Lab 5: xv6 lazy page allocation

1950787 杨鑫

实验目的：

- 了解 lazy allocation 的原理和机制
- 通过修改操作系统内部文件实现lazy allocation
- 体会 lazy allocaion 的必要性和优点

实验步骤：

首先切换分支至 lazy branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout lazy
make clean
```

a. Eliminate allocation from sbrk

实验目的：

您的第一个任务是从 sbrk(n) 系统调用实现中删除页面分配，这是在 sysproc.c 中的函数 sys_sbrk() 里。 sbrk(n) 系统调用将进程的内存大小增加 n 字节，然后返回新分配区域的开始（即旧大小）。 您的新 sbrk(n) 应该只是将进程的大小 (myproc()->sz) 增加 n 并返回旧大小。 它不应该分配内存——所以你应该删除对 growproc() 的调用（但你仍然需要增加进程的大小）。

实验分析：

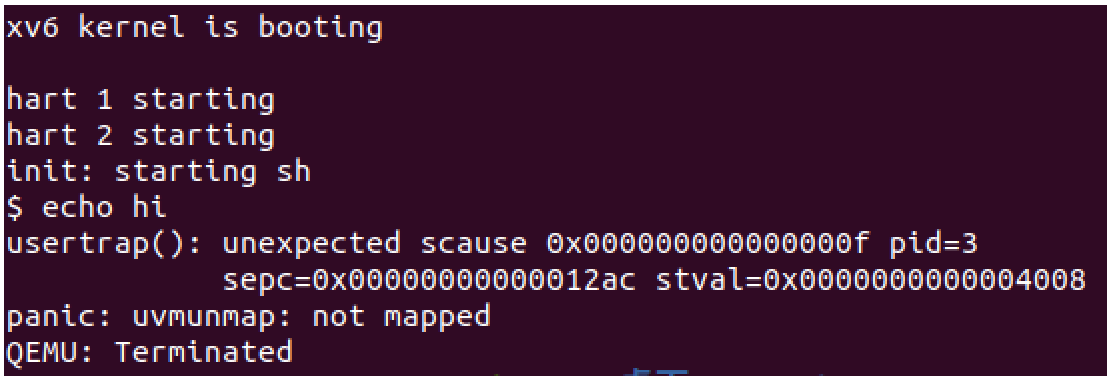
该实验只需要将 sbrk() 里面的实现增加内存大小的部分删除即可，但是要保证他的大小要发生改变。同时需要正常执行缩小内存的部分。

实验核心代码：

sysproc.c

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  struct proc *p = myproc();

  if(n < 0) {
    uvmunmap(p->pagetable, PGROUNDUP(addr + n), (PGROUNDUP(addr) - 
PGROUNDUP(addr + n)) / PGSIZE, 1);
  }

  p->sz += n;
  return addr;
}
```

测试：



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
QEMU: Terminated
```

可以看到，usertrap() 捕获了一个它不知道如何处理的异常。与示例结果一致。

b. Lazy allocation

实验目的：

修改trap.c中的代码，通过将新分配的物理内存页面映射到故障地址，然后返回用户空间让进程继续执行，来响应来自用户空间的页面故障。 您应该在产生“usertrap(): ...”消息的 printf 调用之前添加代码。修改您需要的任何其他 xv6 内核代码以使 echo hi 工作。

实验分析：

在 usertrap() 里面，当检测到 page fault 的时候（即 r_scause() 为 13 或者 15）为虚拟地址分配物理页并添加映射；同时在 uvmunmap() 里销毁进程时，对于尚未分配实际物理页的虚拟地址不做处理。

实验核心代码：

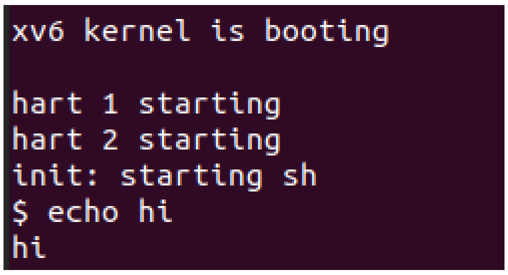
trap.c

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if (r_scause() == 13 || r_scause() == 15) {
    uint64 stval = r_stval();
    if (stval >= p->sz) p->killed = 1;
    else {
      if (PGROUNDDOWN(p->trapframe->sp) != PGROUNDUP(stval)) {
        char *mem = kalloc();
        if(mem == 0){
          p->killed = 1;
        } else {
          memset(mem, 0, PGSIZE);
          if(mappages(p->pagetable, PGROUNDDOWN(stval), PGSIZE, (uint64)mem,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            p->killed = 1;
          }
        }
      } else p->killed = 1;
    }

  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

vm.c

