|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LINUX实验报告**  Project 3 | | | | |
|  | 课 程 名 称： | ： | Linux操作系统实践 |  |
| 年 级： | ： | 2018级 |
| 组 内 成 员： | ： | 朱张弛 10185102126 |
| ： | 汪子凡 10185102153 |
| ： | 白彬惠 10185102121 |
| 2020年12月 | | | | |

目录

[（一） Project 3a:Locks & Threads](#_Toc5221)

[一、 实验目的](#_Toc28357)

[二、 问题重述](#_Toc13149)

[三、 算法设计](#_Toc17052)

[四、 实验结果](#_Toc13625)

[1. Counter](#_Toc5633)

[2. Hash](#_Toc15545)

[3. List](#_Toc21650)

[4. 总结](#_Toc29561)

[（二） Project3b：xv6 VM Layout](#_Toc1056)

[一、 实验目的](#_Toc26623)

[二、 问题重述](#_Toc8471)

[三、 算法分析](#_Toc5411)

[四、 实现过程](#_Toc12985)

[五、 问题总结](#_Toc9658)

1. **Project 3a:Locks & Threads**
2. 实验目的
3. To get a feel for threads, locks, and performance.
4. To build thread-safe versions of three common data structures: counter, list and hash table.
5. To make a nice report by comparing different lock implementations and concurrency levels.
6. 问题重述
7. 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)

1. the mutex you build should define a mutex\_t data structure and two routines:

mutex\_acquire(mutex\_t \*lock)

mutex\_release(mutex\_t \*lock)

1. 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);

1. write up a report on some performance comparison experiments.
2. 算法设计
3. 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);

1. 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;

}

}

1. 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]);

}

1. 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;

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);

}

}

1. 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]));

}

}

1. 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;

}

1. 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;

}

1. 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);

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;

