```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include "job.h"

int jobid = 0;
int siginfo = 1;
int fifo;
int globalfd;

struct waitqueue *head = NULL;
struct waitqueue *next = NULL, *current = NULL;

void schedule()
{
	struct jobinfo *newjob = NULL;
	struct jobcmd cmd;
	int count = 0;

	bzero(&cmd, DATALEN);
	if ((count = read(fifo, &cmd, DATALEN)) < 0)
		error_sys("read fifo failed");

#ifdef DEBUG

	if (count) {
		printf("cmd cmdtype\t%d\n"
			"cmd defpri\t%d\n"
			"cmd data\t%s\n",
			cmd.type, cmd.defpri, cmd.data);
	}// else
	 //	printf("no data read\n");

#endif

	switch (cmd.type) {
	case ENQ:
		do_enq(newjob,cmd);
		break;
	case DEQ:
		do_deq(cmd);
		break;
	case STAT:
		do_stat(cmd);
		break;
	default:
		break;
	}
	/* Update jobs in waitqueue */

	updateall();
```

```
        /* select the highest priority job to run */

        next = jobselect();

        /* stop current job, run next job */

        jobswitch();
}

int allocjid()
{
        return ++jobid;
}

void updateall()
{
        struct waitqueue *p;

        /* update running job's run_time */
        if (current)
                current->job->run_time += 1;    /* add 1 represent 100 ms */

        /* update ready job's wait_time */
        for (p = head; p != NULL; p = p->next) {
                p->job->wait_time += 100;

                if (p->job->wait_time >= 1000 && p->job->curpri < 3)
                        p->job->curpri++;
        }
}

struct waitqueue* jobselect()
{
        struct waitqueue *p, *prev, *select, *selectprev;
        int highest = -1;

        select = NULL;
        selectprev = NULL;

        if (head) {
                for (prev = head, p = head; p != NULL; prev = p, p = p->next) {

                        if (p->job->curpri > highest) {
                                select = p;
                                selectprev = prev;
                                highest = p->job->curpri;
                        }
                }

                selectprev->next = select->next;

                if (select == selectprev) head = NULL;
        }

        return select;
}
```

```
void jobswitch()
{
	struct waitqueue *p;
	int i;

	if (current && current->job->state == DONE) {		/* current job finished */

		/* job has been done, remove it */
		for (i = 0; (current->job->cmdarg)[i] != NULL; i++) {
			free((current->job->cmdarg)[i]);
			(current->job->cmdarg)[i] = NULL;
		}

		free(current->job->cmdarg);
		free(current->job);
		free(current);

		current = NULL;
	}

	if (next == NULL && current == NULL)		/* no job to run */

		return;

	else if (next != NULL && current == NULL) {	/* start new job */

	   printf("begin start new job\n");
		current = next;
		next = NULL;
		current->job->state = RUNNING;
		kill(current->job->pid, SIGCONT);
		return;

	} else if (next != NULL && current != NULL) { /* do switch */

		kill(current->job->pid, SIGSTOP);
		current->job->curpri = current->job->defpri;
		current->job->wait_time = 0;
		current->job->state = READY;

		/* move back to the queue */

		if (head) {
			for (p = head; p->next != NULL; p = p->next);

			p->next = current;

		} else {
			head = current;
		}

		current = next;
		next = NULL;
		current->job->state = RUNNING;
		kill(current->job->pid, SIGCONT);

		//printf("\nbegin switch: current jid=%d, pid=%d\n",
		//		current->job->jid, current->job->pid);
```

```
            return;

        } else {    /* next == NULL && current != NULL, no switch */

            return;
        }
}

void sig_handler(int sig, siginfo_t *info, void *notused)
{
        int status;
        int ret;

        switch (sig) {
        case SIGVTALRM:
            schedule();
            return;

        case SIGCHLD:
            ret = waitpid(-1, &status, WNOHANG);
            if (ret == 0 || ret == -1)
                return;

            if (WIFEXITED(status)) {
            #ifdef DEBUG
            //printf("%d %d %d\n", ret, info->si_pid, current->job->pid);
            //do_stat();
            #endif
                current->job->state = DONE;
                printf("normal termation, exit status = %d\tjid = %d, pid = %d\n\n",
                    WEXITSTATUS(status), current->job->jid, current->job->pid);

            }  else if (WIFSIGNALED(status)) {
               printf("abnormal termation, signal number = %d\tjid = %d, pid = %d\n\n",
                    WTERMSIG(status), current->job->jid, current->job->pid);

            } else if (WIFSTOPPED(status)) {
               printf("child stopped, signal number = %d\tjid = %d, pid = %d\n\n",
                    WSTOPSIG(status), current->job->jid, current->job->pid);
            }
            return;

        default:
            return;
        }
}

void do_enq(struct jobinfo *newjob, struct jobcmd enqcmd)
{
        struct      waitqueue *newnode, *p;
        int         i=0, pid;
        char *offset, *argvec, *q;
        char **arglist;
        sigset_t    zeromask;

        sigemptyset(&zeromask);

        /* fill jobinfo struct */
```

```
        newjob = (struct jobinfo *)malloc(sizeof(struct jobinfo));
        newjob->jid = allocjid();
