# lab4-challenge实验报告

# 实现思路

## 线程

实现线程机制的过程可以分为两个部分：

- 实现线程机制的**环境**：定义在实现函数过程中需要的结构、变量等信息；
- 实现线程机制的**函数**：实现线程机制相关函数的具体功能。

### 环境实现

#### 线程控制块

- 根据课上所学习的理论知识，进程是资源分配的单位，线程是调度单位。而在 MOS 操作系统实验中，我们将资源分配与调度在同一个数据结构下实现了，即 `struct Env` 。
- 为了将资源分配的过程与调度的过程分离，需要重新设计相应的数据结构，以实现不同细度的操作。在实现过程中，考虑设置线程控制块 `struct Tcb` ，将线程调度过程中所需要的信息存储在其中，而将资源分配所需要的信息保留在进程控制块 `struct Env` 中：

```
struct Tcb {
    // basic
    struct Trapframe tcb_tf;
    u_int tcb_id;
    u_int tcb_status;
    u_int tcb_pri;
    LIST_ENTRY(Tcb) tcb_sched_link;

    // join
    LIST_ENTRY(Tcb) tcb_joined_link;
    LIST_HEAD(Tcb_joined_list, Tcb);
    struct Tcb_joined_list tcb_joined_list;
    void **tcb_join_value_ptr;
    u_int tcb_detach;

    // exit
    void *tcb_exit_ptr;
    int tcb_exit_value;

    //cancel
    int tcb_cancel_state;
    int tcb_cancel_type;
    u_int tcb_canceled;

    u_int tcb_nop[10];
};
```

```
struct Env {
    LIST_ENTRY(Env) env_link;       // Free list
```

```
    u_int env_id;                   // Unique environment identifier
    u_int env_parent_id;            // env_id of this env's parent
    Pde  *env_pgdir;                // Kernel virtual address of page
dir
    u_int env_cr3;

    // Lab 4 IPC
    u_int env_ipc_value;            // data value sent to us
    u_int env_ipc_from;             // envid of the sender
    u_int env_ipc_recving;          // env is blocked receiving
    u_int env_ipc_dstva;        // va at which to map received page
    u_int env_ipc_perm;     // perm of page mapping received
    u_int env_ipc_waiting_thread_no;

    // Lab 4 fault handling
    u_int env_pgfault_handler;      // page fault state
    u_int env_xstacktop;            // top of exception stack

    // Lab 6 scheduler counts
    u_int env_runs;         // number of times been env_run'ed

    u_int env_thread_count;
    u_int env_nop[496];                 // align to avoid mul
instruction
    struct Tcb env_threads[8];
};
```

- 在线程结构设计中，每个进程最大支持8个线程。

## 变量定义

- 为了方便编写程序，增加可读性，需要设置一些必要的宏定义以及错误号：
  - 宏定义：

```
/* include/env.h */
#define THREAD_MAX              8
#define THREAD_CAN_CANCEL       1
#define THREAD_CANNOT_CANCEL    0
#define THREAD_CANCEL_ASYN      0
#define THREAD_CANCEL_DEFER     1
#define THREAD_CANCEL_EXIT      99
```

  - 错误号：

```
#define E_THREAD_MAX            13
#define E_THREAD_NOT_CREATE     14
#define E_THREAD_NOT_FOUND      15
#define E_THREAD_CANNOT_CANCEL  16
#define E_THREAD_CANNOT_JOIN    17
```

# 函数实现

## 系统调用

- 由于 MOS 操作系统采用微内核设计，因此线程机制也将在用户态实现，故需要设计一些系统调用来完成内核操作：

```
u_int syscall_get_tcbid();
int syscall_thread_alloc();
int syscall_thread_destroy(u_int tcbid);
int syscall_set_thread_status(u_int tcbid, u_int status);
int syscall_thread_join(u_int tcbid, void **retval);
```

- `u_int syscall_get_tcbid();` 获取当前线程的线程id
  - 返回值：当前线程id

```
u_int sys_get_tcbid(void)
{
    return curtcb->tcb_id;
}
```

- `int syscall_thread_alloc();` 创建新线程
  - 返回值：成功 - 新线程id；失败 - 错误号

首先通过调用 `thread_alloc` 在当前进程下创建一个新的线程，然后根据进程的基本信息（存储在进程的0号线程中）为新线程进行相关设置。

```
int sys_thread_alloc(void)
{
    int r;
    struct Tcb *t;

    if (curenv)
        r = thread_alloc(curenv, &t);
    else
        r = -E_BAD_ENV;
    if (r < 0)
        return r;

    if (curenv)
        t->tcb_pri = curenv->env_threads[0].tcb_pri;
    else
        t->tcb_pri = 1;
    t->tcb_status = ENV_NOT_RUNNABLE;
    t->tcb_tf.pc = t->tcb_tf.cp0_epc;
    t->tcb_tf.regs[2] = 0;

    return t->tcb_id & 0x7;
}
```

