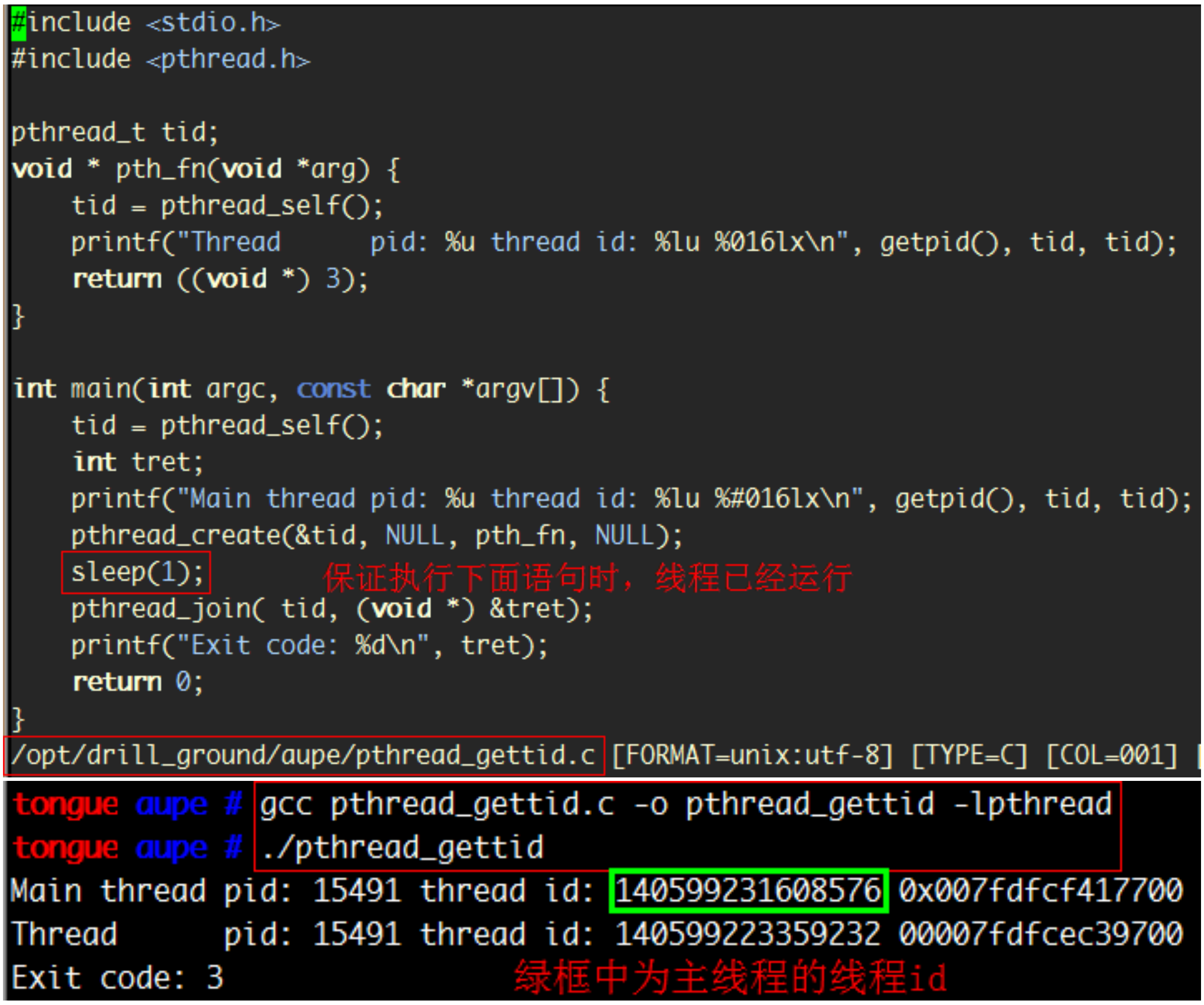
第十一章 线程



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

pthread_t tid;
void * pth_fn(void *arg) {
    tid = pthread_self();
    printf("Thread      pid: %u thread id: %lu %016lx\n", getpid(), tid, tid);
    return ((void *) 3);
}

int main(int argc, const char *argv[]) {
    tid = pthread_self();
    int tret;
    printf("Main thread pid: %u thread id: %lu %#016lx\n", getpid(), tid, tid);
    pthread_create(&tid, NULL, pth_fn, NULL);
    sleep(1);
    pthread_join( tid, (void *) &tret);
    printf("Exit code: %d\n", tret);
    return 0;
}
/opt/drill_ground/aupe/pthread_gettid.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001]
```

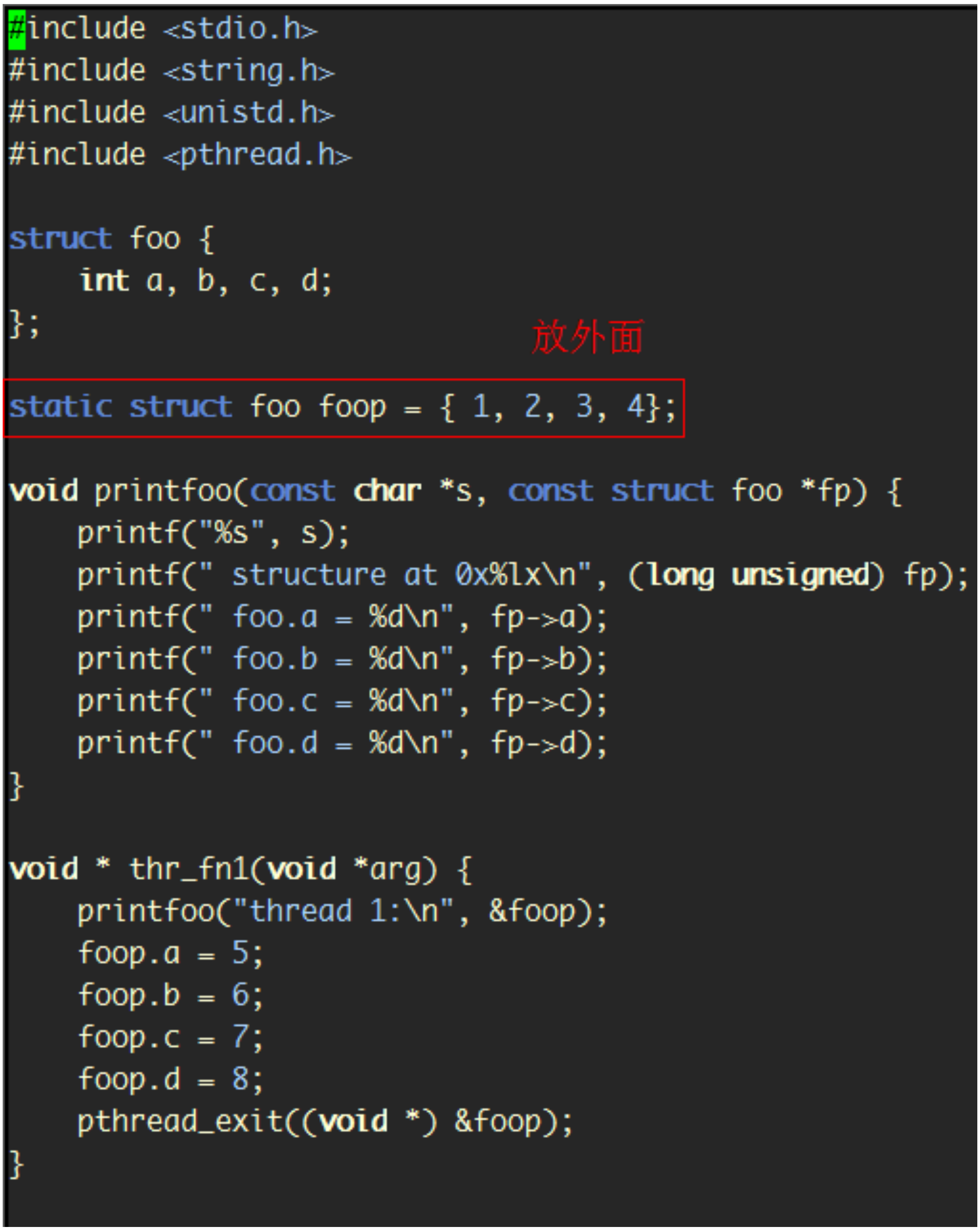
保证执行下面语句时，线程已经运行