}

1. 由于测试较多，因此通过.sh脚本实现测试。

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

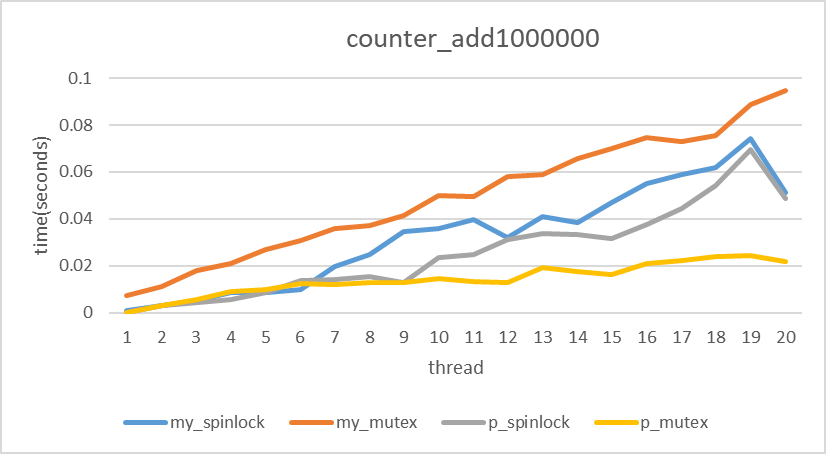
./hash\_test 3 > hash\_res3

./hash\_test 4 > hash\_res4

./hash\_test 5 > hash\_res5

./hash\_test 6 > hash\_res6

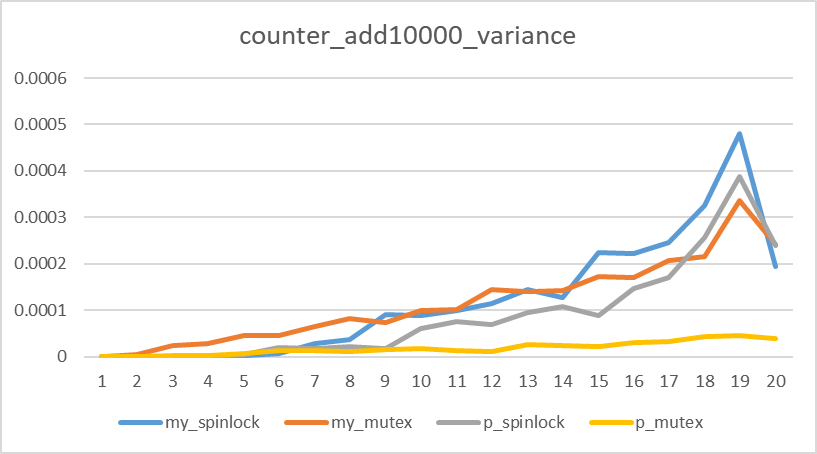
1. 实验结果
2. Counter



在counter中，my\_mutex的耗时远⾼于其它三种，且线程越多，性能差距越明显。

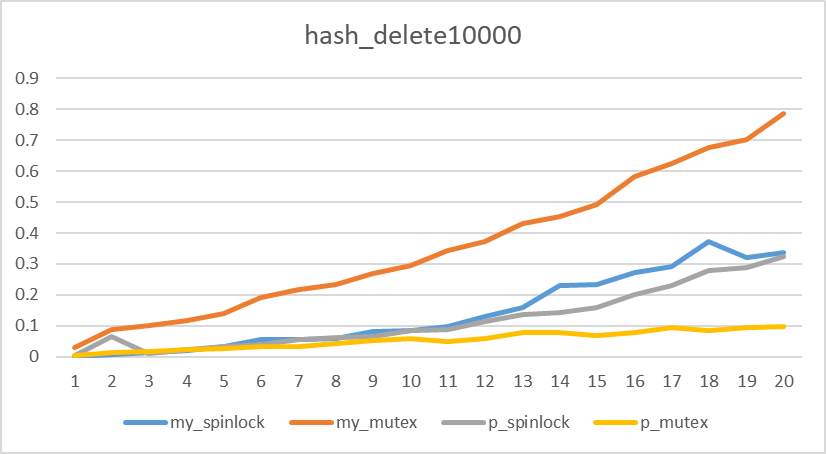
My\_mutex比pthread原有mutex锁性能差；

my\_spinlock与pthread原有spinlock锁性能大致相当。

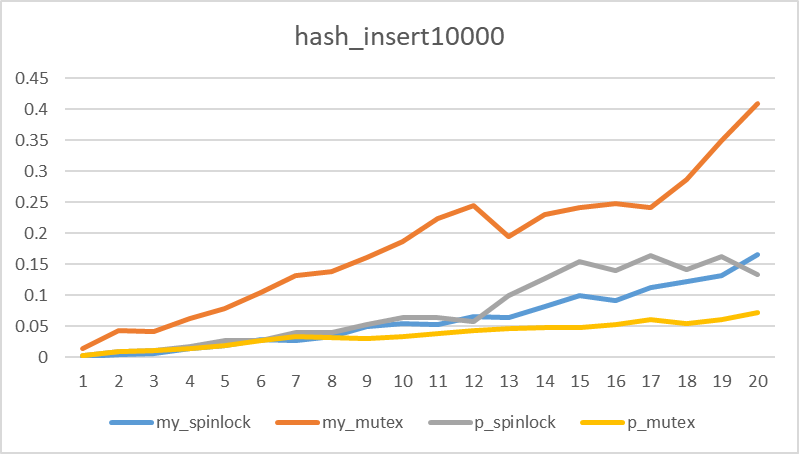


pthreads mutex曲线最为平稳，保持于方差较小，公平性相对较好my\_mutex,my\_spinlock、p\_spinlock 曲线相对来说公平性较差，后期随线程增加显著降低，且my\_spinlock最为明显。

