****

《操作系统程序设计大作业》

设计报告书

**题目：一个用户级线程的实现**

**学 院 计算机科学与工程学院**

**专 业 计算机科学与技术**

**学生姓名 卢科雄**

**学生学号 201930342424**

**指导教师 吴一民**

**课程编号**

**课程学分**

**起始日期**

|  |  |
| --- | --- |
| 教  师  评  语 | 教师签名：  日期： |
| 成  绩  评  定 |  |
| 备  注 |  |

**一个用户级线程的实现**

一**、**选题背景

本次课程设计需要开发一个用户级线程库，功能包括完整的线程创建/删除/joining，互斥锁，条件变量，以及一个基于优先级的调度器。完成此用户级线程库，你可以使用你实现的线程库写多线程的应用，替代linux中的 pthreads.

二**、**方案论证(设计理念)

使用线程库前首先要调用 uthread\_init()。这个函数只执行一次，负责初始化数据结构，比如全局 uthreads 矩阵和 ut\_curthr (这是当前正在执行的线程)。 uthread\_init() 中有一些特殊的代码，处理当前正在执行的进程的上下文 (即调用 uthread\_init()的进程)使其成为一个有效的 uthread 并设置 ut\_curthr为这个uthread，这个uthread称之为主线程，0号线程。详见 create\_first\_thr() 附近的注释信息。所有事情初始化后，即uthread\_init完成后，你可以使用线程 uthread\_create()创建线程。

一旦创建好线程，线程库需要能够调度这些线程。线程需要暂时 yield 处理器给另一个线程 (仍然是可执行的，即处于就绪态)，则需要调用 uthread\_yield()。 线程可以通过 uthread\_wake() 被唤醒。选择另一个线程运行发生在 uthread\_switch() 函数。调度器调度时，采用优先级调度算法，线程优先级可以由 uthread\_setprio()设置，数字越大优先级越高，相同优先级的线程则轮流使用CPU。每个优先级一个队列，相关的数据结构见 uthread\_sched.c 中的runq\_table .

本课程设计所设计实现的 uthreads 行为上与linux pthread 类似。因此你要实现的函数或api 与linux pthread 基本相同。实现时请参见pthread的说明。

