Operating System Labs Project 3

Project 3a: Locks and Threads

Part 0 概述：

该部分涉及较多文件，首先对其用途一个阐述:

```
counter.h       counter.c     ->   计数器源文件
hash.h          hash.c        ->   哈希表源文件
list.h          list.c        ->     链表源文件
lock.h          lock.c        ->   锁函数源文件
muetex.h        mutex.c       ->   互斥锁源文件
spinlock.h      spinlock.c    ->   自旋锁源文件
xchg.c          sys_futex.c   ->        系统调用
main            main.c        ->        测试文件
main.sh                       ->        测试脚本
*.so                          ->   生成的链接库
testout.txt                   ->        测试结果
```

Part1 锁：

本实验一共涉及三个锁，操作系统自身的 pthread_mutex_lock，以及自己撰写的 mutex,spinlock。在 lock.h 中有如下宏定义：

```
#define P_MUTEX_LOCK    1
#define MY_SPIN_LOCK    2
#define MY_MUTEX_LOCK   3

#define LOCK_TYPE       1
```

因为涉及三个锁，所以在测试的时候难免需要进行大量的 if 从句，即

```
if test is spin_lock
   then acquire struct's spinlock
else
   acquire struct's mutex
```

整体下来，从撰写到测试，都有大量这样的冗余语句，而且 list 与 hash 结构体中都需要存储四种锁以及指针，大大增加了结构体所占字节，因此以 list 为例子，我们这样定义结构体：

```
typedef struct __list_t
{
    node_t *head;
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_t lock;
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_t lock;
```

```
            #else
                mutex_t lock;
            #endif
        }list_t;
```

通过这样的定义方式，链表自始至终内部都只有一种锁，并且都命名为 lock，在后续的测试以及调用中，可以免去 if 的枚举。当我们需要更换测试锁的类型时，可以通过 shell 中的如下语句：

```
sed -i 's/#define LOCK_TYPE      1/#define LOCK_TYPE      2/g' lock.h
```

来将 lock.h 中的宏定义进行替换，然后 touch main.c 来重新构建。后续我们将看到，这样的定义方式可以省去大量的代码。

同样，对于 lock.c 的部分我们也通过这种宏定义方式：

```
void lock_init(void* lock)
{
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_init((pthread_mutex_t*) lock, NULL);
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_init((spinlock_t*)lock);
    #else
        mutex_init((mutex_t*)lock);
    #endif
}

void lock_acquire(void *lock)
{
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_lock((pthread_mutex_t*)lock);
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_acquire((spinlock_t*)lock);
    #else
        mutex_acquire((mutex_t*)lock);
    #endif
}

void lock_release(void *lock)
{
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_unlock((pthread_mutex_t*)lock);
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_release((spinlock_t*)lock);
    #else
        mutex_release((mutex_t*)lock);
    #endif
}
```

Part2 自旋锁与互斥锁：

1.自旋锁

自旋锁的内部结构体之后一个 flag 来表示锁的状态:

```
typedef struct __spinlock_t {
    flag;
} spinlock_t;
```

而对于自旋锁的函数，也是简单的修改 flag 和系统调用 xchg

```
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}

void spinlock_acquire(spinlock_t *lock) {
    while (xchg(&lock->flag, 1) == 1);
}

void spinlock_release(spinlock_t *lock) {
    lock->flag = 0;
}
```

2. 互斥锁

互斥锁在定义上与自旋锁无差别

```
typedef struct __mutex_t {
    int flag;
} mutex_t;
```

但是在实现上，多了对 sys_futex 的调用

```
void mutex_init(mutex_t *lock) {
    lock->flag = 0;
}

void mutex_acquire(mutex_t *lock)
{
    if (xchg(&lock->flag, 1) == 0) return;
    else
    {
        sys_futex(&lock->flag, FUTEX_WAIT, 1, 0, 0 ,0
);
        mutex_acquire(lock);
    }
```

```
}

void mutex_release(mutex_t *lock)
{
    xchg(&lock->flag, 0);
    sys_futex(&lock->flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

Part3 测试方法：

我们将在 Part4-Part6 给出三种数据结构的定义与测试结果，因此将在给出测试结果之前先阐述我们的测试方式。我们的测试部分设计三个文件：

main.c main.sh testout.txt

我们在 main.c 中定义了基本的测试方式：

根据命令行参数确定测试何种数据结构，根据 lock.h 中的 LOCK_TYPE 值确定调用何种锁，然后分别在三种操作数：1w,10w,100w 的数量级下，枚举线程数 1-20，通过 rand 的方式随机调用各种命令，然后记录用时。

单个 main.c 仅能完成一种锁，以及一种数据结构的测试，而我们有三种锁和三种数据结构，因此需要配合 main.sh 脚本来完成测试：

```
#!/usr/bin/bash
echo "ECNU OS Lab3a Result" > testout.txt
#提示信息
echo -e "Test Start...\n"
echo "Pthread Mutex Lock is testing..."
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
make
#修改命令行参数，下同
./main 1 >> testout.txt
./main 2 >> testout.txt
./main 3 >> testout.txt
echo -e "Pthread Mutex Lock finished!\n\n"
#修改 LOCK_TYPE 的值，下同
sed -
i 's/#define LOCK_TYPE      1/#define LOCK_TYPE      2/g' lock.h
touch main.c
echo "My Spin Lock is testing..."
make
./main 1 >> testout.txt
./main 2 >> testout.txt
./main 3 >> testout.txt
echo -e "My Spin Lock finished!\n\n"
```

```
sed -
i 's/#define LOCK_TYPE      2/#define LOCK_TYPE      3/g' lock.h
#通过 touch 来使 make 重新构建 main.c，否则 LOCK_TYPE 的修改不会应用到 main
touch main.c
echo "My Mutex Lock is testing..."
make
./main 1 >> testout.txt
./main 2 >> testout.txt
./main 3 >> testout.txt
echo -e "My Mutex Lock finished!\n\n"

sed -
i 's/#define LOCK_TYPE      3/#define LOCK_TYPE      1/g' lock.h
touch main.c
echo -e "All Finished!\n"
echo -e "Please read testout.txt to get test result!\n"
```

测试的结果将被放在 testout.txt 中，有如下样式:

