哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2023 年秋季

课程名称：操作系统

实验名称：页表

实验性质：课内实验

实验时间：　　地点：

学生班级：计算机 6 班

学生学号：210110612

学生姓名：章懿

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 年 9 月

一、 回答问题

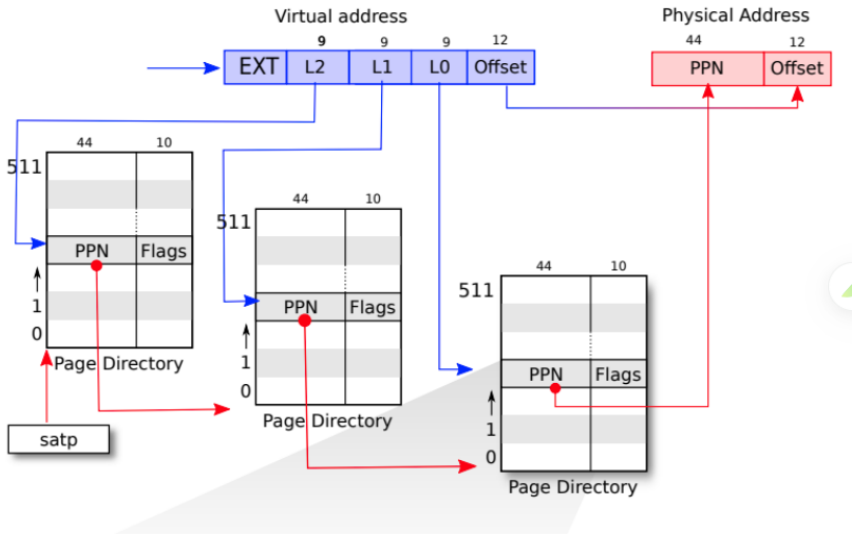
1. 查阅资料，简要阐述页表机制为什么会被发明，它有什么好处？

答：
1. 内存虚拟化：页表机制允许操作系统为每个进程创建一个独立的虚拟地址空间。这种虚拟化使得每个进程都认为它拥有一整块连续的内存，而实际上这些内存可能分散在物理内存的不同位置。这种机制简化了程序的编写和管理。
2. 内存保护：页表机制还可以提供内存保护，确保一个进程不能访问其他进程的内存。这种隔离保护了系统的稳定性和安全性。
3. 有效利用内存：页表机制通过允许物理内存的共享和重用，提高了内存的利用效率。例如，通过页交换（swapping）和分页（paging），可以将不常用的数据移出物理内存，为更需要内存的进程腾出空间。
4. 支持更大的地址空间：通过使用虚拟内存，页表机制允许系统支持远大于物理内存大小的地址空间，从而允许运行更大或更多的应用程序。
5. 简化内存管理：页表为操作系统提供了一种灵活且统一的方法来管理内存，无论物理内存的实际大小或配置如何。

2. 按照步骤，阐述 SV39 标准下，给定一个 64 位虚拟地址为 0xFFFFFFE789ABCDEF 的时候，是如何一步一步得到最终的物理地址的？（页表内容可以自行假设）

