LINUX 实验报告

Project 3

课　程　名　称：　Linux 操作系统实践

年　　　　　级：　2018 级

朱张弛　10185102126

组　内　成　员：　汪子凡　10185102153

白彬惠　10185102121

2020 年 12 月

目录

（一）Project 3a:Locks & Threads

一、实验目的

1. To get a feel for threads, locks, and performance.

2. To build thread-safe versions of three common data structures: counter, list and hash table.

3. To make a nice report by comparing different lock implementations and concurrency levels.

二、问题重述

1. The lock you build should define a `spinlock_t` data structure, which contains any values needed to build your lock, and two routines:

```
spinlock_acquire(spinlock_t *lock)
spinlock_release(spinlock_t *lock)
```

2. the mutex you build should define a `mutex_t` data structure and two routines:

```
mutex_acquire(mutex_t *lock)
mutex_release(mutex_t *lock)
```

3. you will use your locks to build three concurrent data structures. The three data structures you will build are a thread-safe counter, list, and hash table.

To build the counter, you should implement the following code:

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

To build the list, you should implement the following routines:

```
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);
```

To build the hash table, you should implement the following code:

```
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);
```

4. write up a report on some performance comparison experiments.

三、算法设计

1. lock.h

spinlock 与 mutex 都包含 flag 表示是否占用。

lock_init 等函数将不同锁的操作包装起来便于调用。

type 用于记录当前在使用哪种锁。

```
extern unsigned int type;
typedef struct __spinlock_t
{
    unsigned int flag;
}spinlock_t;
void spinlock_init(spinlock_t *lock);
void spinlock_acquire(spinlock_t *lock);
void spinlock_release(spinlock_t *lock);


//mutex
typedef struct __mutex_t
{
    unsigned int flag;
}mutex_t;
void mutex_init(mutex_t *lock);
```

```
void mutex_acquire(mutex_t *lock);
void mutex_release(mutex_t *lock);
void lock_init(void *lock);
void lock_acquire(void *lock);
void lock_release(void *lock);
```

2. lock.c

实现了自己定义的 spinlock 与 mutex，并包装。

```
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;
}

//mutex
void mutex_init(mutex_t *lock) {
    lock->flag = 0;
}
void mutex_acquire(mutex_t *lock) {
    if (xchg(&(lock->flag), 1) == 0) {
        return;
    }
    while (1) {
        if (xchg(&(lock->flag), 1) == 0)return;
        sys_futex((void*)(&(lock->flag)), FUTEX_WAIT, 1, NULL, NULL, 0);
    }
}
void mutex_release(mutex_t *lock) {
    xchg(&(lock->flag), 0);
    sys_futex((void*)(&(lock->flag)), FUTEX_WAKE, 1, NULL, NULL, 0);
}
void lock_init(void *lock) {
    switch (type) {
    case 0:
        spinlock_init((spinlock_t*)lock);
        break;
    case 1:
        mutex_init((mutex_t*)lock);
        break;
```

```
    case 2:
        pthread_spin_init((pthread_spinlock_t*)lock, PTHREAD_PROCESS_SHARED);
        break;
    case 3:
        pthread_mutex_init((pthread_mutex_t*)lock, NULL);
        break;
    }
}
void lock_acquire(void *lock) {
    switch (type) {
    case 0:
        spinlock_acquire((spinlock_t*)lock);
        break;
    case 1:
        mutex_acquire((mutex_t*)lock);
        break;
    case 2:
        pthread_spin_lock((pthread_spinlock_t*)lock);
        break;
    case 3:
        pthread_mutex_lock((pthread_mutex_t*)lock);
        break;
    }
}
void lock_release(void *lock) {
    switch (type) {
    case 0:
        spinlock_release((spinlock_t*)lock);
        break;
    case 1:
        mutex_release((mutex_t*)lock);
        break;
    case 2:
        pthread_spin_unlock((pthread_spinlock_t*)lock);
        break;
    case 3:
        pthread_mutex_unlock((pthread_mutex_t*)lock);
        break;
    }
}
```

3. counter.c

实现了 counter 的初始化，查询，增加，减少。

由于需要测试四种锁，因此在初始化时需要对所有锁都进行初始化。

```
void counter_init(counter_t *c, int value) {
    c->value = value;
    c->locks[0] = (void*)(&(c->spinlock));
    c->locks[1] = (void*)(&(c->mutex));
    c->locks[2] = (void*)(&(c->pspinlock));
    c->locks[3] = (void*)(&(c->pmutex));
    lock_init((c->locks[type]));
}
int counter_get_value(counter_t *c) {
    int res;
    lock_acquire(c->locks[type]);
    res = c->value;
    lock_release(c->locks[type]);
    return res;
}
void counter_increment(counter_t *c) {
    lock_acquire(c->locks[type]);
    c->value++;
    lock_release(c->locks[type]);
}
void counter_decrement(counter_t *c) {
    lock_acquire(c->locks[type]);
    c->value--;
    lock_release(c->locks[type]);
}
```

4. list.c

实现了 list 初始化，插入，删除，查找，以及释放。

```
void list_init(list_t *list) {
    list->head = NULL;
    list->locks[0] = (void*)(&(list->spinlock));
    list->locks[1] = (void*)(&(list->mutex));
    list->locks[2] = (void*)(&(list->pspinlock));
    list->locks[3] = (void*)(&(list->pmutex));
    lock_init(list->locks[type]);
}
void list_insert(list_t *list, unsigned int key) {
    node_t* nw = malloc(sizeof(node_t));
    lock_acquire(list->locks[type]);
    nw->value = key;
    nw->next = list->head;
    list->head = nw;
```

```c
    lock_release(list->locks[type]);
}
void list_delete(list_t *list, unsigned int key) {
    lock_acquire(list->locks[type]);
    node_t* fa = NULL;
    node_t* p = list->head;
    while (p != NULL) {
        if (p->value == key) {
            if (p == list->head) {
                list->head = p->next;
            }
            else {
                fa->next = fa->next->next;
            }
            //printf("%p\n", p);
            free(p);
            break;
        }
        fa = p;
        p = p->next;
    }
    lock_release(list->locks[type]);
}
void *list_lookup(list_t *list, unsigned int key) {
    lock_acquire(list->locks[type]);
    node_t* p = list->head;
    while (p != NULL) {
        if (p->value == key) {
            lock_release(list->locks[type]);
            return (void*)p;
        }
        p = p->next;
    }
    lock_release(list->locks[type]);
    return NULL;
}
void list_free(list_t *list) {
    node_t* p = list->head;
    node_t* f;
    while (p != NULL) {
        f = p;
        p = p->next;
        free(f);
    }
```

```
}
```

5. hash.c

在 list 的基础上实现，hash 的初始化，插入，删除，查找，释放。

```
void hash_init(hash_t *hash, int size) {
    hash->size = size;
    int i;
    for (i = 0; i < size; i++)
        list_init(&(hash->hash_table[i]));
    hash->locks[0] = (void*)(&(hash->spinlock));
    hash->locks[1] = (void*)(&(hash->mutex));
    hash->locks[2] = (void*)(&(hash->pspinlock));
    hash->locks[3] = (void*)(&(hash->pmutex));
    lock_init(hash->locks[type]);
}
void hash_insert(hash_t *hash, unsigned int key) {
    lock_acquire(hash->locks[type]);
    list_insert(&(hash->hash_table[key%hash->size]), key);
    lock_release(hash->locks[type]);
}
void hash_delete(hash_t *hash, unsigned int key) {
    lock_acquire(hash->locks[type]);
    list_delete(&(hash->hash_table[key%hash->size]), key);
    lock_release(hash->locks[type]);
}
void *hash_lookup(hash_t *hash, unsigned int key) {
    void* res = NULL;
    lock_acquire(hash->locks[type]);
    res = list_lookup(&(hash->hash_table[key%hash->size]), key);
    lock_release(hash->locks[type]);
    return res;
}
void hash_free(hash_t *hash) {
    int i;
    for (i = 0; i < hash->size; i++) {
        list_free(&(hash->hash_table[i]));
    }
}
```