Uthreads中的线程有6种状态，定义如下：

*typedef enum {*

*UT\_NO\_STATE, /\* 无效的线程状态 \*/*

*UT\_ON\_CPU, /\* 线程正在执行 \*/*

*UT\_RUNNABLE, /\* 线程可运行，就绪 \*/*

*UT\_WAIT, /\* 线程被阻塞 \*/*

*UT\_ZOMBIE, /\* 线程处于 zombie 状态，即已结束，但需要回收资源 \*/*

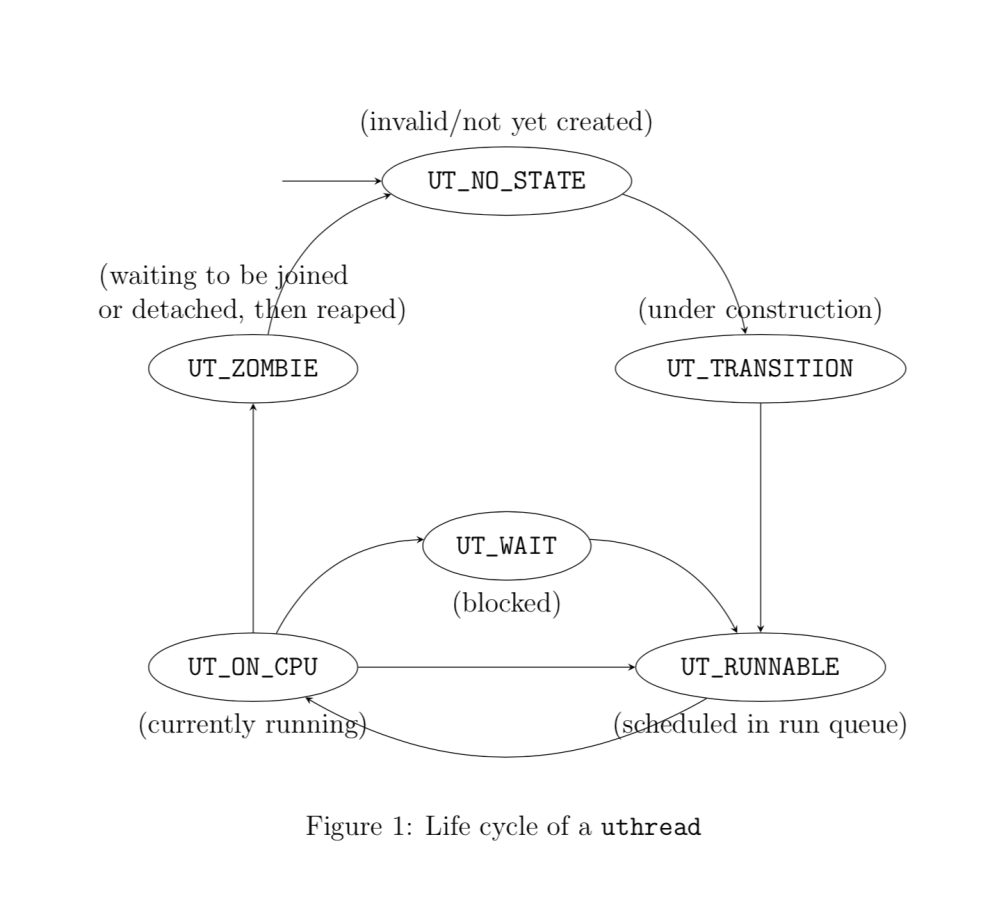
*UT\_TRANSITION, /\* 线程处于创建状态 \*/*

*UT\_JOINABLE, //线程结束时需要一个线程回收其资源*

*UT\_DETACHABLE, //线程结束时，不需要一个线程回收其资源*

*UT\_NUM\_THREAD\_STATES //线程状态数目*

*} uthread\_state\_t;*



Uthread 线程转换图

在uthreads 中永远不会有多于一个线程同时执行。在 uthreads中，使用 uthread\_makecontext() 创建线程的机器上下文。如果需要改变当前正在执行线程 (在调度器)，需要调用 uthread\_swapcontext()。这使得当前 CPU 上下文被保存，新的上下文被执行。保存的上下文作为newctx 参数在之后调用 uthread\_swapcontext() 时被重新执行。

Uthread中有一个线程称之为Reaper 线程，也就是1号线程，负责清理已结束线程占用的资源。值得注意的是 reaper 并不清理已经结束但还没有joined的non-detached 线程，而是让 uthread\_join()去完成。我们在 uthread.c中给出了一个完整的 reaper 作为 reaper，仔细查看其实现，以理解其意义.

三**、**过程论述

下面分别从需要修改的四个文件中介绍我对程序的修改：

**uthread.c**

**uthread\_init**

void uthread\_init(void) {

    for(uthread\_id\_t i = 0;i < UTH\_MAX\_UTHREADS;i++){

        uthreads[i].ut\_state = UT\_NO\_STATE;

        uthreads[i].ut\_id = i;

    }

    /\* 以下函数代码不需要修改 \*/

    uthread\_sched\_init();  //不需要修改，初始化优先级的队列

    reaper\_init1();        //创建互斥锁和条件变量

    create\_first\_thr();    //不需要修改，创建第一个线程，即主线程

    reaper\_init2();        //不需要修改，创建reap线程

}

首先是uthread\_init函数，这个函数只在用户进程启动时调用一次，需要实现的功能是初始化全局的数据结构和变量。

在这个线程库中，uthreads数组用于存储线程库中的所有线程，由于在刚初始化时，所有线程还未被创建，因此在初始化时只需设置线程的状态和id，简单起见将ut\_id设置为线程的数组下标，规定将未创建的线程设置为UT\_NO\_STATE状态。

uthread\_create

int uthread\_create(uthread\_id\_t \*uidp, uthread\_func\_t func,

               long arg1, char \*arg2[], int prio) {

    uthread\_id\_t id= uthread\_alloc();

    if ( id == -1 )return -1;

    else{

        \*uidp = id;

        uthread\_t \*thr = &uthreads[id];