```
----------------Pthread Mutex Lock Test-------------


/*operation times :   10000*/
threads number :  1   list test time    : 0.001322
threads number :  2   list test time    : 0.002174
threads number :  3   list test time    : 0.003649
threads number :  4   list test time    : 0.011089
threads number :  5   list test time    : 0.014327
threads number :  6   list test time    : 0.020812
threads number :  7   list test time    : 0.016215
threads number :  8   list test time    : 0.021721
threads number :  9   list test time    : 0.022127
threads number : 10   list test time    : 0.024213
threads number : 11   list test time    : 0.027122
threads number : 12   list test time    : 0.028135
threads number : 13   list test time    : 0.025842
threads number : 14   list test time    : 0.028120
threads number : 15   list test time    : 0.029462
threads number : 16   list test time    : 0.031137
threads number : 17   list test time    : 0.030213
threads number : 18   list test time    : 0.034126
threads number : 19   list test time    : 0.039626
threads number : 20   list test time    : 0.040809
```

我们通过 python 中的 matplotlib 将其可视化

最后给出我们测试的源代码：

```
#include <pthread.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>

#include "lock.h"
#include "hash.h"
#include "list.h"
#include "counter.h"

struct timeval timer;
//pthread_t *p;
double begin,end;
counter_t *counter;
list_t *list;
hash_t *hash;
int operation,threads;
pthread_t p[20];
void* counter_test(void);
void* list_test(void);
void* hash_test(void);
int main(int argc,char** argv)
{
    int test_type = atoi(argv[1]);
    srand((unsigned)time(0));
    if(LOCK_TYPE == 1)
        printf("\n------------Pthread Mutex Lock Test---------\n");
    else if(LOCK_TYPE == 2)
        printf("\n------------My Spin Lock Test---------------\n");
    else
        printf("\n------------My Mutex Lock Test--------------\n");
    int i;
    if(test_type == 1)
    {
        for(operation = (int)1e4;operation <= 1e6;operation *= 10)
        {
            printf("\n\n/*operation times : %7d*/\n",operation);
            for(threads = 1;threads <= 20;threads++)
            {
                printf("threads number : %2d   ",threads);
```

```
                gettimeofday(&timer,NULL);
                begin = timer.tv_sec+timer.tv_usec/1e6;
                counter = (counter_t*)malloc(sizeof(counter_t));
                counter_init(counter,0);
                //p = (pthread_t*)malloc(sizeof(pthread_t)*threads);
                for(i = 0;i < threads;i++)
                    pthread_create(&p[i],NULL,(void*)counter_test,NU
LL);
                for(i = 0;i < threads;i++)
                    pthread_join(p[i],NULL);
                //free(p);
                counter_clear(counter);
                gettimeofday(&timer,NULL);
                end = timer.tv_sec+timer.tv_usec/1e6;
                printf("counter test time : %.6f\n",end-begin);
            }
        }
    }

    else if(test_type == 2)
    {
        for(operation = (int)1e4;operation <= 1e6;operation *= 10)
        {
            printf("\n\n/*operation times : %7d*/\n",operation);
            for(threads = 1;threads <= 10;threads++)
            {
                printf("threads number : %2d   ",threads);
                gettimeofday(&timer,NULL);
                begin = timer.tv_sec+timer.tv_usec/1e6;

                list = (list_t*)malloc(sizeof(list_t));
                list_init(list);
                //p = (pthread_t*)malloc(sizeof(pthread_t)*threads);
                for(i = 0;i < threads;i++)
                    pthread_create(&p[i],NULL,(void*)list_test,NULL)
;
                for(i = 0;i < threads;i++)
                    pthread_join(p[i],NULL);
                //free(p);
                list_clear(list);
                gettimeofday(&timer,NULL);
                end = timer.tv_sec+timer.tv_usec/1e6;
                printf("list test time    : %.6f\n",end-begin);
            }
```

```
        }
    }

    else if(test_type == 3)
    {
        for(operation = (int)1e4;operation <= 1e6;operation *= 10)
        {
            printf("\n\n/*operation times : %7d*/\n",operation);
            printf("threads number : %d   ",threads);
            for(threads = 1;threads <= 10;threads++)
            {
                printf("threads number : %2d   ",threads);
                gettimeofday(&timer,NULL);
                begin = timer.tv_sec+timer.tv_usec/1e6;

                hash = (hash_t*)malloc(sizeof(hash_t));
                hash_init(hash,20);
                //p = (pthread_t*)malloc(sizeof(pthread_t)*threads);
                for(i = 0;i < threads;i++)
                    pthread_create(&p[i],NULL,(void*)hash_test,NULL)
;
                for(i = 0;i < threads;i++)
                    pthread_join(p[i],NULL);
                //free(p);
                hash_clear(hash);
                gettimeofday(&timer,NULL);
                end = timer.tv_sec+timer.tv_usec/1e6;
                printf("hash test time    : %.6f\n",end-begin);
            }
        }
    }

    return 0;
}
void* counter_test(void)
{
    int i;
    for(i = 0;i < operation;i++)
    {
        int cmd = rand()%3;
        if(cmd == 0) counter_get_value(counter);
        else if(cmd == 1) counter_increment(counter);
        else counter_decrement(counter);
    }
```