6. counter_test.c

用于测试 counter。用四种锁，线程个数由 1 到 20，每种线程个数下测试 10 次，

每次测试给每个线程分配任务，并记录该线程完成所需时间。

每次测试得到该次所需总时间，以及各个线程所需时间的方差。这一线程个数下的测试结果为所有次尝试得到结果取平均值。

花费总时间用于比较不同锁的效率。

方差用于比较不同锁的公平性。

```
#define MAX 10000 //操作次数
#define T 10      //尝试次数，取平均
#define MXTHREAD 20   //最多线程数
unsigned int type;
int thread_count;
counter_t counter;
struct timeval tp1;
struct timeval tmp;
double start, end;
double aver[MXTHREAD + 10], vari[MXTHREAD + 10], cost[MXTHREAD + 10];
void* t_counter(void* rank);
pthread_t thread_handles[MXTHREAD + 10];
int i, t, j;
int main(int argc, char const *argv[])
{
    for (type = 0; type < 4; type++) {
        if (type == 0)printf("-----------Here is test of spinlock: -----------\n");
        else if (type == 1)printf("-----------Here is test of mutex: -----------\n");
        else if (type == 2)printf("-----------Here is test of pthread_spinlock:
-----------\n");
        else printf("-----------Here is test of pthread_mutex: -----------\n");
        for (thread_count = 1; thread_count <= MXTHREAD; thread_count++) {
            aver[thread_count] = vari[thread_count] = 0.0;
            printf("thread = %d\n", thread_count);
            for (t = 0; t < T; t++) {
                counter_init(&counter, 0);
                gettimeofday(&tp1, NULL);
                start = tp1.tv_sec + tp1.tv_usec / 1000000.0;
                for (i = 0; i < thread_count; i++)
                    pthread_create(&thread_handles[i], NULL, t_counter, (void*)i);
                for (i = 0; i < thread_count; i++)
                    pthread_join(thread_handles[i], NULL);
                gettimeofday(&tp1, NULL);
                end = tp1.tv_sec + tp1.tv_usec / 1000000.0;
                aver[thread_count] += end - start;
                double sum = 0.0;
```

```
                for (i = 0; i < thread_count; i++)
                    sum += cost[i];
                sum /= thread_count;
                double var = 0.0;
                for (i = 0; i < thread_count; i++) {
                    var += (cost[i] - sum)*(cost[i] - sum);
                }
                var /= thread_count;
                vari[thread_count] += var;
            }
            aver[thread_count] /= T;
            vari[thread_count] /= T;
        }
        for (i = 1; i <= MXTHREAD; i++)printf("%lf%c", aver[i], " \n"[i == MXTHREAD]);
        for (i = 1; i <= MXTHREAD; i++)printf("%lf%c", vari[i], " \n"[i == MXTHREAD]);
    }
    return 0;
}
void* t_counter(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        counter_increment(&counter);
    }
    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
```

7. list_test.c

实现对 list 的测试，与 counter_list 类似，但由于 list 有插入删除，随机插入，随机删除等多种测试，所以通过传入的 main 函数的参数判断应给线程分配哪种测试任务。

```
void* t_list1(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
```

```
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        list_insert(&list, i);
    }
    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
void* t_list2(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        list_insert(&list, i);
    }
    for (i = MAX - 1; i >= 0; i--) {
        list_delete(&list, i);
        //printf("%d\n", i);
        //printf("%p\n", list.head);
    }

    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
void* t_list3(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        list_insert(&list, rand() % 100000);
    }
    for (i = 0; i < MAX; i++) {
        list_delete(&list, rand() % 100000);
    }
    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
```

```
    return NULL;
}
```

```
switch ((argv[1][0] - '0')) {
case 1:
    pthread_create(&thread_handles[i], NULL, t_list1, (void*)i);
    break;
case 2:
    pthread_create(&thread_handles[i], NULL, t_list2, (void*)i);
    break;
case 3:
    pthread_create(&thread_handles[i], NULL, t_list3, (void*)i);
    break;
}
```

8. hash_test.c

实现了对 hash 的测试，与 list_test.c 类似。改变了线程分配的测试任务。

```
void* t_hash1(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        hash_insert(&hash, i);
    }
    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
void* t_hash2(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        hash_insert(&hash, i);
    }
    for (i = 0; i < MAX; i++) {
        hash_delete(&hash, i);
    }
    gettimeofday(&tmp, NULL);
```

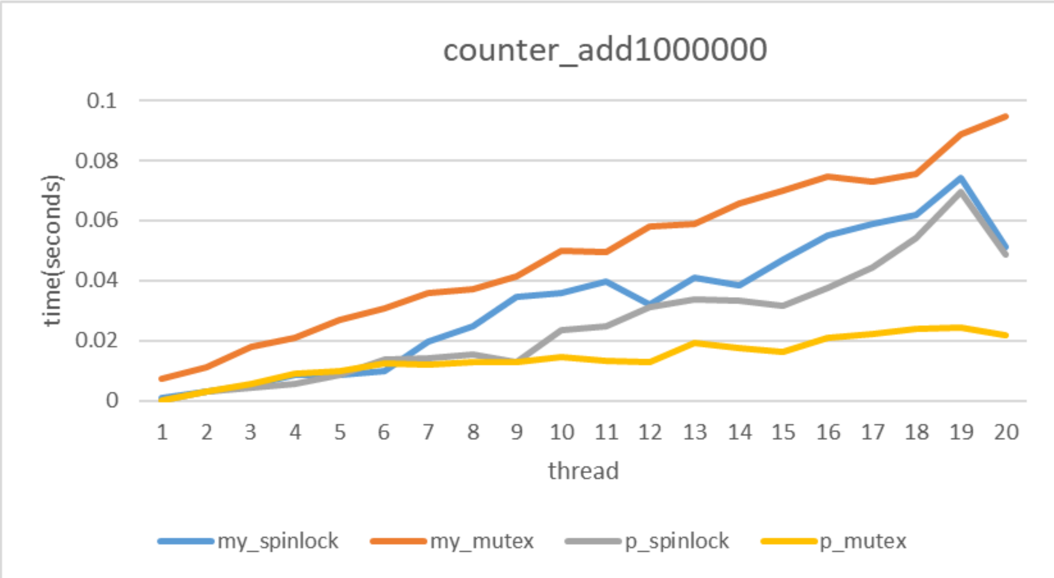
```c
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
void* t_hash3(void* rank) {
    int i;
    int rk = (int)rank;
    double mstart, mend;
    gettimeofday(&tmp, NULL);
    mstart = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    for (i = 0; i < MAX; i++) {
        hash_insert(&hash, rand() % 100000);
    }
    for (i = 0; i < MAX; i++) {
        hash_delete(&hash, rand() % 100000);
    }
    gettimeofday(&tmp, NULL);
    mend = tmp.tv_sec + tmp.tv_usec / 1000000.0;
    cost[rk] = mend - mstart;
    return NULL;
}
```