        thr->ut\_stack = alloc\_stack();

        if(thr->ut\_stack == NULL)return -1;

        if ( thr->ut\_stack ){

            uthread\_makecontext(&thr->ut\_ctx,thr->ut\_stack, UTH\_STACK\_SIZE, func, arg1, arg2);

            memset(&thr->ut\_link, 0, sizeof(list\_link\_t));

            thr->ut\_id = id;

            thr->ut\_state = UT\_RUNNABLE;

            thr->ut\_errno = 0;

            thr->ut\_has\_exited = 0;

            thr->ut\_exit = NULL;

            thr->ut\_detach\_state = UT\_DETACHABLE;

            thr->ut\_waiter = NULL;

            thr->ut\_prio = -1;

            uthread\_setprio(thr->ut\_id, prio);

            return 0;

        }

    }

    return -1;

}

uthread\_create函数用于创建线程，输入的参数有线程的id，创建线程执行指定的函数 <func>，函数的参数为 <arg1> 和 <arg2>，优先级为 <prio>。

首先，使用 uthread\_alloc 找到一个有效的 id，如果找不到，即函数返回值为-1，该函数返回-1表示创建失败。

接着为线程分配栈，若分配失败，内存为null，返回错误。分配成功时，使用uthread\_makecontext() 创建线程的上下文，将运行的函数和参数输入至线程中，再根据uthread的数据结构将该线程的所有变量初始化，将线程的状态设置为UT\_RUNNABLE就绪状态，将线程设置为不可抢占式，最后将优先级设置为-1，方便后续与已经创建的线程进行区分，并调用uthread\_setprio函数设置优先级，返回成功。

uthread\_exit

void uthread\_exit(void \*status) {

    ut\_curthr->ut\_exit = status;

    ut\_curthr->ut\_has\_exited = 1;

    if(ut\_curthr->ut\_detach\_state == UT\_DETACHABLE){

        make\_reapable(ut\_curthr);

        ut\_curthr->ut\_state = UT\_ZOMBIE;

    }else if (ut\_curthr->ut\_detach\_state == UT\_JOINABLE){

        if (ut\_curthr->ut\_waiter)

        {

            uthread\_wake(ut\_curthr->ut\_waiter);

        }

    }

    uthread\_switch();

    PANIC("returned to a dead thread");

}

用于结束当前线程，首先将判断线程是否已经结束，如果已经结束，则抛出错误；若没有结束，将线程的状态设置为退出，并将输入的退出码赋值给当前的线程。接着判断现在是否有线程等待着该线程结束，若没有就修改线程状态并切换线程；若有，先判断当前线程是否可抢占，若不可抢占，则调用make\_reapable()将其放入清理线程，清理队列并唤醒清理清理线程；若可抢占，唤醒当期线程的等待线程。最后切换线程。

uthread\_join

int uthread\_join(uthread\_id\_t uid, void \*\*return\_value) {

    if(uid >= UTH\_MAX\_UTHREADS){

        return -1;

    }

    uthread\_t \*thr = &uthreads[uid];

    if(thr->ut\_detach\_state==UT\_DETACHABLE)

    {

        if(return\_value!=NULL)\*return\_value=NULL;

        return -1;

    }

    if(thr->ut\_waiter != NULL || thr->ut\_state == UT\_NO\_STATE){

        return -1;

    }

    if(thr->ut\_state != UT\_ZOMBIE){

        thr->ut\_waiter = ut\_curthr;

        ut\_curthr->ut\_state = UT\_WAIT;

        uthread\_switch();

    }

    \*return\_value = thr->ut\_exit;

    make\_reapable(thr);

    return 0;

}

该函数用于等待指定的线程结束。首先获取指定的线程信息，若线程为不可抢占线程，则无法join，返回错误。接着判断是否存在多个线程试图join同一个线程或指定线程不存在，若出现这种情况则返回错误。然后判断指定线程是否已经结束，若仍未结束，将当前线程状态改为UT\_WAIT,将当前线程设置为指定线程的等待线程，调用uthread\_switch等待线程结束；若线程已结束或等待到线程结束，设置线程的返回值并彻底清理指定的线程。

uthread\_self

uthread\_id\_t uthread\_self(void) {

    assert(ut\_curthr != NULL);

    return ut\_curthr->ut\_id;