```
tongue aupe # gcc pthread_gettid.c -o pthread_gettid -lpthread
tongue aupe # ./pthread_gettid
Main thread pid: 15491 thread id: 140599231608576 0x007fdfcf417700
Thread      pid: 15491 thread id: 140599223359232 00007fdfcec39700
Exit code: 3
```

绿框中为主线程的线程id

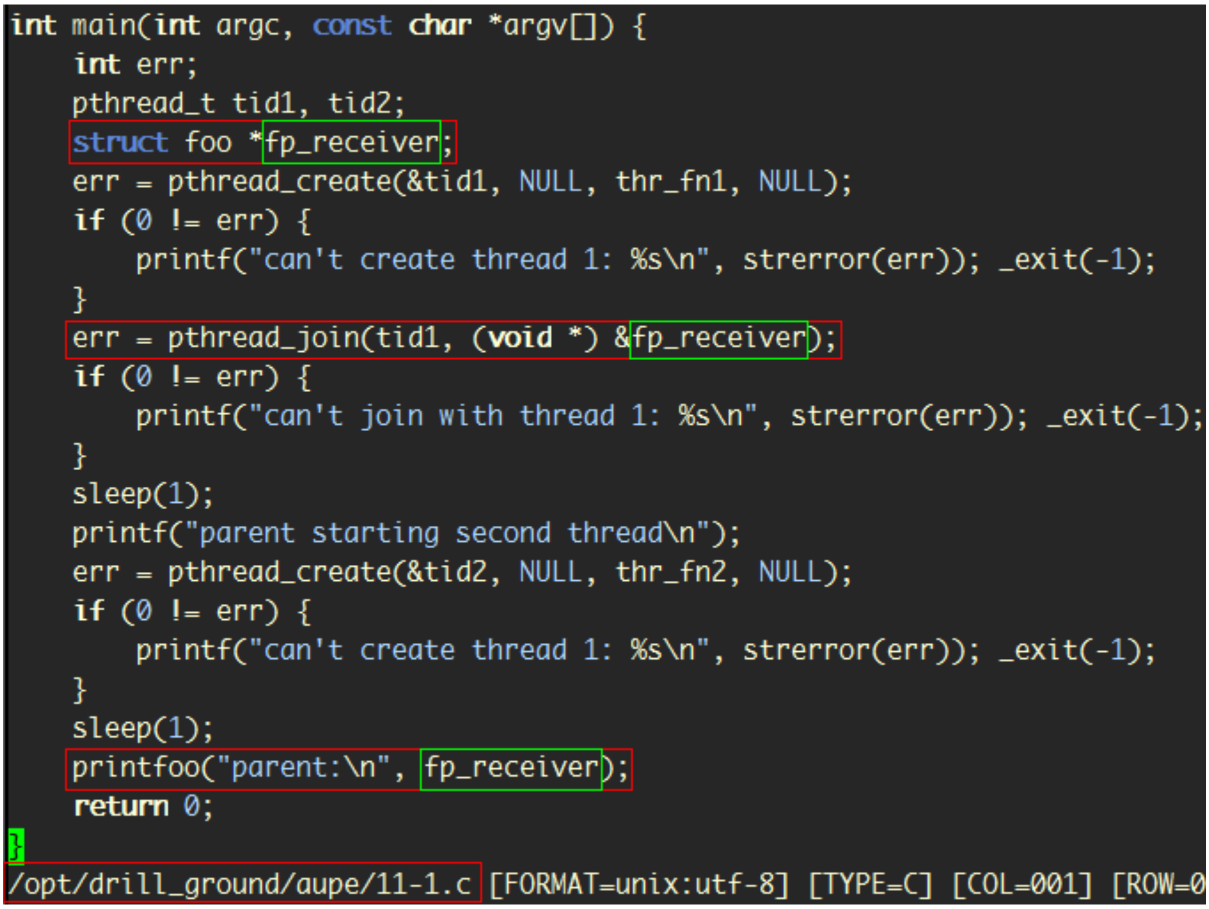
习题：

11-1、通过线程把结构体传给其接受者；

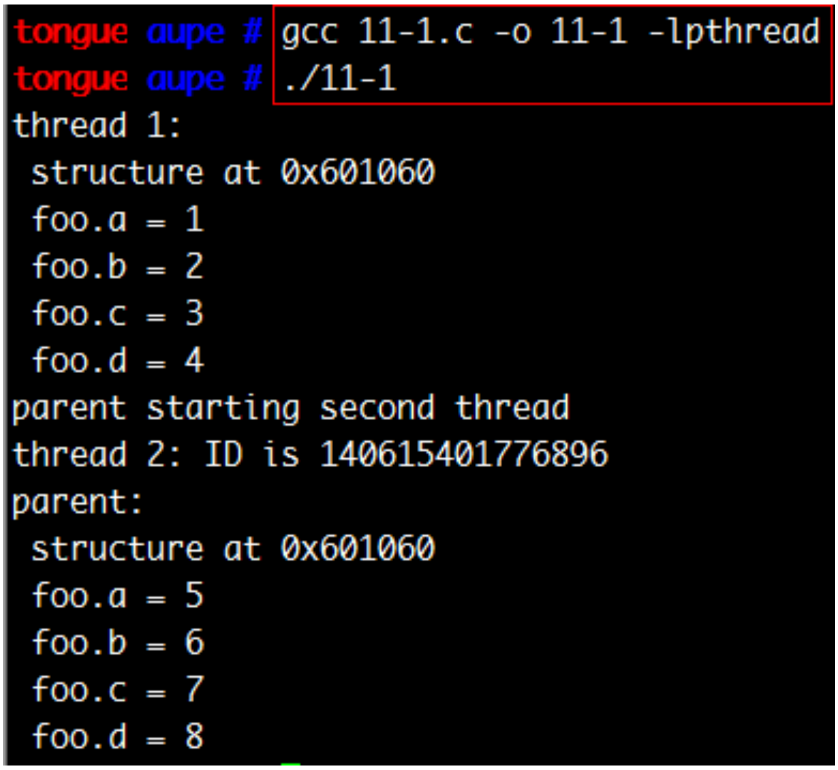


```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

static struct foo foop = { 1, 2, 3, 4};

