Lab4挑战性任务 线程与信号量

19373135 田旗舰

一、实验要求

1.线程

要求实现全用户地址空间共享，线程之间可以互相访问栈内数据。可以保留少量仅限于本线程可以访问的空间用以保存线程相关数据。（POSIX 标准有规定相关接口也可以实现）。包含以下函数：

- pthread_create 创建线程
- pthread_exit 退出线程
- pthread_cancel 撤销线程
- pthread_join 等待线程结束

2.信号量

POSIX 标准的信号量分为有名和无名信号量。无名信号量可以用于进程内同步与通信，有名信号量既可以用于进程内同步与通信，也可以用于进程间同步与通信。要求至少实现无名信号量的全部功能。包含以下函数：

- sem_init 初始化信号量
- sem_destroy 销毁信号量
- sem_wait 对信号量P操作（阻塞）
- sem_trywait 对信号量P操作（非阻塞）
- sem_post 对信号量V操作
- sem_getvalue 取得信号量的值

二、具体实现

1.数据结构

对于线程的数据结构一开始有两种想法，一是单独定义线程的数据结构，二是借助进程的数据结构 struct Env，增加部分属性以实现线程。单独定义线程的数据结构的好处是进程与线程的区分更加明显，操作也更加灵活，但考虑到在之前的实验中，资源的分配和调度都是以进程为单位进行的，重新定义线程需要修改大量代码，而借助进程的数据结构则可以十分方便地实现线程。故采用借助进程的数据结构的思路，这样一来，进程的第一个线程即为进程本身，只能由创建进程的方法（init或fork）产生，而其他的线程由第一个线程通过pthread_create创建。具体的数据结构及含义如下：

```
/*include/env.h*/
struct Env {
    //...
    struct Env *tcb_parent;//线程的进程
    struct Env *tcb_children[NTCB];//进程的线程
    u_int tcb_childnum;//进程的线程数
    u_int tcb_status;//线程的状态，0代表RUNNABLE，1代表DEAD
    void *retval;//线程返回值
    LIST_ENTRY(Env) env_blocked_link;//阻塞线程链表
}
```

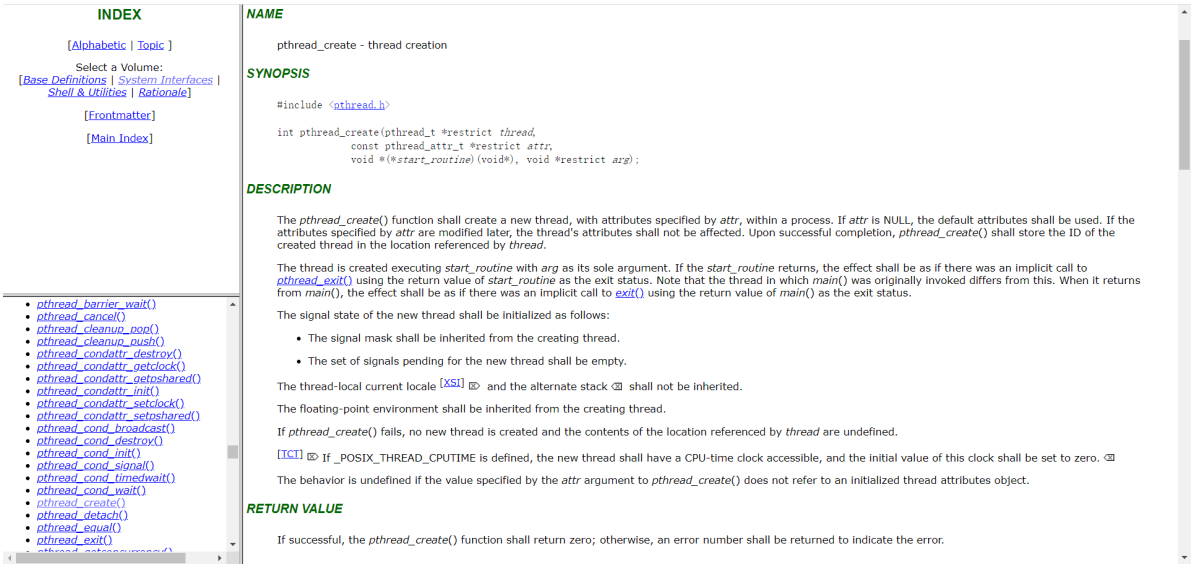
信号量的数据结构相对简单，根据POSIX中相关接口的描述以及理论课介绍，信号量的数据结构定义如下：

```
/*include/semaphore.h*/
typedef struct {
    int count;//资源数
    struct Env_list queue;//阻塞队列
    void *pshared;//0代表进程内共享，非0代表进程间共享
} sem_t
```

2.线程

pthread_create

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- pthread_barrier_wait()
- pthread_cancel()
- pthread_cleanup_pop()
- pthread_cleanup_push()
- pthread_condattr_destroy()
- pthread_condattr_getclock()
- pthread_condattr_getpshared()
- pthread_condattr_init()
- pthread_condattr_setclock()
- pthread_condattr_setpshared()
- pthread_cond_broadcast()
- pthread_cond_destroy()
- pthread_cond_init()
- pthread_cond_signal()
- pthread_cond_timedwait()
- pthread_cond_wait()
- pthread_create()
- pthread_detach()
- pthread_equal()
- pthread_exit()

NAME

pthread_create - thread creation

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
            const pthread_attr_t *restrict attr,
            void *(*start_routine)(void*), void *restrict arg);
```

DESCRIPTION

The *pthread_create*() function shall create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are modified later, the thread's attributes shall not be affected. Upon successful completion, *pthread_create*() shall store the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to *pthread_exit*() using the return value of *start_routine* as the exit status. Note that the thread in which *main*() was originally invoked differs from this. When it returns from *main*(), the effect shall be as if there was an implicit call to *exit*() using the return value of *main*() as the exit status.

The signal state of the new thread shall be initialized as follows:

- The signal mask shall be inherited from the creating thread.
- The set of signals pending for the new thread shall be empty.

The thread-local current locale [XSI] ⌦ and the alternate stack ⌫ shall not be inherited.

The floating-point environment shall be inherited from the creating thread.

If *pthread_create*() fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

[TCT] ⌦ If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero. ⌫

The behavior is undefined if the value specified by the *attr* argument to *pthread_create*() does not refer to an initialized thread attributes object.

RETURN VALUE

If successful, the *pthread_create*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

共有4个参数：

- pthread_t *thread：新创建线程的id。
- const pthread_attr_t *attr：线程的属性（本次实验中并未涉及）。
- void *(*start_routine)(void*)：线程执行的函数，当执行结束时，隐式调用pthread_exit()。
- void *arg：上述函数的参数。

具体实现：

在我的实现中，pthread_create通过调用官方代码中留下的接口sfork，来实现创建线程的功能。sfork的实现与fork类似，不同之处为从UTEXT到USTACKTOP的空间页面共享（UTEXT之下定义为线程控制块的空间和有名信号量的空间，因此不能共享），即将duppage修改为自定义的函数libpage，此外，还需要设置新线程的入口函数地址，以及相应的参数，这里的入口地址设置为自定义的thread_wrapper函数，其主要功能为将线程指针指向自身、调用线程执行的函数start_routine，返回后调用pthread_exit()，满足POSIX定义中隐式调用pthread_exit()。成功创建后，将*thread指向新线程的id，返回0。

```c
/*user/pthread.c*/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine), void *arg) {
    u_int newthreadid = sfork((u_int) thread_wrapper, (u_int) start_routine,
(u_int) arg, USTACKTOP - PDMAP * (env->tcb_childnum + 1));//每个线程设置PDMAP大小的
栈
    *t = newthreadid;
    return 0;
}
```