```
    return NULL;
}

void* list_test(void)
{
    int i;
    for(i = 0;i < operation;i++)
    {
        int cmd = rand()%3;
        if(cmd == 0) list_insert(list,i%1000);
        else if(cmd == 1) list_delete(list,i%1000);
        else list_lookup(list,i%1000);
    }
    return NULL;
}

void* hash_test(void)
{
    int i;
    for(i = 0;i < operation;i++)
    {
        int cmd = rand()%3;
        if(cmd == 0) hash_insert(hash,i%1000);
        else if(cmd == 1) hash_delete(hash,i%1000);
        else hash_lookup(hash,i%1000);
    }
    return NULL;
}
```

Part4 计数器：

计数器的定义采用了我们在 Part1 中提到的方式

counter.h

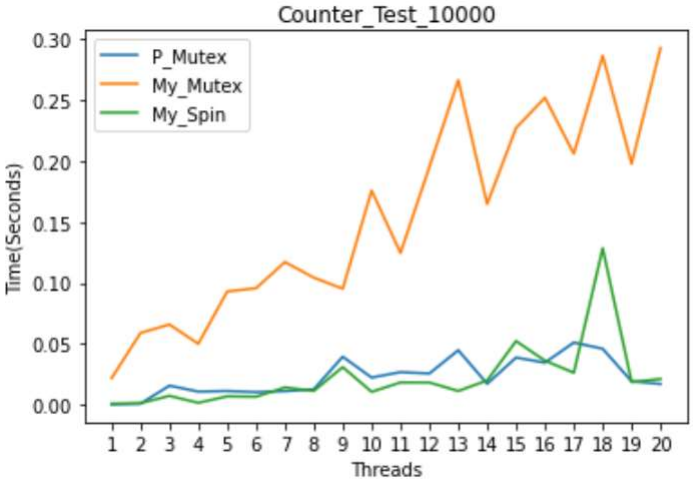
```
#ifndef _COUNTER_H
#define _COUNTER_H

#include "lock.h"

typedef struct __counter_t {
    int val;
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_t lock;
```

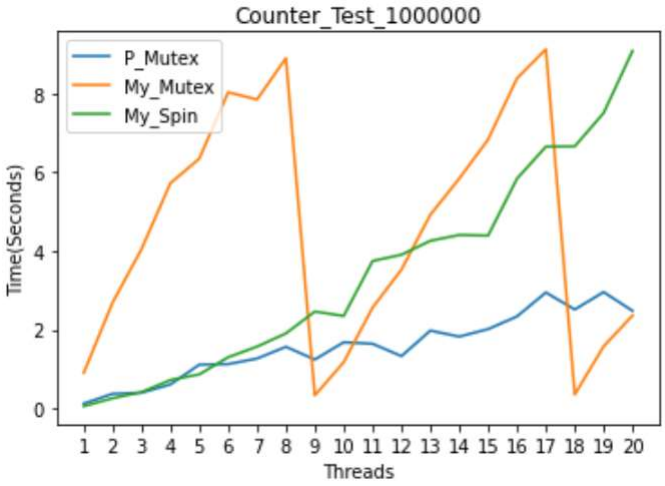
```
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_t lock;
    #else
        mutex_t lock;
    #endif
}counter_t;

void counter_init(counter_t *c, int val);
int counter_get_value(counter_t *c);
void counter_increment(counter_t *c);
void counter_decrement(counter_t *c);

//for testing
void counter_clear(counter_t* c);
#endif
```

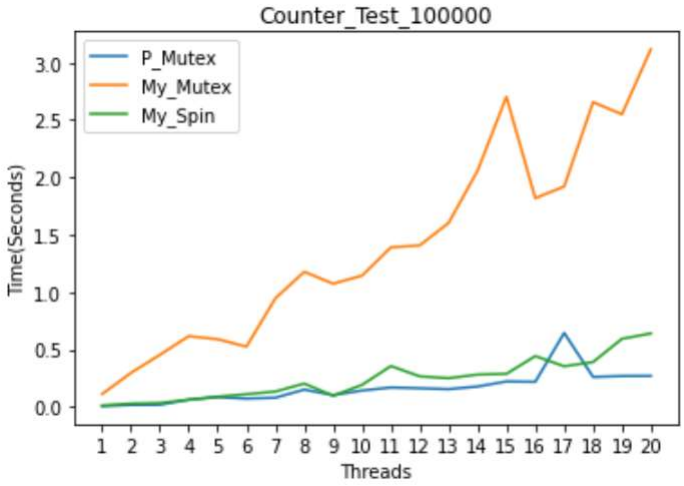
counter.c

```
#include "lock.h"
#include "counter.h"

#define LOCK (void*)&c->lock
void counter_init(counter_t *c, int val)
{
    c->val = val;
    lock_init(LOCK);
}

int counter_get_value(counter_t *c)
{
    lock_acquire(LOCK);
    int result = c->val;
    lock_release(LOCK);
    return result;
}

void counter_increment(counter_t *c)
{
    lock_acquire(LOCK);
    c->val++;
    lock_release(LOCK);
}

void counter_decrement(counter_t *c)
{
```

```
    lock_acquire(LOCK);
    c->val--;
    lock_release(LOCK);
}

void counter_clear(counter_t* c)
{
    free(c);
}
```







Part5 链表：

list.h

```
#ifndef __LIST_H_
#define __LIST_H_

#include "lock.h"

typedef struct __node_t
{
    unsigned int val;
    struct __node_t *next;
}node_t;

typedef struct __list_t
{
    node_t *head;
    #if LOCK_TYPE == P_MUTEX_LOCK
        pthread_mutex_t lock;
    #elif LOCK_TYPE == MY_SPIN_LOCK
        spinlock_t lock;
    #else
        mutex_t lock;
    #endif
}list_t;

void list_init(list_t *list);
void list_insert(list_t *list, unsigned int key);
```

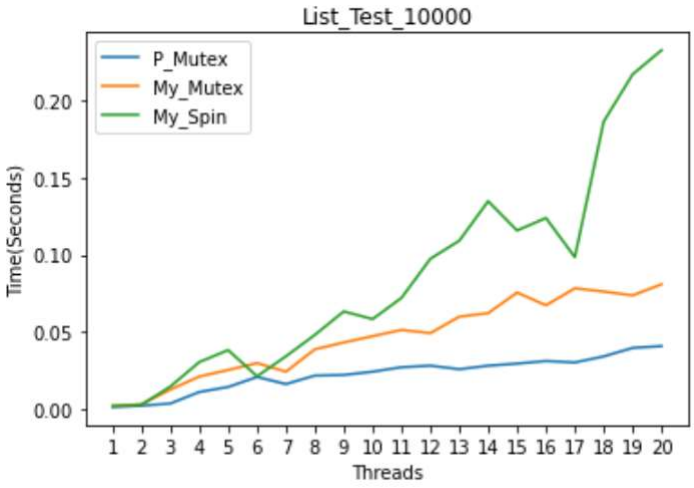
```
void list_delete(list_t *list, unsigned int key);
void *list_lookup(list_t *list, unsigned int key);
void list_clear(list_t *list);