        newjob->defpri = enqcmd.defpri;
        newjob->curpri = enqcmd.defpri;
        newjob->ownerid = enqcmd.owner;
        newjob->state = READY;
        newjob->create_time = time(NULL);
        newjob->wait_time = 0;
        newjob->run_time = 0;
        arglist = (char**)malloc(sizeof(char*)*(enqcmd.argnum+1));
        newjob->cmdarg = arglist;
        offset = enqcmd.data;
        argvec = enqcmd.data;
        while (i < enqcmd.argnum) {

                if (*offset == ':') {

                        *offset++ = '\0';
                        q = (char*)malloc(offset - argvec);
                        strcpy(q,argvec);
                        arglist[i++] = q;
                        argvec = offset;

                } else
                        offset++;
        }

        arglist[i] = NULL;

#ifdef DEBUG

        printf("enqcmd argnum %d\n",enqcmd.argnum);
        for (i = 0; i < enqcmd.argnum; i++)
                printf("parse enqcmd:%s\n",arglist[i]);

#endif

        /* add new job to the queue */

        newnode = (struct waitqueue*)malloc(sizeof(struct waitqueue));
        newnode->next = NULL;
        newnode->job = newjob;

        if (head) {
                for (p = head; p->next != NULL; p = p->next);

                p->next = newnode;
        } else
                head = newnode;

        /* create process for the job */

        if ((pid = fork()) < 0)
                error_sys("enq fork failed");

        /* In child process */
```

```
        if (pid == 0) {

                newjob->pid = getpid();

                /* block the child wait for run */

                raise(SIGSTOP);

#ifdef DEBUG

                printf("begin running\n");
                for (i = 0; arglist[i] != NULL; i++)
                        printf("arglist %s\n",arglist[i]);

#endif

                /* dup the globalfile descriptor to stdout */
                dup2(globalfd,1);
                if (execv(arglist[0],arglist) < 0)
                        printf("exec failed\n");

                exit(1);

        } else {

                newjob->pid = pid;
                printf("\nnew job: jid=%d, pid=%d\n", newjob->jid, newjob->pid);

        }
}

/* bug to fix */
void do_deq(struct jobcmd deqcmd)
{
        [illegible]
        [illegible]

        [illegible]

[illegible]
        [illegible]

        [illegible]
        [illegible]


        [illegible]
                [illegible]


        [illegible]
```

[illegible]

```
}

void do_stat()
{
	/*
	* Print job statistics of all jobs:
	* 1. job id
	* 2. job pid
	* 3. job owner
	* 4. job run time
	* 5. job wait time
	* 6. job create time
```

```
        * 7. job state
        */

        struct waitqueue *p;
        char timebuf[BUFLEN];

        printf( "JID\tPID\tOWNER\tRUNTIME\tWAITTIME\tCREATTIME\tSTATE\n");

        if (current) {
                strcpy(timebuf,ctime(&(current->job->create_time)));
                timebuf[strlen(timebuf) - 1] = '\0';
                printf("%d\t%d\t%d\t%d\t%d\t%s\t%s\n",
                        current->job->jid,
                        current->job->pid,
                        current->job->ownerid,
                        current->job->run_time,
                        current->job->wait_time,
                        timebuf,
                        "RUNNING" );
        }

        for (p = head; p != NULL; p = p->next) {
                strcpy (timebuf,ctime(&(p->job->create_time)));
                timebuf[strlen(timebuf) - 1] = '\0';
                printf("%d\t%d\t%d\t%d\t%d\t%s\t%s\n",
                        p->job->jid,
                        p->job->pid,
                        p->job->ownerid,
                        p->job->run_time,
                        p->job->wait_time,
                        timebuf,
                        "READY" );
        }

        printf("\n");
}

int main()
{
        struct timeval interval;
        struct itimerval new,old;
        struct stat statbuf;
        struct sigaction newact,oldact1,oldact2;

        if (stat(FIFO,&statbuf) == 0) {

                /* if fifo file exists, remove it */

                if (remove(FIFO) < 0)
                        error_sys("remove failed");
        }

        if (mkfifo(FIFO,0666) < 0)
                error_sys("mkfifo failed");

        /* open fifo in nonblock mode */

        if ((fifo = open(FIFO,O_RDONLY|O_NONBLOCK)) < 0)
```

```
        error_sys("open fifo failed");

    /* open global file for job output */

    if ((globalfd = open("/dev/null",O_WRONLY)) < 0)
        error_sys("open global file failed");

    /* setup signal handler */
    newact.sa_sigaction = sig_handler;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = SA_SIGINFO;

    sigaction(SIGCHLD,&newact,&oldact1);
    sigaction(SIGVTALRM,&newact,&oldact2);

    /* timer interval: 0s, 100ms */

    interval.tv_sec = 0;
    interval.tv_usec = 100;

    new.it_interval = interval;
    new.it_value = interval;
    setitimer(ITIMER_VIRTUAL,&new,&old);

    printf("OK! Scheduler is starting now!!\n");

    while (siginfo == 1);

    close(fifo);
    close(globalfd);
    return 0;
}
```