1. Hash



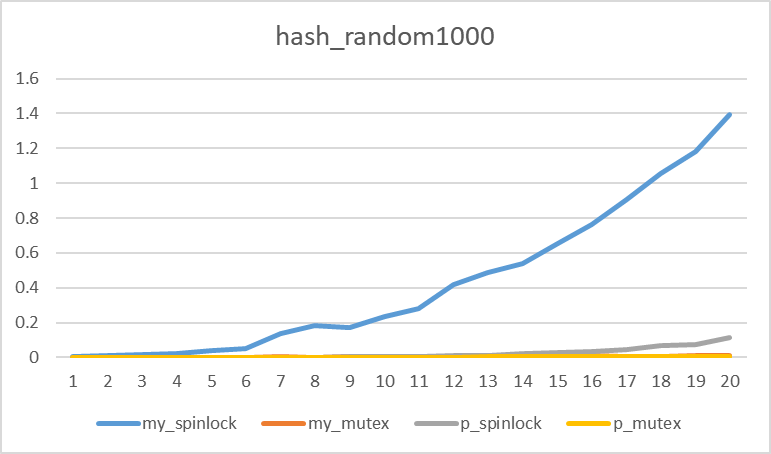
Hash表删除时，p\_mutex性能明显优于 spinlock与pthread spinlock，且十分稳定。Spinlock与pthread spinlock结果不相 上下，随着线程增多而耗费时间显著增多。



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；

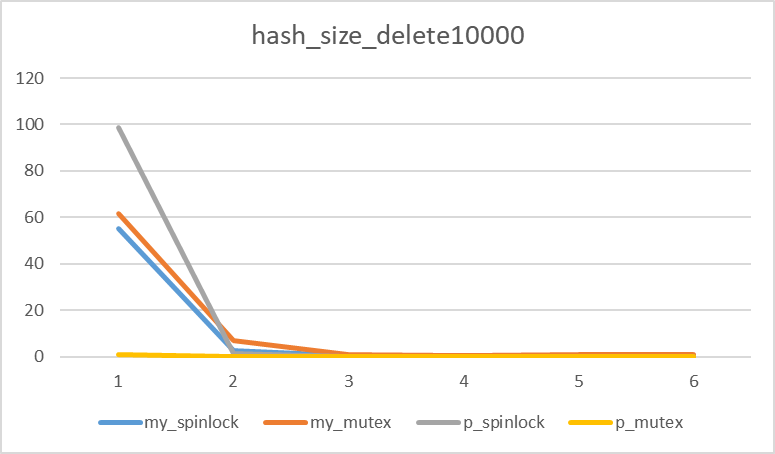
My\_spinlock与p\_spinlock性能不相上下，且随线程增大，p\_spinlock耗费时间会呈下降趋势。

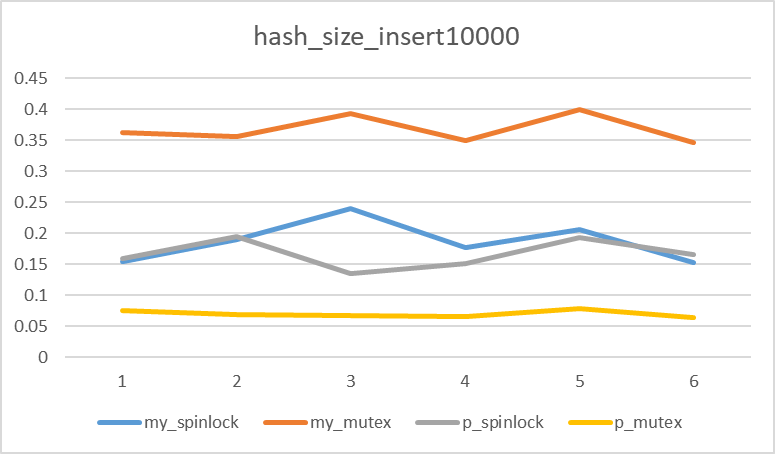


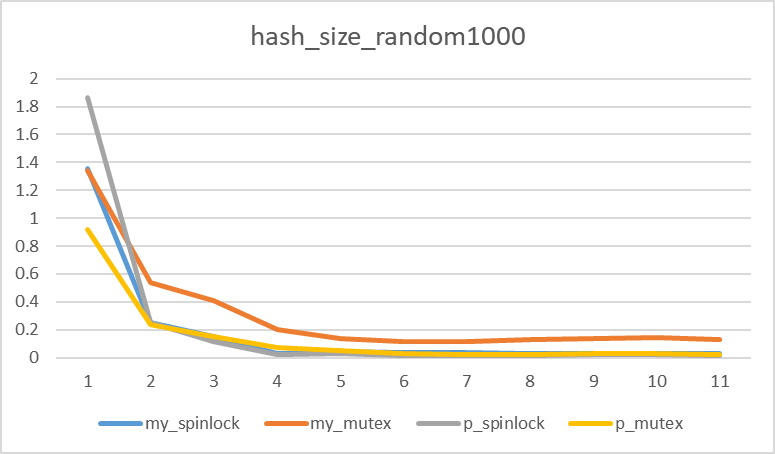
Hash表随机插入删除时，my\_spinlock性能最差，且随线程增多耗费时间显著增多