//for testing && debugging
int list_size(list_t *list);
void list_print(list_t *list);
#endif
```

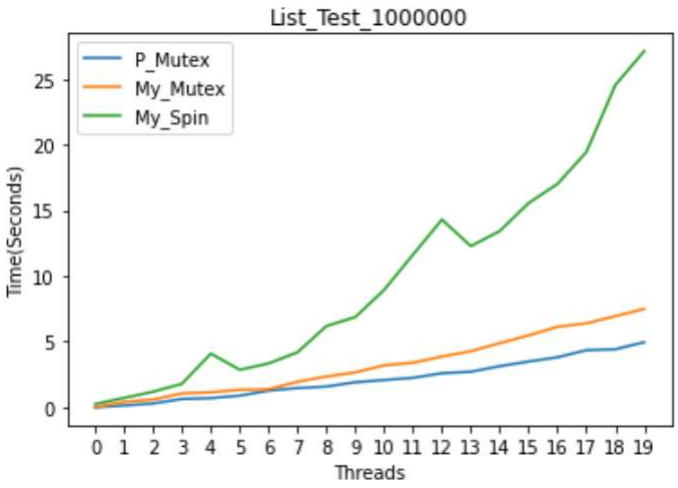
list.c

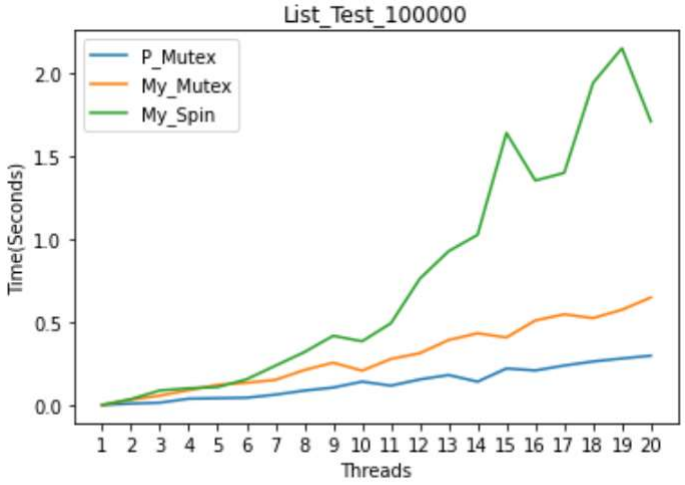
```
#include "list.h"
#include "lock.h"

#define LOCK (void*)&list->lock

void list_init(list_t *list)
{
    list->head == NULL;
    lock_init(LOCK);
}

void list_insert(list_t *list, unsigned int key)
{
    lock_acquire(LOCK);
    node_t* node = (node_t*)malloc(sizeof(node_t));
    node->val = key;
    node->next = list->head;
    list->head = node;
    lock_release(LOCK);
}

void list_delete(list_t *list, unsigned int key)
{
    if(list->head == NULL) return ; //none list
    lock_acquire(LOCK);
    node_t *pre = NULL,*node = list->head;
    while(node != NULL)
    {
        if(node->val == key) break;
        pre = node;
        node = node->next;
    }
    if(node == NULL) return ; //not exist
```

```
    if(pre == NULL) list->head = node->next; //head is t
arget
    else pre->next = node->next; //normal
    free(node);
}

void *list_lookup(list_t *list, unsigned int key)
{
    lock_acquire(LOCK);
    node_t *node = list->head;
    while(node != NULL)
    {
        if(node->val == key) break;
        node = node->next;
    }
    lock_release(LOCK);
    return node;
}

void list_clear(list_t *list)
{
    node_t *node = list->head;
    node_t *p;
    while(node != NULL)
    {
        p = node;
        node = node->next;
        free(p);
    }
}

int list_size(list_t *list)
{
    int size = 0;
    node_t *node = list->head;
    while(node != NULL)
    {
        size++;
        node = node->next;
    }
    return size;
}