- `int syscall_thread_destroy(u_int tcbid);` 销毁线程
  - 返回值：成功 - 0；失败 - 错误号

- 参数：`tcbid`：线程id

先将线程从对应的阻塞队列中删除，再将其销毁。

```
int sys_thread_destroy(int sysno, u_int tcbid)
{
    int r;
    struct Tcb *t;
    struct Tcb *tmp;

    if ((r = tcbid2tcb(tcbid, &t)) < 0) {
        return r;
    }
    if (t->tcb_status == ENV_FREE) {
        return -E_INVAL;
    }

    while (!LIST_EMPTY(&t->tcb_joined_list)) {
        tmp = LIST_FIRST(&t->tcb_joined_list);
        LIST_REMOVE(tmp, tcb_joined_link);
        *(tmp->tcb_join_value_ptr) = t->tcb_exit_ptr;
        sys_set_thread_status(0, tmp->tcb_id, ENV_RUNNABLE);
    }

    printf("[%08x] destroying a thread %08x\n", curenv->env_id, t->tcb_id);
    thread_destroy(t);
    return 0;
}
```

- `int syscall_set_thread_status(u_int tcbid, u_int status);` 设置线程状态
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `tcbid`：线程id
    - `status`：新状态

```
int sys_set_thread_status(int sysno, u_int tcbid, u_int status)
{
    int r;
    struct Tcb *t;

    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE && status != ENV_FREE)
        return -E_INVAL;
    r = tcbid2tcb(tcbid, &t);
    if (r < 0)
        return r;
    if (t->tcb_status != ENV_RUNNABLE && status == ENV_RUNNABLE)
        LIST_INSERT_HEAD(&tcb_sched_list[0], t, tcb_sched_link);
    if (t->tcb_status == ENV_RUNNABLE && status != ENV_RUNNABLE)
        LIST_REMOVE(t, tcb_sched_link);
    t->tcb_status = status;
    return 0;
```

```
}
```

- `int syscall_thread_join(u_int tcbid, void **retval);` 阻塞等待线程终止，获取线程终止状态
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `tcbid` ：线程id
    - `retval` ：存储线程终止状态

  将该线程插入当前线程的阻塞队列中，并为其保存现场，最后调度线程。

```
int sys_thread_join(int sysno, u_int tcbid, void **retval)
{
    int r;
    struct Tcb *t;

    r = tcbid2tcb(tcbid, &t);
    if (r < 0)
        return r;
    if (t->tcb_detach)
        return -E_THREAD_CANNOT_JOIN;

    if (t->tcb_status == ENV_FREE) {
        if (retval != 0) {
            *retval = t->tcb_exit_ptr;
        }
        return 0;
    }

    LIST_INSERT_HEAD(&t->tcb_joined_list, curtcb, tcb_joined_link);
    curtcb->tcb_join_value_ptr = retval;
    sys_set_thread_status(0, curtcb->tcb_id, ENV_NOT_RUNNABLE);
    struct Trapframe *tf = (struct Trapframe *)(KERNEL_SP -
sizeof(struct Trapframe));
    tf->regs[2] = 0;
    tf->pc = tf->cp0_epc;
    sys_yield();
    return 0;
}
```

## 线程操作函数

- 线程操作函数是实现线程机制的重点，是直接操作线程对象的函数：

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void
* (*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
int pthread_detach(pthread_t thread);
int pthread_join(pthread_t thread, void **retval);
```

- `int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);` 创建一个新线程
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `thread` ：传出参数，保存新分配的线程id
    - `attr` ：线程默认属性，由于为实现相关功能，因此在使用时传入 `NULL`
    - `(*start_routine)(void *)` ：函数指针，指向线程主函数（线程体），当该函数运行结束时线程结束
    - `arg` ：线程主函数执行期间所使用的参数，如要传多个参数，可以用数组或结构封装

  通过系统调用获取新线程的线程号并配置参数进程对应的线程，为新线程设置栈空间，执行入口，返回地址等信息。

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg) {
    int newthread = syscall_thread_alloc();
    if (newthread < 0) {
        thread = 0;
        return -newthread;
    }

    struct Tcb *t = &env->env_threads[newthread];
    t->tcb_tf.regs[29] = USTACKTOP - 4 * BY2PG * newthread - 4;
    t->tcb_tf.pc = start_routine;
    t->tcb_tf.regs[4] = arg;
    t->tcb_tf.regs[31] = exit;
    syscall_set_thread_status(t->tcb_id, ENV_RUNNABLE);
    *thread = t->tcb_id;
    return 0;
}
```

- `void pthread_exit(void *retval);` 终止当前线程
  - 参数：`retval` ：存储线程终止状态

```
void pthread_exit(void *retval) {
    u_int tcbid = syscall_get_tcbid();
    struct Tcb *t = &env->env_threads[tcbid & 0x7];
    t->tcb_exit_ptr = retval;
    syscall_thread_destroy(tcbid);
}
```