答：

已知查询过程如图：



由虚拟地址知：

L0=010111100=188

L1= 001001101=77

L2= 110011110=414

offset=0xDEF

故先寻找到 satp 所指向的根页表的第 414 项，得到次页表的基地址页帧号；

找到次页表的第 77 项，得到叶子页表的基地址页帧号；

找到叶子页表的第 188 项，得到对应的物理地址页帧号,假设为 0xAAAAAAAAAAA。

则最终的物理地址为：0xAAAAAAAAAAADEF

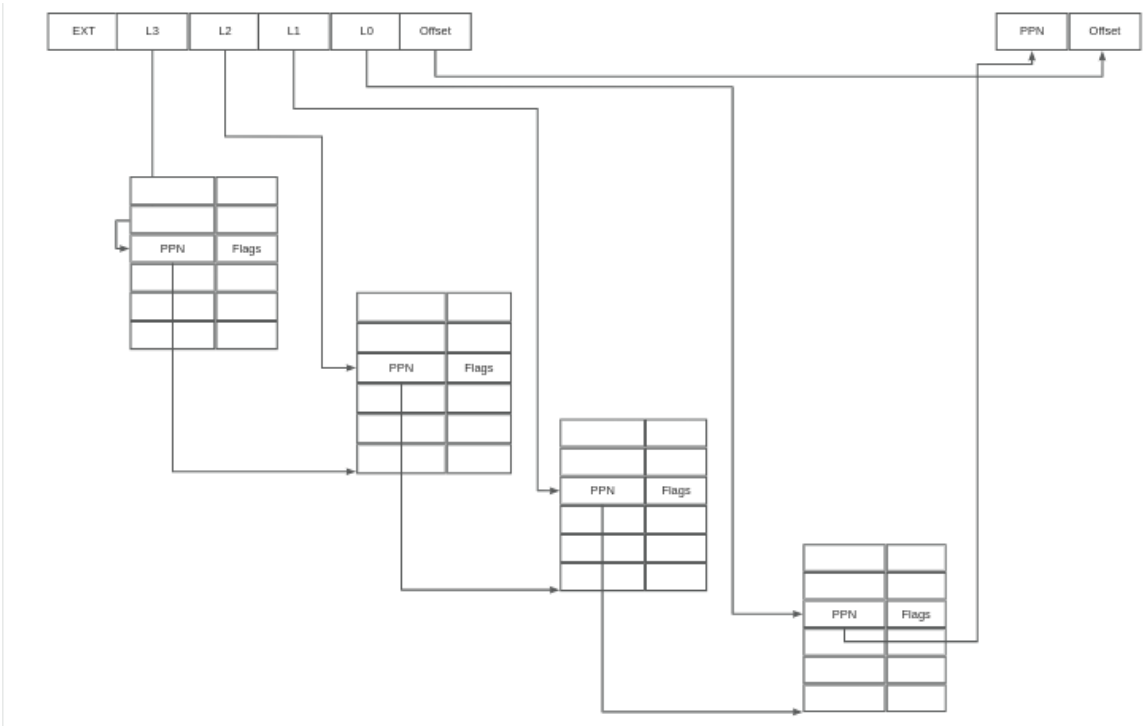
3. 我们注意到，SV39 标准下虚拟地址的 L2, L1, L0 均为 9 位。这实际上是设计中的必然结果，它们只能是 9 位，不能是 10 位或者是 8 位，你能说出其中的理由吗？（提示：一个页目录的大小必须与页的大小等大）

答：

已知一个页目录的大小必须与页的大小等大，每个页表项的大小是 8Bytes，每个页的大小为 4KB，故页表有 512 个页目录项，512 个目录项需要 9 位的索引，因此 L2，L1，L0 均为 9 位。

4. 在“实验原理”部分，我们知道 SV39 中的 39 是什么意思。但是其实还有一种标准，叫做 SV48，采用了四级页表而非三级页表，你能模仿“实验原理”部分示意图，画出 SV48 页表的数据结构和翻译的模式图示吗？（SV39 原图请参考指导书）