9. 由于测试较多，因此通过.sh 脚本实现测试。

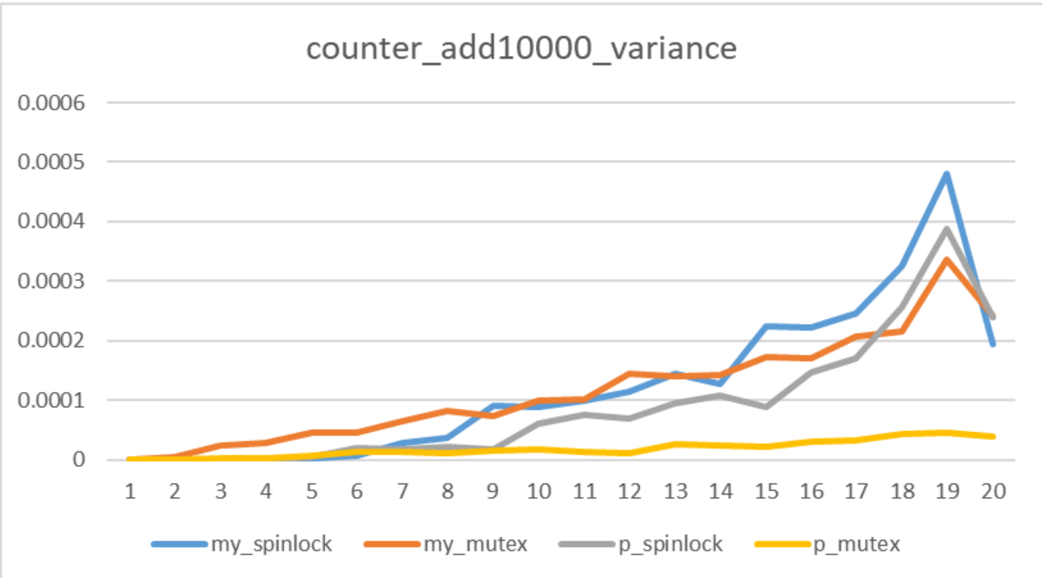
```sh
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
make test_counter
make test_list
make test_hash
./counter_test > counter_res
./list_test 1 > list_res1
./list_test 2 > list_res2
./list_test 3 > list_res3
./hash_test 1 > hash_res1
./hash_test 2 > hash_res2
./hash_test 3 > hash_res3
./hash_test 4 > hash_res4
./hash_test 5 > hash_res5
./hash_test 6 > hash_res6
```

四、实验结果

1. Counter

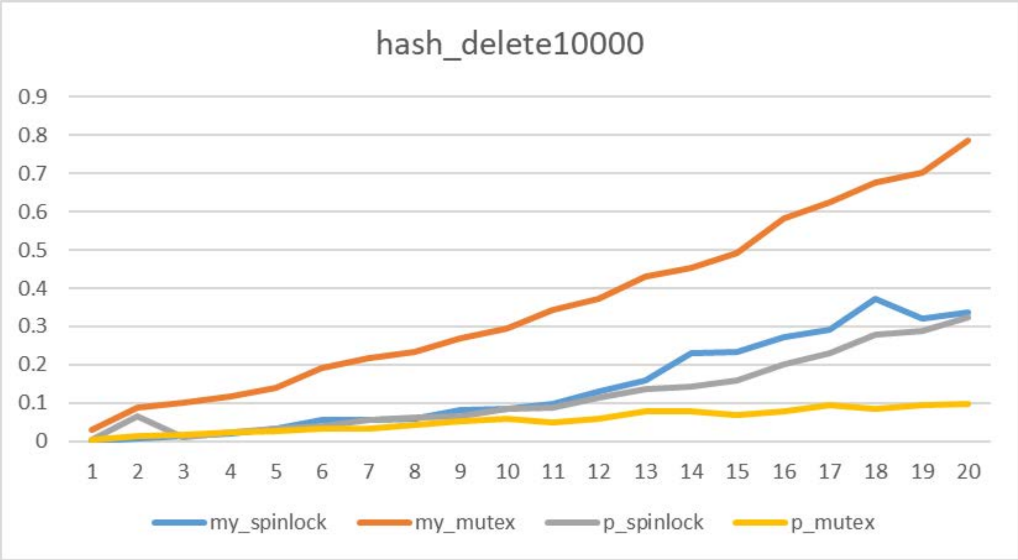


在 counter 中，my_mutex 的耗时远 于其它三种，且线程越多，性能差距越明显。
My_mutex 比 pthread 原有 mutex 锁性能差；
my_spinlock 与 pthread 原有 spinlock 锁性能大致相当。

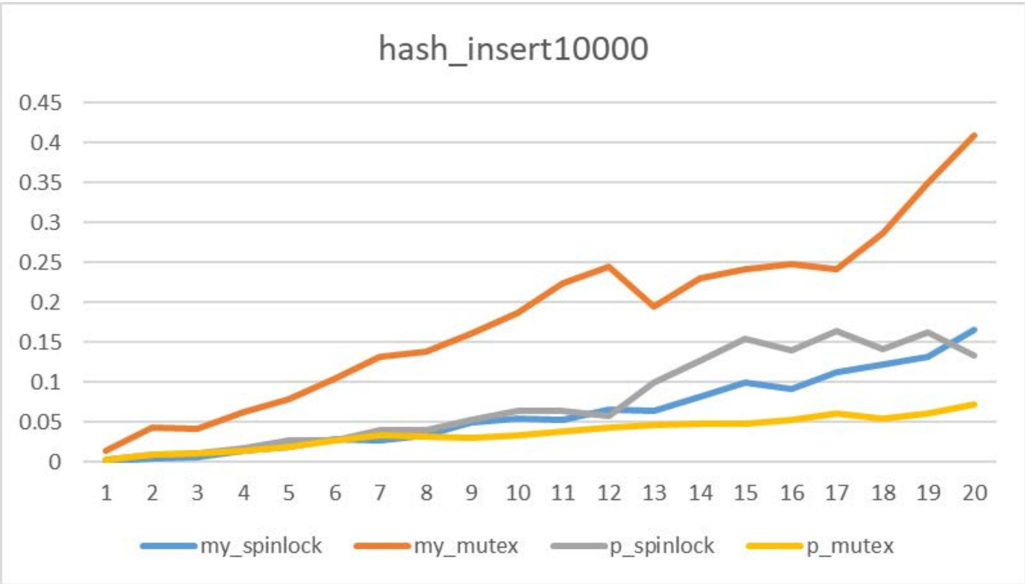


pthreads mutex 曲线最为平稳，保持于方差较小，公平性相对较好
my_mutex,my_spinlock、p_spinlock 曲线相对来说公平性较差，后期随线程增加显著降低，且 my_spinlock 最为明显。