- `int pthread_cancel(pthread_t thread);` 撤销指定线程
  - 若指定线程不可撤销则调用无效
  - 若线程为异步撤销类型，则立即撤销该线程；若为延迟撤销类型，则等待调用 `pthread_testcancel` 时撤销
  - 返回值：成功 - 0；失败 - 错误号

首先判断线程能否撤销，再根据撤销类型执行立即终止或延迟终止。

```
int pthread_cancel(pthread_t thread) {
    struct Tcb *t = &env->env_threads[thread & 0x7];
```

```
    if ((t->tcb_id != thread) || (t->tcb_status == ENV_FREE)) {
        return -E_THREAD_NOT_FOUND;
    }

    if (t->tcb_cancel_state == THREAD_CANNOT_CANCEL) {
        return -E_THREAD_CANNOT_CANCEL;
    }

    t->tcb_exit_value = -THREAD_CANCEL_EXIT;
    if (t->tcb_cancel_type == THREAD_CANCEL_ASYN) {
        syscall_thread_destroy(thread);
    } else {
        t->tcb_canceled = 1;
    }

    return 0;
}
```

- `int pthread_setcancelstate(int state, int *oldstate);` 设置线程撤销状态（可撤销，不可撤销）
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `state` ：线程新的撤销状态
    - `oldstate` ：存储线程旧的撤销状态

```
int pthread_setcancelstate(int state, int *oldstate) {
    u_int tcbid = syscall_get_tcbid();
    struct Tcb *t = &env->env_threads[tcbid & 0x7];

    if ((state != THREAD_CAN_CANCEL) && (state != THREAD_CANNOT_CANCEL))
{
        return -E_INVAL;
    }

    if (t->tcb_id != tcbid) {
        return -E_INVAL;
    }

    if (oldstate != 0) {
        *oldstate = t->tcb_cancel_state;
    }

    t->tcb_cancel_state = state;
    return 0;
}
```

- `int pthread_setcanceltype(int type, int *oldtype);` 设置线程撤销类型（异步撤销，延迟撤销）
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：

- `type`：线程新的撤销类型
- `oldtype`：存储线程旧的撤销类型

```
int pthread_setcanceltype(int type, int *oldtype) {
    u_int tcbid = syscall_get_tcbid();
    struct Tcb *t = &env->env_threads[tcbid & 0x7];

    if ((type != THREAD_CANCEL_ASYN) && (type != THREAD_CANCEL_DEFER)) {
        return -E_INVAL;
    }

    if (t->tcb_id != tcbid) {
        return -E_INVAL;
    }

    if (oldtype != 0) {
        *oldtype = t->tcb_cancel_type;
    }

    t->tcb_cancel_type = type;
    return 0;
}
```

- `void pthread_testcancel(void);` 尝试撤销当前线程

  若进程可以撤销且为延迟撤销，则撤销该进程，否则调用无效

```
void pthread_testcancel() {
    u_int tcbid = syscall_get_tcbid();
    struct Tcb *t = &env->env_threads[tcbid & 0x7];

    if (t->tcb_id != tcbid) {
        user_panic("something wrong in pthread_testcancel!\n");
    }

    if ((t->tcb_canceled) &&
        (t->tcb_cancel_state == THREAD_CAN_CANCEL) &&
        (t->tcb_cancel_type == THREAD_CANCEL_DEFER)) {
        t->tcb_exit_value = -THREAD_CANCEL_EXIT;
        syscall_thread_destroy(t->tcb_id);
    }
}
```

- `int pthread_detach(pthread_t thread);` 实现线程分离
  - 返回值：成功 - 0；失败 - 错误号

  在实现线程分离后，其他线程无法再对其执行 `pthread_join` 操作，终止状态也无法被其他线程获取

```
int pthread_detach(pthread_t thread) {
    struct Tcb *t = &env->env_threads[thread & 0x7];
    int r, i;

    if (t->tcb_id != thread) {
```

```
        return -E_THREAD_NOT_FOUND;
    }

    if (t->tcb_status == ENV_FREE) {
        u_int sp = USTACKTOP - BY2PG * 4 * (thread & 0x7);
        for(i = 1; i <= 4; ++i) {
            r = syscall_mem_unmap(0, sp - i * BY2PG);
            if (r < 0)
                return r;
        }
        user_bzero(t, sizeof(struct Tcb));
    } else {
        t->tcb_detach = 1;
    }
    return 0;
}
```

- `int pthread_join(pthread_t thread, void **retval);` 阻塞等待线程终止，获取线程终止状态，并返回是否成功
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `thread` ：线程id
    - `retval` ：存储线程终止状态

```
int pthread_join(pthread_t thread, void **retval) {
    int r = syscall_thread_join(thread, retval);
    return r;
}
```