```c
/*user/pthread.c*/
static void thread_wrapper(void *(*start_routine)(void *), void *arg, u_int
envid) {
    *thread = envs + ENVX(envid);//thread为指向当前线程的指针
    void *retval = (*start_routine)(arg);
    pthread_exit(retval);
}
```

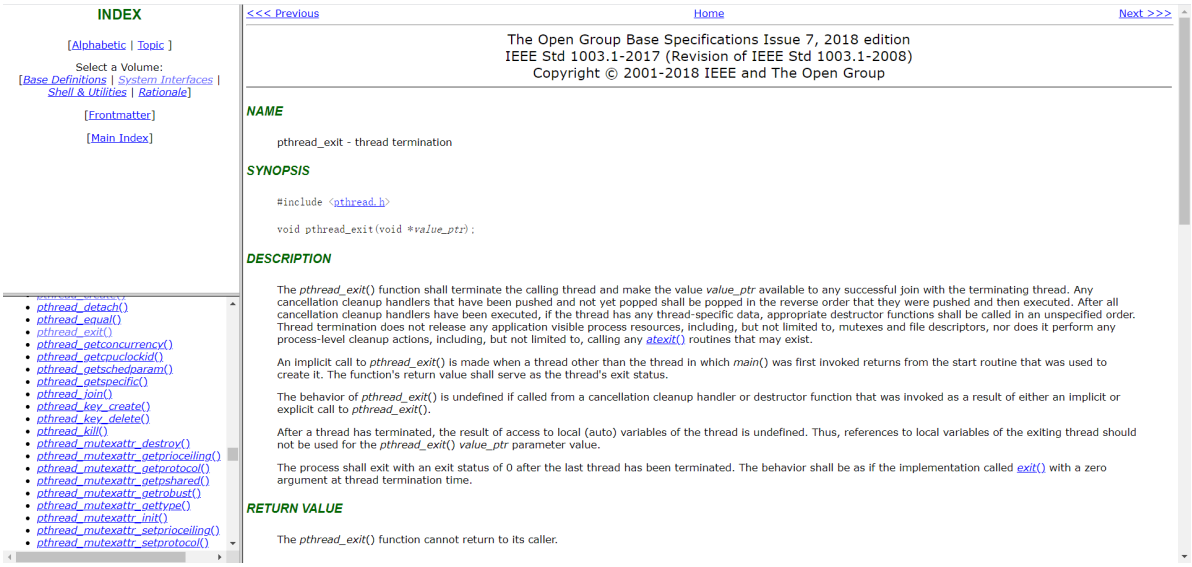
```c
/*user/fork.c*/
static void libpage(u_int newenvid, u_int pn) {
    u_int addr;
    u_int perm;
    addr = (pn << PGSHIFT);
    perm = (((Pte*)(*vpt))[pn] & 0xfff);
    u_int envid = env->env_id;
    u_int temp = UTEXT - 3 * BY2PG;
    if ((perm & PTE_V) == 0) {
        return;
    }
    if ((perm & PTE_COW) != 0) { //写时复制与共享页不能共存
        perm = perm & (~PTE_COW);
        syscall_mem_alloc(envid, temp, PTE_V | PTE_R | perm);
        user_bcopy((void *)addr, (void *)temp, BY2PG);
        syscall_mem_map(envid, temp, envid, addr, PTE_V | PTE_R | perm);
        syscall_mem_unmap(envid, temp);
    }
    if ((perm & PTE_LIBRARY) == 0) {
        perm = perm | PTE_LIBRARY;
        syscall_mem_map(envid, addr, envid ,addr, perm);
    }
    syscall_mem_map(envid, addr, newenvid, addr, perm);
}
```

```c
/*user/fork.c*/
int sfork(u_int wrapper, u_int routine, u_int arg, u_int stack) {
    u_int newenvid;
    struct Env *envchild;
    int i;
    int ret;
    set_pgfault_handler(pgfault);
    if (env->tcb_parent != NULL) {
        user_panic("sfork failed! env %x is a threadA\n", env->env_id);
    }
    newenvid = syscall_env_alloc();
    for (i = UTEXT; i < USTACKTOP; i = i + BY2PG) {
```

```
        if ((((Pte *)(*vpd))[i >> PDSHIFT] & PTE_V) && (((Pte *)(vpt))[i >>
PGSHIFT] & PTE_V)) {
            libpage(newenvid, VPN(i));
        }
    }
    envchild = envs + ENVX(newenvid);
    env->tcb_children[env->tcb_childnum] = envchild;
    env->ycb_childnum++;
    ret = syscall_mem_alloc(newenvid, UXSTACKTOP - BY2PG, PTE_V | PTE_R);
    if (ret < 0) {
        return ret;
    }
    envchild->tcb_parent = env;
    envchild->env_tf.regs[29] = stack;
    envchild->env_tf.pc = wrapper;
    envchild->env_tf.regs[4] = routine;
    envchild->env_tf.regs[5] = arg;
    envchild->env_tf.regs[6] = newenvid;
    ret = syscall_set_env_status(nesenvid, ENV_RUNNABLE);
    if (ret < 0) {
        return ret;
    }
    return newenvid;
}
```

pthread_exit

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- pthread_detach()
- pthread_equal()
- pthread_exit()
- pthread_getconcurrency()
- pthread_getcpuclockid()
- pthread_getschedparam()
- pthread_getspecific()
- pthread_join()
- pthread_key_create()
- pthread_key_delete()
- pthread_kill()
- pthread_mutexattr_destroy()
- pthread_mutexattr_getprioceiling()
- pthread_mutexattr_getprotocol()
- pthread_mutexattr_getpshared()
- pthread_mutexattr_getrobust()
- pthread_mutexattr_gettype()
- pthread_mutexattr_init()
- pthread_mutexattr_setprioceiling()
- pthread_mutexattr_setprotocol()

<<< Previous Home Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

NAME

pthread_exit - thread termination

SYNOPSIS

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

DESCRIPTION

The *pthread_exit*() function shall terminate the calling thread and make the value *value_ptr* available to any successful join with the terminating thread. Any cancellation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any *atexit*() routines that may exist.

An implicit call to *pthread_exit*() is made when a thread other than the thread in which *main*() was first invoked returns from the start routine that was used to create it. The function's return value shall serve as the thread's exit status.

The behavior of *pthread_exit*() is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to *pthread_exit*().

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the *pthread_exit*() *value_ptr* parameter value.

The process shall exit with an exit status of 0 after the last thread has been terminated. The behavior shall be as if the implementation called *exit*() with a zero argument at thread termination time.

RETURN VALUE

The *pthread_exit*() function cannot return to its caller.

共有1个参数：

- void *value_ptr：线程返回值

具体实现：

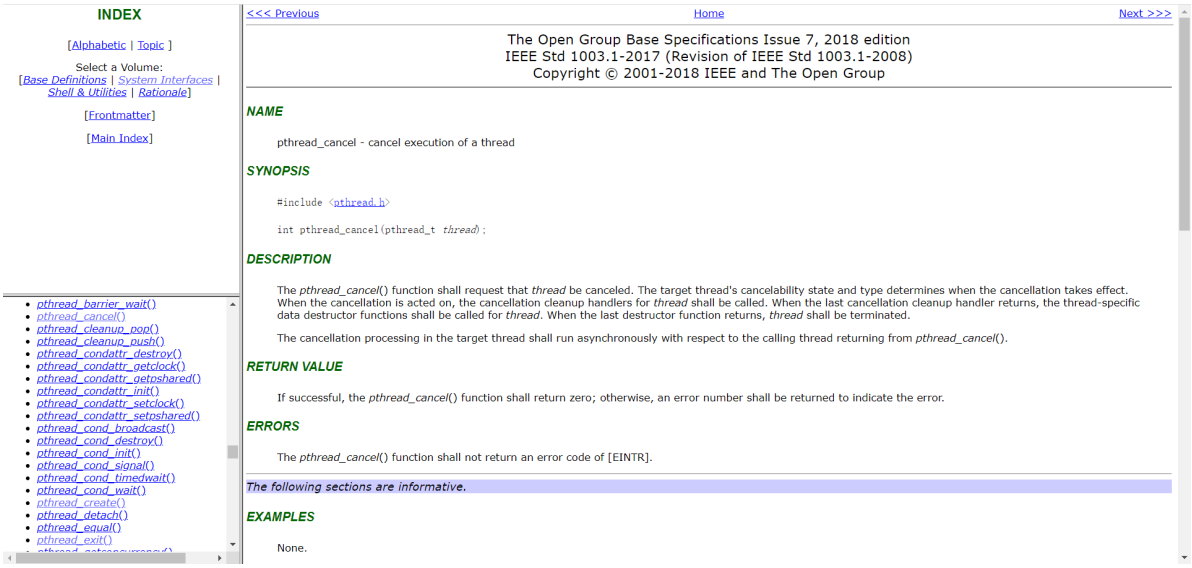
由于线程是借助进程的数据结构实现，因此只需要调用撤销进程的系统调用即可。此外，由于与pthread_cancel有相似性，因此提取出thread_exit()函数，用于将指定的线程（当前线程或其他线程）状态设置为DEAD，调用syscall_env_destroy销毁线程。