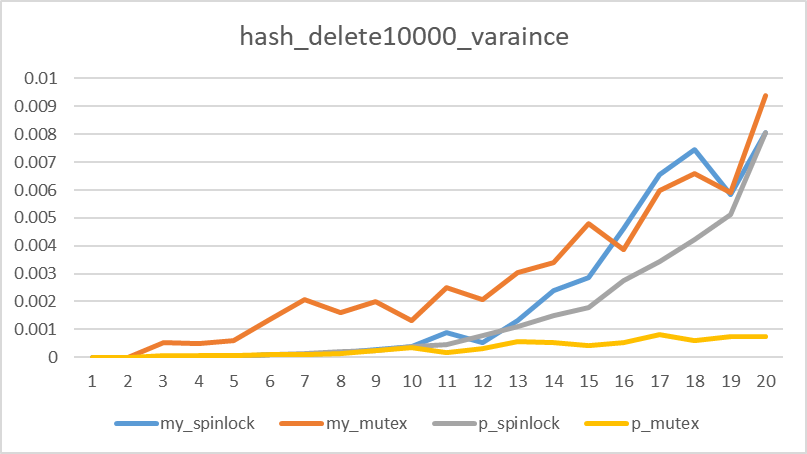
其余性能相差不大，且没有明显变化趋势

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

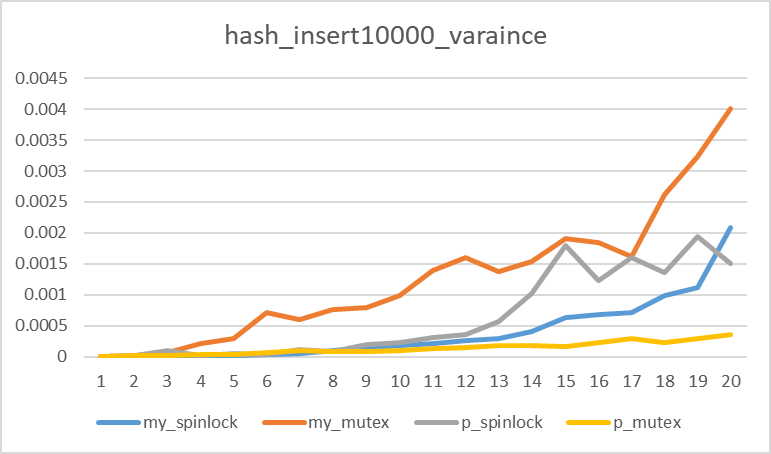




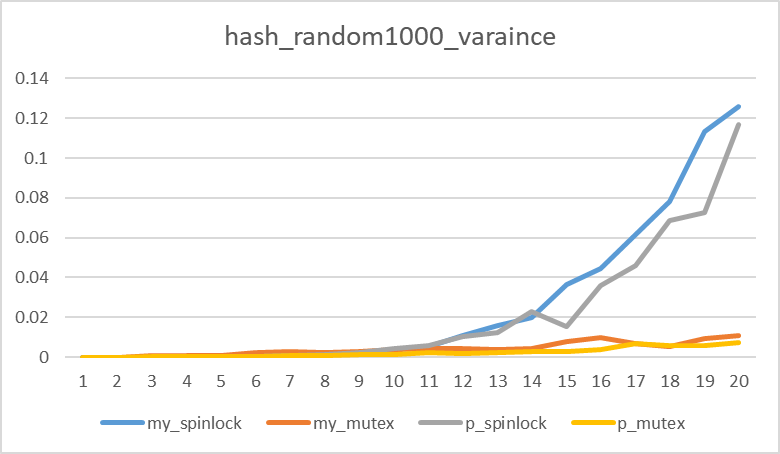




P\_mutex稳定性较高始终保持一定水平，而my\_spinlock,my\_mutex,p\_spinlock相差不大，都随线程增多有大幅度上升趋势。



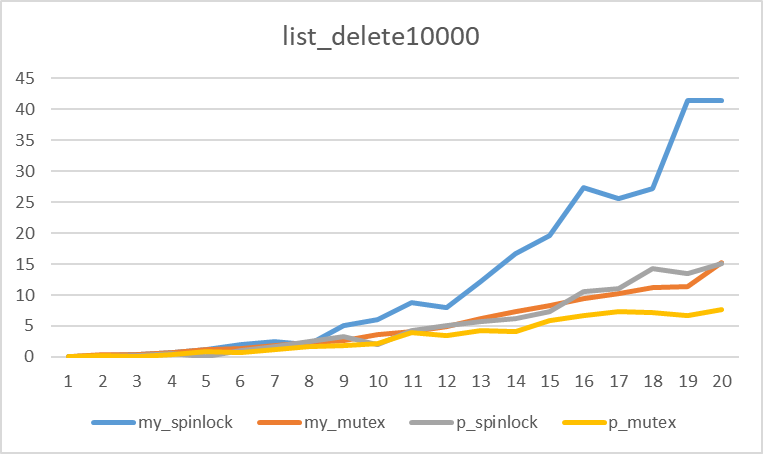
P\_mutex稳定性较高始终保持一定水平，而my\_spinlock, p\_spinlock相差不大，都随线程增多有一定程度上升趋势；而my\_mutex在短暂下降后呈现总体上升趋势。