## 其他函数

- 在将调度的基本单位改为线程后，原本与进程调度相关的函数都需要进行相应修改。
- 为配合线程操作函数实现对应功能，需要额外实现一些相关函数：

```
int thread_alloc(struct Env *e, struct Tcb **t);
void thread_destroy(struct Tcb *t);
void thread_free(struct Tcb *t);
u_int mktcbid(struct Tcb *t);
int tcbid2tcb(u_int tcbid, struct Tcb **ptcb);
```

- `int thread_alloc(struct Env *e, struct Tcb **t);` 分配新线程
  - 返回值：成功 - 线程id；失败 - 错误号
  - 参数：
    - `e` ：线程所属进程
    - `t` ：存储新线程控制块地址

```
int thread_alloc(struct Env *e, struct Tcb **new) {
    if (e->env_thread_count >= THREAD_MAX) {
        return E_THREAD_MAX;
    }
```

```
    u_int thread_num = e->env_thread_count;
    u_int i = 0;
    while (e->env_threads[thread_num].tcb_status != ENV_FREE) {
        thread_num++;
        thread_num = thread_num % THREAD_MAX;
        i++;
        if (i >= THREAD_MAX) {
            return E_THREAD_MAX;
        }
    }

    e->env_thread_count++;
    struct Tcb *t = &e->env_threads[thread_num];
    t->tcb_id = mktcbid(t);
    printf("the new thread id is %x\n", t->tcb_id);
    t->tcb_status = ENV_RUNNABLE;
    t->tcb_exit_ptr = (void *)0;
    t->tcb_tf.cp0_status = 0x1000100c;
    t->tcb_tf.regs[29] = USTACKTOP - 4 * BY2PG * (t->tcb_id & 0x7);
    t->tcb_cancel_state = THREAD_CANNOT_CANCEL;
    t->tcb_cancel_type = THREAD_CANCEL_ASYN;
    t->tcb_canceled = 0;
    t->tcb_exit_value = 0;
    t->tcb_exit_ptr = (void *)&t->tcb_exit_value;
    t->tcb_detach = 0;
    LIST_INIT(&t->tcb_joined_list);
    *new = t;
    return 0;
}
```

- `void thread_destroy(struct Tcb *t);` 销毁线程
  - 参数：`t`：线程控制块

```
void thread_destroy(struct Tcb *t) {
    if (t->tcb_status == ENV_RUNNABLE)
        LIST_REMOVE(t, tcb_sched_link);
    thread_free(t);
    if (curtcb == t) {
        curtcb == NULL;
        bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
              (void *)TIMESTACK - sizeof(struct Trapframe),
              sizeof(struct Trapframe));
        printf("i am a thread ... i am killed ...\n");
        sched_yield();
    }
}
```

- `void thread_free(struct Tcb *t);` 释放线程资源，在 `thread_destroy` 中调用
  - 参数：`t`：线程控制块

```c
void thread_free(struct Tcb *t) {
    struct Env *e = ROUNDDOWN(t, BY2PG);
    printf("[%08x] free thread %08x\n", e->env_id, t->tcb_id);
    e->env_thread_count--;
    if (e->env_thread_count <= 0) {
        env_free(e);
    }
    t->tcb_status = ENV_FREE;
}
```

- `u_int mktcbid(struct Tcb *t)`
  - 返回值：新线程id
  - 参数：`t`：线程控制块

```c
u_int mktcbid(struct Tcb *t) {
    struct Env *e = ROUNDDOWN(t, BY2PG);
    u_int tcb_num = ((u_int)t - (u_int)e - BY2PG / 2) / 256;
    return ((e->env_id << 3) | tcb_num);
}
```

- `int tcbid2tcb(u_int tcbid, struct Tcb **ptcb);` 将线程id转换为对应进程
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `tcbid`：线程id
    - `ptcb`：存储对应线程控制块

```c
int tcbid2tcb(u_int tcbid, struct Tcb **ptcb) {
    struct Env *e;
    struct Tcb *t;

    if (tcbid == 0) {
        *ptcb = curtcb;
        return 0;
    }

    e = &envs[ENVX(tcbid >> 3)];
    t = &e->env_threads[tcbid & 0x7];
    if (t->tcb_status == ENV_FREE || t->tcb_id != tcbid) {
        *ptcb = 0;
        return -E_BAD_ENV;
    }

    *ptcb = t;
    return 0;
}
```

# 信号量

# 环境实现

## 信号量结构

- 为了完整描述信号量的信息，需要定义信号量结构：

```
struct Sem {
    u_int sem_envid;
    char sem_name[20];
    int sem_value;
    int sem_status;
    int sem_shared;
    int sem_wait_count;
    u_int sem_head_index;
    u_int sem_tail_index;
    struct Tcb *sem_wait_list[10];
};
```