```
/*user/pthread.c*/
void pthread_exit(void *retval) {
    thread_exit(*thread, retval);
}
```

```
/*user/pthread.c*/
static void thread_exit(struct Env *e, void *retval) {
    e->retval = retval;
    e->tcb_status = 1;//1代表DEAD
    syscall_env_destroy(e->env_id);
}
```

pthread_cancel

POSIX定义如下：



共有1个参数：

- thread：撤销线程的id。

具体实现：

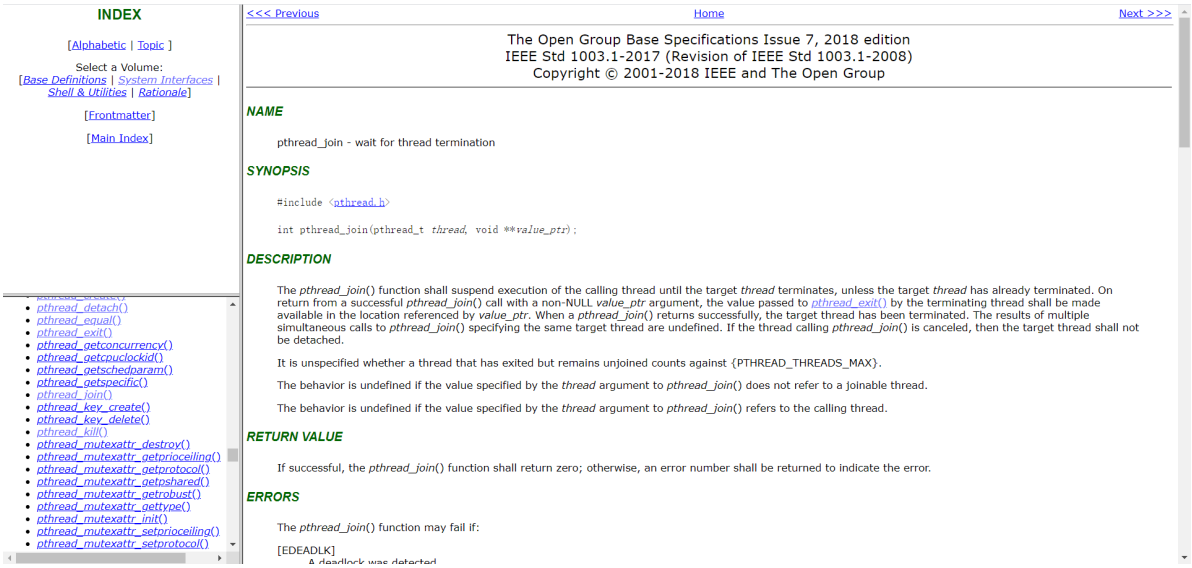
与pthread_exit相似，调用了自定义的thread_exit函数，区别是不是直接撤销自身线程，而是撤销id所代表的线程。撤销完成后返回0。

```
/*user/pthread.c*/
int pthread_cancel(pthread_t thread) {
    if(thread == env->env_id) {
        thread_exit(env, NULL);
    } else {
        int i;
        for (i = 0; i < env->tcb_childnum; i++) {
            if (thread == env->tcb_children[i]->env_id) {
                thread_exit(env->tcb_children[i], NULL);
                break;
            }
        }
    }
    return 0;
```