}

用于返回当前线程的id

uthread\_alloc

static uthread\_id\_t uthread\_alloc(void) {

    for(uthread\_id\_t i = 0;i < UTH\_MAX\_UTHREADS;i++){

        if(uthreads[i].ut\_state == UT\_NO\_STATE)return i;

    }

    return -1;

}

遍历uthreads数组中的所有线程，找到第一个状态为UT\_NO\_STATE的线程，并返回线程的下标。

uthread\_destroy

static void uthread\_destroy(uthread\_t \*uth) {

    assert(uth->ut\_state == UT\_ZOMBIE);

    free\_stack(uth->ut\_stack);

    uth->ut\_state = UT\_NO\_STATE;

    uth->ut\_has\_exited = 1;

    uth->ut\_waiter = NULL;

}

释放线程的内存，将线程的状态设置为NO\_STATE，退出状态设置为1，将线程的等待线程设置为空。

**uthread\_sched.c**

uthread\_yield

void uthread\_yield(void) {

    ut\_curthr->ut\_state = UT\_RUNNABLE;

    utqueue\_enqueue(&runq\_table[ut\_curthr->ut\_prio],ut\_curthr);

    uthread\_switch();

}

该函数用于使正在运行的线程让出CPU，由于此时线程仍然处于可被调度的UT\_RUNNABLE状态，修改线程的状态，并将线程重新加入到就绪队列中，调用uthread\_switch使更高优先级的线程得以执行。

uthread\_wake

void uthread\_wake(uthread\_t \*uthr) {

    if(uthr->ut\_state != UT\_RUNNABLE){

        uthr->ut\_state = UT\_RUNNABLE;

        utqueue\_enqueue(&runq\_table[uthr->ut\_prio],uthr);

    }

}

该函数用于唤醒指定的线程，如果线程已经处于就绪态，则不作处理，否则将线程设置为就绪态并加入就绪队列。

uthread\_setprio

int uthread\_setprio(uthread\_id\_t id, int prio) {

    if(id >= UTH\_MAX\_UTHREADS || prio > UTH\_MAXPRIO || prio < 0)return 0;

    if(uthreads[id].ut\_state == UT\_ON\_CPU || uthreads[id].ut\_state == UT\_WAIT){

        uthreads[id].ut\_prio = prio;

        return 1;

    }else if(uthreads[id].ut\_state == UT\_TRANSITION){

        uthreads[id].ut\_state = UT\_RUNNABLE;

    }else if(uthreads[id].ut\_state == UT\_RUNNABLE){

        if(uthreads[id].ut\_prio != prio){

            if(uthreads[id].ut\_prio != -1)utqueue\_remove(&runq\_table[uthreads[id].ut\_prio],&uthreads[id]);

            utqueue\_enqueue(&runq\_table[prio],&uthreads[id]);

            uthreads[id].ut\_prio = prio;

        }

        return 1;

    }

    return 0;

}

该函数用于改变指定线程的优先级，首先判断输入线程的id是否合规，判断线程是否正在占用CPU或被阻塞，如果是则可直接改变其优先级；若线程处于UT\_TRANSITION状态，则要改变其状态为UT\_RUNNABLE。

若线程状态为UT\_RUNNABLE，且其优先级与改变的优先级不一样，便更改其优先级并调整其处于优先级队列中的位置。注意在uthread\_create中创建的线程还未进入优先级队列，因此可以直接放入其所对应的优先级队列中。

uthread\_switch

void uthread\_switch() {

    int i;

    for(i = UTH\_MAXPRIO;i >= 0;i--){

        // return;

        if(!utqueue\_empty(&runq\_table[i])){

            uthread\_t\* old = ut\_curthr;

            uthread\_t\* new = utqueue\_dequeue(&runq\_table[i]);

            if(old != new){

                ut\_curthr = new;

                ut\_curthr->ut\_state = UT\_ON\_CPU;

                uthread\_swapcontext(&old->ut\_ctx,&ut\_curthr->ut\_ctx);

