Lab5 xv6 lazy page allocation

- 4）实验心得

O/S在使用页表硬件时可以使用的许多巧妙技巧之一是延迟分配用户空间堆内存。Xv6应用程序使用 `sbrk()` 系统调用向内核请求堆内存。在我们给您的内核中， `sbrk()` 分配物理内存并将其映射到进程的虚拟地址空间中。内核为大型请求分配和映射内存可能需要很长时间。例如，考虑一个千兆字节由262144个4096字节的页面组成；这是一个巨大的分配数量，即使每个分配都很便宜。此外，一些程序分配的内存比实际使用的内存多（例如，用于实现稀疏阵列），或者在使用之前就分配好内存。在这些情况下，为了让 `sbrk()` 更快地完成，复杂的内核会惰性地分配用户内存。也就是说， `sbrk()` 不分配物理内存，只是记住分配了哪些用户地址，并在用户页表中将这些地址标记为无效。当进程第一次尝试使用任何给定的延迟分配内存页面时，CPU会生成一个页面错误，内核会通过分配物理内存、将其归零并映射来处理该错误。您将在本实验室中将此延迟分配功能添加到xv6中。

1. Eliminate allocation from sbrk() (easy)

1）实验目的

> Your first task is to delete page allocation from the `sbrk(n)` system call implementation, which is the function `sys_sbrk()` in sysproc.c. The `sbrk(n)` system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new `sbrk(n)` should just increment the process's size (myproc()->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to `growproc()` (but you still need to increase the process's size!).
> 您的第一个任务是从 `sbrk(n)` 系统调用实现中删除页面分配，该调用是sysproc.c中的函数 `sys_sbrk()` 。 `sbrk(n)` 系统调用将进程的内存大小增加n个字节，然后返回新分配区域的起始位置（即旧大小）。新的 `sbrk(n)` 应该只将进程的大小（myproc（）->sz）增加n，然后返回旧的大小。它不应该分配内存，所以应该删除对 `growproc()` 的调用（但仍然需要增加进程的大小！）
> 启动xv6，然后在shell中键入 `echo hi` 。 您应该会看到以下内容：

```
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000001258 stval=0x0000000000004008
va=0x0000000000004000 pte=0x0000000000000000
panic: uvmunmap: not mapped
```

> “ `usertrap（）：...`”消息来自用户陷阱处理程序 `trap.c` ；它捕获了一个不知道如何处理的异常。确保您了解为什么发生此页面错误。“ stval = 0x0..04008”指示导致页面错误的虚拟地址为0x4008

2）实验步骤

编写代码

在 kernel/sysproc.c 文件中修改 sys_sbrk() 函数, 删去对 growproc() 函数调用，并为 myproc()->sz 增加 n

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

  if(argint(0, &n) < 0)
    return -1;

  // lab5-1
  struct proc* p = myproc();
  addr = p->sz;
  p->sz += n;
  // 错误处理
  if (n < 0) {
    p->sz = uvmdealloc(p->pagetable, addr, addr + n);
  }
  // if(growproc(n) < 0)
  //   return -1;
  return addr;
}
```

测试程序

在存放 Makefile 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 echo hi 进行测试

```
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012ac stval=0x0000000000004008
panic: uvmunmap: not mapped
```

结果符合预期

3）实验中遇到的问题和解决方法

没什么问题，操作比较简单，提示也非常明确

4）实验心得

这里出现 `usertrap()` 是因为还没有进行实际的内存分配，知识虚晃一枪给 `p->sz` 增加了而已， `growproc(n)` 也没有调用

2. Lazy allocation (moderate)

1）实验目的

> Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the `printf` call that produced the "usertrap(): ..." message. Modify whatever other xv6 kernel code you need to in order to get `echo hi` to work.
> 修改trap.c中的代码以响应来自用户空间的页面错误，方法是在错误地址映射新分配的物理内存页面，然后返回到用户空间，让进程继续执行。您应该在生成“usertrap（）：…”消息的 `printf` 调用之前添加代码。为了让 `echo hi` 正常工作，您需要修改任何其他xv6内核代码。

2）实验步骤

编写代码

处理page fault

根据提示, 当 `r_scause() == 13 or 15` 即表示一个page fault。同时， `r_stval()` 返回
在 `kernel/trap.c` 文件中的 `usertrap()` 函数进行page fault的处理