```
// Remove npages of mappings starting from va. va must be
// page-aligned. The mappings must exist.
// Optionally free the physical memory.
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      continue;
    if((*pte & PTE_V) == 0)
      continue;
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

测试：



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo hi
hi
```

可以看到，现在已经可以正常执行 echo 指令，说明测试成功

c. Lazytests and Usertests

实验目的：

我们为您提供了lazytests，这是一个xv6 用户程序，用于测试一些可能对您的惰性内存分配器造成压力的特定情况。 修改你的内核代码，让所有的惰性测试和用户测试都通过。

要求：

- 处理负数的 sbrk() 参数。
- 如果一个进程在高于任何使用 sbrk() 分配的虚拟内存地址上发生页面错误，则终止该进程。
- 正确处理 fork() 中的父子内存副本。
- 处理进程将有效地址从 sbrk() 传递给系统调用（例如读或写），但尚未分配该地址的内存的情况。
- 正确处理内存不足：如果 kalloc() 在页面错误处理程序中失败，则终止当前进程。
- 处理用户堆栈下方无效页面上的错误。

实验分析：

首先对于 sbrk 参数为负数时，则按之前的支持缩小内存；在 copyout 和 copyin 使用到尚未分配的物理页的虚拟地址时，需要按需分配物理页面；还需处理在用户堆栈下方发生页面失效的错误。

实验核心代码：

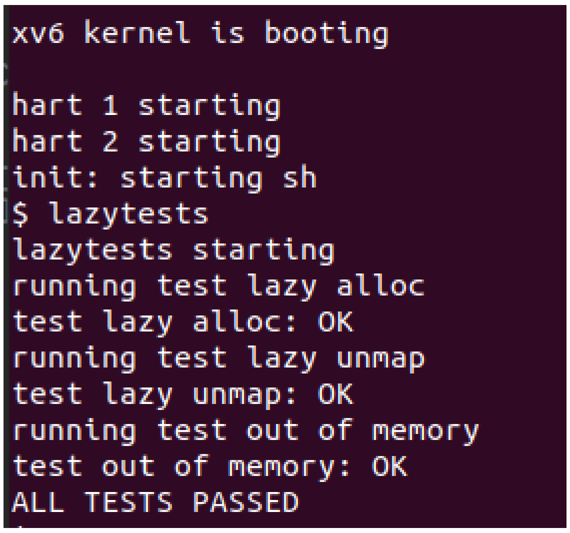
trap.c

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if (r_scause() == 13 || r_scause() == 15) {
    uint64 stval = r_stval();
    if (stval >= p->sz) p->killed = 1;
    else {
      if (PGROUNDDOWN(p->trapframe->sp) != PGROUNDUP(stval)) {
        char *mem = kalloc();
        if(mem == 0){
          p->killed = 1;
        } else {
          memset(mem, 0, PGSIZE);
          if(mappages(p->pagetable, PGROUNDDOWN(stval), PGSIZE, (uint64)mem,
PTE_W|PTE_X|PTE_R|PTE_U) != 0){
            p->killed = 1;
          }
        }
      } else p->killed = 1;
    }

  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

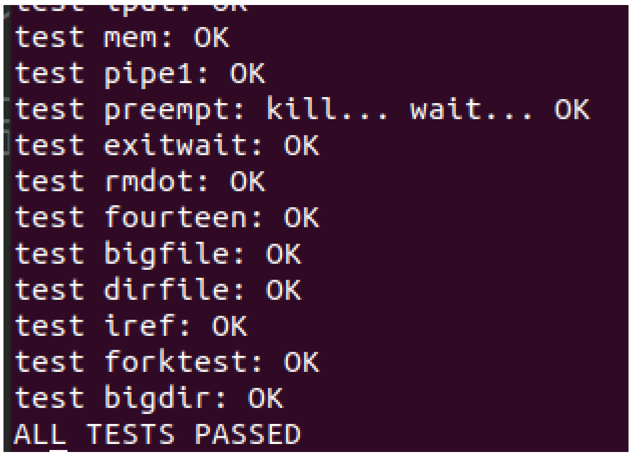
vm.c

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0) {
      if (dstva >= myproc()->sz) return -1;
      char* mem = kalloc();
      pa0 = (uint64)mem;
      memset(mem, 0, PGSIZE);
      mappages(pagetable, va0, PGSIZE, pa0, PTE_W|PTE_R|PTE_U|PTE_X);
    }
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}


// Copy from user to kernel.
// Copy len bytes to dst from virtual address srcva in a given page table.
// Return 0 on success, -1 on error.
int
copyin(pagetable_t pagetable, char *dst, uint64 srcva, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(srcva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0) {
      if (srcva >= myproc()->sz) return -1;
        char* mem = kalloc();
        pa0 = (uint64)mem;
        memset(mem, 0, PGSIZE);
        mappages(pagetable, va0, PGSIZE, pa0, PTE_W|PTE_R|PTE_U|PTE_X);
    }
    n = PGSIZE - (srcva - va0);
    if(n > len)
      n = len;
    memmove(dst, (void *)(pa0 + (srcva - va0)), n);

    len -= n;
    dst += n;
    srcva = va0 + PGSIZE;
  }
  return 0;
}
```

