy(assignJob[jobs.jobid].status, "Waiting");
                    strcpy(jobs.value, strtok(NULL, ""));
                    jobs.value[strcspn(jobs.value, "\n")] = 0;
                    strcpy(assignJob[jobs.jobid].value, jobs.value);
                    char *token = strtok(jobs.value, " ");
                    while (token != NULL)
                    {
                        if (strcmp(token, " ") != 0)
                        {
                            subargs[count++] = token;
                        }
                        token = strtok(NULL, " ");
                    }
                    subargs[count++] = NULL;
                    pthread_create(&tid[job_counter], NULL, reCompute,
&(jobs.jobid));
                }
            }
        }
    }
    free(line);
}
```