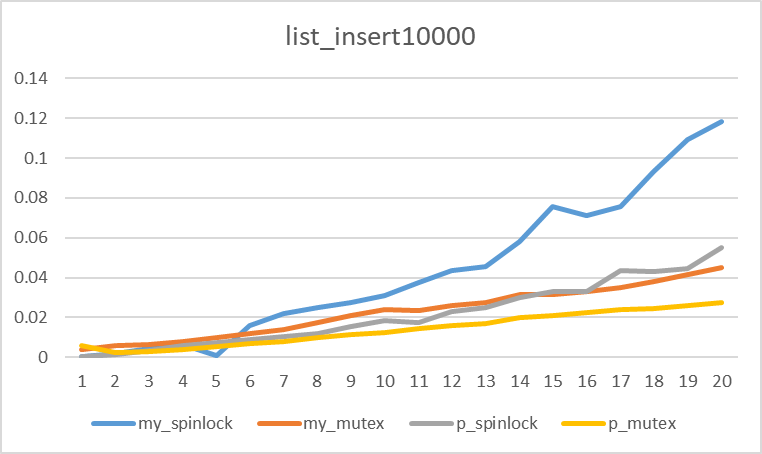
P\_mutex曲线相对平稳，保持与方差较小，公平性相对较好；

My\_spinlock与p\_spinlock曲线公平性较差，随线程增加上升‘

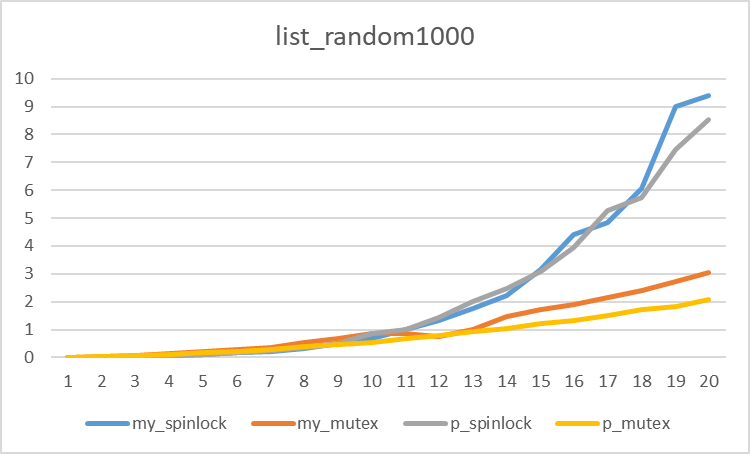
1. List



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；



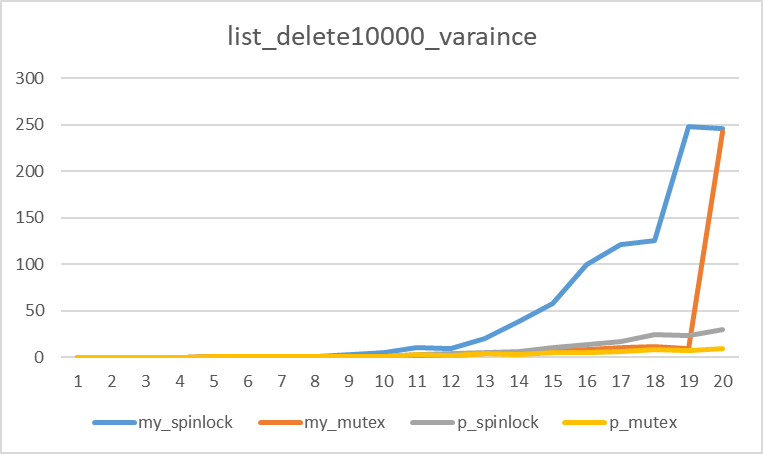
链表插入时，随进程增多，my\_spinlock后期上升明显，较其余三种耗时长，性能降低；my\_mutex性能要低于p\_mutex，线程越多，而这差距越大

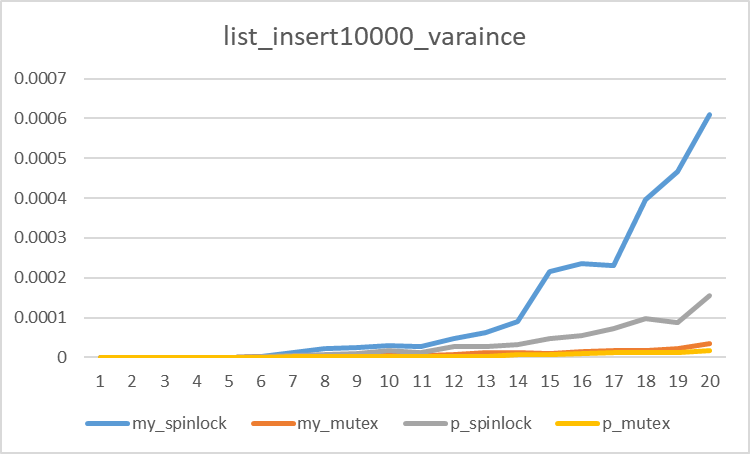


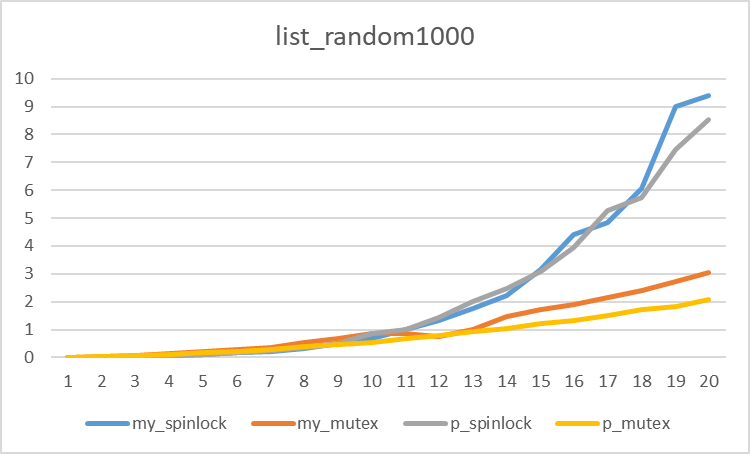
链表随机插入删除时，my\_mutex与p\_mutex性能明显优于my\_spinlock与

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）。

1. 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；

(2)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。

(3)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。

(4)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。

(5)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。

1. **Project3b：xv6 VM Layout**
2. 实验目的
3. To familiarize you with the xv6 virtual memory system.
4. To add a few new VM features to xv6 that are common in modern OSes.
5. 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.