- 信号量结构设计中，最大支持10个线程同时等待该信号量。

## 变量定义

- 为了方便编写程序，增加可读性，需要设置一些必要的宏定义以及错误号：
  - 宏定义：

```
/* include/env.h */
#define SEM_INVALID 0
#define SEM_VALID   1
```

  - 错误号：

```
/* include/error.h */
#define E_SEM_ERROR     18
#define E_SEM_NOT_FOUND 19
#define E_SEM_EAGAIN    20
```

# 函数实现

## 系统调用

- 信号量在用户态实现，故也需要设计一些系统调用来完成内核操作：

```
int syscall_sem_destroy(sem_t *sem);
int syscall_sem_wait(sem_t *sem);
int syscall_sem_trywait(sem_t *sem);
int syscall_sem_post(sem_t *sem);
int syscall_sem_getvalue(sem_t *sem, int *valp);
```

- `int syscall_sem_destroy(sem_t *sem);` 销毁信号量
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：

    - `sem`：信号量

```
int sys_sem_destroy(int sysno, sem_t *sem)
{
    if (sem->sem_envid != curenv->env_id && sem->sem_shared == 0) {
        return -E_SEM_NOT_FOUND;
    }

    if (sem->sem_status == SEM_INVALID) {
        return 0;
    }

    sem->sem_status = SEM_INVALID;
    return 0;
}
```

- `int syscall_sem_wait(sem_t *sem);` 对信号量的 P 操作（阻塞）
    - 返回值：成功 - 0；失败 - 错误号
    - 参数：
        - `sem`：信号量

尝试对信号量进行 P 操作，如果失败，则将当前线程添加入信号量的阻塞队列中并为其保存现场，最后调度线程。

```
int sys_sem_wait(int sysno, sem_t *sem)
{
    if (sem->sem_status == SEM_INVALID) {
        return -E_SEM_ERROR;
    }

    if (sem->sem_value > 0) {
        sem->sem_value--;
        return 0;
    }

    if (sem->sem_wait_count >= 10) {
        return -E_SEM_ERROR;
    }

    sem->sem_wait_list[sem->sem_head_index] = curtcb;
    sem->sem_head_index = (sem->sem_head_index + 1) % 10;
    sem->sem_wait_count++;
    sys_set_thread_status(0, 0, ENV_NOT_RUNNABLE);
    struct Trapframe *tf = (struct Trapframe *)(KERNEL_SP -
sizeof(struct Trapframe));
    tf->regs[2] = 0;
    tf->pc = tf->cp0_epc;
    sys_yield();
    return -E_SEM_ERROR;
}
```

- `int syscall_sem_trywait(sem_t *sem);` 对信号量的 P 操作（非阻塞）
    - 返回值：成功 - 0；失败 - 错误号

- 参数：
  - `sem`：信号量

```
int sys_sem_trywait(int sysno, sem_t *sem)
{
    if (sem->sem_status == SEM_INVALID) {
        return -E_SEM_ERROR;
    }

    if (sem->sem_value > 0) {
        sem->sem_value--;
        return 0;
    }
    return -E_SEM_EAGAIN;
}
```

- `int syscall_sem_post(sem_t *sem);` 对信号量的 V 操作
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量

  如果信号量等待队列中有线程，则将其唤醒，设置为就绪状态。

```
int sys_sem_post(int sysno, sem_t *sem)
{
    if (sem->sem_status == SEM_INVALID) {
        return -E_SEM_ERROR;
    }

    if (sem->sem_value > 0 || sem->sem_wait_count == 0) {
        sem->sem_value ++;
        return 0;
    }

    struct Tcb *t;
    sem->sem_wait_count--;
    t = sem->sem_wait_list[sem->sem_tail_index];
    sem->sem_wait_list[sem->sem_tail_index] = 0;
    sem->sem_tail_index = (sem->sem_tail_index + 1) % 10;
    sys_set_thread_status(0, t->tcb_id, ENV_RUNNABLE);
    return 0;
}
```

- `int syscall_sem_getvalue(sem_t *sem, int *valp);` 读取信号量的值
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量
    - `valp`：存储信号量的值

```c
int sys_sem_getvalue(int sysno, sem_t *sem, int *valp)
{
    if (sem->sem_status == SEM_INVALID) {
        return -E_SEM_ERROR;
    }
    if (valp == 0) {
        return -E_INVAL;
    }

    *valp = sem->sem_value;
    return 0;
}
```

## 信号量操作函数

- 信号量操作函数是实现信号量机制的重点，是直接操作信号量结构的函数：

```c
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *valp);
```