答：如图所示：



二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写。

任务一：

1. 首先把 vmprint()放在 kernel/vm.c 中

如图所示，采用了三个函数分别遍历根页表、次级页表和叶子页表：

```c
void vmgrandchildprint(pagetable_t pgtbl, uint64 L1, uint64 L2) {
  // 遍历页表项
  for(int i = 0; i < 512; i++) {
    pte_t pte = pgtbl[i]; //获取第i条PTE
    // /* 判断PTE的Flag位，若虚拟地址有映射到物理地址，则进行打印
    if(pte & PTE_V){
      // this PTE points to a lower-level page table.
      uint64 pa = PTE2PA(pte); // 将PTE转为为物理地址
      uint64 va = (pa & 0xFFF) + (L1 << 30) + (L2 << 21) + (i << 12);
      printf("||   ||   ||idx: %d: va: %p -> pa: %p, flags: ", i, va, pa);
      (pte & PTE_R) ? printf("r") : printf("-");
      (pte & PTE_W) ? printf("w") : printf("-");
      (pte & PTE_X) ? printf("x") : printf("-");
      (pte & PTE_U) ? printf("u") : printf("-");
      printf("\n");
    }
  }
}

void vmchildprint(pagetable_t pgtbl, uint64 L1) {
  // 遍历页表项
 for(int i = 0; i < 512; i++) {
    pte_t pte = pgtbl[i]; //获取第i条PTE

    /* 判断PTE的Flag位，如果还有下一级页表(即当前是次页表)，
       则调用vmgrandchildprint打印 */
    if(pte & PTE_V){
      // this PTE points to a lower-level page table.
      uint64 grandchild = PTE2PA(pte); // 将PTE转为为物理地址
      printf("||   ||idx: %d: pa: %p, flags: ----\n", i, grandchild);
      vmgrandchildprint((pagetable_t)grandchild, L1, (uint64)i); // 调用vmchildprint
    }
  }
}
void vmprint(pagetable_t pgtbl) {
  // 打印 vmprint 的参数，即获得的页表参数具体的值
  printf("page table %p\n", pgtbl);
  // 遍历页表项
  for(int i = 0; i < 512; i++) {
    pte_t pte = pgtbl[i]; //获取第i条PTE

    /* 判断PTE的Flag位，如果还有下一级页表(即当前是根页表)，
       则调用vmchildprint打印 */
    if(pte & PTE_V){
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
      printf("||idx: %d: pa: %p, flags: ----\n", i, child);
      vmchildprint((pagetable_t)child, (uint64)i); // 调用vmchildprint
    }
  }
}
```

在 kernel/defs.h 中定义 vmprint() 的接口

```c
// 添加释放进程的内核独立页表的函数
void            freegrandchildkpgtbl(pagetable_t pagetable);
void            freechildkpgtbl(pagetable_t pagetable);
void            freekpgtbl(pagetable_t pagetable);
```

任务二：

1. 修改 kernel/proc.h 中的 struct proc， 增加两个新成员：pagetable_t k_pagetable;和 uint64 kstack_pa;

```
    // 给每个进程中设置一个内核独立页表和内核栈的物理地址
    pagetable_t k_pagetable;
    uint64 kstack_pa;
  };
```

2. 仿照 kvminit() 函数重新写一个创建内核页表的函数

```
pagetable_t kvmsingleinit() {
  pagetable_t k_pagetable = (pagetable_t)kalloc();
  memset(k_pagetable, 0, PGSIZE);

  // uart registers
  kvmsinglemap(k_pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W);

  // virtio mmio disk interface
  kvmsinglemap(k_pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

  // 不要映射CLINT，否则会在任务三发生地址重合问题不要映射CLINT，否则会在任务三发生地址重合问题

  // PLIC
  kvmsinglemap(k_pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W);

  // map kernel text executable and read-only.
  kvmsinglemap(k_pagetable, KERNBASE, KERNBASE, (uint64)etext - KERNBASE, PTE_R | PTE_X);

  // map kernel data and the physical RAM we'll make use of.
  kvmsinglemap(k_pagetable, (uint64)etext, (uint64)etext, PHYSTOP - (uint64)etext, PTE_R | PTE_W);

  // map the trampoline for trap entry/exit to
  // the highest virtual address in the kernel.
  kvmsinglemap(k_pagetable, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

  return k_pagetable;
}
```

同时仿照 kvmmap 写一个完成映射的函数：

```
void kvmsinglemap(pagetable_t k_pagetable, uint64 va, uint64 pa, uint64 sz, int perm) {
  if (mappages(k_pagetable, va, sz, pa, perm) != 0) panic("kvmsinglemap");
}
```

3. 修改 procinit 函数

即增加了 p->kstack_pa = (uint64)pa;