void list_print(list_t *list)
```

```
{
    printf("List:\n");
    node_t *node = list->head;
    int index = 0;
    while(node != NULL)
    {
        printf("node %d: %d\n",index++,node->val);
        node = node->next;
    }
}
```







Part6 哈希表：

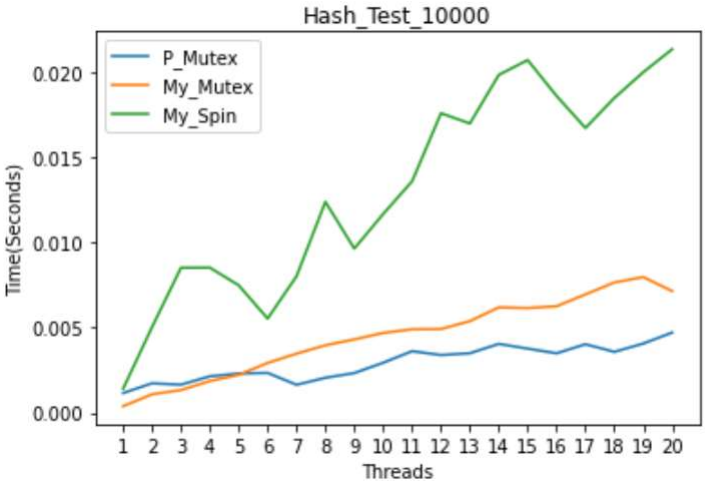
hash.h

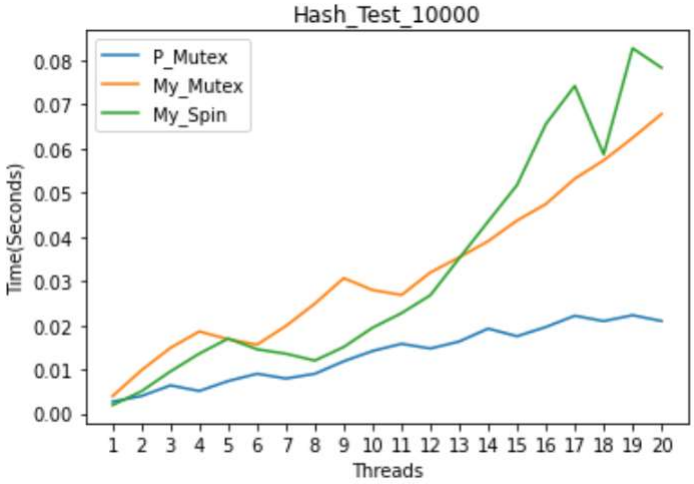
```
#ifndef _HASH_H
#define _HASH_H

#include "lock.h"
#include "list.h"

typedef struct __hash_t {
    list_t base[50];
    int size;
} hash_t;

void hash_init(hash_t *hash, int size);
void hash_insert(hash_t *hash, unsigned int key);
void hash_delete(hash_t *hash, unsigned int key);
void *hash_lookup(hash_t *hash, unsigned int key);

//for testing & debugging
int hash_total_size(hash_t *hash);
void hash_clear(hash_t *hash);


#endif
```

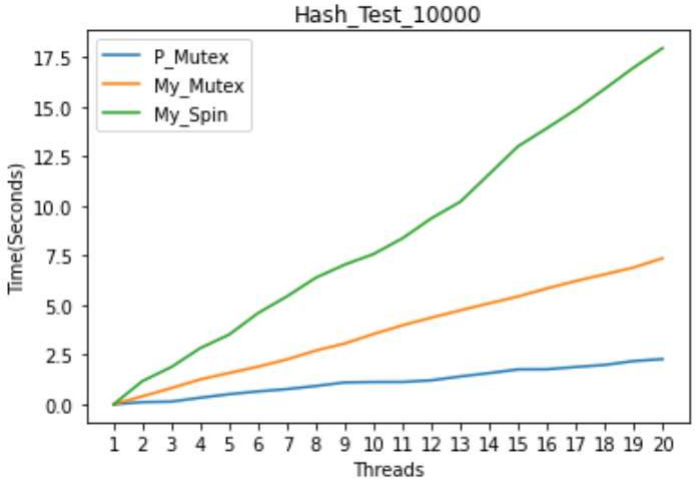
hash.c

```
#include "lock.h"
```

```c
#include "list.h"
#include "hash.h"

void hash_init(hash_t *hash, int size)
{
    hash->size = size;
    int i;
    for (i = 0; i < size; i++)
        list_init(&hash->base[i]);
}

void hash_insert(hash_t *hash, unsigned int key) {
    int index = key%hash->size;
    list_insert(&hash->base[index], key);
}

void hash_delete(hash_t *hash, unsigned int key) {
    int index = key%hash->size;
    list_delete(&hash->base[index], key);
}

void *hash_lookup(hash_t *hash, unsigned int key) {
    int index = key%hash->size;
    return list_lookup(&hash->base[index], key);
}

int hash_total_size(hash_t *hash)
{
    int i, result = 0;
    for (i = 0; i < hash->size; i++)
        result += list_size(&hash->base[i]);
    return result;
}

void hash_clear(hash_t *hash)
{
    int i;
    for (i = 0; i < hash->size; ++i)
        list_clear(&hash->base[i]);
}
```


Hash_Test_10000
P_Mutex
My_Mutex
My_Spin
Time(Seconds)
0.020
0.015
0.010
0.005
0.000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Threads


Hash_Test_10000
P_Mutex
My_Mutex
My_Spin
Time(Seconds)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Threads


Hash_Test_10000
P_Mutex
My_Mutex
My_Spin
Time(Seconds)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Threads

Part7 结论：

1.整体上看，但线程足够多，操作数足够大时，耗时有关系:

My_Spin > My_Mutex > P_Mutex

2.对于每种锁而言，其都有不稳定性，即其耗时并不会随着线程增加而增加，除了有时会减少以外，还经常出现较大跳动，最明显的为 My_Spin，他在 Counter_Test_1000000 的测试中出现了难以解释的跳跃。

3.对于每种数据结构而言，List 和 Hash 在数据量大时是三条趋于稳定上涨的曲线，而 Counter 测试即使在大数据量也无法稳定。

4.P_Mutex 作为系统调用，每次使用都需要调用 CPU，所以即使他的效率优异，但是在线程数少，操作数小的时候，其在 CPU 上的耗时使得其优势无法体现，但是在大数据量下，其速度远优于其他两种锁。

5.My_Spin 是作为简单的锁，在所有测试下，他几乎都有着最大的耗时，初次之后，其稳定性极差。

6.My_Mutex 额外加入了 sys_futex 的使用，其效果是显著的，虽然耗时不能与 P_Mutex 相比，但是好于 My_Spin 并且有着与 P_Mutex 一样的优异的稳定性。

Project 3b: xv6 VM Layout

Part 0 涉及文件:

1. xv6/kernel/defs.h
2. xv6/kernel/proc.h
3. xv6/kernel/exe.c
4. xv6/kernel/makefile.mk
5. xv6/kernel/proc.c
6. xv6/kernel/syscall.c
7. xv6/kernel/vm.c
8. xv6/kernel/trap.c
9. xv6/user/makefile.mk

Part A: Null-pointer Deference：

在该部分我们需要做两个工作：
(1)为空指针添加引用
(2)设置 code 段的起始地址为 0x2000

1.添加空指针的引用

因为要求有些抽象，因此我们从测试文件本身入手，发现在 null.c 与 null2.c 当中，进行了如下指针的引用：