- `int sem_init(sem_t *sem, int pshared, unsigned int value);` 创建并初始化信号量
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：传出参数，保存新创建的信号量
    - `pshared`：信号量是否共享
    - `value`：信号量的值

```c
int sem_init(sem_t *sem, int pshared, unsigned int value) {
    if (sem == 0) {
        return -E_SEM_ERROR;
    }

    sem->sem_head_index = 0;
    sem->sem_tail_index = 0;
    sem->sem_envid = env->env_id;
    sem->sem_name[0] = '\0';
    sem->sem_value = value;
    sem->sem_shared = pshared;
    sem->sem_status = SEM_VALID;
    sem->sem_wait_count = 0;
    int i;
    for (i = 0; i < 10; i++) {
        sem->sem_wait_list[i] = 0;
    }
    return 0;
}
```

- `int sem_destroy(sem_t *sem);` 销毁信号量

- 返回值：成功 - 0；失败 - 错误号
- 参数：
  - `sem`：信号量

```c
int sem_destroy(sem_t *sem) {
    return syscall_sem_destroy(sem);
}
```

- `int sem_wait(sem_t *sem);` 对信号量的 P 操作（阻塞）
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量

```c
int sem_wait(sem_t *sem) {
    return syscall_sem_wait(sem);
}
```

- `int sem_trywait(sem_t *sem);` 对信号量的 P 操作（非阻塞）
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量

```c
int sem_trywait(sem_t *sem) {
    return syscall_sem_trywait(sem);
}
```

- `int sem_post(sem_t *sem);` 对信号量的 V 操作
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量

```c
int sem_post(sem_t *sem) {
    return syscall_sem_post(sem);
}
```

- `int sem_getvalue(sem_t *sem, int *valp);` 读取信号量的值
  - 返回值：成功 - 0；失败 - 错误号
  - 参数：
    - `sem`：信号量
    - `valp`：存储信号量的值

```c
int sem_getvalue(sem_t *sem, int *valp) {
    return syscall_sem_getvalue(sem, valp);
}
```

# 功能测试

## 线程

### `pthreadtest`

- 测试内容：线程创建，创建时传参，栈空间共享
- 测试程序：

```c
#include "lib.h"

void *pthreadtest(void *arg) {
    int arg1 = ((int *)arg)[0];
    int arg2 = ((int *)arg)[1];
    int arg3 = ((int *)arg)[2];
    int *stacki = (int *)((int *)arg)[3];

    writef("arg 1 is %d\n", arg1);
    writef("arg 2 is %d\n", arg2);
    writef("arg 3 is %d\n", arg3);
    if (arg1 == 1 && arg2 == 2 && arg3 == 3) {
        writef("passing args success!\n");
    } else {
        user_panic("passing args error!\n");
    }

    (*stacki) = 1;
    writef("[[%x]] stacki = %d\n", syscall_get_tcbid(), *stacki);
}

void umain() {
    writef("Test start ... \n");
    pthread_t thread;
    int tcbid;
    int i = 0;
    int args[4];
    args[0] = 1;
    args[1] = 2;
    args[2] = 3;
    args[3] = &i;

    tcbid = pthread_create(&thread, NULL, pthreadtest, (void *)args);
    if (tcbid == 0) {
        writef("thread create success!\n");
    } else {
        user_panic("thread create error!\n");
    }

    while(1) {
        writef("");
        if (i != 0) break;
        syscall_yield();
    }
    writef("[[%x]] i = %d\n", syscall_get_tcbid(), i);
```

```
}
```

- 测试结果：

```
Test start ...

the new thread id is 2001

thread create success!

pageout:        @@@___0x7f3f9ff4___@@@  ins a page

arg 1 is 1

arg 2 is 2

arg 3 is 3

passing args success!

[[2001]] stacki = 1

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

i am a thread ... i am killed ...

[[2000]] i = 1

[00000400] destroying a thread 00002000

[00000400] free thread 00002000

[00000400] free env 00000400

i am a thread ... i am killed ...
```

## pthreadexit

- 测试内容：线程终止
- 测试程序：

```c
#include "lib.h"

void *pthreadexit(void *arg) {
    writef("pthreadexit start ...\n");
    writef("[[%x]] arg %d\n", syscall_get_tcbid(), *((int *)arg));
    pthread_exit(arg);
    user_panic("thread didn't exit!\n");
}

void umain() {
    writef("Test start ... \n");
    pthread_t thread;
    int tcbid;
```

```
    int arg = 99;
    tcbid = pthread_create(&thread, NULL, pthreadexit, (void *)(&arg));
    if (tcbid == 0) {
        writef("thread create success!\n");
    } else {
        user_panic("thread create error!\n");
    }
    while (env->env_threads[thread & 0x7].tcb_status != ENV_FREE) {
        writef("");
    }
    writef("get retval %d, expecting %d\n", *((int *)env->env_threads[thread & 0x7].tcb_exit_ptr), arg);
    if (*((int *)env->env_threads[thread & 0x7].tcb_exit_ptr) == arg) {
        writef("retval correct!\n");
    } else {
        user_panic("retval incorrect!\n");
    }
}
```