void printfoo(const char *s, const struct foo *fp) {
    printf("%s", s);
    printf(" structure at 0x%lx\n", (long unsigned) fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void * thr_fn1(void *arg) {
    printfoo("thread 1:\n", &foop);
    foop.a = 5;
    foop.b = 6;
    foop.c = 7;
    foop.d = 8;
    pthread_exit((void *) &foop);
}
```

放外面



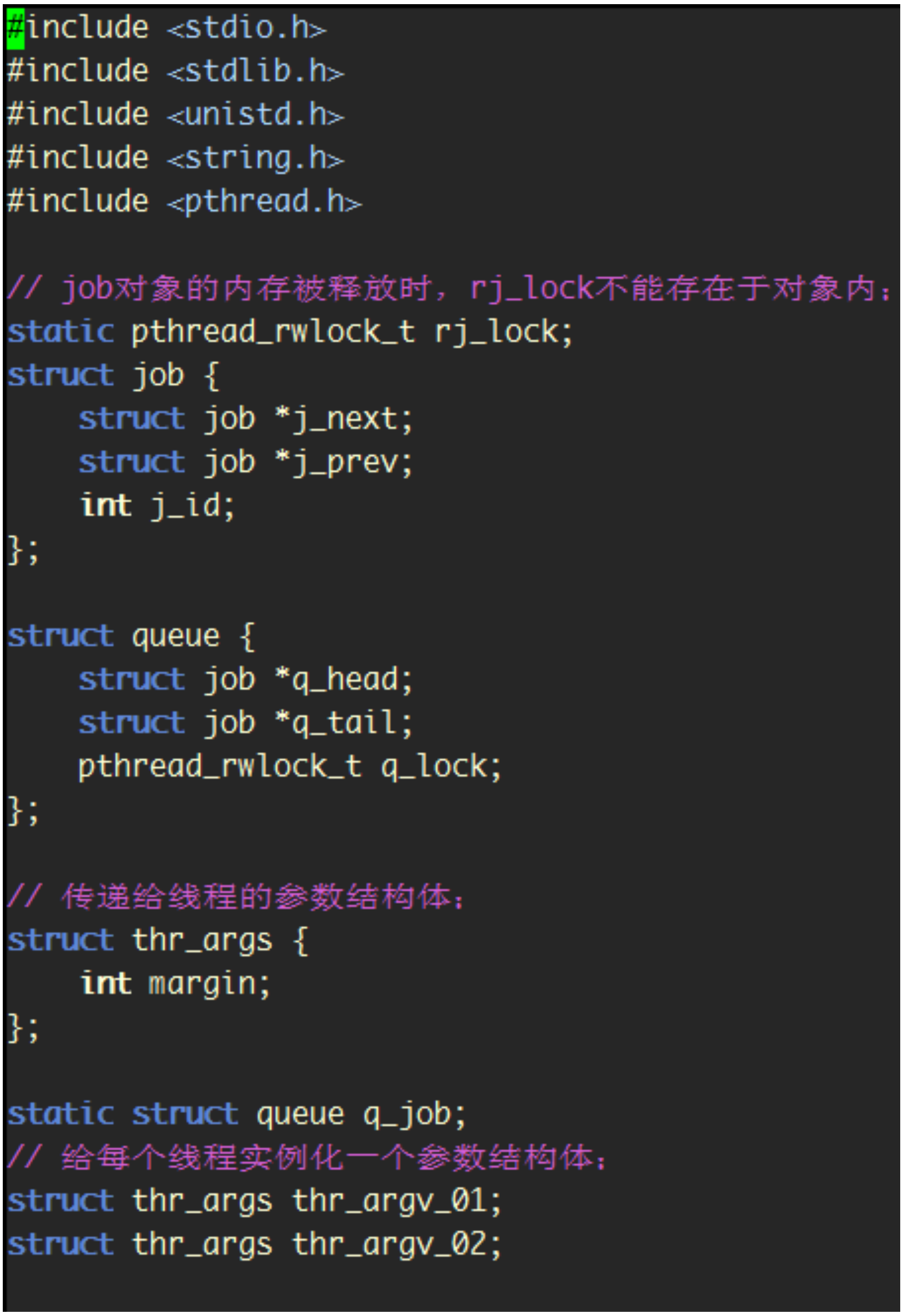
```c
int main(int argc, const char *argv[]) {
    int err;
    pthread_t tid1, tid2;
    struct foo *fp_receiver;
    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (0 != err) {
        printf("can't create thread 1: %s\n", strerror(err)); _exit(-1);
    }
    err = pthread_join(tid1, (void *) &fp_receiver);
    if (0 != err) {
        printf("can't join with thread 1: %s\n", strerror(err)); _exit(-1);
    }
    sleep(1);
    printf("parent starting second thread\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (0 != err) {
        printf("can't create thread 1: %s\n", strerror(err)); _exit(-1);
    }
    sleep(1);
    printfoo("parent:\n", fp_receiver);
    return 0;
}
/opt/drill_ground/aupe/11-1.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=0
```



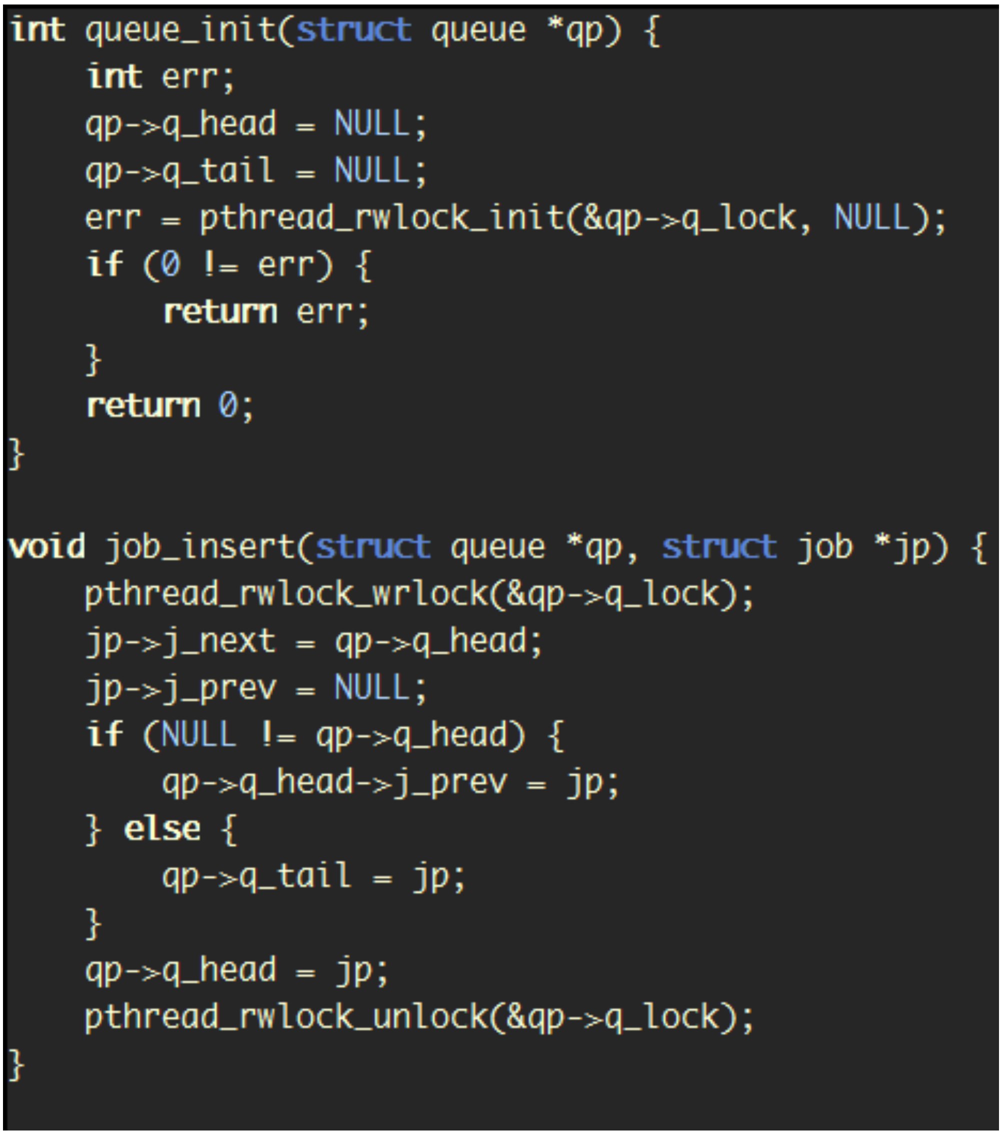
```
tongue aupe # gcc 11-1.c -o 11-1 -lpthread
tongue aupe # ./11-1
thread 1:
  structure at 0x601060
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 140615401776896
parent:
  structure at 0x601060
  foo.a = 5
  foo.b = 6
  foo.c = 7
  foo.d = 8
```

11-2、其实是增加一个方法，该方法可对未被执行的作业更改其运作线程；需要注意的

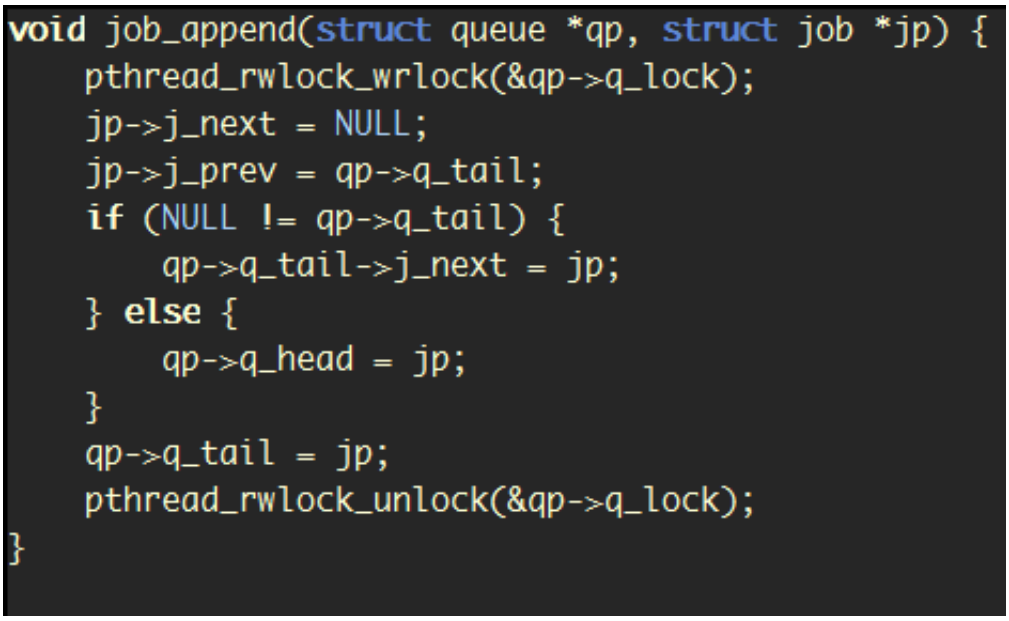
地方是，在A线程更新job_A的j_id时，B线程可能在这时会执行对该对象删除的操作；所以，要避免这种情况的发生，需要放置一把job对象占用锁，同一时刻只允许一种方法来对job执行更新或删除的操作；



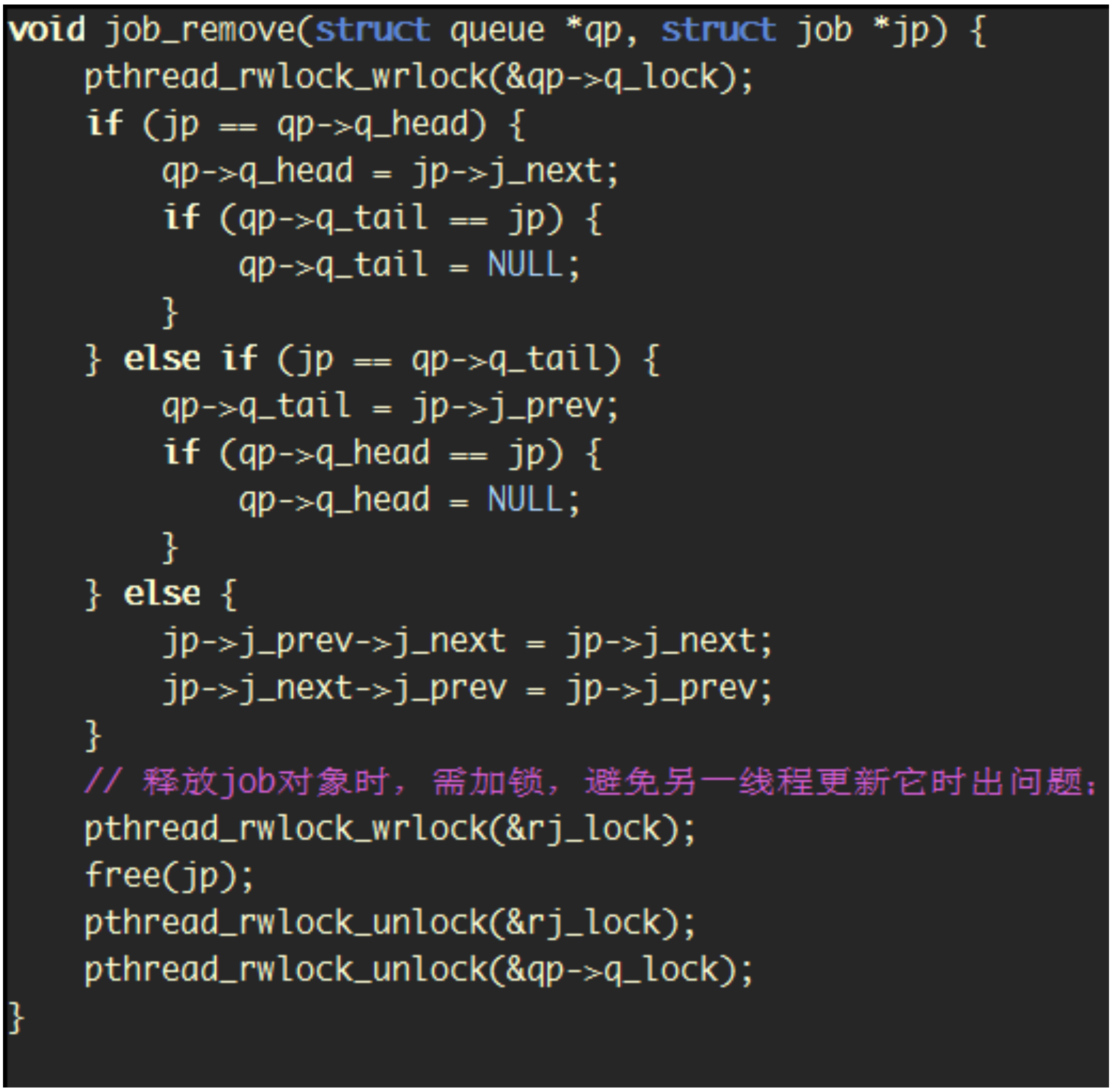
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

// job对象的内存被释放时，rj_lock不能存在于对象内；
static pthread_rwlock_t rj_lock;
struct job {
    struct job *j_next;
    struct job *j_prev;
    int j_id;
};

struct queue {
    struct job *q_head;
    struct job *q_tail;
    pthread_rwlock_t q_lock;
};

// 传递给线程的参数结构体；
struct thr_args {
    int margin;
};

static struct queue q_job;
// 给每个线程实例化一个参数结构体；
struct thr_args thr_argv_01;
struct thr_args thr_argv_02;
```