```
}
```

pthread_join

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- pthread_detach()
- pthread_equal()
- pthread_exit()
- pthread_getconcurrency()
- pthread_getcpuclockid()
- pthread_getschedparam()
- pthread_getspecific()
- pthread_join()
- pthread_key_create()
- pthread_key_delete()
- pthread_kill()
- pthread_mutexattr_destroy()
- pthread_mutexattr_getprioceiling()
- pthread_mutexattr_getprotocol()
- pthread_mutexattr_getpshared()
- pthread_mutexattr_getrobust()
- pthread_mutexattr_gettype()
- pthread_mutexattr_init()
- pthread_mutexattr_setprioceiling()
- pthread_mutexattr_setprotocol()

<<< Previous Home Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

NAME

pthread_join - wait for thread termination

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread*, void ***value_ptr*);

DESCRIPTION

The *pthread_join*() function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join*() call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*() by the terminating thread shall be made available in the location referenced by *value_ptr*. When a *pthread_join*() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join*() specifying the same target thread are undefined. If the thread calling *pthread_join*() is canceled, then the target thread shall not be detached.

It is unspecified whether a thread that has exited but remains unjoined counts against {PTHREAD_THREADS_MAX}.

The behavior is undefined if the value specified by the *thread* argument to *pthread_join*() does not refer to a joinable thread.

The behavior is undefined if the value specified by the *thread* argument to *pthread_join*() refers to the calling thread.

RETURN VALUE

If successful, the *pthread_join*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_join*() function may fail if:

[EDEADLK]
A deadlock was detected

共2个参数：

- pthread_t thread：要等待线程的id
- void **value_ptr：要等待线程执行结束后的返回值

具体实现：

通过while循环判断要等待线程的status是否为DEAD，若是，则将retval赋值给*value_ptr，否则，调用syscall_yield重新调度，返回循环开头再次判断。成功执行后返回0。

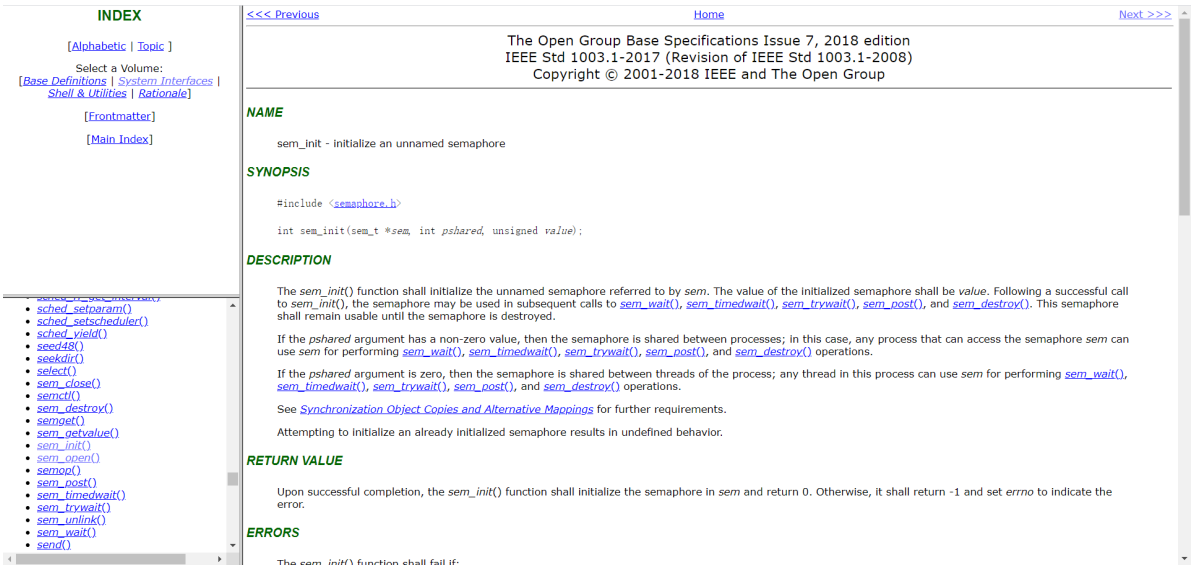
```
/*user/pthread.c*/
int pthread_join(pthread_t thread, void **value_ptr) {
    struct Env *e;
    if (thread == env->env_id) {
        e = env;
    } else {
        int i;
        for (i = 0; i < env->tcb_childnum; i++) {
            if (thread == env->tcb_children[i]->env_id) {
                e = env->tcb_children[i];
                break;
            }
        }
    }
    while(e->tcb_status == 0) {
        syscall_yield();
    }
    if(vlaue_ptr != NULL) {
        *value_ptr = e->retval;
    }
    return 0;
}
```

3.信号量

为了保证对信号量操作为原子操作，与信号量相关的函数均采用系统调用的方式实现（实验操作系统的系统调用为中断屏蔽，因此可以保证原子操作）。

sem_init

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- sched_setparam()
- sched_setscheduler()
- sched_yield()
- seed48()
- seekdir()
- select()
- sem_close()
- semctl()
- sem_destroy()
- semget()
- sem_getvalue()
- sem_init()
- sem_open()
- semop()
- sem_post()
- sem_timedwait()
- sem_trywait()
- sem_unlink()
- sem_wait()
- send()

<<< Previous | Home | Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

NAME

sem_init - initialize an unnamed semaphore

SYNOPSIS

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);

DESCRIPTION

The *sem_init*() function shall initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore shall be *value*. Following a successful call to *sem_init*(), the semaphore may be used in subsequent calls to sem_wait(), sem_timedwait(), sem_trywait(), sem_post(), and sem_destroy(). This semaphore shall remain usable until the semaphore is destroyed.

If the *pshared* argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing sem_wait(), sem_timedwait(), sem_trywait(), sem_post(), and sem_destroy() operations.

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing sem_wait(), sem_timedwait(), sem_trywait(), sem_post(), and sem_destroy() operations.

See Synchronization Object Copies and Alternative Mappings for further requirements.

Attempting to initialize an already initialized semaphore results in undefined behavior.

RETURN VALUE

Upon successful completion, the *sem_init*() function shall initialize the semaphore in *sem* and return 0. Otherwise, it shall return -1 and set *errno* to indicate the error.

ERRORS

The *sem_init*() function shall fail if:

共有3个参数：

- sem_t *sem：指向无名信号量的指针。
- int pshared：若非0，则可以在进程间共享信号量。
- unsigned value：信号量初值。

具体实现：

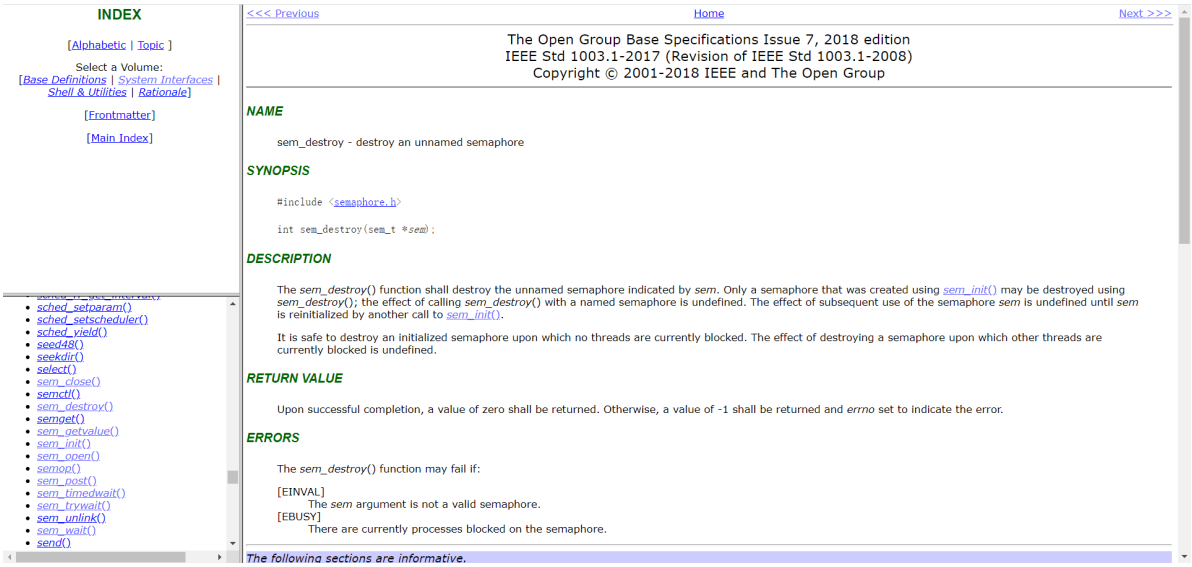
若pshared为0，即为无名信号量，只需将sem指向的无名信号量各个属性初始化即可，value赋值给count，初始化阻塞队列，pshared赋值给pshared。

若pshared非0，即为有名信号量，则从有名信号量的空间（定义为UTEXT-BY2PG）分配出一个信号量，并将sem指向的信号量的pshare指针指向分配出的信号量，再初始化该分配出的信号量，以此实现进程间共享信号量。初始化完成后，返回0。

```
/*lib/syscall_all.c*/
int sys_sem_init(int sysno, sem_t *s, int pshared, u_int value) {
    sem_t *sem;
    sem_t *sems = (sem_t *) USEM;
    if (pshared != 0) {//判断是否为有名信号量
        sem = sems->pshared++;
        s->pshared = sem;
    } else {
        sem = s;
        s->pshared = NULL;
    }
    sem->count = value;
    LIST_INIT(&sem->queue);
    return 0;
}
```

sem_destroy

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- sched_setparam()
- sched_setscheduler()
- sched_yield()
- seed48()
- seekdir()
- select()
- sem_close()
- semctl()
- sem_destroy()
- semget()
- sem_getvalue()
- sem_init()
- sem_open()
- semop()
- sem_post()
- sem_timedwait()
- sem_trywait()
- sem_unlink()
- sem_wait()
- send()

<<< Previous | Home | Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

NAME

sem_destroy - destroy an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

DESCRIPTION

The *sem_destroy*() function shall destroy the unnamed semaphore indicated by *sem*. Only a semaphore that was created using *sem_init*() may be destroyed using *sem_destroy*(); the effect of calling *sem_destroy*() with a named semaphore is undefined. The effect of subsequent use of the semaphore *sem* is undefined until *sem* is reinitialized by another call to *sem_init*().

It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of -1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sem_destroy*() function may fail if:

[EINVAL]
The *sem* argument is not a valid semaphore.
[EBUSY]
There are currently processes blocked on the semaphore.

The following sections are informative.

共有1个参数：

- sem_t *sem：指向信号量的指针。

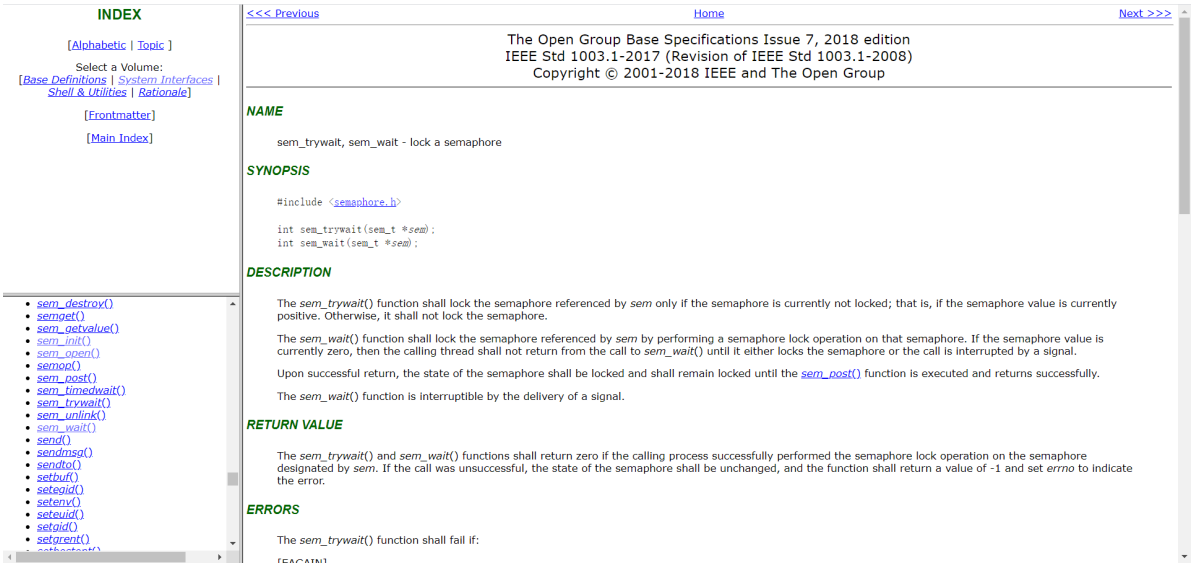
具体实现：

先判断sem指向的信号量的pshare指针是否为空，若为空，则为无名信号量，无需操作，若不为空，则为有名信号量，需要将sem指向pshare指向的信号量。通过while循环，当信号量的阻塞队列不为空时，将队首的信号量移出，重新调度，直至信号量的阻塞队列为空。销毁完成后，返回0。