```
uint * nullp = (uint*)0;
uint * badp  = (uint*)(4096+1);
```

这两个指针分别访问了 0x0 与 0x1001 地址，分别对应了 null 与 unmapped pages，因此我们应该对这样的指针引用进行异常处理。

继续分析 null.c 中的代码，发现他是在 fork 后的子进程中来进行空指针的检验的，于是在 kernel/proc.c 中找到 fork 函数。fork 函数的流程如下：
(1)调用 allocproc 来分配一个进程
(2)调用 copyuvm 函数来复制父进程的页表
(3)若以上均成功，那么将父进程的全部信息复制给子进程，并重设寄存器 eax 的值，使得子进程调用 fork 时，返回的是 0

注意到 copyuvm 函数是 fork 中的关键部分，他的实现在 kernel/vm.c 文件当中，而函数的流程如下：
(1)调用 setupkvm 来获取页表目录
(2)从 0 开始枚举到 sz，以 PGSIZE 为步长枚举每一个页
(3)在每一个页中，通过 walkpgdir 函数将一个页中的虚拟地址映射为物理地址，然后通过 mappages 初始化页表项:计算起始地址和结束地址，

然后在起始地址与结束地址之间进行虚拟地址到物理地址的转换。

也就是说整个流程是 fork->copyuvm->mappages->walkpdir，在整个流程的执行过程中，并没有找到与地址检验相关的部分，然后我们将目光放到了参数本身。每个系统调用都有不同的参数，那么在内核中的系统调用函数又是如何找到这些参数？参数或许是库函数在陷入内核前压栈的，所以可以根据 trapframe 中的用户栈 esp 来找到各种参数，xv6 使用了工具函数 argint、argptr 和 argstr 来获得第 n 个系统调用参数。

在 syscall.c 中，我们找到了这三个函数的定义，他们分别用于获取整数，指针和字符串起始地址。argint 利用用户空间的 %esp 寄存器定位第 n 个参数:%esp 指向系统调用结束后的返回地址。参数就恰好在%esp 之上(%esp+4)。因此第 n 个参数就在%esp+4+4*n。

argint 调用 fetchint 从用户内存地址读取值到 *ip。fetchint 可以简单地将这个地址直接转换成一个指针，因为用户和内核共享同一个页表，但是内核必须检验这个指针的确指向的是用户内存空间的一部分。内核已经设置好了页表来保证本进程无法访问它的私有地址以外的内存：如果一个用户尝试读或者写高 p->sz 的地址，处理器会产生一个段中断，这个中断会杀死此进程，正如我们之前所见。但是现在，我们在内核态中执行，用户提供的任何地址都是有权访问的，因此必须要检查这个地址是在 p->sz 之下的。

argptr 和 argint 的目标是相似的：它解析第 n 个系统调用参数。argptr 调用 argint 来把第 n 个参数当做是整数来获取，然后把这个整数看做指针，检查它的确指向的是用户地址空间。注意 argptr 的源码中有两次检查。首先，用户的栈指针在获取参数的时候被检查。然后这个获取到得参数作为用户指针又经过了一次检查。

argstr 是最后一个用于获取系统调用参数的函数。它将第 n 个系统调用参数解析为指针。它确保这个指针是一个 NUL 结尾的字符串并且整个完整的字符串都在用户地址空间中。

所以我们要做的工作就很清晰了：修改 fetchint，fetchstr 和 argptr 函数，在这三个函数中，遇到了异常都会进行通过 return -1 来杀死进程，所以我们要判断的就是：

(1)是否小于 0x2000

(2)是否超过上界

(3)是否在栈与堆的中间间隔，该部分与 Stack Rearrangement 相关因此在该 part 先未列出，在 B 部分会补充上。

```
int
fetchint(struct proc *p, uint addr, int *ip)
```

```
{
    if(addr >= p->sz || (addr+4) > p->sz)
        return -1;
   if(addr >= USERTOP || (addr+4) > USERTOP || addr < 0x2000)
        return -1;

    *ip = *(int*)(addr);
    return 0;
}


int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;
  if(addr >= USERTOP || addr < 0x2000)
    return -1;
  if(addr >= p->sz)
    return -1;
  *pp = (char*)addr;
  if(addr < p->sz) ep = (char*)p->sz;
  else ep = (char*)USERTOP;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}

int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= USERTOP || (uint)i+size > USERTOP
                        || (uint)i < 0x2000)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size > proc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

2.设置 code 段起始地址为 0x2000

解决了该部分后，我们还需要设置 code 的起始地址段为 0x2000，该部分直接修改数字即可，但需要修改多处：

(1)kernel/makefile.mk: line123

```
--entry=start --section-start=.text=0x2000 \
```

该部分为 initcode 中的段起始地址，故需修改。

(2)kernel/exec.c:

```
sz = 0x2000;
```

位于 exec 函数，程序加载起始地址，也应修改。

(3)kernel/proc.c:

```
p->sz = PGSIZE + 0x2000;
```

位于 userintit 函数，表示进程的起始的末尾地址，因为起始端变为 0x2000，故结束地址也应由 PGSIZE 变为 PGSIZE+0x2000

```
p->tf->eip = 0x2000;
```

位于 userintit 函数 eip 寄存器，是 CPU 取址寄存器，其值应为 code 起始地址，也就是 0x2000

(4)kernel/vm.c:

```
mappages(pgdir,(void*)0x2000,PGSIZE,PADDR(mem),PTE_W|PTE_U);
```

mappages 函数，其第二个参数表示地址映射时的起始地址，所以需要变更为 0x2000

```
for(i = 0x2000; i < sz; i += PGSIZE)
```