把内核栈的物理地址 pa 拷贝到 PCB 新增的成员 kstack_pa 中

```
void procinit(void) {
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  for (p = proc; p < &proc[NPROC]; p++) {
    initlock(&p->lock, "proc");

    // Allocate a page for the process's kernel stack.
    // Map it high in memory, followed by an invalid
    // guard page.
    char *pa = kalloc();
    if (pa == 0) panic("kalloc");
    uint64 va = KSTACK((int)(p - proc));
    kvmmap(va, (uint64)pa, PGSIZE, PTE_R | PTE_W); // 保留内核栈在全局页表kernel_pagetable的映射
    p->kstack = va; // 在内核页表建立内核栈的映射
    p->kstack_pa = (uint64)pa; // 把内核栈的物理地址pa拷贝到PCB新增的成员kstack_pa中
  }
  kvminithart();
}
```

4. 修改 allocproc 函数

在 allocproc 函数中设置内核页表，并且将内核栈映射到页表 k_pagetable 中。

```
193    // 设置内核页表
194    pagetable_t k_pagetable = kvmsingleinit();
195
196    // 将内核栈映射到页表k_pagetable中
197    kvmsinglemap(k_pagetable, p->kstack, p->kstack_pa, PGSIZE, PTE_R | PTE_W);
198    p->k_pagetable = k_pagetable;
199
200    sync_pagetable(p); // 把进程的用户页表映射到内核页表中
201    return p;
202  }
```

5. 修改调度器（scheduler），使得切换进程的时候切换内核页表

在在进程切换的同时也要切换页表将其放入寄存器 satp 中，使用以下代码：

```
w_satp(MAKE_SATP(p->k_pagetable));
sfence_vma();
```

当目前没有进程运行的时候，scheduler() 应该要 satp 载入全局的内核页

具体操作如下图所示：

```
for (p = proc; p < &proc[NPROC]; p++) {
  acquire(&p->lock);
  if (p->state == RUNNABLE) {
    // Switch to chosen process.  It is the process's job
    // to release its lock and then reacquire it
    // before jumping back to us.
    p->state = RUNNING;
    c->proc = p;
    w_satp(MAKE_SATP(p->k_pagetable));
    sfence_vma();
    swtch(&c->context, &p->context);

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
    kvminithart();

    found = 1;
  }
  release(&p->lock);
}
```

6. 修改 freeproc()函数来释放对应的内核页表

编写了 freekpgtbl 函数来释放内核页表：

```
246  static void freeproc(struct proc *p) {
247    if (p->trapframe) kfree((void *)p->trapframe);
248    p->trapframe = 0;
249    if (p->pagetable) proc_freepagetable(p->pagetable, p->sz);
250    p->pagetable = 0;
251    p->sz = 0;
252    p->pid = 0;
253    p->parent = 0;
254    p->name[0] = 0;
255    p->chan = 0;
256    p->killed = 0;
257    p->xstate = 0;
258    p->state = UNUSED;
259    freekpgtbl(p->k_pagetable);
260  }
```

freekpgtbl 如下：

仍然使用三个函数分别释放根页表，次页表和叶子页表。

```c
void freegrandchildkpgtbl(pagetable_t pagetable) {
  // 将该叶子页表释放
  for(int i = 0; i < 512; i++) {
    pagetable[i] = 0; // 清零
  }
  kfree((void*)pagetable); // 释放pagetable指向的物理页
}

void freechildkpgtbl(pagetable_t pagetable) {
  for(int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i]; //获取第i条PTE
    if(pte & PTE_V) {
      /* 判断PTE的Flag位，如果还有下一级页表(即当前是次页表)，
       则调用释放页表项，并将对应的PTE清零 */
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
      freegrandchildkpgtbl((pagetable_t)child); // 调用freegrandchildkpgtbl
    }
    pagetable[i] = 0; // 清零
  }
  kfree((void*)pagetable); // 释放pagetable指向的物理页
}

void freekpgtbl(pagetable_t pagetable) {
  for(int i = 0; i < 512; i++) {
    pte_t pte = pagetable[i]; //获取第i条PTE
    if(pte & PTE_V) {
      /* 判断PTE的Flag位，如果还有下一级页表(即当前是根页表)，
       则调用释放页表项，并将对应的PTE清零 */
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte); // 将PTE转为为物理地址
      freechildkpgtbl((pagetable_t)child); // 调用freechildkpgtbl
    }
    pagetable[i] = 0; // 清零
  }
  kfree((void*)pagetable); // 释放pagetable指向的物理页
}
```

任务三：

1.