            }

            return;

        }

    }

}

该函数用于找到优先级队列中优先级最高的线程，并切换到该线程。首先按照高到低的顺序遍历优先级队列，若当前的队列非空，说明有可调度的线程，将其出队，并判断该线程是否就是当前线程，若是则直接返回，否则设置线程为当前运行的线程，并使用uthread\_swapcontext() 切换到它。

uthread\_cond.c

uthread\_cond\_init

void uthread\_cond\_init(uthread\_cond\_t \*cond) {

    utqueue\_init(&cond->uc\_waiters);

}

该函数用于初始化指定的条件变量，条件变量中有一个等待者队列对象，该函数通过初始化waiter队列的方式对条件变量进行初始化

uthread\_cond\_wait

void uthread\_cond\_wait(uthread\_cond\_t \*cond, uthread\_mtx\_t \*mtx) {

    uthread\_mtx\_unlock(mtx);

    ut\_curthr->ut\_state = UT\_WAIT;

    utqueue\_enqueue(&cond->uc\_waiters,ut\_curthr);

    uthread\_switch();

    uthread\_mtx\_lock(mtx);

}

这个函数用于等待指定的条件变量。首先将条件变量解锁，再修改当前线程的状态，将其加入条件变量的等待队列中，切换线程，将等待变量上锁。

uthread\_cond\_broadcast

void uthread\_cond\_broadcast(uthread\_cond\_t \*cond) {

    while(!utqueue\_empty(&cond->uc\_waiters)){

        uthread\_wake(utqueue\_dequeue(&cond->uc\_waiters));

    }

}

该函数用于唤醒等待于该条件变量的所有线程。根据条件变量的等待队列，一一遍历所有的线程，将其从等待队列中出队，并唤醒线程。

uthread\_cond\_signal

void uthread\_cond\_signal(uthread\_cond\_t \*cond) {

    if(!utqueue\_empty(&cond->uc\_waiters)){

        uthread\_wake(utqueue\_dequeue(&cond->uc\_waiters));

    }

}

该函数用于唤醒等待此条件变量的一个线程，如果等待队列非空，从等待队列拿出一个线程并唤醒。

**uthread\_mtx.c**

uthread\_mtx\_init

void uthread\_mtx\_init(uthread\_mtx\_t \*mtx) {

    mtx->m\_owner = NULL;

    utqueue\_init(&mtx->m\_waiters);

}

该函数用于初始化指定的mutex，首先将mtx的拥有者置为空，并调用utqueue\_init函数初始化mutex的等待队列

uthread\_mtx\_lock

void uthread\_mtx\_lock(uthread\_mtx\_t \*mtx) {

    if(mtx->m\_owner == NULL){

        mtx->m\_owner = ut\_curthr;

    }else{

        if(mtx->m\_owner == ut\_curthr)return;

        ut\_curthr->ut\_state = UT\_WAIT;

        utqueue\_enqueue(&mtx->m\_waiters,ut\_curthr);

        uthread\_switch();

    }

}

该函数用于将指定的锁上锁，首先判断是否有线程占有该锁，如果没有，将锁的拥有者改为当前线程。

若没有，将当前线程的状态改为阻塞态，将线程放入锁的等待队列，切换线程。

uthread\_mtx\_trylock

int uthread\_mtx\_trylock(uthread\_mtx\_t \*mtx) {

    if(mtx->m\_owner == NULL){

        mtx->m\_owner = ut\_curthr;

        return 1;

    }

    if(mtx->m\_owner == ut\_curthr){

        return 1;

    }

    return 0;

}

该函数用于线程试图对指定的锁上锁，若锁没有被线程占有，则当前线程占有该锁，并返回成功；若锁已被当前线程占有，直接返回成功；否则返回上锁失败。

uthread\_mtx\_unlock

void uthread\_mtx\_unlock(uthread\_mtx\_t \*mtx) {

    if(mtx->m\_owner == NULL){

        return;

    }

    if(mtx->m\_owner == ut\_curthr){

        if(utqueue\_empty(&mtx->m\_waiters)){

            mtx->m\_owner = NULL;

            return;