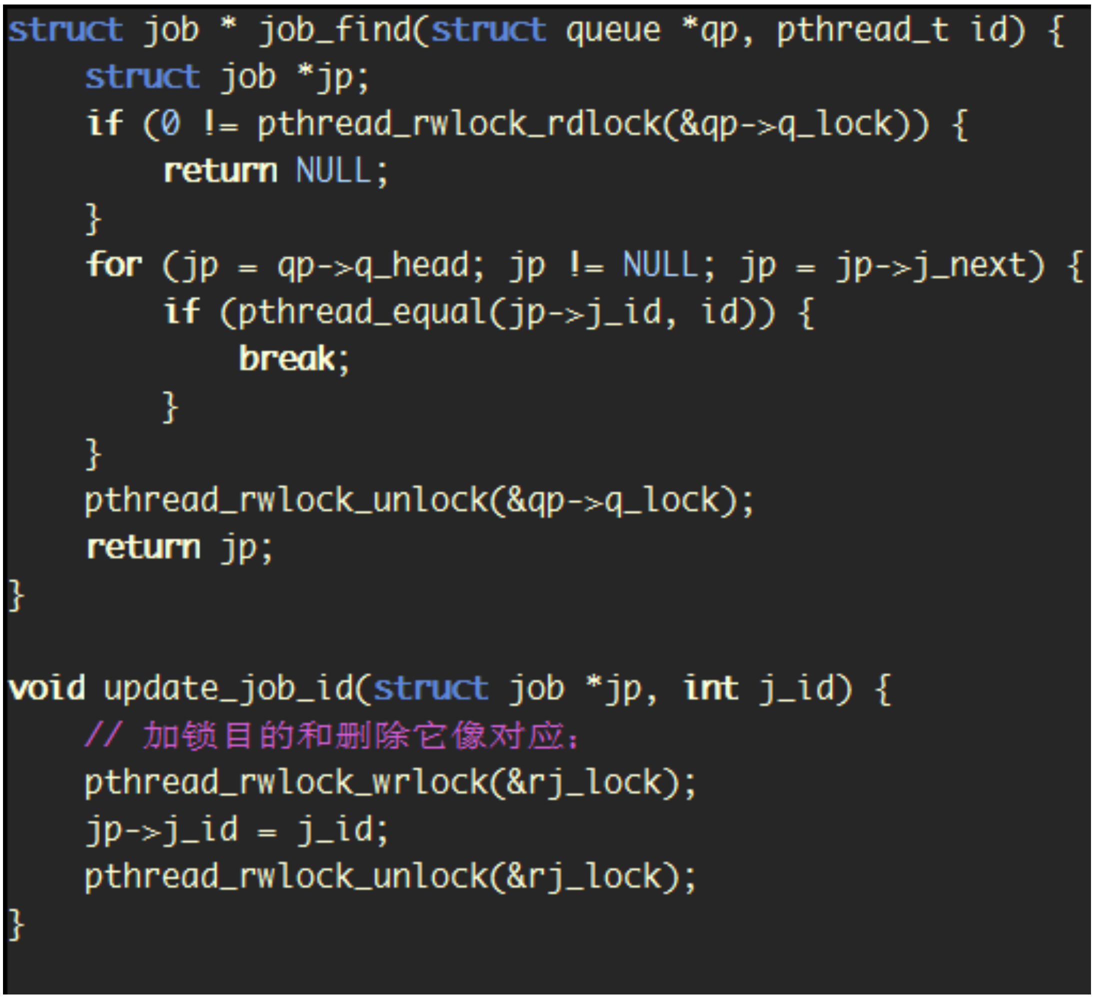
```c
int queue_init(struct queue *qp) {
    int err;
    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (0 != err) {
        return err;
    }
    return 0;
}

void job_insert(struct queue *qp, struct job *jp) {
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (NULL != qp->q_head) {
        qp->q_head->j_prev = jp;
    } else {
        qp->q_tail = jp;
    }
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}
```



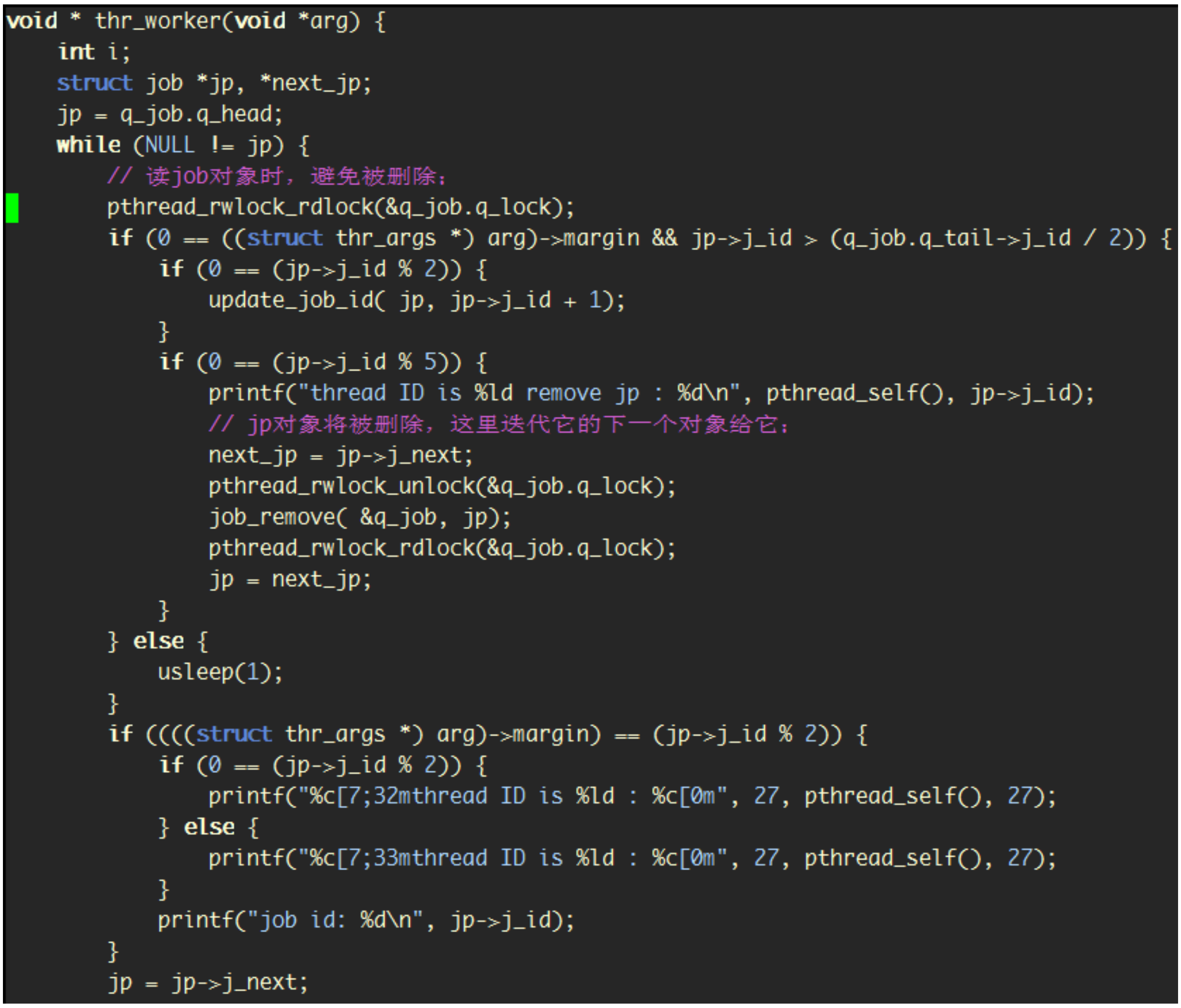
```c
void job_append(struct queue *qp, struct job *jp) {
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (NULL != qp->q_tail) {
        qp->q_tail->j_next = jp;
    } else {
        qp->q_head = jp;
    }
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}
```