2. Hash



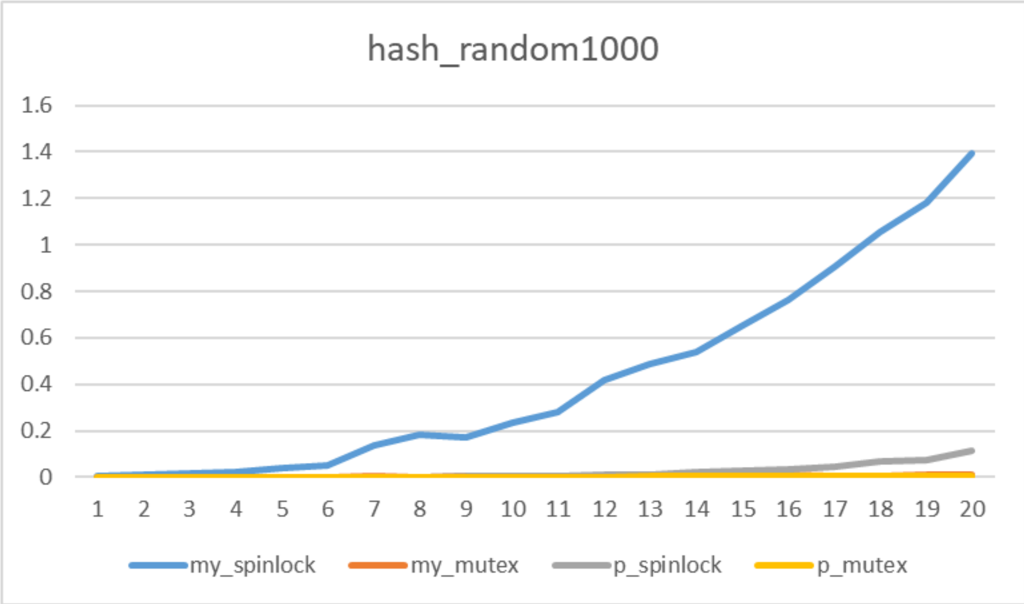
Hash 表删除时，p_mutex 性能明显优于 spinlock 与 pthread spinlock，且十分稳定。Spinlock 与 pthread spinlock 结果不相 上下，随着线程增多而耗费时间显著增多。



Hash 表插入时，my_spinlock 性能高于 my_mutex;

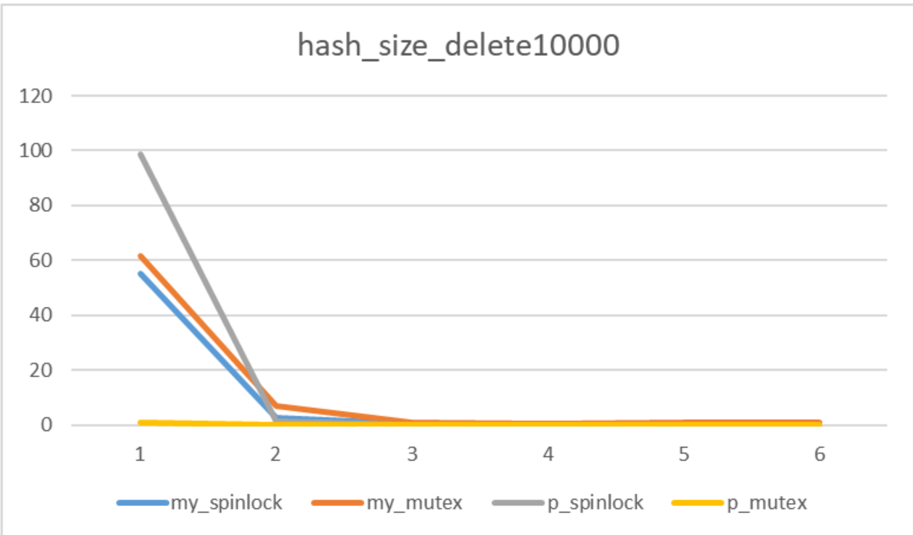
P_mutex 性能高于 my_mutex,并随着线程增多，差距有所增大;

My_spinlock 与 p_spinlock 性能不相上下，且随线程增大，p_spinlock 耗费时间会呈下降趋势。

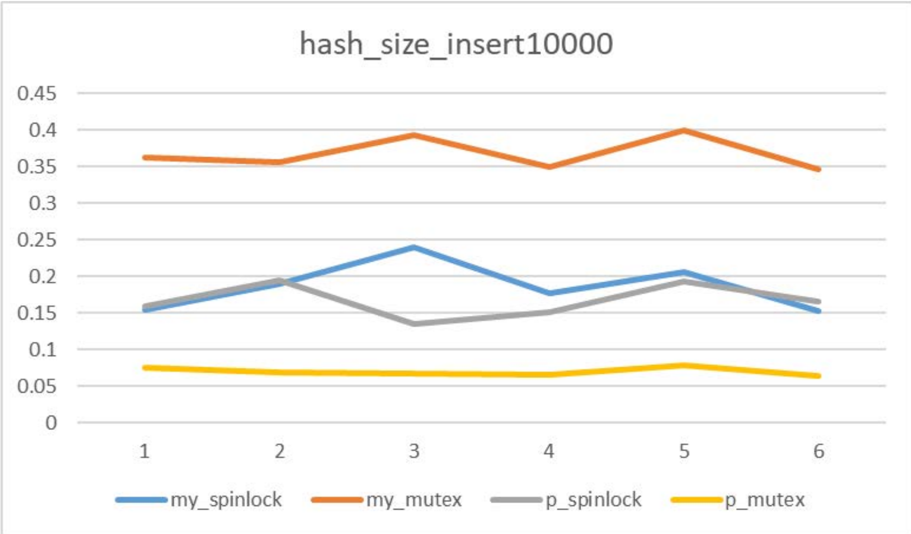


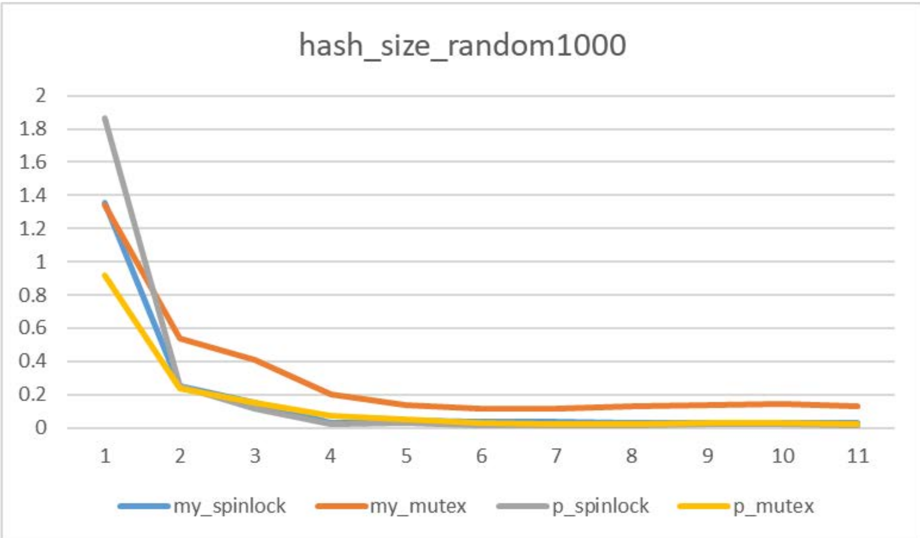
Hash 表随机插入删除时，my_spinlock 性能最差，且随线程增多耗费时间显著增多其余性能相差不大，且没有明显变化趋势

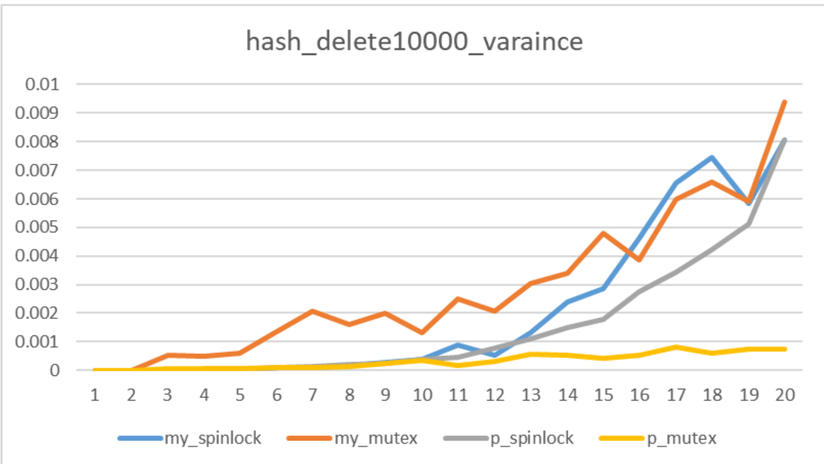
取 size 为 10，100，1000，10000，100000，1000000 对 Hash 表进行操作所得数据如下图：