```
/*lib/syscall_all.c*/
int sys_sem_destroy(int sysno, sem_t *s) {
    sem_t *sem;
    struct Env *e;
    if (s->pshared != 0) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    sem->count = 0;
    while (!LIST_EMPTY(&sem->queue)) {
        e = LIST_FIRST(&sem->queue);
        LIST_REMOVE(e, env_blocked_link);
        e->env_status = ENV_RUNNABLE;
    }
    return 0;
}
```

sem_wait

POSIX定义如下：



INDEX

[Alphabetic | Topic]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

- sem_destroy()
- semget()
- sem_getvalue()
- sem_init()
- sem_open()
- semop()
- sem_post()
- sem_timedwait()
- sem_trywait()
- sem_unlink()
- sem_wait()
- send()
- sendmsg()
- sendto()
- setbuf()
- setegid()
- setenv()
- seteuid()
- setgid()
- setgrent()

<<< Previous Home Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

NAME

sem_trywait, sem_wait - lock a semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

DESCRIPTION

The *sem_trywait*() function shall lock the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it shall not lock the semaphore.

The *sem_wait*() function shall lock the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to *sem_wait*() until it either locks the semaphore or the call is interrupted by a signal.

Upon successful return, the state of the semaphore shall be locked and shall remain locked until the sem_post() function is executed and returns successfully.

The *sem_wait*() function is interruptible by the delivery of a signal.

RETURN VALUE

The *sem_trywait*() and *sem_wait*() functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of -1 and set *errno* to indicate the error.

ERRORS

The *sem_trywait*() function shall fail if:

[EAGAIN]

共有1个参数：

- sem_t *sem：指向信号量的指针。

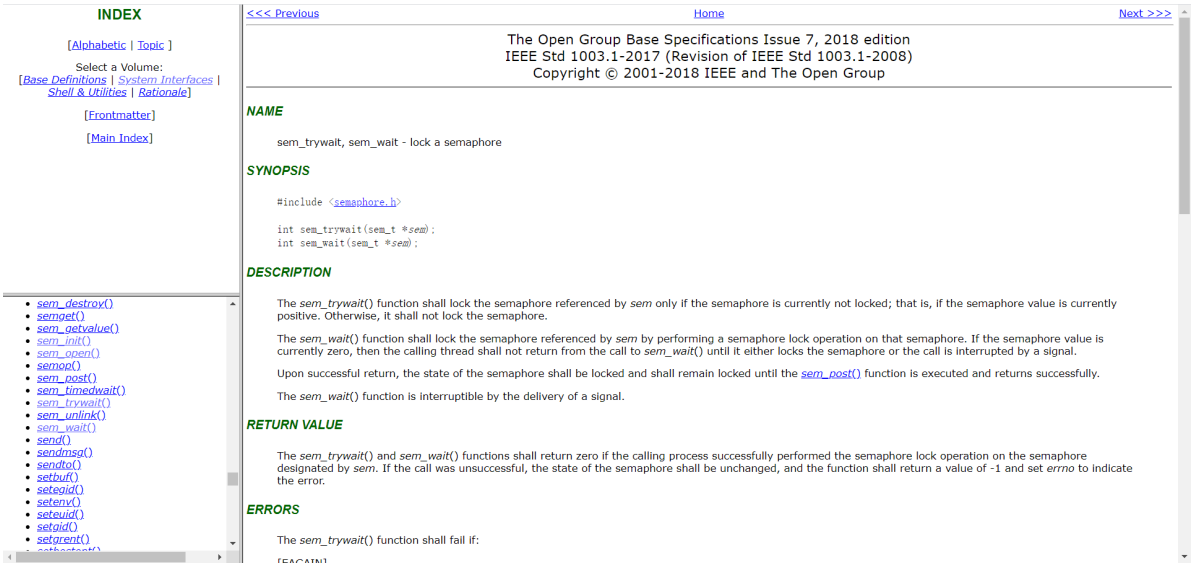
具体实现：

先判断sem指向的信号量的pshare指针是否为空，若为空，则为无名信号量，无需操作，若不为空，则为有名信号量，需要将sem指向pshare指向的信号量。将信号量的count减1，若count小于0，则将当前线程加入阻塞队列，调度状态设置为ENV_NOT_RUNNABLE（我的调度设计中阻塞进程/线程并不移出调度队列，而是在调度算法中判断是否为可调度状态），重新调度。阻塞完成后返回0。

```c
/*lib/syscall_all.c*/
int sys_sem_wait(int sysno, sem_t *s) {
    sem_t *sem;
    if (s->pshared != 0) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    sem->count--;
    if (sem->count < 0) {
        LIST_INSERT_TAIL(&sem->queue, curenv, env_blocked_link);
        curenv->env_status = ENV_NOT_RUNNABLE;
        sys_yield();
    }
    return 0;
}
```

sem_trywait

POSIX定义如下：



共有1个参数：

- sem_t *sem：指向信号量的指针。

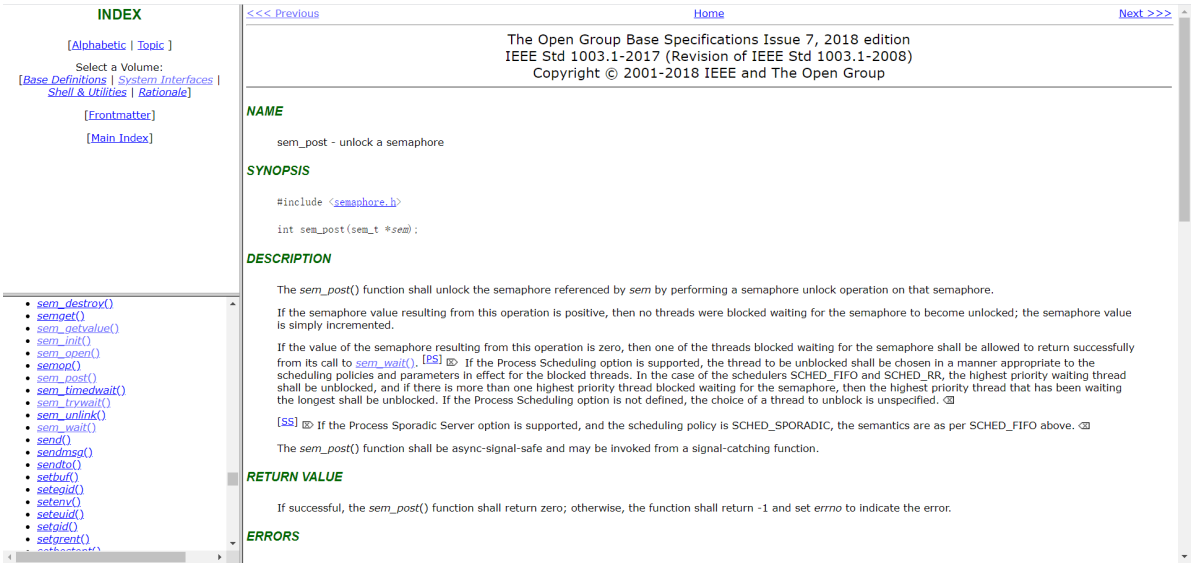
具体实现：

同sem_trywait，区别是减1后若count小于0，并不阻塞线程，而是返回错误码-1。

```
/*lib/syscall_all.c*/
int sys_sem_trywait(int sysno, sem_t *s) {
    sem_t *sem;
    if (s->pshared != 0) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    if (sem->count <= 0) {
        return -1;
    }
    sem->count--;
    return 0;
}
```

sem_post

POSIX定义如下：



共有1个参数：

- sem_t *sem：指向信号量的指针。

具体实现：

先判断sem指向的信号量的pshare指针是否为空，若为空，则为无名信号量，无需操作，若不为空，则为有名信号量，需要将sem指向pshare指向的信号量。将信号量的count加1，若count小于等于0，则取出阻塞队列的队首线程，调度状态设置为ENV_RUNNABLE，重新调度。完成后返回0。