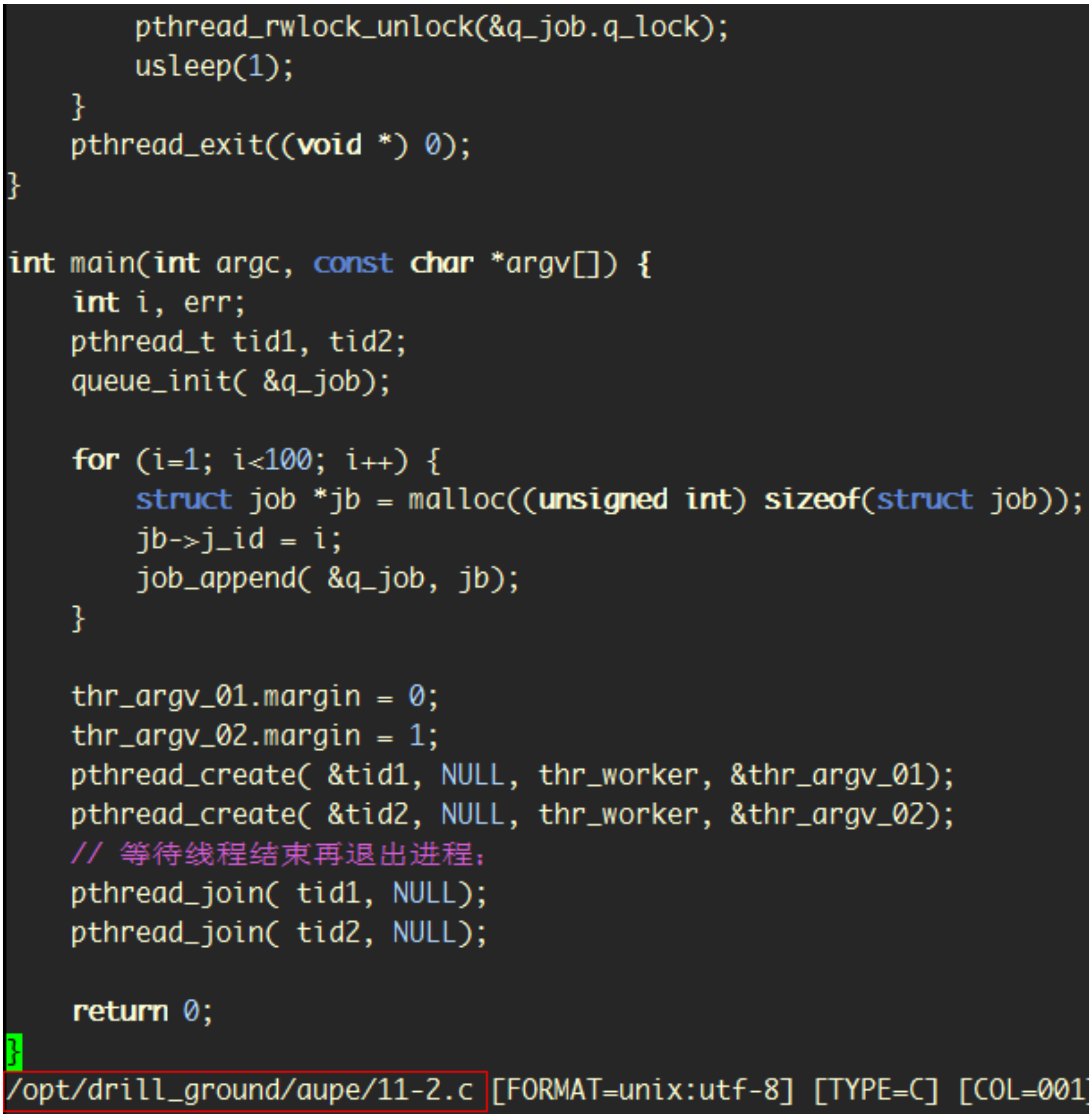
```c
void job_remove(struct queue *qp, struct job *jp) {
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp) {
            qp->q_tail = NULL;
        }
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        if (qp->q_head == jp) {
            qp->q_head = NULL;
        }
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    // 释放job对象时，需加锁，避免另一线程更新它时出问题；
    pthread_rwlock_wrlock(&rj_lock);
    free(jp);
    pthread_rwlock_unlock(&rj_lock);
    pthread_rwlock_unlock(&qp->q_lock);
}
```



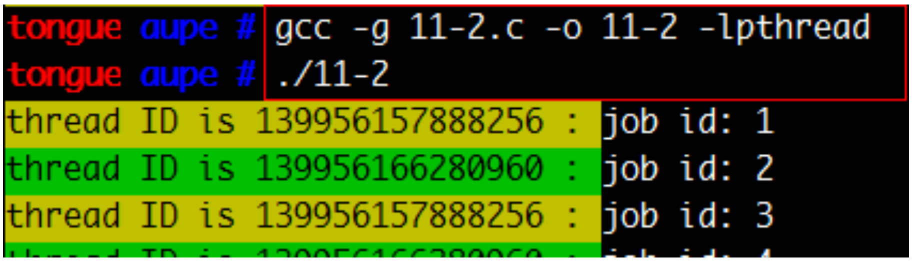
```c
struct job * job_find(struct queue *qp, pthread_t id) {
    struct job *jp;
    if (0 != pthread_rwlock_rdlock(&qp->q_lock)) {
        return NULL;
    }
    for (jp = qp->q_head; jp != NULL; jp = jp->j_next) {
        if (pthread_equal(jp->j_id, id)) {
            break;
        }
    }
    pthread_rwlock_unlock(&qp->q_lock);
    return jp;
}

void update_job_id(struct job *jp, int j_id) {
    // 加锁目的和删除它像对应;
    pthread_rwlock_wrlock(&rj_lock);
    jp->j_id = j_id;
    pthread_rwlock_unlock(&rj_lock);
}
```



```
void * thr_worker(void *arg) {
    int i;
    struct job *jp, *next_jp;
    jp = q_job.q_head;
    while (NULL != jp) {
        // 读job对象时，避免被删除；
        pthread_rwlock_rdlock(&q_job.q_lock);
        if (0 == ((struct thr_args *) arg)->margin && jp->j_id > (q_job.q_tail->j_id / 2)) {
            if (0 == (jp->j_id % 2)) {
                update_job_id( jp, jp->j_id + 1);
            }
            if (0 == (jp->j_id % 5)) {
                printf("thread ID is %ld remove jp : %d\n", pthread_self(), jp->j_id);
                // jp对象将被删除，这里迭代它的下一个对象给它；
                next_jp = jp->j_next;
                pthread_rwlock_unlock(&q_job.q_lock);
                job_remove( &q_job, jp);
                pthread_rwlock_rdlock(&q_job.q_lock);
                jp = next_jp;
            }
        } else {
            usleep(1);
        }
        if ((((struct thr_args *) arg)->margin) == (jp->j_id % 2)) {
            if (0 == (jp->j_id % 2)) {
                printf("%c[7;32mthread ID is %ld : %c[0m", 27, pthread_self(), 27);
            } else {
                printf("%c[7;33mthread ID is %ld : %c[0m", 27, pthread_self(), 27);
            }
            printf("job id: %d\n", jp->j_id);
        }
        jp = jp->j_next;
```