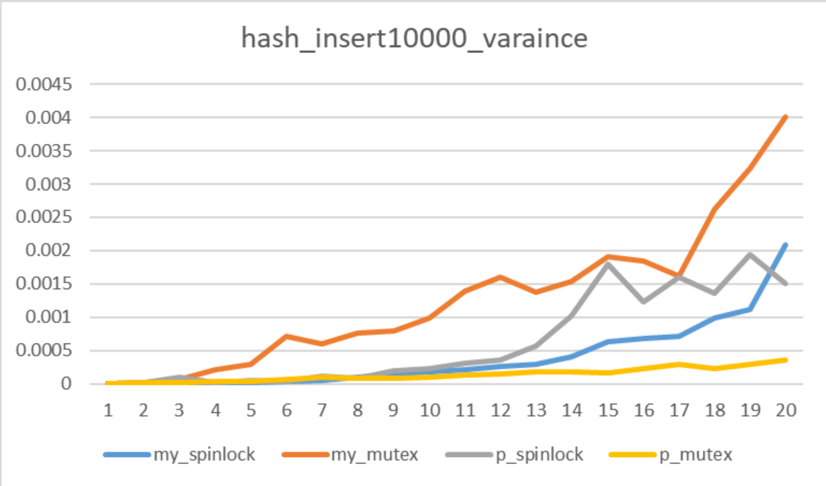
P_mutex 稳定性较高始终保持一定水平，而 my_spinlock,my_mutex,p_spinlock 相差



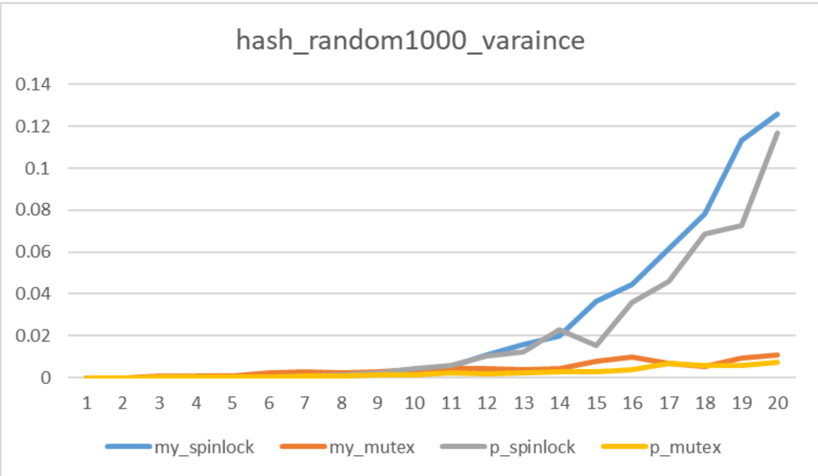




P_mutex 稳定性较高始终保持一定水平，而 my_spinlock,my_mutex,p_spinlock 相差不大，都随线程增多有大幅度上升趋势。



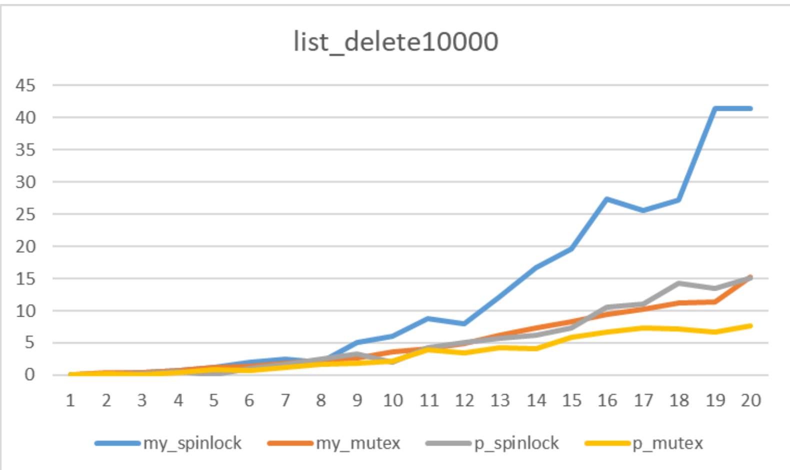
P_mutex 稳定性较高始终保持一定水平，而 my_spinlock, p_spinlock 相差不大，都随线程增多有一定程度上升趋势；而 my_mutex 在短暂下降后呈现总体上升趋势。



P_mutex 曲线相对平稳，保持与方差较小，公平性相对较好；

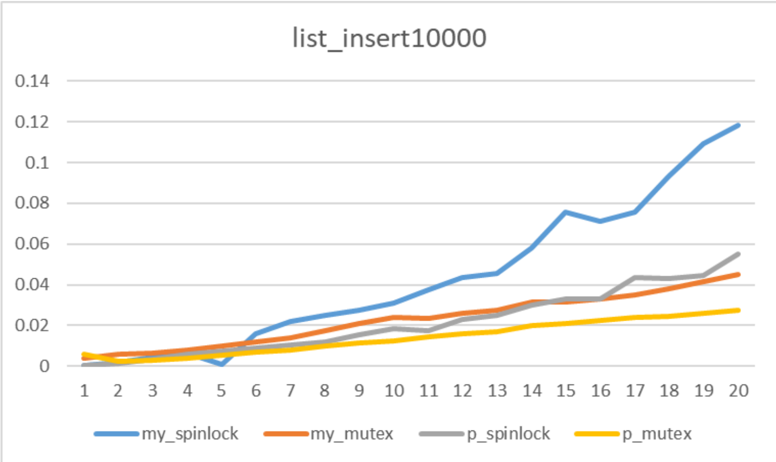
My_spinlock 与 p_spinlock 曲线公平性较差，随线程增加上升 ‘