测试：



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
```

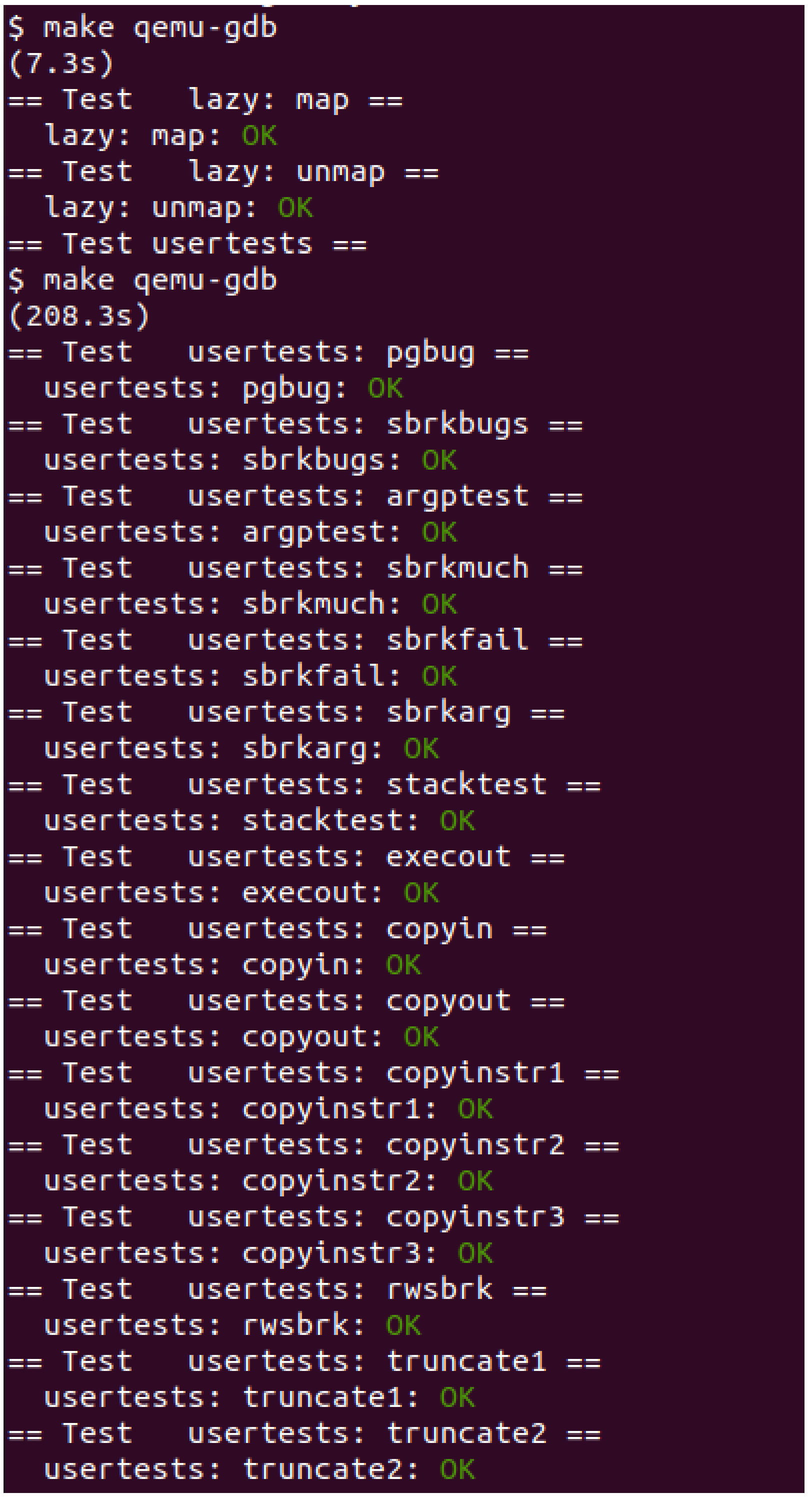