- 测试结果：

```
Test start ...

the new thread id is 2001

thread create success!

pageout:        @@@___0x7f3f9ff8___@@@  ins a page

pthreadexit start ...

[[2001]] arg 99

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

i am a thread ... i am killed ...

get retval 99, expecting 99

retval correct!

[00000400] destroying a thread 00002000

[00000400] free thread 00002000

[00000400] free env 00000400

i am a thread ... i am killed ...
```

## pthreadcancel

- 测试内容：线程撤销（可撤销，不可撤销，异步撤销，延迟撤销）
- 测试程序：

```
#include "lib.h"

void *test1(void *args) {
    int oldval;
    pthread_setcancelstate(1, &oldval);
    writef("[[%x]] oldstate %d\n",syscall_get_tcbid(), oldval);
    while(1);
}

void *test2(void *args) {
    int oldval;
    int tcbid = syscall_get_tcbid();
    pthread_setcanceltype(1, &oldval);
    writef("[[%x]] oldtype %d and newtype 1\n", tcbid, oldval);
    pthread_setcancelstate(1, &oldval);
    writef("[[%x]] oldstate %d and newstate 1\n", tcbid, oldval);
    while (1) {
        writef("[[%x]] try cancel ...\n", tcbid);
        pthread_testcancel();
        writef("cannot be canceled!\n");
        syscall_yield();
    }
}

void umain() {
    int arg = 1, ret, tcbid;
    pthread_t thread1, thread2;

    tcbid = pthread_create(&thread1, NULL, test1, (void *)arg);
    if (tcbid == 0) {
        writef("thread create success!\n");
    } else {
        user_panic("thread create error!\n");
    }
    syscall_yield();

    ret = pthread_cancel(thread1);
    if (ret == 0) {
        writef("cancel success!\n");
    } else {
        user_panic("cancel fail!\n");
    }

    tcbid = pthread_create(&thread2, NULL, test2, (void *)arg);
    if (tcbid == 0) {
        writef("thread create success!\n");
    } else {
        user_panic("thread create error!\n");
    }

```

```
    while (pthread_cancel(thread2) < 0);
    writef("thread testcancel success!\n");
    syscall_yield();
}
```

- 测试结果：

```
Test start ...

the new thread id is 2001

thread create success!

pageout:        @@@___0x7f3f9ff4___@@@  ins a page

[[2001]] oldstate 0

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

cancel success!

the new thread id is 2001

thread create success!

[[2001]] oldtype 0 and newtype 1

[[2001]] oldstate 0 and newstate 1

[[2001]] try cancel ...

cannot be canceled!

[[2001]] try cancel ...

cannot be canceled!

thread testcancel success!

[00000400] destroying a thread 00002000

[00000400] free thread 00002000

i am a thread ... i am killed ...

[[2001]] try cancel ...

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

[00000400] free env 00000400

i am a thread ... i am killed ...
```

## pthreadjoin

- 测试内容：线程阻塞和分离
- 测试程序：

```c
#include "lib.h"

void *test1(void *args) {
    int father_tcbid = *((int *)args);
    int tcbid = syscall_get_tcbid();
    u_int *retval;
    writef("test1 start ...\n");
    if (pthread_join(father_tcbid, &retval) == 0) {
        writef("[[%x]] join success!\n", tcbid);
    } else {
        writef("[[%x]] join fail!\n", tcbid);
        pthread_exit(NULL);
    }
    writef("[[%x]] father %x exit with retval %d\n", tcbid, father_tcbid, *retval);
}

void *test2(void *args) {
    u_int father_tcbid = *((int *)args);
    int tcbid = syscall_get_tcbid();
    u_int *retval;
    writef("test2 start ...\n");
    if (pthread_join(father_tcbid, &retval) == 0) {
        writef("[[%x]] join success!\n", tcbid);
    } else {
        writef("[[%x]] join fail!\n", tcbid);
        pthread_exit(NULL);
    }
    writef("[[%x]] father %x exit with retval %d\n", tcbid, father_tcbid, *retval);
}

void umain() {
    u_int arg = syscall_get_tcbid();
    int ret = 99;
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, test1, (void *)(&arg));
    syscall_yield();

    pthread_detach(arg);
    pthread_create(&thread2, NULL, test2, (void *)(&arg));
    syscall_yield();
    syscall_yield();

    pthread_exit(&ret);
}
```

- 测试结果：

```
Test start ...

the new thread id is 2001

pageout:        @@@___0x7f3f9ff4___@@@  ins a page

test1 start ...

the new thread id is 2002

pageout:        @@@___0x7f3f5ff4___@@@  ins a page

test2 start ...

[[2002]] join fail!

[00000400] destroying a thread 00002002

[00000400] free thread 00002002

i am a thread ... i am killed ...

[00000400] destroying a thread 00002000

[00000400] free thread 00002000

i am a thread ... i am killed ...

[[2001]] join success!

[[2001]] father 2000 exit with retval 99

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

[00000400] free env 00000400

i am a thread ... i am killed ...
```