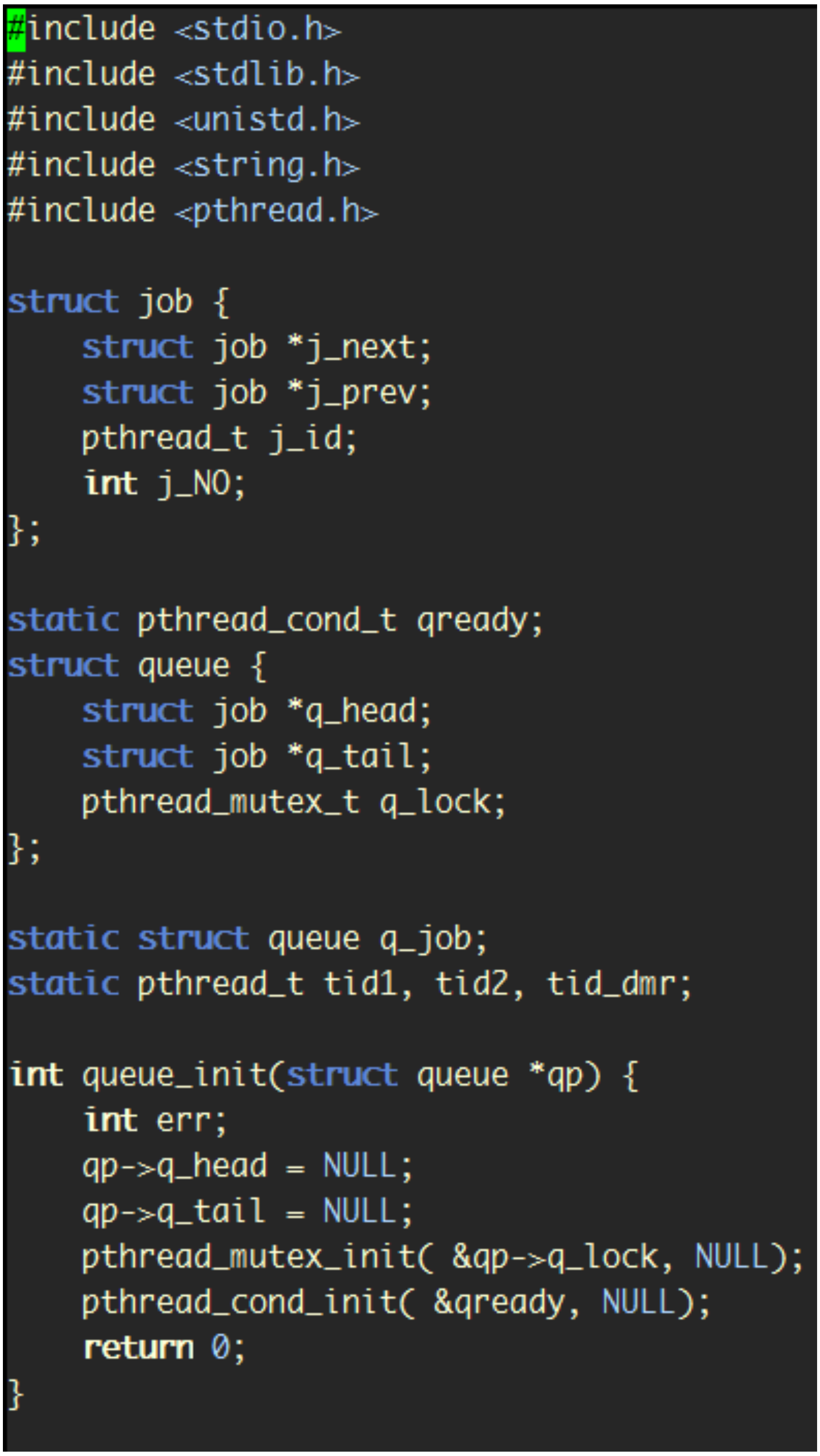


```
        pthread_rwlock_unlock(&q_job.q_lock);
        usleep(1);
    }
    pthread_exit((void *) 0);
}

int main(int argc, const char *argv[]) {
    int i, err;
    pthread_t tid1, tid2;
    queue_init( &q_job);

    for (i=1; i<100; i++) {
        struct job *jb = malloc((unsigned int) sizeof(struct job));
        jb->j_id = i;
        job_append( &q_job, jb);
    }

    thr_argv_01.margin = 0;
    thr_argv_02.margin = 1;
    pthread_create( &tid1, NULL, thr_worker, &thr_argv_01);
    pthread_create( &tid2, NULL, thr_worker, &thr_argv_02);
    // 等待线程结束再退出进程;
    pthread_join( tid1, NULL);
    pthread_join( tid2, NULL);

    return 0;
}
/opt/drill_ground/aupe/11-2.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001]
```

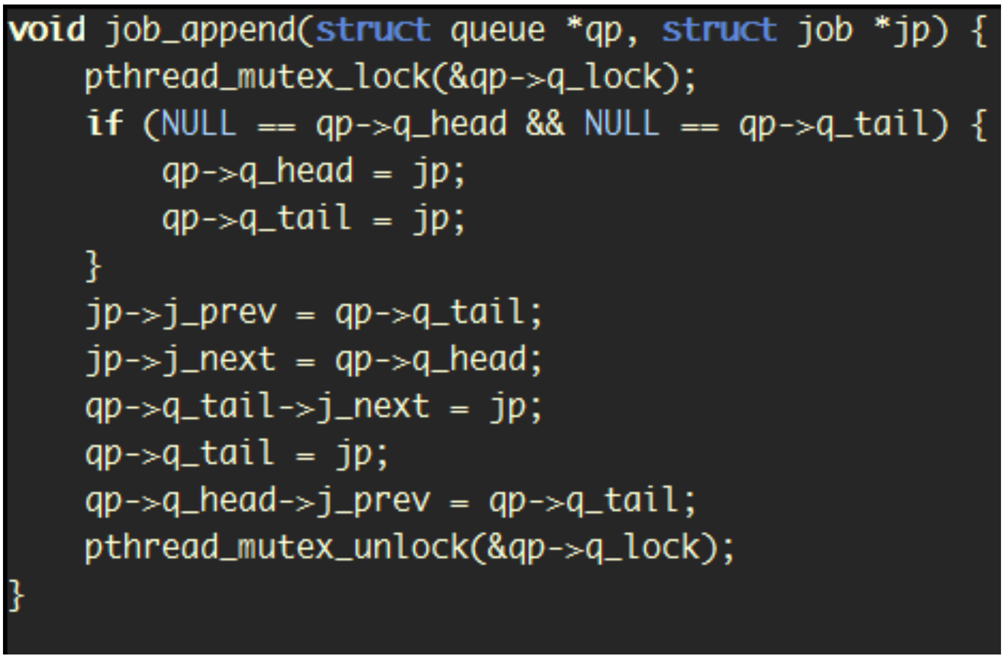


```
tongue aupe # gcc -g 11-2.c -o 11-2 -lpthread
tongue aupe # ./11-2
thread ID is 139956157888256 : job id: 1
thread ID is 139956166280960 : job id: 2
thread ID is 139956157888256 : job id: 3
thread ID is 139956166280960 : job id: 4
```