写一个 sync_pagetable 函数把进程的用户页表映射到内核页表中，同时在 defs.h 中声明。

```c
//该函数将进程的用户页表映射到内核页表中

int
sync_pagetable(pagetable_t uvm, pagetable_t kvm, uint64 old_sz, uint64 new_sz){
 pte_t *pte;
 uint64 pa, i;
 uint flags;

 old_sz = PGROUNDUP(old_sz);
 if (new_sz <= old_sz) return 0;

 for(i = old_sz; i < new_sz; i += PGSIZE){
  if((pte = walk(uvm, i, 0)) == 0) //找到PTE的物理地址
   panic("sync_pagetable: pte should exist");
  if((*pte & PTE_V) == 0)
   panic("sync_pagetable: page not present");

  // 清除PTE_U的标记位
  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);
  if(mappages(kvm, i, PGSIZE, pa, flags&(~PTE_U)) != 0){ //调用proc_mappages完成映射,并保存相关信息
   return -1;
  }
 }
 return 0;
}
```

```c
111  int             sync_pagetable(pagetable_t, pagetable_t, uint64, uint64);
```

2. 用函数 copyin_new() （在 kernel/vmcopyin.c 中定义）代替 copyin()，copyinstr_new() 以代替 copyinstr()

因为修改了 PTE_U 标记位，故直接替换即可：

```c
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  return copyin_new(pagetable, dst, srcva, len);
```

```c
int
copyinstr(pagetable_t pagetable, char *dst, uint64 srcva, uint64 max)
{
  return copyinstr_new(pagetable, dst, srcva, max);
```

3. 在 fork()、exec()和 growproc()这三个函数里将改变后的进程页表同步

fork()中：

```c
435   //独立内核页表加上用户页表的映射
436   // 父进程用户空间的页表也全部拷贝一遍给子进程
437   if(sync_pagetable(np->pagetable, np->k_pagetable, 0, np->sz) < 0){
438     freeproc(np);
439     release(&np->lock);
440     return -1;
441   }
```

exec()中：

需要删除旧的映射并将新页面映射到内核页表：

```
124     // 修改独立内核页表
125     //删除旧的映射并将新页面映射到内核页表
126     uvmunmap(p->k_pagetable, 0, PGROUNDUP(oldsz)/PGSIZE, 0);
127     if(sync_pagetable(p->pagetable, p->k_pagetable, 0, p->sz) < 0)
128       goto bad;
129
130
131     return argc; // this ends up in a0, the first argument to main(argc, argv)
132
133   bad:
134     if(pagetable)
135       proc_freepagetable(pagetable, sz);
136     if(ip){
137       iunlockput(ip);
138       end_op();
139     }
140     return -1;
141   }
```

growproc()中：

```
367 int growproc(int n) {
368   uint sz;
369   struct proc *p = myproc();
370
371   sz = p->sz;
372   if(n > 0){
373     if((sz = uvmalloc(p->pagetable, sz, sz + n)) == 0) {
374       return -1;
375     }
376     // mapper user page to kernel page table
377     if((sync_pagetable(p->pagetable, p->k_pagetable, p->sz, sz)) < 0){
378       return -1;
379     }
380   } else if(n < 0){
381     sz = uvmdealloc(p->pagetable, sz, sz + n);
382     uvmunmap(p->k_pagetable,PGROUNDUP(sz),(PGROUNDUP(p->sz)-PGROUNDUP(sz))/PGSIZE,0);
383   }
384   p->sz = sz;
385
386   return 0;
387 }
```