```
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
```

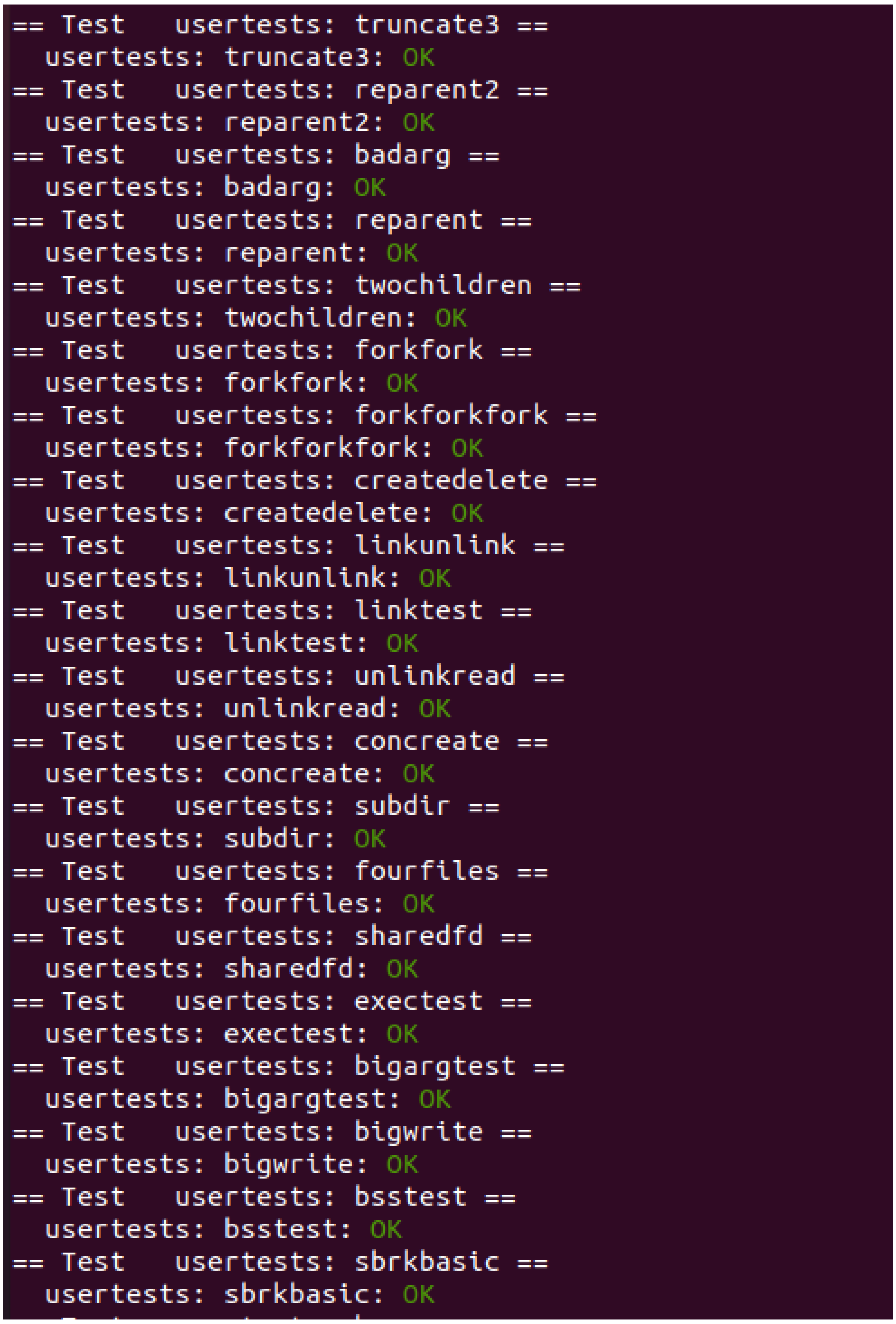