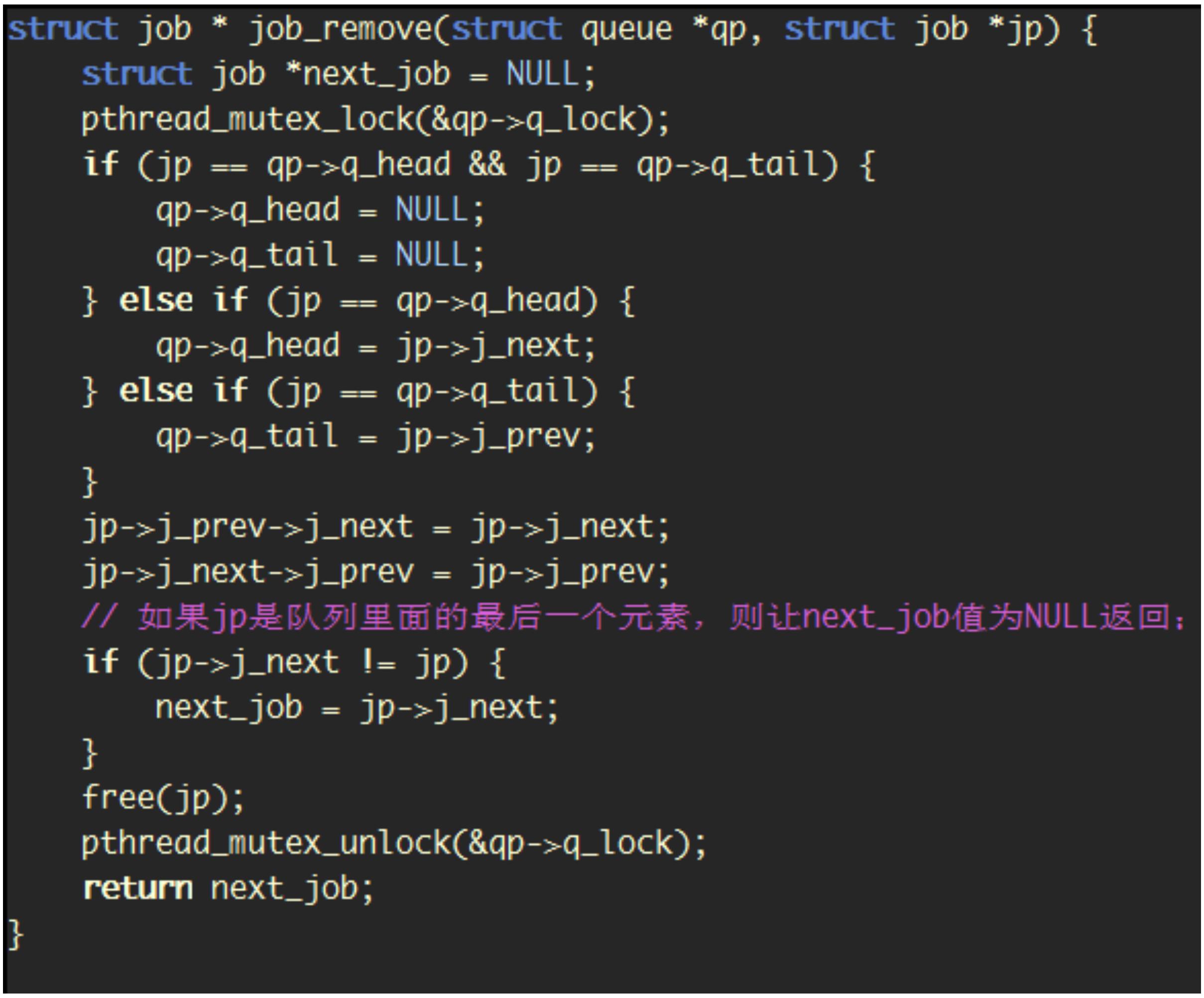
11-3、相比于上题，做了些许调整；



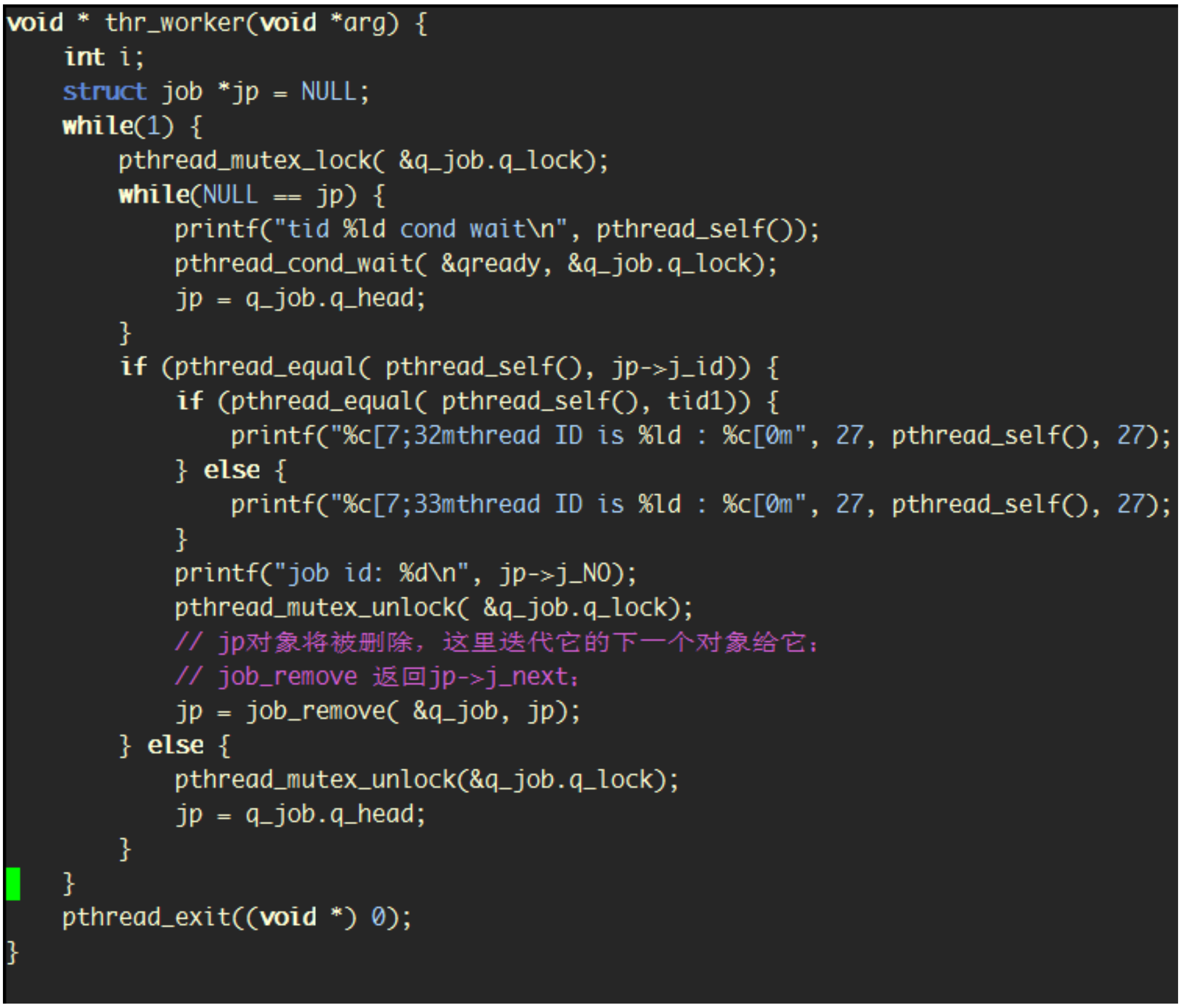
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t j_id;
    int j_NO;
};

static pthread_cond_t qready;
struct queue {
    struct job *q_head;
    struct job *q_tail;
    pthread_mutex_t q_lock;
};

static struct queue q_job;
static pthread_t tid1, tid2, tid_dmr;

int queue_init(struct queue *qp) {
    int err;
    qp->q_head = NULL;
    qp->q_tail = NULL;
    pthread_mutex_init( &qp->q_lock, NULL);
    pthread_cond_init( &qready, NULL);
    return 0;
}
```



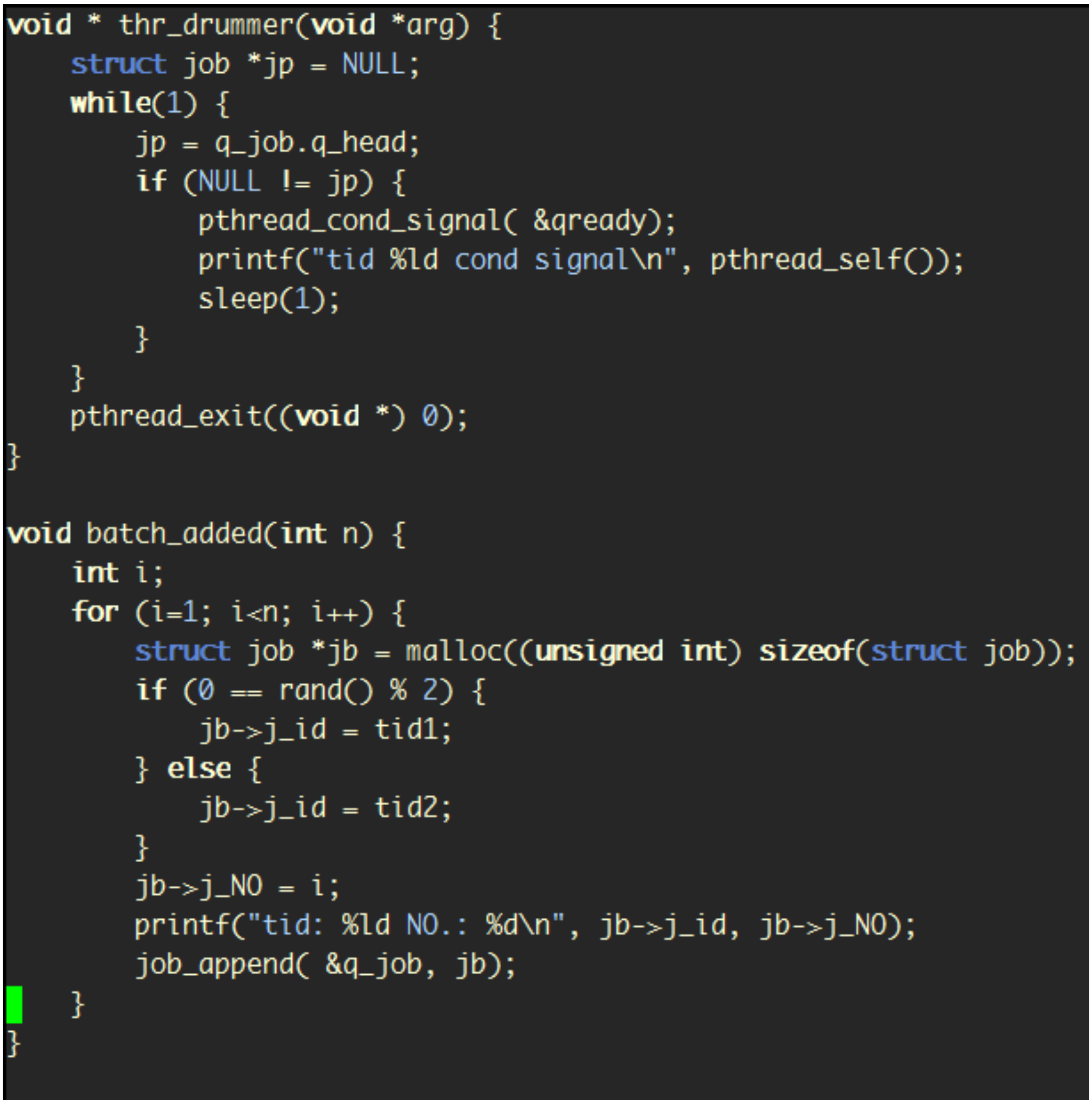
```c
void job_append(struct queue *qp, struct job *jp) {
    pthread_mutex_lock(&qp->q_lock);
    if (NULL == qp->q_head && NULL == qp->q_tail) {
        qp->q_head = jp;
        qp->q_tail = jp;
    }
    jp->j_prev = qp->q_tail;
    jp->j_next = qp->q_head;
    qp->q_tail->j_next = jp;
    qp->q_tail = jp;
    qp->q_head->j_prev = qp->q_tail;
    pthread_mutex_unlock(&qp->q_lock);
}
```



