```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>

#define MX_JS 1024
#define INI 0
#define R_ERR -1
#define R_SUCC 0
#define EQUAL(a, b) (strcmp(a, b) == 0)

int MxJsInQue = MX_JS;
int CUR_JS_RNG = INI;
int MX_CNCRT_JS;
pthread_t tid;
struct jb_que *queue;
struct job **jobsrunning;
sem_t rnrs;
sem_t js_que;


enum job_status
{
    WAITING,
    RUNNING,
    SUCCESS,
    FAILED
};

char *getstatus(int status);
struct job *jobinit(int jid, char *command);
int sv_jb_inf(struct job *job);
struct job *copyjob(struct job *job);
struct job *dequeue(struct jb_que *queue);
int dlt_que(struct jb_que *queue);
int en_que(struct jb_que *queue, struct job *job);
int addrngjb(struct job *job);
void set_jobs(int argc, char *argv[]);
struct job *removerunningjob(struct job *job);
void *wt_fr_res(void *arg);
void *jb_hdlr(void *arg);
void *inpt_hdlr();
void createnewlogfile();

struct job
{
    int jid;
    char *jb_cmnd;
    int crnt_stts;
    int ext_stat;
    int pid;
    pthread_t tid;
    time_t strt_tme;
    time_t end_tm;
};
struct jb_que
{
    int cpcty;
    int sz;
    int fnt;
    int rear;
    struct job **jobs;
};

struct job *jobinit(int jid, char *command)
{
    struct job *newjob = (struct job *)malloc(sizeof(struct job));
    newjob->jid = jid;
    newjob->jb_cmnd = strdup(command);
    newjob->crnt_stts = INI;
    newjob->ext_stat = INI;
    newjob->pid = INI;
    newjob->tid = INI;
    newjob->strt_tme = INI;
    newjob->end_tm = INI;
    return newjob;
}
char *getstatus(int status)
{
    switch (status)
    {
    case WAITING:
        return "WAITING";
    case RUNNING:
        return "RUNNING";
    case SUCCESS:
        return "SUCCESS";
    case FAILED:
        return "FAILED";
    default:
        return "UNKNOWN";
    }
}
int en_que(struct jb_que *queue, struct job *job)
{
    if (queue == NULL || job == NULL)
    {
        return R_ERR;
    }
    if (queue->sz == queue->cpcty)
    {
        return R_ERR;
    }
    queue->rear = (queue->rear + 1) % queue->cpcty;
    queue->jobs[queue->rear] = job;
    queue->sz++;
    return R_SUCC;
}

struct job *copyjob(struct job *job)
{
    struct job *newjob = (struct job *)malloc(sizeof(struct job));
    newjob->jid = job->jid;
    newjob->jb_cmnd = strdup(job->jb_cmnd);
    newjob->crnt_stts = job->crnt_stts;
    newjob->ext_stat = job->ext_stat;
    newjob->pid = job->pid;
    newjob->tid = job->tid;
    newjob->strt_tme = job->strt_tme;
    newjob->end_tm = job->end_tm;
    return newjob;
}


int sv_jb_inf(struct job *job)
{
    char info[MX_JS];
    char *status, *strt_tme, *end_tm;
    status = getstatus(job->crnt_stts);
    char buffer[80];
    struct tm *tm_info;
    tm_info = localtime(&job->strt_tme);
    strftime(buffer, 80, "%a %b %d %H:%M:%S %Y", tm_info);
    strt_tme = strdup(buffer);
    tm_info = localtime(&job->end_tm);
    strftime(buffer, 80, "%a %b %d %H:%M:%S %Y", tm_info);
    end_tm = strdup(buffer);
    strt_tme[strcspn(strt_tme, "\n")] = INI;
    end_tm[strcspn(end_tm, "\n")] = INI;
    FILE *fp;
    fp = fopen("log.txt", "a");
    if (fp == NULL)
    {
        printf("Error opening file.\n");
        return 1;
    }
    sprintf(info, "%d\t%s\t%s\t%s\t%s\n",
            job->jid, job->jb_cmnd,
            strt_tme, end_tm, status);
    fputs(info, fp);
    fclose(fp);
    return R_SUCC;
}
struct job *dequeue(struct jb_que *queue)
{
    if (queue == NULL)
    {
        return NULL;
    }
    if (queue->sz == 0)
    {
        return NULL;
    }
    struct job *job = queue->jobs[queue->fnt];
    queue->fnt = (queue->fnt + 1) % queue->cpcty;
    queue->sz--;
    return job;
}
void createnewlogfile()
{
    FILE *fp;
    fp = fopen("log.txt", "w");
    if (fp == NULL)
    {
        printf("Error opening file.\n");