```
/*lib/syscall_all.c*/
int sys_sem_post(int sysno, sem_t *s) {
    sem_t *sem;
    if (s->pshared != 0) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    if (sem->count++ < 0) {
        e = LIST_FIRST(&sem->queue);
        LIST_REMOVE(e, env_blocked_link);
        e->env_status = ENV_RUNNABLE;
    }
    return 0;
}
```

sem_getvalue

POSIX定义如下：

共有2个参数：

sem_t *restrict sem：指向信号量的指针（restrict：只读，不改变状态）。

int *restrict sval：信号量的值。

具体实现：

先判断sem指向的信号量的pshare指针是否为空，若为空，则为无名信号量，无需操作，若不为空，则为有名信号量，需要将sem指向pshare指向的信号量。将信号量的count赋值给sval即可，完成后返回0。

```
/*lib/syscall_all.c*/
int sys_sem_getvalue(int sysno, sem_t *s, int *sval) {
    sem_t *sem;
    if (s->pshared != 0) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    *sval = sem->count;
    return 0;
}
```

三、重要机制

1.线程共享内存

与进程各自占据独立的内存空间不同，同一个线程需要共享地址空间。进程占据独立内存空间的实现机制是在fork时通过duppage设置PTE_COW，进而在修改页面时进行写时复制，而线程的创建则是由pthread_create调用sfork实现的，因此在sfork，不能像fork那样通过duppage设置PTE_COW，而是将需要共享的地址空间（UTEXT~USTACKTOP）设置为PTE_LIBRARY，以此来实现共享地址空间。值得注意的是，如果进程/线程在执行过程中出现了缺页中断，如果按照原有的缺页中断处理机制进行处理，进程的其他线程不会映射到替换后的页面，而是仍然映射之前的页面，因此，如果需要修改现有的缺页中断处理机制，我的做法是在env.c中增加env_library_page函数，当需要共享的地址空间（UTEXT~USTACKTOP）在出现缺页中断时，执行该函数，使得该进程的所有线程映射为替换后的页面，并设置为PTE_LIBRARY。具体实现如下：

```
/*lib/env.c*/
void env_library_page(u_int va) {
    struct Env *e;
    if (curenv->tcb_parent == NULL && curenv->tcb_childnum == 0) {
        return;
    } else if (curenv->tcb_parent == NULL){
        e = curenv;
    } else {
        e = curenv->tcb_parent;
    }
    Pte *ppte = NULL;
    struct Page *ppage = NULL;
    u_int perm = PTE_V | PTE_R | PTE_LIBRARY;
    int i,r;
    ppage = page_lookup(curenv->env_pgdir, va, &ppte);
    r = page_insert(e->env_pgdir, ppage, va, perm);
    if (r < 0){
        panic("env_library_page - page insert error!\n");
    }
    for (i = 0; i < e->tcb_childnum; i++) {
        if((e->tcb_children[i])->tcb_status == 0) {
            ppage = page_lookup(curenv->env_pgdir, va, &ppte);
            r = page_insert((e->tcb_children[i])->env_pgdir, ppage, va, perm);
            if (r < 0) {
                panic("env_library_page - page insert error!\n");
           }
        }
    }
}
```

2.有名信号量

由于实现了线程共享内存，因此无名信号量的实现相对容易，只需将信号量的指针传递给不同线程执行的函数即可。而进程间由于内存空间独立，因此需要在创建进程时留出特定的空间（定义为UTEXT - BY2PG）使有名信号量可以在进程间共享，并设置第一个有名信号量的pshare指针进行计数，当有新进程创建时，分配出新的有名信号量，当进程内初始化有名信号量时，通过第一个有名信号量的pshare指针获取，并将该信号量的pshare指针指向获取的共享信号量。在执行信号量相关的系统调用时，先判断当前信号量的pshare指针是否指向了共享信号量（是否为NULL），若指向了共享信号量，则说明该信号量是共享信号量，需要对共享信号量进行操作。相关代码如下：

```
/*include/mmu.h*/
/*定义有名信号量的地址空间*/
//...
#define UTHREAD (UTEXT - BY2PG)
//...
```

```c
/*user/libos.c*/
/*创建进程时为有名信号量分配空间*/
void libmain(int argc, char **argv) {
    //...
    int ret;
    ret = syscall_mem_alloc(0, USEM, PTE_V | PTE_R | PTE_LIBRARY);
    if (ret < 0) {
        user_panic("libmain - syscall_mem_alloc error!\n");
    }
    sem_t *sems = (sem_t *)USEM;
    sems->pshared = sems + 1;
    //...
}
```

```c
/*lib/syscall_all.c*/
/*执行信号量的系统调用时判断是否为有名信号量*/
int sys_sem_init(int sysno, sem_t *s, int pshared, u_int value) {
    sem_t *sem;
    sem_t *sems = (sem_t *)USEM;
    if (pshared != 0) {
        sem = sems->pshared;
        sems->pshared++;
        s->pshared = sem;
    } else {
        sem = s;
        s->pshared = NULL;
    }
    //...
}
int sys_sem_xxx(int sysno, sem_t s...) {
    sem_t *sem;
    sem_t *sem = (sem_t *)USEM;
    if (s->pshared != NULL) {
        sem = (sem_t *)s->pshared;
    } else {
        sem = s;
    }
    //...
}
```

四、测试

本次实验的测试方法采用自行编写能通过逻辑约束打印不同输出的测试程序进行测试，通过断言（user_assert）、大循环模拟特定的线程执行顺序等技巧，根据所打印的输出判断功能的正确性。为了方便，将不同函数的测试通过不同的函数进行测试，并集合到一个文件中。共有6个测试函数，以及对有名信号量的简单测试：

pthread_create_test

测试线程创建是否正确。

```c
static void *pthread_create_test_routine1(void *arg) {
        user_assert(arg == 190616);
        writef("pthread_create_test_routine1 passed!\n");
        return NULL;
}

static void *pthread_create_test_routine2(void *arg) {
        user_assert(arg == 19373);
        writef("pthread_create_test_routine2 passed!\n");
        return NULL;
}

static void *pthread_create_test_routine3(void *arg) {
        user_assert(arg == 135);
        writef("pthread_create_test_routine3 passed!\n");
        return NULL;
}

static void pthread_create_test() {
        pthread_t t1, t2, t3;
        pthread_create(&t1, NULL, pthread_create_test_routine1, 190616);
        pthread_create(&t2, NULL, pthread_create_test_routine2, 19373);
        pthread_create(&t3, NULL, pthread_create_test_routine3, 135);
        writef("pthread_create test passed!\n");
        return;
}
```

测试逻辑：创建多个线程并传递不同的参数，在线程内通过user_assert和writef判断线程是否正确执行以及传递参数是否正确。

正确结果：创建的三个线程user_assert通过，并输出相应内容。

```
hello challenge!

------------------------pthread_create_test begin------------------------

pthread_create_test_routine1 passed!

pthread_create_test_routine2 passed!

pthread_create_test_routine3 passed!

pthread_create test passed!

------------------------pthread_create_test end------------------------
```

pthread_exit_test

测试线程退出是否正确。

```c
static void *pthread_exit_test_routine(void *arg) {
        pthread_exit((void *) 2021);
        user_panic("exit fail!\n");
}

static void pthread_exit_test() {
        pthread_t t1;
        pthread_create(&t1, NULL, pthread_exit_test_routine, NULL);
        void *retval;
        pthread_join(t1, &retval);
        user_assert(retval == (void *) 2021);
        writef("pthread_exit test passed!\n");
        return;
}
```

测试逻辑：创建线程后执行pthread_exit，通过其后的user_panic语句判断线程是否正常退出，并在主线程中通过user_assert判断线程的返回值是否正确。

正确结果：线程执行到pthread_exit后即退出，不会执行user_panic语句。

```
------------------------pthread_exit_test begin------------------------

pthread_exit test passed!