```c
struct job * job_remove(struct queue *qp, struct job *jp) {
    struct job *next_job = NULL;
    pthread_mutex_lock(&qp->q_lock);
    if (jp == qp->q_head && jp == qp->q_tail) {
        qp->q_head = NULL;
        qp->q_tail = NULL;
    } else if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
    }
    jp->j_prev->j_next = jp->j_next;
    jp->j_next->j_prev = jp->j_prev;
    // 如果jp是队列里面的最后一个元素，则让next_job值为NULL返回；
    if (jp->j_next != jp) {
        next_job = jp->j_next;
    }
    free(jp);
    pthread_mutex_unlock(&qp->q_lock);
    return next_job;
}
```

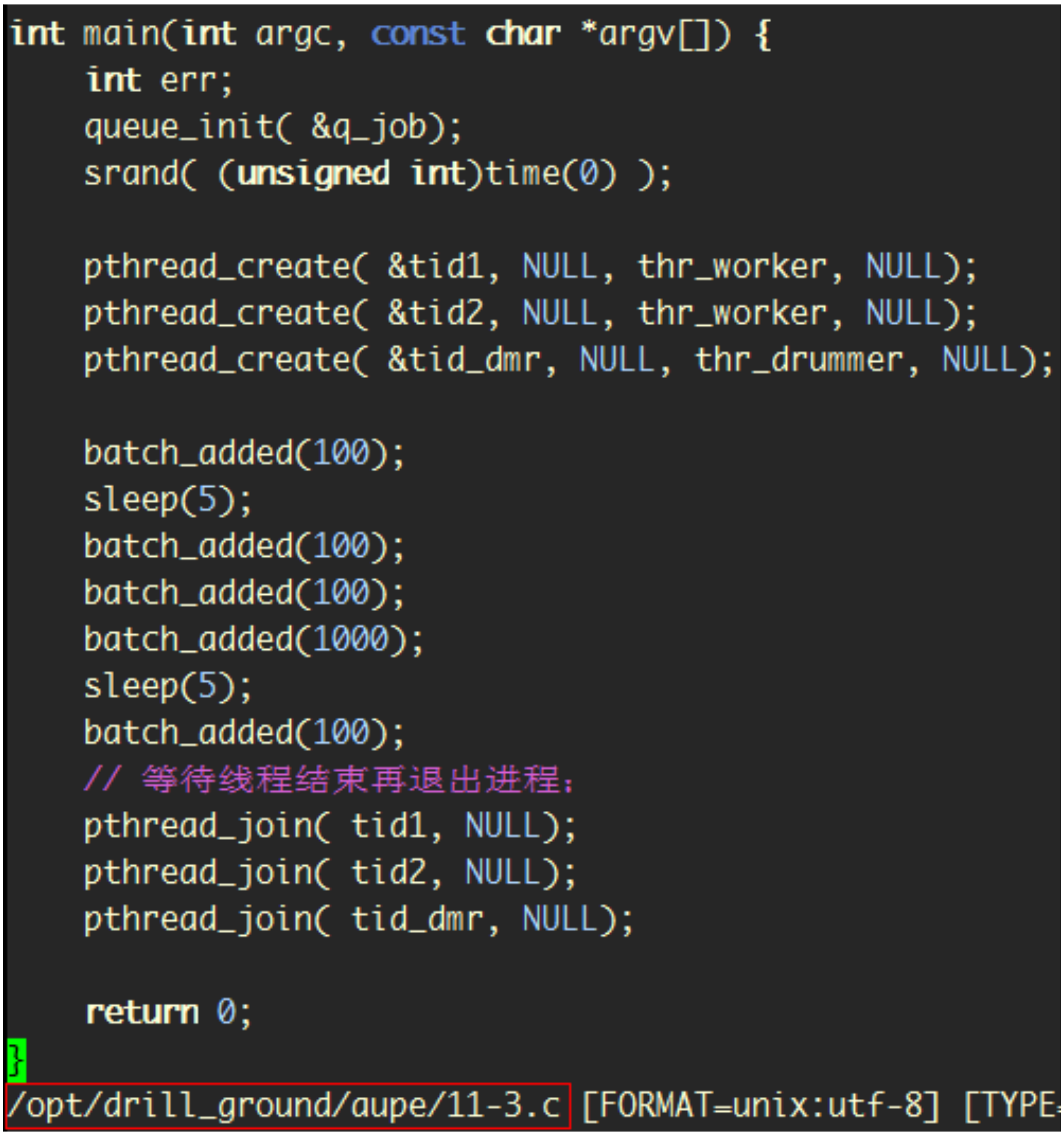


```
void * thr_worker(void *arg) {
    int i;
    struct job *jp = NULL;
    while(1) {
        pthread_mutex_lock( &q_job.q_lock);
        while(NULL == jp) {
            printf("tid %ld cond wait\n", pthread_self());
            pthread_cond_wait( &qready, &q_job.q_lock);
            jp = q_job.q_head;
        }
        if (pthread_equal( pthread_self(), jp->j_id)) {
            if (pthread_equal( pthread_self(), tid1)) {
                printf("%c[7;32mthread ID is %ld : %c[0m", 27, pthread_self(), 27);
            } else {
                printf("%c[7;33mthread ID is %ld : %c[0m", 27, pthread_self(), 27);
            }
            printf("job id: %d\n", jp->j_NO);
            pthread_mutex_unlock( &q_job.q_lock);
            // jp对象将被删除，这里迭代它的下一个对象给它；
            // job_remove 返回jp->j_next；
            jp = job_remove( &q_job, jp);
        } else {
            pthread_mutex_unlock(&q_job.q_lock);
            jp = q_job.q_head;
        }
    }
    pthread_exit((void *) 0);
}
```

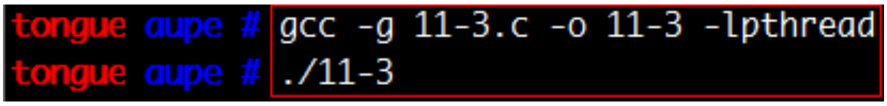


```
void * thr_drummer(void *arg) {
    struct job *jp = NULL;
    while(1) {
        jp = q_job.q_head;
        if (NULL != jp) {
            pthread_cond_signal( &qready);
            printf("tid %ld cond signal\n", pthread_self());
            sleep(1);
        }
    }
    pthread_exit((void *) 0);
}

void batch_added(int n) {
    int i;
    for (i=1; i<n; i++) {
        struct job *jb = malloc((unsigned int) sizeof(struct job));
        if (0 == rand() % 2) {
            jb->j_id = tid1;
        } else {
            jb->j_id = tid2;
        }
        jb->j_NO = i;
        printf("tid: %ld NO.: %d\n", jb->j_id, jb->j_NO);
        job_append( &q_job, jb);
    }
}
```

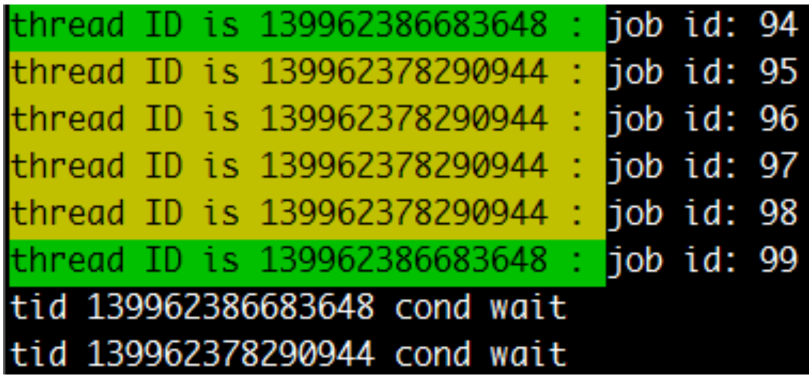
11-4、该题应该指的是程序清单11-9中的enqueue_msg方法，该方法和题中第二种序列相像。



```
int main(int argc, const char *argv[]) {
    int err;
    queue_init( &q_job);
    srand( (unsigned int)time(0) );

    pthread_create( &tid1, NULL, thr_worker, NULL);
    pthread_create( &tid2, NULL, thr_worker, NULL);
    pthread_create( &tid_dmr, NULL, thr_drummer, NULL);

    batch_added(100);
    sleep(5);
    batch_added(100);
    batch_added(100);
    batch_added(1000);
    sleep(5);
    batch_added(100);
    // 等待线程结束再退出进程;
    pthread_join( tid1, NULL);
    pthread_join( tid2, NULL);
    pthread_join( tid_dmr, NULL);

    return 0;
}
/opt/drill_ground/aupe/11-3.c [FORMAT=unix:utf-8] [TYPE
```



```
tongue aupe # gcc -g 11-3.c -o 11-3 -lpthread
tongue aupe # ./11-3
```



```
thread ID is 139962386683648 : job id: 94
thread ID is 139962378290944 : job id: 95
thread ID is 139962378290944 : job id: 96
thread ID is 139962378290944 : job id: 97
thread ID is 139962378290944 : job id: 98
thread ID is 139962386683648 : job id: 99
tid 139962386683648 cond wait
tid 139962378290944 cond wait
```

11-4、该题应该指的是程序清单11-9中的enqueue_msg方法，该方法和题中第二种序列相像。