        return;
    }
    fclose(fp);
}
int addrngjb(struct job *job)
{
    if (job == NULL)
    {
        return R_ERR;
    }
    if (CUR_JS_RNG == MX_CNCRT_JS)
    {
        return R_ERR;
    }
    jobsrunning[CUR_JS_RNG] = job;
    CUR_JS_RNG++;
    return R_SUCC;
}
struct job *removerunningjob(struct job *job)
{
    if (job == NULL)
    {
        return NULL;
    }
    int i;
    for (i = INI; i < CUR_JS_RNG; i++)
    {
        if (jobsrunning[i]->jid == job->jid)
        {
            break;
}
    }
    if (i == CUR_JS_RNG)
    {
        return NULL;
    }
    struct job *jobtoremove = copyjob(jobsrunning[i]);
    for (; i < CUR_JS_RNG - 1; i++)
    {
        jobsrunning[i] = jobsrunning[i + 1];
    }
    CUR_JS_RNG--;
    return jobtoremove;
}

int dlt_que(struct jb_que *queue)
{
    if (queue == NULL)
    {
        return R_ERR;
    }
    int i;
    for (i = INI; i < queue->sz; i++)
    {
        struct job *job = queue->jobs[(queue->fnt + i) % queue-
>cpcty];
        free(job->jb_cmnd);
        free(job);
    }
    free(queue->jobs);
    free(queue);
    queue = NULL;

}
void *wt_fr_res(void *arg)
{
    struct job *job = (struct job *)arg;
    if (job == NULL)
    {
        return NULL;
    }
    int status;
    waitpid(job->pid, &status, 0);
    job->end_tm = time(NULL);
    if (WIFEXITED(status))
    {
        job->ext_stat = WEXITSTATUS(status);
        if (job->ext_stat == 0)
        {
            job->crnt_stts = R_SUCC;
        }
        else
        {
            job->crnt_stts = FAILED;
        }
    }
    else
    {
```

```c
        job->crnt_stts = FAILED;
    }
    sv_jb_inf(job);
    removerunningjob(job);
    sem_post(&rnrs);
}
void *jb_hdlr(void *arg)
{
    while (1)
    {
        sem_wait(&js_que);
        struct job *job = dequeue(queue);
        sem_wait(&rnrs);
        job->crnt_stts = RUNNING;
        job->strt_tme = time(NULL);
        addrngjb(job);
        int pid = fork();
        if (pid == 0)
        {
            char *outputfile = (char *)malloc(sizeof(char) * MX_JS);
            char *errorfile = (char *)malloc(sizeof(char) * MX_JS);
            char *command = job->jb_cmnd;
            sprintf(outputfile, "%d.out", job->jid);
            sprintf(errorfile, "%d.err", job->jid);
            freopen(outputfile, "w", stdout);
            freopen(errorfile, "w", stderr);
            char *args[100];
            char *token;
            int i = INI;
            token = strtok(command, " ");
            while (token != NULL)
            {
                args[i] = token;
                token = strtok(NULL, " ");
                i++;
            }
            args[i] = NULL;

            execvp(args[0], args);
            exit(EXIT_FAILURE);
        }
        else if (pid > 0)
        {
            job->pid = pid;
            job->tid = pthread_self();
            pthread_t tid;
            pthread_create(&tid, NULL, wt_fr_res, (void *)job);
        }
        else
        {
            printf("fork failed\n");
            job->crnt_stts = FAILED;
        }
    }
    return NULL;
}


void *inpt_hdlr()
{
    char *line = NULL, *token = NULL, *command = NULL;
    size_t len = INI;
    ssize_t read;
    int jid = INI;
    while (1)
    {
        printf("scheduler>");
        read = getline(&line, &len, stdin) != -1;
        if (line[read - 1] == '\n')
        {
            line[read - 1] = '\0';
        }
        token = strtok(line, " \t\r\n\0");
        if (EQUAL(token, "submit"))
        {
            command = strtok(NULL, "\r\n\0");
            struct job *newjob = jobinit(jid++, command);
            if (en_que(queue, newjob) == R_ERR)
            {
                printf("waiting queue is full. wait for some time.