4. 最后将第一个进程也将用户页表映射到内核页表中

```
335 void userinit(void) {
336   struct proc *p;
337
338   p = allocproc();
339   initproc = p;
340
341   // allocate one user page and copy init's instructions
342   // and data into it.
343   uvminit(p->pagetable, initcode, sizeof(initcode));
344   p->sz = PGSIZE;
345
346   // prepare for the very first "return" from kernel to user.
347   p->trapframe->epc = 0;      // user program counter
348   p->trapframe->sp = PGSIZE;  // user stack pointer
349
350   safestrcpy(p->name, "initcode", sizeof(p->name));
351   p->cwd = namei("/");
352
353   p->state = RUNNABLE;
354
355   //独立内核页表加上用户页表的映射
356   //user2kernelpage(p);
357   // 包含第一个进程的用户页表
358   sync_pagetable(p->pagetable, p->k_pagetable,0,PGSIZE) ;
359
360   release(&p->lock);
361 }
```

三、 实验结果截图

请给出任务一、任务二、任务三和 make grade 的运行结果截图。

任务一：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
page table 0x0000000087f6e000
||idx: 0: pa: 0x0000000087f6a000, flags: ----
||   ||idx: 0: pa: 0x0000000087f69000, flags: ----
||   ||   ||idx: 0: va: 0x0000000000000000 -> pa: 0x0000000087f6b000, flags: rwxu
||   ||   ||idx: 1: va: 0x0000000000001000 -> pa: 0x0000000087f68000, flags: rwx-
||   ||   ||idx: 2: va: 0x0000000000002000 -> pa: 0x0000000087f67000, flags: rwxu
||idx: 255: pa: 0x0000000087f6d000, flags: ----
||   ||idx: 511: pa: 0x0000000087f6c000, flags: ----
||   ||   ||idx: 510: va: 0x0000003fffffe000 -> pa: 0x0000000087f76000, flags: rw--
||   ||   ||idx: 511: va: 0x0000003ffffff000 -> pa: 0x0000000080007000, flags: r-x-
init: starting sh
$
```

```
== Test pte printout == pte printout: OK (1.9s)
```

任务二：

```
$ kvmtest
kvmtest: start
kvmtest: OK
```

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3235
            sepc=0x0000000000005406 stval=0x0000000000005406
usertrap(): unexpected scause 0x000000000000000c pid=3236
            sepc=0x0000000000005406 stval=0x0000000000005406
OK
```

```
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: OK
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

任务三：

usertests：

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
```

```
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

make grade 如下：

```
make[1]: 离开目录“/home/yikise/OS_hitsz”
== Test pte printout ==
$ make qemu-gdb
pte printout: OK (1.7s)
== Test count copyin ==
$ make qemu-gdb
count copyin: OK (1.1s)
    (Old xv6.out.count failure log removed)
== Test kernel pagetable test ==
$ make qemu-gdb
kernel pagetable test: OK (1.0s)
    (Old xv6.out.kvm failure log removed)
== Test usertests ==
$ make qemu-gdb
(76.6s)
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: all tests ==
  usertests: all tests: OK
Score: 100/100
yikise@yikise-Lenovo-XiaoXinPro-14IHU-2021:~/OS_hitsz$
```

四、 实验总结

请总结 xv6 4 个实验的收获，给出对 xv6 实验内容的建议。
注：本节为酌情加分项。

1. util 实验：
   - 收获：util 实验通常涵盖与进程管理和系统调用相关的基本概念。了解如何创建和管理进程，以及如何使用 xv6 的系统调用接口。
2. syscall 实验：

- 收获：syscall 实验涵盖操作系统中的系统调用概念，以及如何在内核中添加新的系统调用。了解了如何扩展操作系统的功能。

3. lock 实验：

- 收获：lock 实验通常涵盖并发编程的基本概念，包括锁的概念和使用。学会了如何处理并发访问共享资源的问题。

4. pgtbl 实验：

- 收获：pgtbl 实验通常涵盖虚拟内存管理的概念，包括页表的创建和管理。了解了如何将虚拟地址映射到物理地址。