可以通过 lazytests 和 usertests

实验评分：

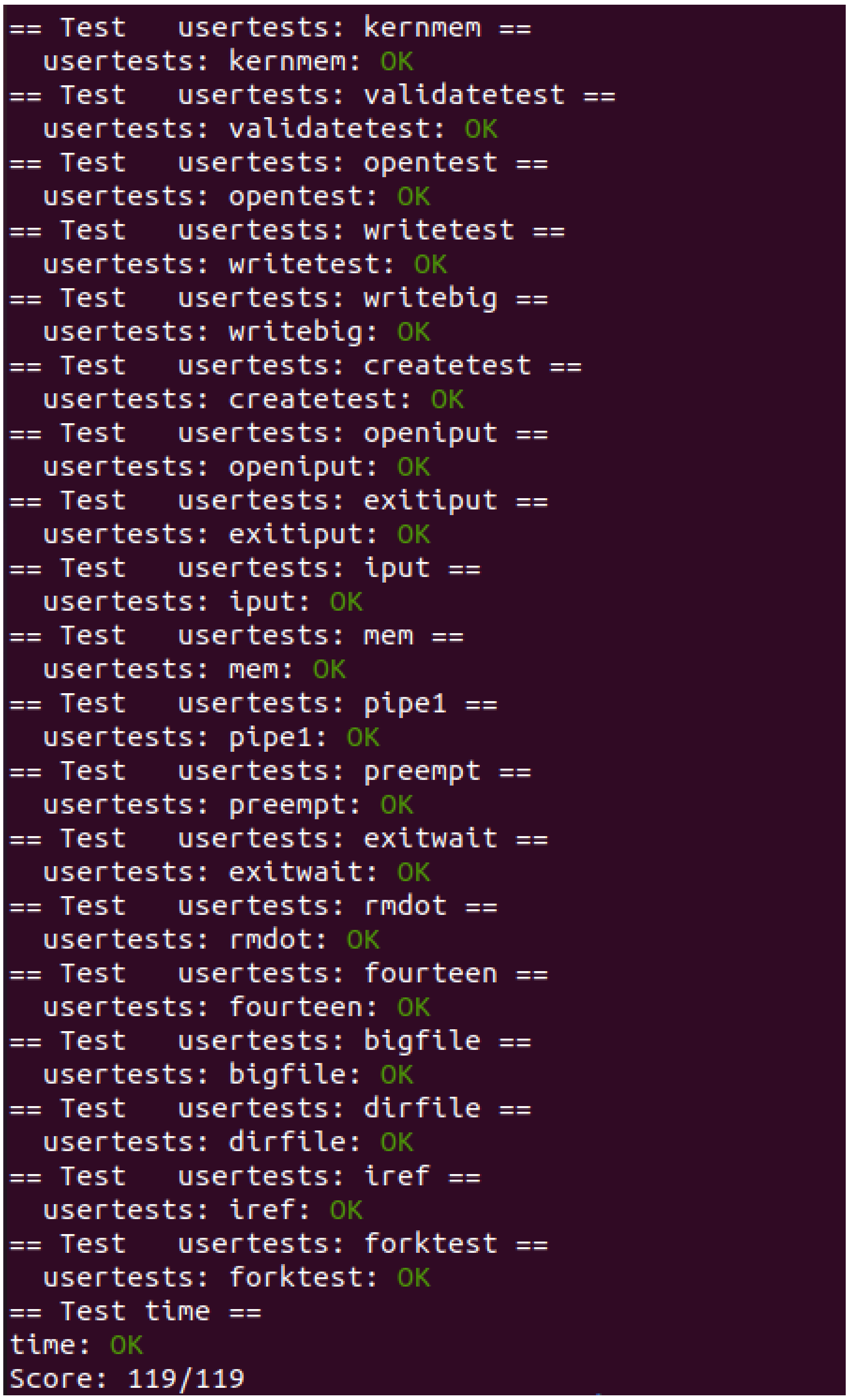
按照实验要求对本次实验的所有小实验进行评分，结果如图：