3. List

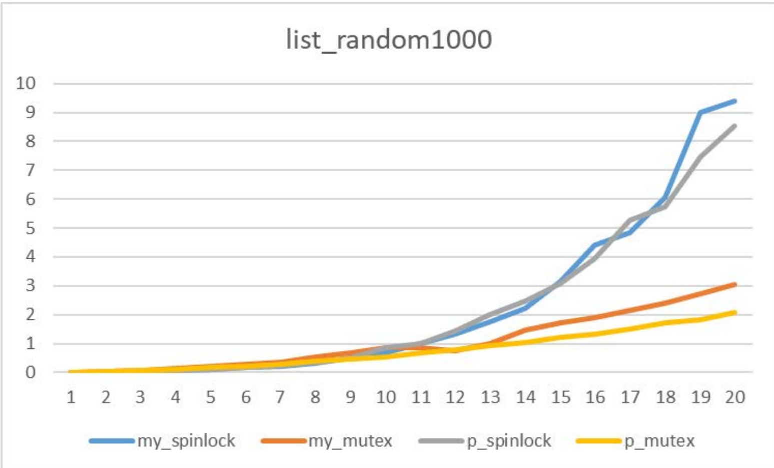


链表删除时，my_mutex 与 p _mutex 性能明显优于 spinlock 与 p_spinlock, 且十分稳

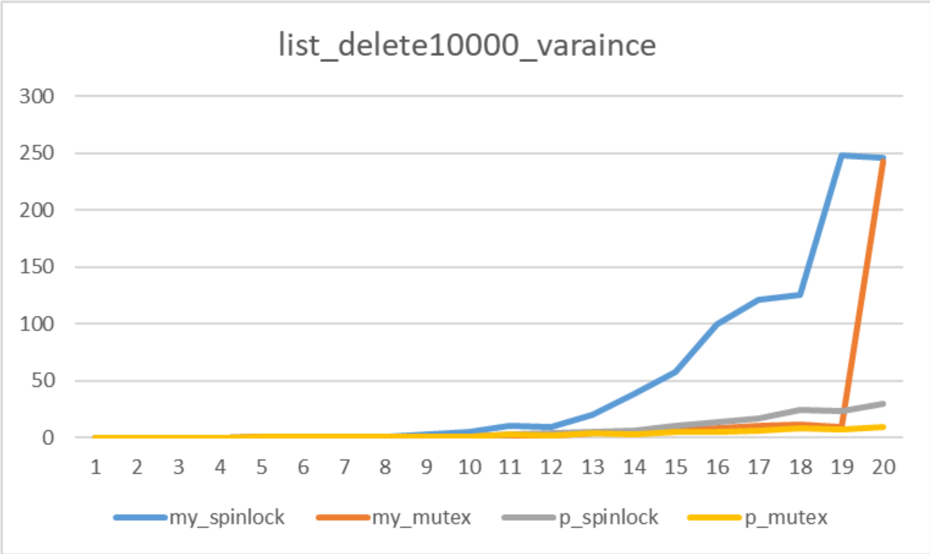
定；My_spinlock 随着时间增加耗费时间显著增多；

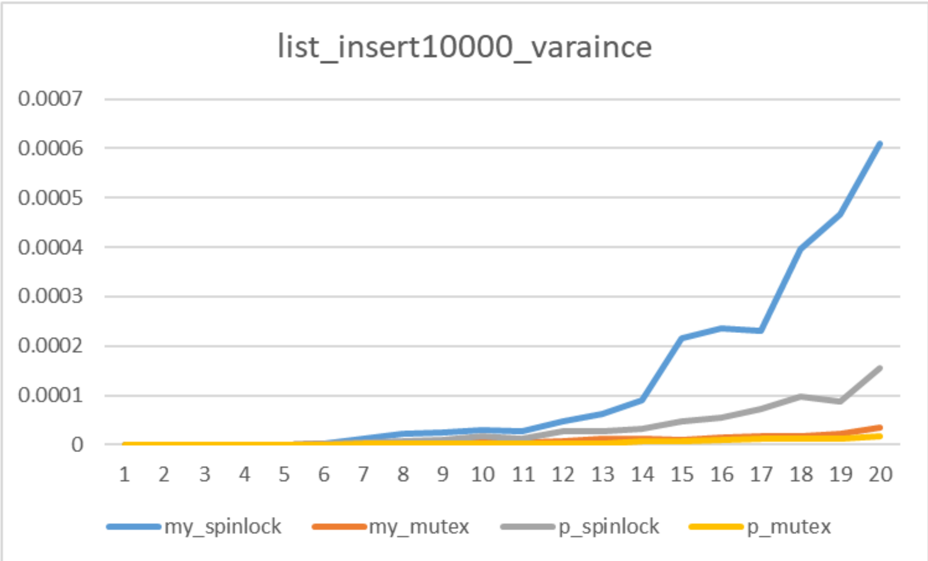


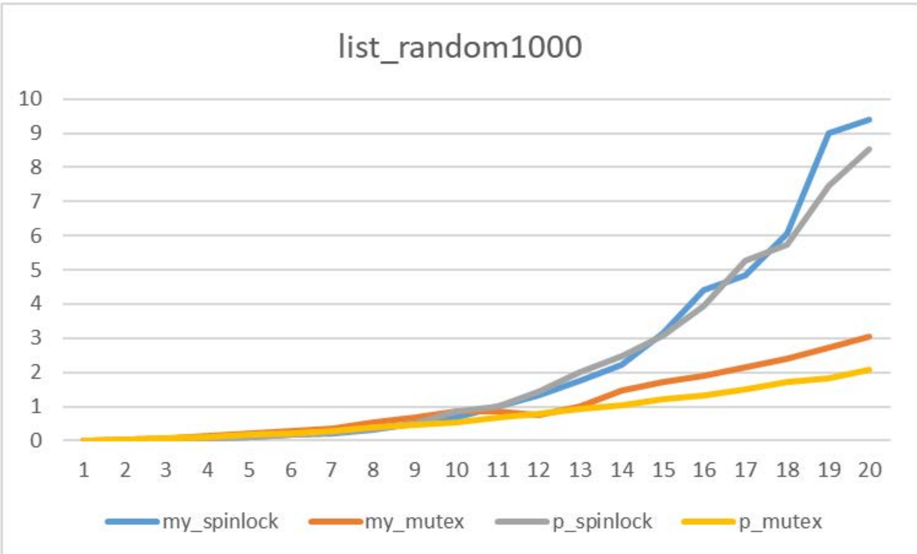
链表插入时，随进程增多，my_spinlock 后期上升明显，较其余三种耗时长，性能降低；my_mutex 性能要低于 p_mutex，线程越多，而这差距越大



链表随机插入删除时，my_mutex 与 p_mutex 性能明显优于 my_spinlock 与 P_spinlock，随着线程增多而耗费时间稍有增多。My_spinlock 与 p_spinlock 结果不相上下，随着线程增多而耗费时间显著增多。







mutex、pthreads mutex 曲线相对平稳，保持于方差较小，公平性相对较好，随线程增加稍有降低。spinlock、pthread spinlock 曲线相对来说公平性较差，后期随线

程增加显著降低，且 pthread spinlock 比 spinlock 更明显。在 list 中，spinlock 的性能是最差的，这与线程的执行函数所需要的时间密不可分（无论是插入、查找还是删除平均都需要 O(n)的时间复杂度）。

4. 总结

（1）一个进程进入临界区的操作时间越短，spinlock 的表现就越好；反之 mutex 表现更好；

⑵当操作简单时，自旋锁的性能往往优于互斥锁；当操作复杂且面临大规模数据时，互斥锁的性能优于自旋锁。

⑶进行相同操作时，哈希表会比单纯的链表操作快，且 size 越大（小于次数）越快。

⑷互斥锁在线程增加时的时间改变小；自旋锁在线程增加时的时间改变大。

⑸由于随机插入和删除时，删除操作的 key 不稳定，遍历次数不定，故参考性低于仅插入的实验。

（二）Project3b：xv6 VM Layout

一、实验目的

1. To familiarize you with the xv6 virtual memory system.
2. To add a few new VM features to xv6 that are common in modern OSes.

二、问题重述

In this project, you'll be changing xv6 to support a few features virtually every modern OS does. The first is causing an exception to occur when your program dereferences a null pointer; the second is rearranging the address space so as to place the stack at the high end.

Your job here will be to figure out how xv6 sets up a page table, and then change it to leave the first two pages (0x0 - 0x2000) unmapped. The code segment should be starting at 0x2000.

one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently with the `sz` field in the `proc` struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the address space). We recommend keeping this field to track the size of the code and heap, but doing some other accounting to track the stack, and changing all relevant code (i.e., that used to deal with `sz`) to now work with your new accounting.