        }else{

            uthread\_t\* uthr = utqueue\_dequeue(&mtx->m\_waiters);

            if(uthr){

                mtx->m\_owner = uthr;

                uthread\_wake(uthr);

            }

        }

    }else{

        return;

    }

}

该函数用于释放锁，首先判断是否有线程正在占有锁，若没有，表示不用释放，直接返回，否则判断锁的拥有者是否为当前线程，若不是，则无法操作；若是，则进一步判断是否有其他线程在等待锁，有则释放锁并将唤醒等待的线程，将锁的所有者换为该线程，否则直接释放锁。

四**、**结果分析

joined with thread 2, exited 0.

joined with thread 3, exited 0.

joined with thread 4, exited 0.

joined with thread 5, exited 0.

joined with thread 6, exited 0.

joined with thread 7, exited 0.

joined with thread 8, exited 0.

joined with thread 9, exited 0.

joined with thread 10, exited 0.

joined with thread 11, exited 0.

joined with thread 12, exited 0.

joined with thread 13, exited 0.

joined with thread 14, exited 0.

joined with thread 15, exited 0.

joined with thread 16, exited 0.

joined with thread 17, exited 0.

第 5 个线程，第0 次循环

第 5 个线程，第1 次循环

第 5 个线程，第2 次循环

第 5 个线程，第3 次循环

第 5 个线程，第4 次循环

第 5 个线程，第5 次循环

第 5 个线程，第6 次循环

第 5 个线程，第7 次循环

第 5 个线程，第8 次循环

第 5 个线程，第9 次循环

第 5 个线程结束.