```
$ make qemu-gdb
(7.3s)
== Test   lazy: map ==
  lazy: map: OK
== Test   lazy: unmap ==
  lazy: unmap: OK
== Test usertests ==
$ make qemu-gdb
(208.3s)
== Test   usertests: pgbug ==
  usertests: pgbug: OK
== Test   usertests: sbrkbugs ==
  usertests: sbrkbugs: OK
== Test   usertests: argptest ==
  usertests: argptest: OK
== Test   usertests: sbrkmuch ==
  usertests: sbrkmuch: OK
== Test   usertests: sbrkfail ==
  usertests: sbrkfail: OK
== Test   usertests: sbrkarg ==
  usertests: sbrkarg: OK
== Test   usertests: stacktest ==
  usertests: stacktest: OK
== Test   usertests: execout ==
  usertests: execout: OK
== Test   usertests: copyin ==
  usertests: copyin: OK
== Test   usertests: copyout ==
  usertests: copyout: OK
== Test   usertests: copyinstr1 ==
  usertests: copyinstr1: OK
== Test   usertests: copyinstr2 ==
  usertests: copyinstr2: OK
== Test   usertests: copyinstr3 ==
  usertests: copyinstr3: OK
== Test   usertests: rwsbrk ==
  usertests: rwsbrk: OK
== Test   usertests: truncate1 ==
  usertests: truncate1: OK
== Test   usertests: truncate2 ==
  usertests: truncate2: OK
```