三、算法分析

发现有两点需要实现：

①需要对空指针的解引进行处理，防止内存的外泄，并且将地址空间的前两页(0x0000 - 0x2000)不进行映射，地址从 0x2000 开始。

当在 Linux 对空指针进行解引时，发现会产生段错误：

```c
//test.c
#include<stdio.h>

int main()
{
    int *nullp = NULL;
    printf("%d\n", *nullp);
}
```

```
wangzifan@ubuntu: ~
wangzifan@ubuntu:~$ gcc -o test test.c
wangzifan@ubuntu:~$ ./test
Segmentation fault (core dumped)
wangzifan@ubuntu:~$
```

当在 xv6 对空指针进行解引时，发现空指针会解引到地址 0x0000 的值，造成内存的外泄，测试的程序可以用测试数据中的 null.c：

```c
/* null pointer dereference should kill process */
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
   int ppid = getpid();

   if (fork() == 0) {
      uint * nullp = (uint*)0;
      printf(1, "null dereference: ");
      printf(1, "%x %x\n", nullp, *nullp);
      // this process should be killed
      printf(1, "TEST FAILED\n");
      kill(ppid);
      exit();
   } else {
      wait();
   }

   printf(1, "TEST PASSED\n");
   exit();
}
```

在 qemu 中运行，发现会输出对应地址 0x0000 的值：

```
$ tester
tester
null dereference: 0 83E58955
TEST FAILED


test null FAILED (0 of 10)
 (null)
 tester failed

Skipped 8 tests.
To keep testing after failing a test, use flag '-c' or '--continue'
wangzifan@ubuntu:~/xv6/project3test$
```

②需要对一个进程的地址空间进行重新的安排，原来栈和堆都放在低端，并且栈固定长度为 1 页：

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

重排之后，栈会放在高端，向低位增长，堆在低端，向高端增长，两者之间要空出 5 页；结合前面的要求，前两页不需要映射：

```
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
```

四、实现过程

①在原来的 xv6 中，proc 结构体中的 sz 记录进程所使用的内存总大小。而由于现在的进程一半在高端(栈)，一半在低端，所以**新增变量 sz_stack(以字节为单位) 记录高端栈的大小，sz 记录其他部分占据低位的内存大小(从 0x2000 开始)**。

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this pr
ocess
  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)
  //modified here
  uint sz_stack;
};
```

②在 exec.c 中，进程将代码填充进地址空间，并完成了堆栈的初始化。所以在这里需要将地址空间起始位置由 0x0000 改变为 0x2000：

```
// Load program into memory.
  //sz = 0;   modified here
  sz = 0x2000;
```

将原先在低端给栈分配一页，改为在高端先给栈分配一页，由于原来栈就是从高位向低位增长的，所以不需要修改。

```
// Allocate a one-page stack at the next page boundary
  // sz = PGROUNDUP(sz);  modified here
  //if((sz = allocuvm(pgdir, sz, sz + PGSIZE)) == 0)
  //  goto bad;
  uint tmp = allocuvm(pgdir, USERTOP-PGSIZE, USERTOP);
  if(tmp == 0)
    goto bad;

   // Push argument strings, prepare rest of stack in ustack.

  //sp = sz;  modified here
  sp = PGROUNDUP(tmp);

  同时新增的元素 st_stack 也需要相应的初始化:

//modified here
  proc->sz_stack = PGSIZE;
```

③vm.c 中用到了 sz，是关于虚拟内存的函数，也需要修改。
Inituvm()函数中，将 initcode 加载到了地址空间 0x0000 的地方，这里需要修改成 0x2000:

```
// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
```

```
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;
  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  //modiefied here

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

Copyuvm()函数中子进程根据父进程的页表进行复制，这里首先需要将复制从0x0000 开始改成从 0x20000 开始：

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz, uint sz_stack)
{
  pde_t *d;
  pte_t *pte;
  uint pa, i;
  char *mem;

  if((d = setupkvm()) == 0)
    return 0;
  //for(i = 0; i < sz; i += PGSIZE){
  //modified here
   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;
  }
```

此外，修改过的 xv6 高端还有栈，因此对这部分的页表也需要复制，**所以这里需要新增一个参数 sz_stack 来记录栈的大小进行复制：**

```
for(i = (uint)PGROUNDUP(USERTOP - sz_stack); i < USERTOP; 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;
  }
```

④pro.c 文件中是一些关于进程的函数，也需要进行相应的修改。
Userinit()用来创建第一个用户进程，需要对它的成员变量进行修改，sz 的值要加上 0x2000，sz_stack 要初始化为 0，将寄存器 esp 和 eip 也需要修改：

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];
  p = allocproc();
  acquire(&ptable.lock);
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_si
ze);

  //modified here
  p->sz = PGSIZE + 0x2000;
  p->sz_stack = 0;

  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
```

```
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  //p->tf->esp = PGSIZE;
  p->tf->esp = p->sz;           //modified here
  //p->tf->eip = 0;  // beginning of initcode.S
  //modified here
  p->tf->eip = 0x2000;
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
  release(&ptable.lock);
}
```

Fork()函数中涉及了子进程复制父进程的地址空间(用到了之前 copyuvm 函数)，因此需要多穿传一个参数同时复制低端和高端，并且 proc 中添加的 sz_stack 也需要赋值：

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  //modified here   add a paramenter
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz, proc->sz_stack)) == 
 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  //modified here
  np->sz_stack = proc->sz_stack;
  np->parent = proc;
```

```
  *np->tf = *proc->tf;
...
```

在 proc.c 中，原来有一个函数 growproc() 来增长，在这里判断需要注意堆和栈的差值是否小于 5 个页面：

```
// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;
  //modified here
  if(proc->sz + proc->sz_stack + n + 5 * PGSIZE > USERTOP)
      return -1;
  sz = proc->sz;
  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;
}
```

而原来栈的大小就是一个页面，不会增长，而在改进的 xv6 中栈是可以增长的（只需要保证堆和栈的差大于等于 5 个页面），所以需要**新增函数 growstack()**，来给栈增长：

```
// add a new function for stack-growing
int growstack(struct proc *p)
{
   if(allocuvm(p->pgdir, USERTOP - p->sz_stack - PGSIZE, USERTOP - p
->sz_stack) == 0)
     return -1;
   if(USERTOP - p->sz_stack - PGSIZE - p->sz < 5 * PGSIZE)// gap >=
5 pages
     return -1;//gap >= 5 pages
   p->sz_stack += PGSIZE;//  add one page
   switchuvm(p);
```

```
    return 0;
}
```

⑤由于之前在 vm.c 中给函数 copyuvm 增加了参数，还新增了函数 growstack，因此需要在 defs.h 中修改相应的声明：

```
// proc.c
struct proc*    copyproc(struct proc*);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
int             growstack(struct proc *);// modified here

// vm.c
void            seginit(void);
void            kvmalloc(void);
void            vmenable(void);
pde_t*          setupkvm(void);
char*           uva2ka(pde_t*, char*);
int             allocuvm(pde_t*, uint, uint);
int             deallocuvm(pde_t*, uint, uint);
void            freevm(pde_t*);
void            inituvm(pde_t*, char*, uint);
int             loaduvm(pde_t*, char*, struct inode*, uint, uint);
//pde_t*          copyuvm(pde_t*, uint);
//modified here
pde_t*          copyuvm(pde_t*, uint, uint);
void            switchuvm(struct proc*);
void            switchkvm(void);
int             copyout(pde_t*, uint, void*, uint);
```

⑥由于之前栈大小固定不需要增长，使用大于 1 页就会爆栈，但是现在需要动态增长，因此若是超出当前栈的大小，会产生 T_PGFLT 这个系统中断，若是检测

到栈空间不足之后使用之前的函数 growproc() 来尝试分配。

因此在 trap.c 中添加新的 case：