\n");
            }
            else
            {
                printf("job %d added to the queue\n", newjob->jid);
                sem_post(&js_que);
            }
        }
        else if (EQUAL(token, "submithistory"))
        {
            FILE *fp;
            char ch;
            fp = fopen("log.txt", "r");
            if (fp == NULL)
            {
                printf("Error opening file.\n");
                return NULL;
            }
printf("jid\tcommand\tstrt_tme\tend_tm\tstatus\n");
            while ((ch = fgetc(fp)) != EOF)
            {
                printf("%c", ch);
            }
            fclose(fp);
        }
        else if (EQUAL(token, "showjobs"))
        {
            int i;
            printf("jid\tcommand\t\tstatus\n");
            for (i = INI; i < CUR_JS_RNG; i++)
            {
                printf("%d\t%s\t%s\n", jobsrunning[i]->jid,
jobsrunning[i]->jb_cmnd, getstatus(jobsrunning[i]->crnt_stts));
            }
            if (queue == NULL)
            {
                return NULL;
            }
            for (i = INI; i < queue->sz; i++)
            {
                struct job *job = queue->jobs[(queue->fnt + i) %
queue->cpcty];
                printf("%d\t%s\t\t%s\n", job->jid, job->jb_cmnd,
getstatus(job->crnt_stts));
            }
        }
        else if (EQUAL(token, "exit"))
        {
            break;
        }
        else
        {
            printf("invalid command\n");
        }
    }
    free(line);
    return NULL;

}
void set_jobs(int argc, char *argv[])
{
    int temp = INI;
    MX_CNCRT_JS = sysconf(_SC_NPROCESSORS_ONLN);
    if (argc > 1)
    {
        temp = atoi(argv[1]);
        if (temp > MX_CNCRT_JS)
        {
            printf("No.of jobs shouldn't be > than no.of cores\n");
            printf("Set no.of jobs to %d.\n", MX_CNCRT_JS);
        }
        else if (temp < 0)
        {
            printf("Number of jobs cannot be negative.\n");
            printf("Set no. of jobs to %d.\n", MX_CNCRT_JS);
        }
        else
        {
           MX_CNCRT_JS = temp;
        }
    }
}
```

```
int main(int argc, char *argv[])
{
    MX_CNCRT_JS = atoi(argv[1]);
    set_jobs(argc, argv);
    sem_init(&rnrs, 0, MX_CNCRT_JS);
    sem_init(&js_que, 0, 0);

    queue = (struct jb_que *)malloc(sizeof(struct jb_que));
    queue->cpcty = MxJsInQue;
    queue->sz = INI;
    queue->fnt = INI;
    queue->rear = -1;
    queue->jobs = (struct job **)malloc(sizeof(struct job *) *
MxJsInQue);

    jobsrunning = (struct job **)malloc(sizeof(struct job *) *
MX_CNCRT_JS);
    createnewlogfile();
    pthread_create(&tid, NULL, jb_hdlr, NULL);
    inpt_hdlr();
    dlt_que(queue);
    return R_SUCC;

}
```