```
== Test   usertests: truncate3 ==
  usertests: truncate3: OK
== Test   usertests: reparent2 ==
  usertests: reparent2: OK
== Test   usertests: badarg ==
  usertests: badarg: OK
== Test   usertests: reparent ==
  usertests: reparent: OK
== Test   usertests: twochildren ==
  usertests: twochildren: OK
== Test   usertests: forkfork ==
  usertests: forkfork: OK
== Test   usertests: forkforkfork ==
  usertests: forkforkfork: OK
== Test   usertests: createdelete ==
  usertests: createdelete: OK
== Test   usertests: linkunlink ==
  usertests: linkunlink: OK
== Test   usertests: linktest ==
  usertests: linktest: OK
== Test   usertests: unlinkread ==
  usertests: unlinkread: OK
== Test   usertests: concreate ==
  usertests: concreate: OK
== Test   usertests: subdir ==
  usertests: subdir: OK
== Test   usertests: fourfiles ==
  usertests: fourfiles: OK
== Test   usertests: sharedfd ==
  usertests: sharedfd: OK
== Test   usertests: exectest ==
  usertests: exectest: OK
== Test   usertests: bigargtest ==
  usertests: bigargtest: OK
== Test   usertests: bigwrite ==
  usertests: bigwrite: OK
== Test   usertests: bsstest ==
  usertests: bsstest: OK
== Test   usertests: sbrkbasic ==
  usertests: sbrkbasic: OK
```



```
== Test   usertests: kernmem ==
  usertests: kernmem: OK
== Test   usertests: validatetest ==
  usertests: validatetest: OK
== Test   usertests: opentest ==
  usertests: opentest: OK
== Test   usertests: writetest ==
  usertests: writetest: OK
== Test   usertests: writebig ==
  usertests: writebig: OK
== Test   usertests: createtest ==
  usertests: createtest: OK
== Test   usertests: openiput ==
  usertests: openiput: OK
== Test   usertests: exitiput ==
  usertests: exitiput: OK
== Test   usertests: iput ==
  usertests: iput: OK
== Test   usertests: mem ==
  usertests: mem: OK
== Test   usertests: pipe1 ==
  usertests: pipe1: OK
== Test   usertests: preempt ==
  usertests: preempt: OK
== Test   usertests: exitwait ==
  usertests: exitwait: OK
== Test   usertests: rmdot ==
  usertests: rmdot: OK
== Test   usertests: fourteen ==
  usertests: fourteen: OK
== Test   usertests: bigfile ==
  usertests: bigfile: OK
== Test   usertests: dirfile ==
  usertests: dirfile: OK
== Test   usertests: iref ==
  usertests: iref: OK
== Test   usertests: forktest ==
  usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119
```

通过了所有测试

问题以及解决办法：

a. 为何要实现 lazy allocation

当一个进程执行期间所涵盖的内存特别大的时候，但是在一定时间段内使用的内存数又比较少，在这种情况下如果一次性分配所有页面，会造成很大的时间开销和空间开销，甚至会影响其他进程的并发执行效率。所以采用这种按需分配的 lazy allocation 思想，这样会改善程序的执行性能。

b. lazy allocation 的注意事项

在进程使用 lazy allocation 的页面时，会产生 page fault，应当在这时为其分配物理页面并添加映射以解决这个错误。但是如果这个 page fault 的虚拟地址比 sbrk 分配的大的话，则应当杀死此进程。

实验心得：

1. 理解了 lazy allocation 的基本实现，其大致思想是在用户请求分配内存时，只是增加其虚拟地址空间大小，并不实际分配物理内存（sys_sbrk中），然后在进程实际运行时，当使用到未分配物理内存的页面时，会发生异常，从而进入陷阱中，寄存器 scause 指明了异常的种类，stval指明了出错的虚拟地址，对于页面错误，在此时为其分配物理内存并添加页表映射（但是还要对各种异常情况进行监测，如内存分配失败、添加映射失败等），此时再返回用户空间，然后进程继续执行，这是陷阱在 lazy allocation 的作用，同时也让我对处理内存的相关函数有了更多的熟悉。
2. 对于陷阱和内存分配等模块有了更深的认知和了解。