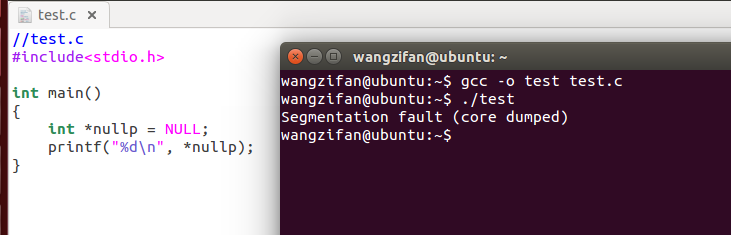
one thing you'll have to be very careful with is how xv6 currently tracks the size of a process's address space (currently with the sz field in the proc struct). There are a number of places in the code where this is used (e.g., to check whether an argument passed into the kernel is valid; to copy the 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.

1. 算法分析

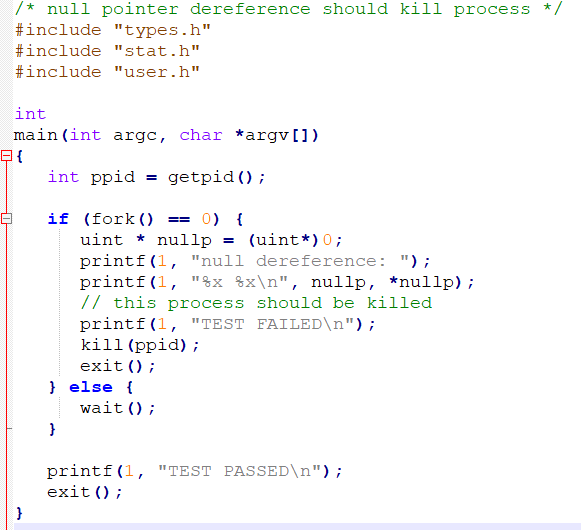
发现有两点需要实现：

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。

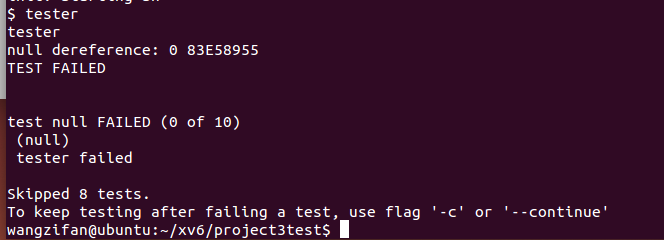
当在Linux对空指针进行解引时，发现会产生段错误：



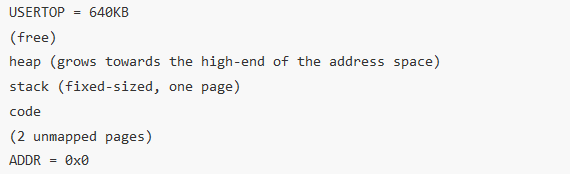
当在xv6对空指针进行解引时，发现空指针会解引到地址 0x0000的值，造成内存的外泄，测试的程序可以用测试数据中的 null.c：



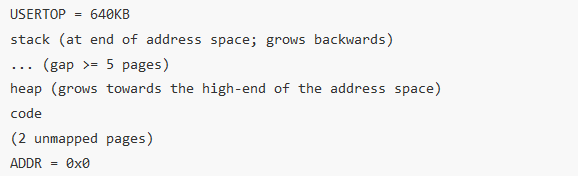
在qemu中运行，发现会输出对应地址0x0000的值：



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：



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：



1. 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)**。

// 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 process

**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\_size);

  //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)) && ((uint)i < (USERTOP - (proc->sz\_stack))))   //在栈和堆之间