------------------------pthread_exit_test end------------------------
```

pthread_cancel_test

测试线程撤销是否正确。

```
45 static void *pthread_cancel_test_routine(void *arg) {
46         writef("this is thread\n");
47         int i = 0;
48         while(i < 500000) {
49                 i++;
50         }
51         user_panic("cancel failed!\n");
52         return NULL;
53 }
54
55 static void pthread_cancel_test() {
56         pthread_t t1;
57         pthread_create(&t1, NULL, pthread_cancel_test_routine, NULL);
58         writef("this is main\n");
59         int i = 0;
60         while(i < 50000) {
61                 i++;
62         }
63         pthread_cancel(t1);
64         while(i < 500000) {
65                 i++;
66         }
67         writef("pthread_cancel test passed!\n");
68         return;
69 }
```

测试逻辑：主线程创建线程后，执行pthread_cancel，同时创建的线程中通过user_panic判断线程是否及时撤销。此外，为防止创建的线程先于主线程调度导致在主线程执行pthread_cancel前就执行了user_panic，分别在主线程和创建的线程中执行不同次数的循环，以保证主线程能够执行pthread_cancel。

正确结果：主线程输出“this is main”，创建的线程输出"this is thread"，主线程执行pthread_cancel后，创建的线程退出，不会执行user_panic。

```
------------------------pthread_cancel_test begin------------------------

this is main

this is thread

pthread_cancel test passed!

------------------------pthread_cancel_test end------------------------
```

pthread_join_test

测试等待线程结束是否正确。

```
71 static void *pthread_join_test_routine(void *arg) {
72         int i = 0;
73         while(i < 50000) {
74                 i++;
75         }
76         return 1952;
77 }
78
79 static void pthread_join_test() {
80         pthread_t t1;
81         void *retval;
82         pthread_create(&t1, NULL, pthread_join_test_routine, NULL);
83         pthread_join(t1, &retval);
84         user_assert(retval == (void *) 1952);
85         writef("pthread_join test passed!\n");
86         return;
87 }
```

测试逻辑：主线程创建线程后，线程内执行大循环，使得在不同步的情况下主线程先于线程执行，从而导致判断创建线程返回值的user_assert失败，增加pthread_join后，主线程等待线程执行结束后再执行，使得返回值判断的user_assert通过。

正确结果：user_assert通过，不会提前终止。

```
------------------------pthread_join_test begin------------------------

pthread_join test passed!

------------------------pthread_join_test end------------------------
```

sem_trywait_test

测试对信号量P操作（非阻塞）和V操作以及获取信号量的值是否正确。

```
89 static void *sem_trywait_test_routine1(void *arg) {
90         sem_t *s = (sem_t *)arg;
91         int value;
92         sem_getvalue(s, &value);
93         user_assert(value == 0);
94         sem_post(s);
95         sem_getvalue(s, &value);
96         user_assert(value == 1);
97         value = sem_trywait(s);
98         user_assert(value == 0);
99         sem_getvalue(s, &value);
100        user_assert(value == 0);
101        value = sem_trywait(s);
102        user_assert(value == -1);
103        sem_getvalue(s, &value);
104        user_assert(value == 0);
105        sem_post(s);
106        sem_getvalue(s, &value);
107        user_assert(value == 1);
108        return NULL;
109 }
```

```
111 static void *sem_trywait_test_routine2(void *arg) {
112         sem_t *s = (sem_t *)arg;
113         int value;
114         sem_post(s);
115         sem_getvalue(s, &value);
116         user_assert(value == 2);
117         value = sem_trywait(s);
118         user_assert(value == 0);
119         sem_getvalue(s, &value);
120         user_assert(value == 1);
121         return NULL;
122 }
123
124 static void sem_trywait_test() {
125         sem_t s;
126         pthread_t t1, t2;
127         sem_init(&s, 0, 0);
128         pthread_create(&t1, NULL, sem_trywait_test_routine1, &s);
129         pthread_join(t1, NULL);
130         pthread_create(&t2, NULL, sem_trywait_test_routine2, &s);
131         pthread_join(t2, NULL);
132         writef("sem_trywait test passed!\n");
133         return;
134 }
```

测试逻辑：创建两个线程，传入同一个信号量，依此进行PV操作，并用sem_getvalue获得信号量的值，通过user_assert判断PV操作后的值是否正确。

正确结果：user_assert通过，不会提前终止。

```
------------------------sem_trywait_test begin------------------------

sem_trywait test passed!

------------------------sem_trywait_test end------------------------
```

sem_wait_test

测试对信号量的P操作（阻塞）是否正确。测试线程共享内存。

```
136 static void *sem_wait_test_routine1(void *arg) {
137         int i =1;
138         int *another = (int *) 0x7cbfdfd0;
139         sem_t *s = (sem_t *) arg;
140         sem_post(s);
141         sem_wait(s+1);
142         writef("thread1: %d at %x, thread2: %d at %x\n", i, &i, *another, another);
143         sem_post(s);
144         sem_wait(s+1);
145         sem_post(s+2);
146         writef("thread1 finished!\n");
147         return NULL;
148 }
149
150 static void *sem_wait_test_routine2(void *arg) {
151         int i = 2;
152         int *another = (int *) 0x7cffdfd0;
153         sem_t *s = (sem_t *) arg;
154         sem_post(s+1);
155         sem_wait(s);
156         writef("thread2: %d at %x, thread1: %d at %x\n", i, &i, *another, another);
157         sem_post(s+1);
158         sem_wait(s);
159         sem_post(s+2);
160         writef("thread2 finished!\n");
161         return NULL;
162 }
164 static void sem_wait_test() {
165         sem_t s[3];
166         sem_init(&s[0], 0, 0);
167         sem_init(&s[1],0, 0);
168         sem_init(&s[2], 0, 0);
169         pthread_t t1, t2;
170         pthread_create(&t1, NULL, sem_wait_test_routine1, s);
171         pthread_create(&t2, NULL, sem_wait_test_routine2, s);
172         sem_wait(&s[2]);
173         writef("sem_wait test passed!\n");
174         return;
175 }
```

测试逻辑：两个线程中各自定义一个整型变量，并赋予不同的值，在正常情况下，线程共享内存，两个线程应该可以通过地址（地址是提前打印而得知的），访问到另一个定义的整型变量，因此可以打印出另一个线程定义的整型变量的值。主线程创建两个线程后，分别向两个线程传递了3个信号量。前两个用于同步两个线程，防止两个线程调度不一致而导致出现一个线程已经执行打印语句，而另一个线程还未定义整型变量的情况，第三个专门用来测试信号量的PV操作是否正确。

正确结果：两个线程可以正常打印自身线程和另一个线程定义的整型变量的值和地址，说明线程内存共享，且同步正常。主线程和两个线程最终正常执行而未被阻塞，说明PV操作正常。

```
------------------------sem_wait_test begin------------------------

thread2: 2 at 7cbfdfd0, thread1: 1 at 7cffdfd0

thread1: 1 at 7cffdfd0, thread2: 2 at 7cbfdfd0

thread1 finished!

thread2 finished!

sem_wait test passed!

------------------------sem_wait_test end------------------------
```

sem_destroy_test

测试销毁信号量是否正确。

```
177 static void *sem_destroy_test_routine1(void *arg) {
178         sem_t *s = (sem_t *) arg;
179         writef("thread1 blocked\n");
180         sem_wait(s);
181         writef("thread1 finished\n");
182         return NULL;
183 }
184
185 static void *sem_destroy_test_routine2(void *arg) {
186         sem_t *s = (sem_t *) arg;
187         writef("thread2 blocked\n");
188         sem_wait(s);
189         writef("thread2 finished\n");
190         return NULL;
191 }
192
193 static void sem_destroy_test() {
194         sem_t s;
195         sem_init(&s, 0, 0);
196         pthread_t t1, t2;
197         pthread_create(&t1, NULL, sem_destroy_test_routine1, &s);
198         pthread_create(&t2, NULL, sem_destroy_test_routine2, &s);
199         int i = 0;
200         while (i < 500000) {
201                 i++;
202         }
203         writef("thread1 and thread2 are blocked!\n");
204         sem_destroy(&s);
205         pthread_join(t1, NULL);
206         pthread_join(t2, NULL);
207         writef("sem_destroy test passed!\n");
```

测试逻辑：设置一个信号量，创建两个线程，使这两个线程阻塞后，调用sem_destroy销毁信号量，并等待两个线程执行结束，若两个线程正常结束，则销毁信号量正确。

正确结果：信号量销毁。

```
------------------------sem_destroy_test begin------------------------

thread1 blocked

thread2 blocked

thread1 and thread2 are blocked!

thread2 finished

thread1 finished

sem_destroy test passed!