# 信号量

- 测试内容：信号量的综合操作
- 测试代码：

```c
#include "lib.h"

void *test1(void *args) {
    sem_t *sem = (sem_t *)(*((u_int *)args));
    int tcbid = syscall_get_tcbid();
    int i, ret, value;

    ret = sem_getvalue(sem, &value);
    if (ret == 0) {
        writef("[[%x]] get value %d\n", tcbid, value);
    } else {
```

```
        user_panic("[[%x]] get value fail!\n", tcbid);
    }

    for (i = 0; i < 2; i++) {
        ret = sem_wait(sem);
        if (ret == 0) {
            writef("[[%x]] sem_wait success!\n", tcbid);
        } else {
            writef("[[%x]] sem_wait fail!\n", tcbid);
        }
        syscall_yield();
    }
}

void *test2(void *args) {
    sem_t *sem = (sem_t *)(*((u_int *)args));
    int tcbid = syscall_get_tcbid();
    int i, ret, value;

    ret = sem_getvalue(sem, &value);
    if (ret == 0) {
        writef("[[%x]] get value %d\n", tcbid, value);
    } else {
        user_panic("[[%x]] get value fail!\n", tcbid);
    }

    for (i = 0; i < 2; i++) {
        ret = sem_trywait(sem);
        if (ret == 0) {
            writef("[[%x]] sem_trywait success!\n", tcbid);
        } else {
            writef("[[%x]] sem_trywait fail!\n", tcbid);
        }
        syscall_yield();
    }
}

void umain() {
    writef("Test start ... \n");
    pthread_t thread1, thread2;
    sem_t sem;
    u_int arg;
    int ret, value;
    int tcbid = syscall_get_tcbid();

    sem_init(&sem, 0, 1);
    arg = &sem;
    pthread_create(&thread1, NULL, test1, (void *)(&arg));
    pthread_create(&thread2, NULL, test2, (void *)(&arg));

    ret = sem_getvalue(&sem, &value);
    if (ret == 0) {
        writef("[[%x]] get value %d\n", tcbid, value);
    } else {
        user_panic("[[%x]] get value fail!\n", tcbid);
```

```
    }

    int i = 0;
    for (i = 0; i < 5; ++i) {
        if (value == 0) {
            writef("[[%x]] sem_post!\n", tcbid);
            sem_post(&sem);
        }
        syscall_yield();
        ret = sem_getvalue(&sem, &value);
        if (ret < 0) {
            user_panic("[[%x]] get value fail!\n", tcbid);
        }
    }

    sem_destroy(&sem);
    ret = sem_getvalue(&sem, &value);
    if (ret < 0) {
        writef("sem_destroy success!\n");
    } else {
        user_panic("sem_destroy fail!\n");
    }
}
```

- 测试结果:

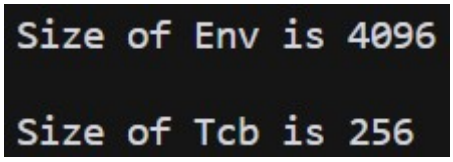
```
Test start ...

the new thread id is 2001

the new thread id is 2002

[[2000]] get value 1

pageout:        @@@___0x7f3f5ff8___@@@  ins a page

[[2002]] get value 1

[[2002]] sem_trywait success!

pageout:        @@@___0x7f3f9ff8___@@@  ins a page

[[2001]] get value 0

[[2002]] sem_trywait fail!

[[2000]] sem_post!

[[2000]] sem_post!

[[2001]] sem_w[00000400] destroying a thread 00002002

[00000400] free thread 00002002

i am a thread ... i am killed ...

ait success!

[[2001]] sem_wait success!

[00000400] destroying a thread 00002001

[00000400] free thread 00002001

i am a thread ... i am killed ...

sem_destroy success!

[00000400] destroying a thread 00002000

[00000400] free thread 00002000

[00000400] free env 00000400

i am a thread ... i am killed ...
```

# 实验难点及解决方案

## 进程控制块和线程控制块的大小

- 在完成线程机制后进行初步测试的过程中发现程序会产生意料之外的结果，新建线程的线程id并不符合人工计算的结果，且还有一定几率创建线程失败。

- 经过排查后，发现是进程控制块和线程控制块的大小设置问题：MOS 操作系统的页大小为4KB，如果进程控制块或线程控制块未进行对齐而被分配在两个页中，则会导致控制块中数据的丢失。
- 为此，考虑在进程控制块和线程控制块中添加 `env_nop` 和 `tcb_nop` 数组作为占位元素，将进程控制块和线程控制块的大小控制在适当值：




## 测试程序设计和编写

- 在完成代码编写后，如何能够完整、准确、简介地测试所实现的功能同样是一个非常重要的任务。
- 在编写测试代码的过程中，我参考了网络上在linux中演示线程操作和信号量机制的程序，并对其加以改编，使其能够测试此实现的正确性。