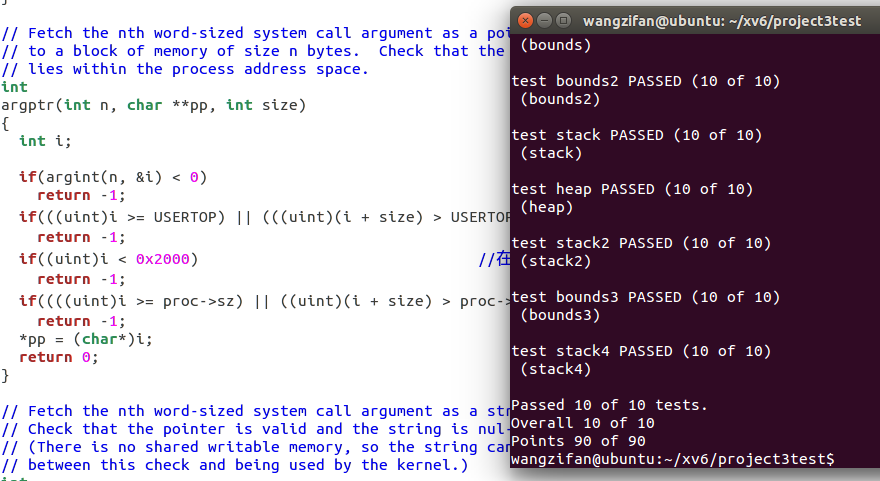
**return** -1;

  \*pp = (**char**\*)i;

**return** 0;

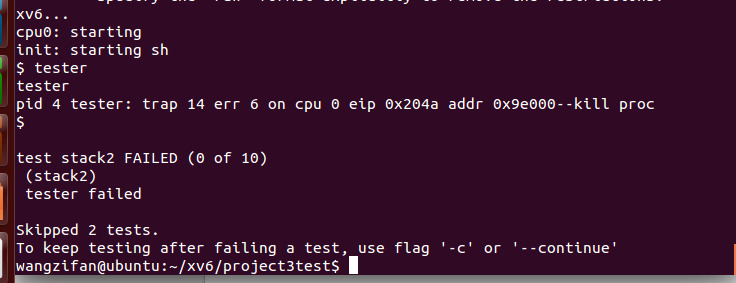
}

最后结果，测试样例通过：

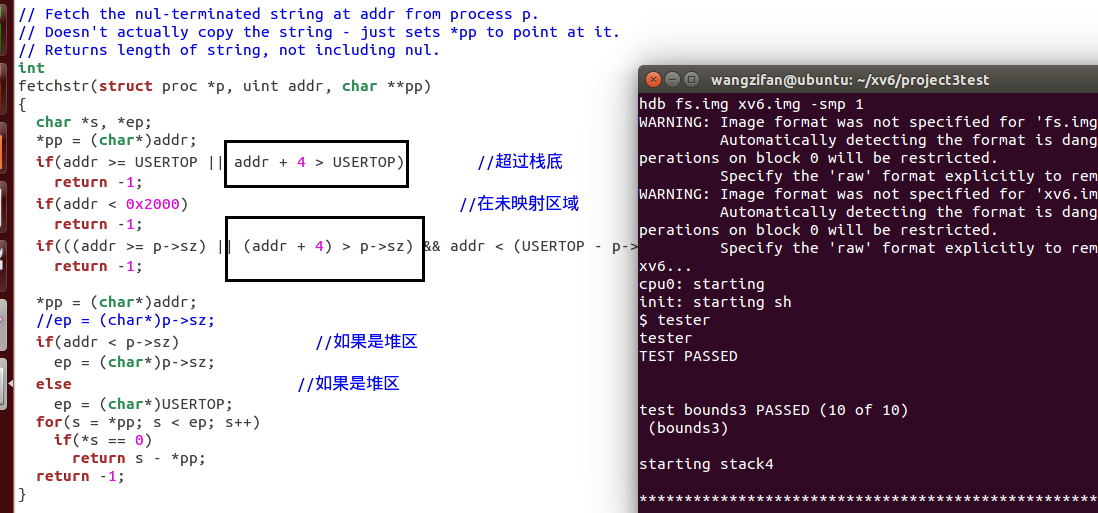


1. **问题总结**

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**。



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。



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。