该循环位于 copyuvm 函数，表示复制父进程页表的过程，循环的初始值应为起始地址 0x2000

(5)user/makefile.mk: line78

```
USER_LDFLAGS += --section-start=.text=0x2000
```

表示代码段的起始地址，应修改为 0x2000

至此，我们便完成了 A 部分的工作。

Part B: Stack Rearrangement：

1. 拓展属性与函数

在重排工作中，结构由

```
USERTOP = 640KB
(free)
heap (grows towards the high-end of the address space)
stack (fixed-sized, one page)
code
(2 unmapped pages)
ADDR = 0x0
```

变为

```
USERTOP = 640KB
stack (at end of address space; grows backwards)
... (gap >= 5 pages)
heap (grows towards the high-end of the address space)
code
(2 unmapped pages)
ADDR = 0x0
```

注意到变化有:stack 与 heap 的位置交换了,此外最重要的是,stack 从 fixed-sized 变为可变的，所以首先我们需要像记录 heap 的 size 一样，记录下 stack 的 size，因此在 proc.h 中的 struct proc 中，新增属性 SizeOfStack,即

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  uint sizeOfStack;            // Size of Stack
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

而对于 sizeOfStack 属性的初始化，更新可以完全参照 sz 属性，也就是说 sz 怎么设置的，我们就怎么设置，只不过注意方向与 sz 相反，边界是 USERTOP 即可。所以与 growproc 相对应，我们需要设置一个 growstack 函数。在 kernel/vm.c 的 copyuvm 函数中，也同样需要新增 sizeOfStack 参数。这些修改在 kernel/def.h 中：

```
int             growstack(struct proc*);
```

```
pde_t*          copyuvm(pde_t*, uint, uint);
```

2. sizeOfStack 的初始化

(1) kernel/proc.c @userintit(void)

p 为初始化的第一个进程，所以在这里为 sizeOfStack 初始化：

```
p->sizeOfStack = 0;
```

此外，eps 寄存器存放当前线程的栈顶指针，而现在的栈顶是 p->sz 而非 PGSIZE 了，所以也需要做修改。

```
p->tf->esp = p->sz;
```

(2) kernel/proc.c @fork(void)

在 fork 函数中，需要为新创建的子进程复制 sizeOfStack 属性

```
np->stk_sz = proc->stk_sz;
```

3. sizeOfStack 的应用

(1) kernel/exec.c @exec(char*,char**)

在该函数中，有如下代码段

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + PGSIZE)) == 0)
    goto bad;
sp = sz;
```

该段代码的意义为，首先对 sz 取整为 PGSIZE 的整数倍，即对齐；然后在 sz 到 sz+PGSIZE 范围内分配一页栈，最后更新堆栈指针 sp。

而现在，栈是自顶向下的，其顶端地址为 USERTOP，所以 allocuvm 内的参数应修改为 USERTOP-PGSIZE 到 USERTOP，sp 也应与新的返回值一致，所以修改为：

```
uint new_sz;
if((new_sz=allocuvm(pgdir,USERTOP-PGSIZE, USERTOP)) == 0)
    goto bad;
new_sz = PGROUNDUP(new_sz);
sp = new_sz;
```

(2) kernel/proc.c @ growproc(int n)

该 Part 还有另一个要求，即 gap >= 5 pages，也就是说在 stack 与 heap 之间要预留 5 个页的大小，我们通过在 growproc 函数中新增一个判断条件：如果在 grow 之后栈顶与堆顶的间距小于 5 个页，那么 return -1 来阻止这个 grow。

```
int
growproc(int n)
{
  uint sz;
  sz = proc->sz;
  if (sz+n+5*PGSIZE > USERTOP-PGSIZE*proc->sizeOfStack)
```

```
    return -1;
  if(n > 0)
  {
    if((sz = allocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  else if(n < 0)
  {
    if((sz = deallocuvm(proc->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  proc->sz = sz;
  switchuvm(proc);
  return 0;
}
```

(3) kernel/proc.c @ growstack(struct proc *p)
该函数为新增函数，用于增长 stack，其结构与 growproc 对称，首先计算增长之后间距是否大于等于 5 个页，然后调用 allocuvm 来分配页，如果分配成功，那么 sizeOfStack 进行加一，并切换到该进程。

```
int
growstack(struct proc *p)
{
  uint sz;
  sz = p->sizeOfStack;
  if(USERTOP-sz*PGSIZE-PGSIZE-p->sz < 5*PGSIZE)
    return -1;
  if((sz = allocuvm(p->pgdir, USERTOP-sz*PGSIZE-
PGSIZE, USERTOP - sz*PGSIZE)) == 0)
    return -1;
  p->sizeOfStack++;
  switchuvm(p);
  return 0;
}
```

(4) kernel/syscall.c @fetchint,@fetchstr,@argptr
上文提到了这些函数可以对非法地址通过 return -1 的方式处理，其中有语句：

```
        if(addr >= p->sz || addr + 4) > p->sz)
            return -1;
```

但是现在这个语句不一定成立了，因为 addr >= p->sz，还有可能是在 stack 区域，所以需要额外新增

```
        addr < (USERTOP - p->sizeOfStack*PGSIZE)
```

需要注意的是在 fetchstr 还需要新增对 ep 寄存器的分支，因为 ep 既可能是 stack 的顶端，也可能是 heap 的。

```
            if(addr < p->sz)
                ep = (char*)p->sz;
            else
                ep = (char*)USERTOP;
```

修改后的三个函数源码如下：