第 2 个线程，第0 次循环

第 3 个线程，第0 次循环

第 4 个线程，第0 次循环

第 6 个线程，第0 次循环

第 7 个线程，第0 次循环

第 8 个线程，第0 次循环

第 9 个线程，第0 次循环

第 10 个线程，第0 次循环

第 11 个线程，第0 次循环

第 12 个线程，第0 次循环

第 13 个线程，第0 次循环

第 14 个线程，第0 次循环

第 15 个线程，第0 次循环

第 16 个线程，第0 次循环

第 17 个线程，第0 次循环

第 2 个线程，第1 次循环

第 3 个线程，第1 次循环

第 4 个线程，第1 次循环

第 6 个线程，第1 次循环

第 7 个线程，第1 次循环

第 8 个线程，第1 次循环

第 9 个线程，第1 次循环

第 10 个线程，第1 次循环

第 11 个线程，第1 次循环

第 12 个线程，第1 次循环

第 13 个线程，第1 次循环

第 14 个线程，第1 次循环

第 15 个线程，第1 次循环

第 16 个线程，第1 次循环

第 17 个线程，第1 次循环

第 2 个线程，第2 次循环

第 3 个线程，第2 次循环

第 4 个线程，第2 次循环

第 6 个线程，第2 次循环

第 7 个线程，第2 次循环

第 8 个线程，第2 次循环

第 9 个线程，第2 次循环

第 10 个线程，第2 次循环

第 11 个线程，第2 次循环

第 12 个线程，第2 次循环

第 13 个线程，第2 次循环

第 14 个线程，第2 次循环

第 15 个线程，第2 次循环

第 16 个线程，第2 次循环

第 17 个线程，第2 次循环

第 2 个线程，第3 次循环

第 3 个线程，第3 次循环

第 4 个线程，第3 次循环

第 6 个线程，第3 次循环

第 7 个线程，第3 次循环

第 8 个线程，第3 次循环

第 9 个线程，第3 次循环

第 10 个线程，第3 次循环

第 11 个线程，第3 次循环

第 12 个线程，第3 次循环

第 13 个线程，第3 次循环

第 14 个线程，第3 次循环

第 15 个线程，第3 次循环

第 16 个线程，第3 次循环

第 17 个线程，第3 次循环

第 2 个线程，第4 次循环

第 3 个线程，第4 次循环

第 4 个线程，第4 次循环

第 6 个线程，第4 次循环

第 7 个线程，第4 次循环

第 8 个线程，第4 次循环

第 9 个线程，第4 次循环

第 10 个线程，第4 次循环

第 11 个线程，第4 次循环

第 12 个线程，第4 次循环

第 13 个线程，第4 次循环

第 14 个线程，第4 次循环

第 15 个线程，第4 次循环

第 16 个线程，第4 次循环

第 17 个线程，第4 次循环

第 2 个线程，第5 次循环

第 3 个线程，第5 次循环

第 4 个线程，第5 次循环

第 6 个线程，第5 次循环

第 7 个线程，第5 次循环

第 8 个线程，第5 次循环

第 9 个线程，第5 次循环

第 10 个线程，第5 次循环

第 11 个线程，第5 次循环

第 12 个线程，第5 次循环

第 13 个线程，第5 次循环

第 14 个线程，第5 次循环

第 15 个线程，第5 次循环

第 16 个线程，第5 次循环

第 17 个线程，第5 次循环

第 2 个线程，第6 次循环

第 3 个线程，第6 次循环

第 4 个线程，第6 次循环

第 6 个线程，第6 次循环

第 7 个线程，第6 次循环

第 8 个线程，第6 次循环

第 9 个线程，第6 次循环

第 10 个线程，第6 次循环

第 11 个线程，第6 次循环

第 12 个线程，第6 次循环

第 13 个线程，第6 次循环

第 14 个线程，第6 次循环

第 15 个线程，第6 次循环

第 16 个线程，第6 次循环

第 17 个线程，第6 次循环

第 2 个线程，第7 次循环

第 3 个线程，第7 次循环

第 4 个线程，第7 次循环

第 6 个线程，第7 次循环

第 7 个线程，第7 次循环

第 8 个线程，第7 次循环

第 9 个线程，第7 次循环

第 10 个线程，第7 次循环

第 11 个线程，第7 次循环

第 12 个线程，第7 次循环

第 13 个线程，第7 次循环

第 14 个线程，第7 次循环

第 15 个线程，第7 次循环

第 16 个线程，第7 次循环

第 17 个线程，第7 次循环

第 2 个线程，第8 次循环

第 3 个线程，第8 次循环

第 4 个线程，第8 次循环

第 6 个线程，第8 次循环

第 7 个线程，第8 次循环

第 8 个线程，第8 次循环

第 9 个线程，第8 次循环

第 10 个线程，第8 次循环

第 11 个线程，第8 次循环

第 12 个线程，第8 次循环

第 13 个线程，第8 次循环

第 14 个线程，第8 次循环

第 15 个线程，第8 次循环

第 16 个线程，第8 次循环

第 17 个线程，第8 次循环

第 2 个线程，第9 次循环

第 3 个线程，第9 次循环

第 4 个线程，第9 次循环

第 6 个线程，第9 次循环

第 7 个线程，第9 次循环

第 8 个线程，第9 次循环

第 9 个线程，第9 次循环

第 10 个线程，第9 次循环

第 11 个线程，第9 次循环

第 12 个线程，第9 次循环

第 13 个线程，第9 次循环

第 14 个线程，第9 次循环

第 15 个线程，第9 次循环

第 16 个线程，第9 次循环

第 17 个线程，第9 次循环

第 2 个线程结束.

第 3 个线程结束.

第 4 个线程结束.

第 6 个线程结束.

第 7 个线程结束.

第 8 个线程结束.

第 9 个线程结束.

第 10 个线程结束.

第 11 个线程结束.

第 12 个线程结束.

第 13 个线程结束.

第 14 个线程结束.

第 15 个线程结束.

第 16 个线程结束.

第 17 个线程结束.

uthreads: no more threads.

uthreads: bye!

五**、**课程设计总结

在这次的课程设计中，我学会了如何在多个线程中实现优先级调度，通过在线程的多个状态中不断转换，我对线程调度的理解又更进了一步。其中我也有遇到问题，比如因为对线程状态的理解不够深刻，在线程退出后将状态改为了UT\_WAIT而不是UT\_ZOMBIE，这导致reaper队列一直不为空，程序最后无法正常退出；通过重新查看代码，我修改了线程退出的函数并将其退出后的状态改为UT\_ZOMBIE，成功解决了问题。