------------------------sem_destroy_test end------------------------
```

name sem test

有名信号量的简单测试。

```
231         sem_t s;
232         writef("------------------------name semaphore test begin------------------------\n");
233         sem_init(&s, 1, 1);
234         if (fork() == 0) {
235                 sem_wait(&s);
236                 writef("this is child env\n");
237         } else {
238                 int i = 0;
239                 while(i < 5000000) {
240                         i++;
241                 }
242                 sem_wait(&s);
243                 writef("this is father env\n");
244         }
245         writef("------------------------name semaphore test end------------------------\n");
246         writef("------------------------congratulations!-----------------------\n");
```

测试逻辑：在当前进程中先创建一个值为1的有名信号量，然后执行fork创建子进程，父子进程分别执行P操作，由于信号量值为1，因此后执行的进程会被阻塞，只有一个进程会正常输出。这里通过大循环使得子进程先于父进程执行。

正确结果：子进程正常执行，而父进程被阻塞。

```
------------------------name semaphore test begin------------------------

this is child env

------------------------name semaphore test end------------------------
```

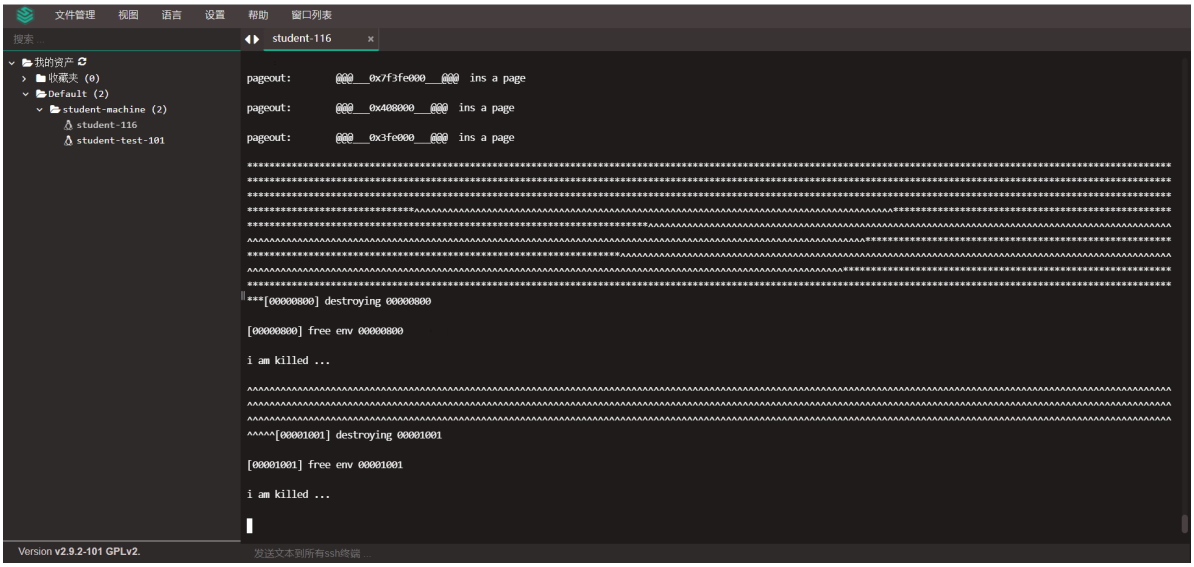
五、附加内容

Lab5的挑战性任务中有一个是解决writef被打断的问题，在本次实验中由于测试需要大量用到writef，而writef被打断会导致难以判断打印结果的正确性，因此需要解决这个问题。

其实通过已经实现的有名信号量即可实现，测试程序如下：

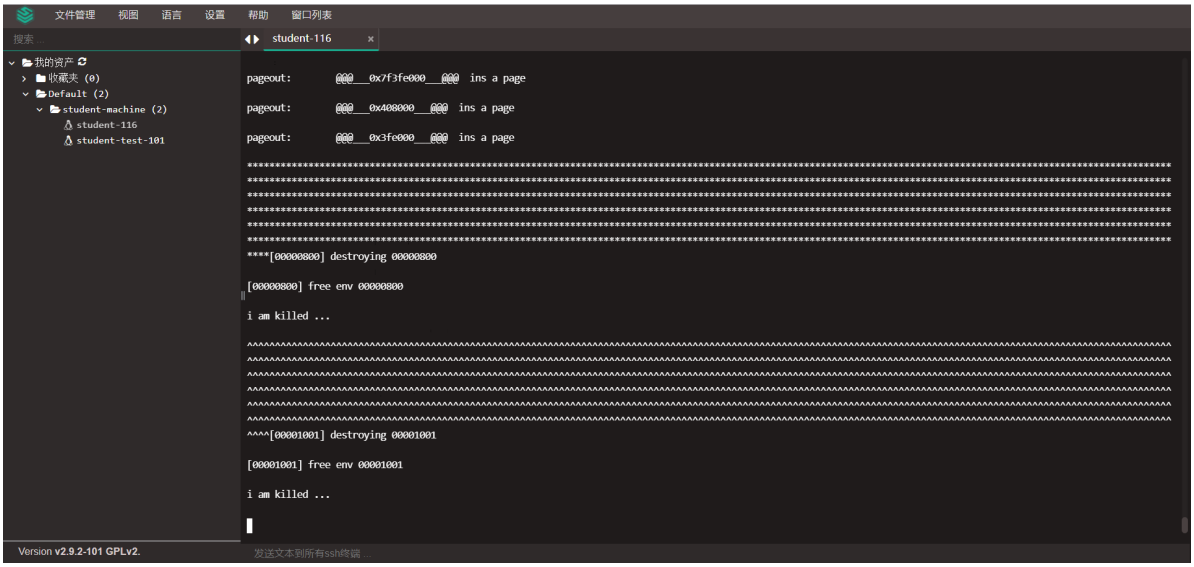
```c
#include "lib.h"

void umain() {
        sem_t s;
        sem_init(&s, 1, 1);
        if (fork() == 0) {
                char c1[1000];
                int i = 0;
                while(i < 1000) {
                        c1[i]= '^';
                        i++;
                }
                //sem_wait(&s);
                writef("%s", c1);
                sem_post(&s);
        } else {
                char c2[1000];
                int j = 0;
                while(j < 1000) {
                        c2[j]= '*';
                        j++;
                }
                //sem_wait(&s);
                writef("%s", c2);
                sem_post(&s);
        }
}
```

将sem_wait注释掉意味着信号量不起作用，此时的执行结果如下：



可见，由于进程的调度，出现了writef被打断的情况。

而加上sem_wait()，使信号量起作用，从而为writef加锁，此时的执行结果如下：



可见，此时两个进程的wirtef分别执行，不会出现被打断的情况，有名信号量很好地解决了这个问题。

六、待改进

1.进程模拟线程的不足

用进程模拟线程固然是一个方便的实现，但使用进程控制块的线程也会有以下几个问题：

- 线程的id与进程id的产生方式相同，无法通过id区分线程还是进程。
- 进程的线程通过静态数组的形式存储，一是浪费空间，二是无法实现动态增长。

通过增加单独的线程控制块线程与进程之间的区别会更加明显，但这样也需要修改大量的代码。

2.有名信号量的不足

本实验中实现的有名信号量仅仅是实现了进程间的信号量共享，还不是真正意义上的“有名”，还有一些数据结构以及POSIX定义的函数有待实现。

七、参考资料

- The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)

  （https://download.csdn.net/download/stupid_boy2007/10702011）
- The Single UNIX® Specification, Version 2 （https://pubs.opengroup.org/onlinepubs/7908799/index.html）
- POSIX线程相关博客（https://blog.csdn.net/weixin_40039738/article/details/81143956）
- POSIX信号量相关博客（https://blog.csdn.net/tennysonsky/article/details/46496201）
- Linux源码分析（https://github.com/theanarkh/read-linux-0.11）
- 理论课课件