```
int
fetchint(struct proc *p, uint addr, int *ip)
{
  if((addr >= p->sz || (addr+4) > p->sz)
  && addr < (USERTOP-p->sizeOfStack*PGSIZE))
    return -1;
  if(addr >= USERTOP || (addr+4) > USERTOP || addr < 0x2000)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}

int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;
  if(addr >= USERTOP || addr < 0x2000)
    return -1;
  if(addr >= p->sz && addr < (USERTOP - p->sizeOfStack*PGSIZE))
    return -1;
  *pp = (char*)addr;
  if(addr < p->sz) ep = (char*)p->sz;
  else ep = (char*)USERTOP;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}

int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
```

```
    return -1;
  if((uint)i >= USERTOP || (uint)i+size > USERTOP
                        || (uint)i < 0x2000)
    return -1;
  if(((uint)i >= proc->sz || (uint)i+size > proc->sz) && (uint)i
 < (USERTOP - proc->sizeOfStack*PGSIZE))
    return -1;
  *pp = (char*)i;
  return 0;
}
```

(5)kernel/vm.c@copyuvm(pde_t *pgdir,uint sz,uint sizeOfStack)
该部分较为简单，因为新增了 stack，所以在 fork 调用 copyuvm 的时候，也需要把父进程的 stack 段复制给子进程，复制过程与 heap 段对称，仅需修改循环的始末地址即可。

```
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint stk_sz)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0x2000; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void*)i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(d, (void*)i,PGSIZE, PADDR(mem),PTE_W|PTE_U) < 0)
      goto bad;
  }

  for(i = PGROUNDUP(USERTOP - (sizeOfStack * PGSIZE)); i < USERT
OP; i += PGSIZE)
{
    if((pte = walkpgdir(pgdir, (void*)i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
```

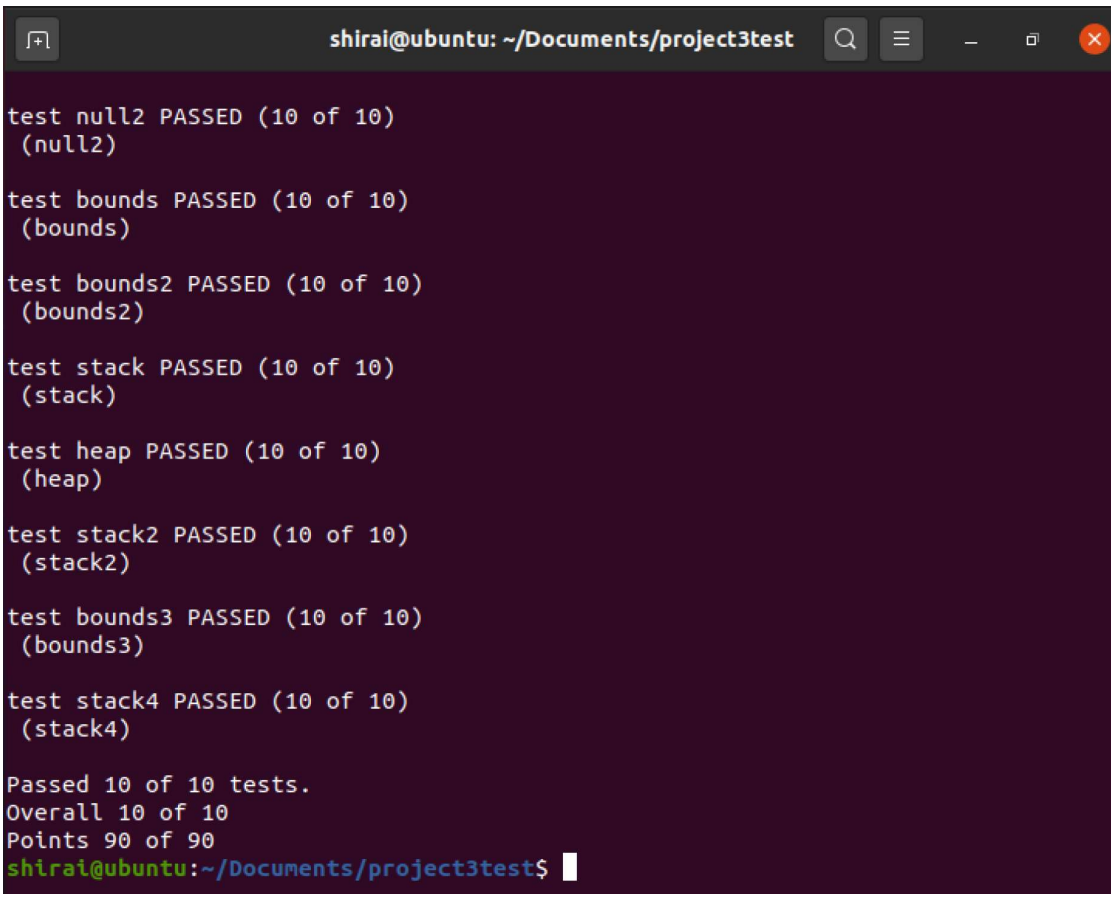
```
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(d, (void*)i, PGSIZE,PADDR(mem),PTE_W|PTE_U) < 0)
      goto bad;
  }
  return d;
bad:
  freevm(d);
  return 0;
}
```

(6)kernel/trap.c@ trap(struct trapframe *tf)

trap 为中断处理函数，因为栈的大小是可变的，因此当栈的空间不足时，tf->trapno 会被置为 T_PGFLT，我们需要对该 case 进行处理，方法就是调用之前撰写好的 growstack 函数，如果返回 0，说明扩增成功，break 结束，否则进入 default 来进行异常处理。

```
case T_PGFLT:
    if(rcr2() >= (USERTOP - proc->sizeOfStack*PGSIZE - PGSIZE))
    {
        if(growstack(proc) == 0)
            break;
    }
```

Part C: Test Result：


shirai@ubuntu: ~/Documents/project3test
test null2 PASSED (10 of 10)
(null2)
test bounds PASSED (10 of 10)
(bounds)
test bounds2 PASSED (10 of 10)
(bounds2)
test stack PASSED (10 of 10)
(stack)
test heap PASSED (10 of 10)
(heap)
test stack2 PASSED (10 of 10)
(stack2)
test bounds3 PASSED (10 of 10)
(bounds3)
test stack4 PASSED (10 of 10)
(stack4)
Passed 10 of 10 tests.
Overall 10 of 10
Points 90 of 90
shirai@ubuntu:~/Documents/project3test$