```
case T_PGFLT:
    if(rcr2() >= (USERTOP - (proc->sz_stack) - PGSIZE))
    {
       if(~growstack(proc))
          break;
   }
```

⑦由于 0x0000 到 0x2000 没有映射，因此在 kernel/makefile.mk 中，要修改 initcode 的 entry：

```
initcode: kernel/initcode.o
    $(LD) $(LDFLAGS) $(KERNEL_LDFLAGS) \
        --entry=start --section-start=.text=0x2000 \
        --output=kernel/initcode.out kernel/initcode.o
    $(OBJCOPY) -S -O binary kernel/initcode.out $@
```

同样用户进程也从 0x2000 开始，也需要在 user/makefile.mk 修改：

```
# location in memory where the program will be loaded
USER_LDFLAGS += --section-start=.text=0x2000
```

⑧对 syscall.c 的系统调用进行修改，对指针的空指针进行正确的解引。

对于 fetchint 函数，**要根据现有的地址空间布局情况来判断当前地址是否合法，**总共有三段不合法位置，注意 int 为 4 字节：

```
// Fetch the int at addr from process p.
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)        //超过栈底
    return -1;
  if(addr < 0x2000)                              //在未映射区域
    return -1;
  if(((addr >= p->sz) || (addr + 4) > p->sz) && addr < (USERTOP - p-
>sz_stack))      //在栈和堆之间
    return -1;

  *ip = *(int*)(addr);
```

```
  return 0;
}
```

对于 fetchstr 函数做类似修改，注意一个 char 为一个字节：

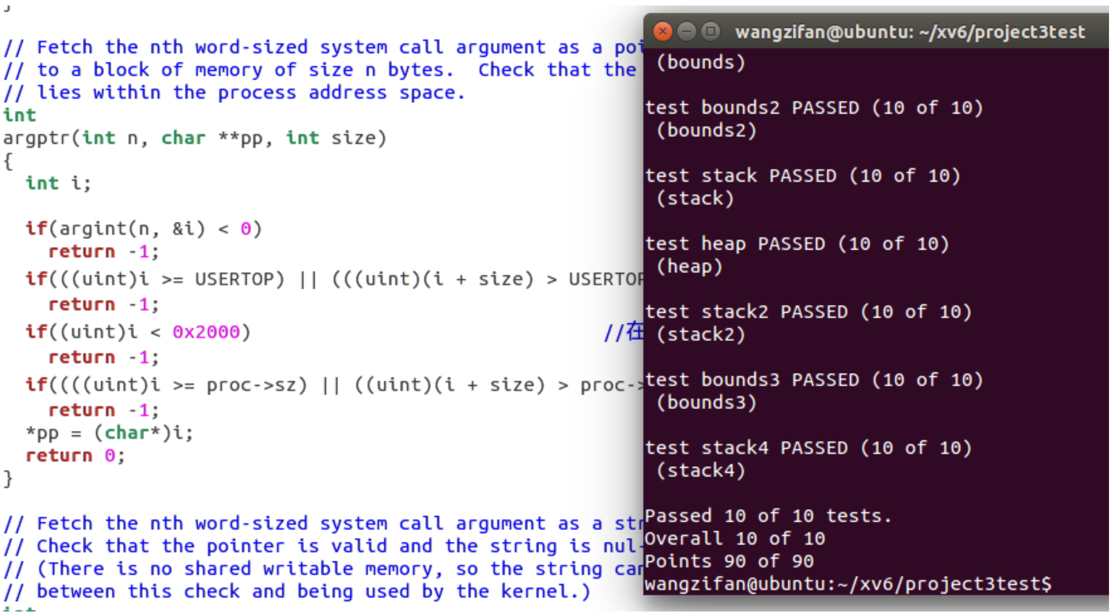
```
// Fetch the nul-terminated string at addr from process p.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;
  *pp = (char*)addr;
  if(addr >= USERTOP)        //超过栈底
    return -1;
  if(addr < 0x2000)                    //在未映射区域
    return -1;
  if(((addr >= p->sz)) && addr < (USERTOP - p->sz_stack))      //在栈
和堆之间
    return -1;

  *pp = (char*)addr;
  //ep = (char*)p->sz;
  if(addr < p->sz)              //如果是堆区
    ep = (char*)p->sz;
  else                          //如果是堆区
    ep = (char*)USERTOP;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

对于 argptr 也要类似判断，这里的大小 size 由参数给定：

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes.  Check that the pointer
// lies within the process address space.
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)))
//超过栈底
    return -1;
  if((uint)i < 0x2000)                              //在未映射区域
    return -1;
  if((((uint)i >= proc->sz) || ((uint)(i + size) > proc->sz)) && ((u
int)i < (USERTOP - (proc->sz_stack))))   //在栈和堆之间
    return -1;
  *pp = (char*)i;
  return 0;
}
```

最后结果，测试样例通过：



五、问题总结

①刚开始设置的时候仅将栈设置了一个页面后，没有实现动态增长，所以没有通过测试样例 stack2.c。为了实现**动态增长**，需要设置对应的函数 growstack，然后在 trap.c 中对于爆栈导致的页面异常做相应的处理，并且一定要在 defs.h 中**增加相应的函数声明**。

```
xv6...
cpu0: starting
init: starting sh
$ tester
tester
pid 4 tester: trap 14 err 6 on cpu 0 eip 0x204a addr 0x9e000--kill proc
$

test stack2 FAILED (0 of 10)
 (stack2)
 tester failed

Skipped 2 tests.
To keep testing after failing a test, use flag '-c' or '--continue'
wangzifan@ubuntu:~/xv6/project3test$
```

②在设置 fetchstr 的时候，将 fetchint 的判断条件复制了过来，没有通过测试样例 bounds3.c，这是因为 fetchint 中 int 是 4 个字节，而 char 类型是 1 个字节，存在某些位置可以存下 char 但是不能存下 int 的情况。

```
// Fetch the nul-terminated string at addr from process p.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;
  *pp = (char*)addr;
  if(addr >= USERTOP || addr + 4 > USERTOP)          //超过栈底
    return -1;
  if(addr < 0x2000)                                   //在未映射区域
    return -1;
  if(((addr >= p->sz) || (addr + 4) > p->sz) && addr < (USERTOP - p->
    return -1;

  *pp = (char*)addr;
  //ep = (char*)p->sz;
  if(addr < p->sz)                  //如果是堆区
    ep = (char*)p->sz;
  else                              //如果是堆区
    ep = (char*)USERTOP;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}
```

```
wangzifan@ubuntu: ~/xv6/project3test
hdb fs.img xv6.img -smp 1
WARNING: Image format was not specified for 'fs.img
         Automatically detecting the format is dang
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to rem
WARNING: Image format was not specified for 'xv6.im
         Automatically detecting the format is dang
perations on block 0 will be restricted.
         Specify the 'raw' format explicitly to rem
xv6...
cpu0: starting
init: starting sh
$ tester
tester
TEST PASSED


test bounds3 PASSED (10 of 10)
 (bounds3)

starting stack4

**************************************************
```

③在增加栈和堆大小时，目前方法是判断当前差距是不是大于 5 个页面，但是未判断加完之后是否大于 5 个页面。这可能还需要对 allocuvm 需要改进。