```
void
usertrap(void)
{
  ...
  if(r_scause() == 8){
    ...
  }
  // lab5-2 page fault的处理
  else if (r_scause() == 13 || r_scause() == 15) {
    char* pa;
    // 分配物理页面
    if ((pa = kalloc()) != 0) {
      // 用0填充
      memset(pa, 0, PGSIZE);
      // 引发page fault的虚拟地址向下取整
      uint64 va = PGROUNDDOWN(r_stval());
      // 页表映射
      if (mappages(p->pagetable, va, PGSIZE, (uint64)pa, PTE_W | PTE_R | PTE_U) != 0) {
        // 映射失败
        kfree(pa);
        printf("usertrap(): mappages() failed\n");
        // 杀死进程
        p->killed = 1;
      }
    }
    // 分配物理页面失败
    else {
      printf("usertrap(): kalloc() failed\n");
      // 杀死进程
      p->killed = 1;
    }
  }
  ...
}
```

处理uvmunmap错误

`uvmunmap()` 函数的功能是释放内存映射，但是页表中有些地址并没有分配实际的内存，没有进行映射，对于这些地址，直接跳过即可

修改 `kernal/vm.c` 文件中的的 `uvmunmap()` 函数，注释掉两行对 `panic` 的调用，并为所在分支添加 `continue`

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  ...
  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      // lab5-2
      // panic("uvmunmap: walk");
      continue;
    if((*pte & PTE_V) == 0)
      // lab5-2
      // panic("uvmunmap: not mapped");
      continue;
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `echo hi` 进行测试

```
$ echo hi
hi
```

结果符合预期， `hi` 可以被正常打印

3）实验中遇到的问题和解决方法

跟着提示做基本没什么问题，但是需要注意的是对特定的page fault的处理需要修改，现在我们正是需要利用这些page fault进行物理页面的分配

4）实验心得

这里用到了RISC-V中的stval寄存器，该寄存器包含导致页面错误的虚拟地址，专门用来存放于trap有关的信息，帮助OS或其他软件更快确定和完成trap的处理。其中的存放非零值有两种情况

1. 内存访问非法：这种情况包括硬件断点(Hardware Breakpoints)、地址未对齐(address misaligned)、访问故障(access-fault，可能是没有权限等)、缺页故障(page-fault)等情况。当这种情况发生时，stval会存储出错位置的虚拟地址。比如缺页故障发生时，stval就会记录到底是对哪个虚拟地址的访问导致了本次缺页故障，内核就可以根据此信息去加载页面进入内存
2. 指令非法访问(illegal instruction)：执行的指令非法时，stval会将这条指令的一部分位记录下来

这里page fault就属于第一种情况
通过 `r_scause()` 返回值判断，确认是page fault然后为当前合理的虚拟地址申请对应的物理地址，判断逻辑为：如果申请物理地址没成功或者虚拟地址超出范围了，那么杀掉进程。如果申请内存成功了，但如果虚拟地址不合法，需要再释放掉这块内存。中断判断时，如果出错(虚拟地址不合法或者没有成功映射到物理地址)，就杀死进程

同时，有些没有实际申请物理空间的，因此没有映射，在接触映射函数进行的时候需要跳过

3. Lazytests and Usertests (moderate)

1）实验目的

> We've supplied you with `lazytests`, an xv6 user program that tests some specific situations that may stress your lazy memory allocator. Modify your kernel code so that all of both `lazytests` and `usertests` pass.
>
> 1. Handle negative `sbrk()` arguments.
>   Kill a process if it page-faults on a virtual memory address higher than any allocated with `sbrk()`.
> 2. Handle the parent-to-child memory copy in `fork()` correctly.
> 3. Handle the case in which a process passes a valid address from `sbrk()` to a system call such as read or write, but the memory for that address has not yet been allocated.
> 4. Handle out-of-memory correctly: if `kalloc()` fails in the page fault handler, kill the current process.
> 5. Handle faults on the invalid page below the user stack.
> 6. Your solution is acceptable if your kernel passes `lazytests` and `usertests`:

```
$  lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap...
usertrap(): ...
test lazy unmap: OK
running test out of memory
usertrap(): ...
test out of memory: OK
ALL TESTS PASSED
$ usertests
...
ALL TESTS PASSED
$
```

> 修改您的内核代码，以便所有 `lazytests` 和 `usertests` 都能通过
>
> 1. 处理负的 `sbrk()` 参数
> 2. 如果某个页面的虚拟内存地址高于使用 `sbrk()` 分配的虚拟内存地址，则杀死该进程
> 3. 正确处理 `fork()` 中的父级到子级内存副本
> 4. 处理进程将有效地址从 `sbrk()` 传递给系统调用（例如读取或写入），但尚未分配该地址的内存的情况
> 5. 正确处理内存不足：如果页面故障处理程序中的 `kalloc()` 失败，请终止当前进程
> 6. 处理用户堆栈下方无效页面上的错误

2）实验步骤

编写代码

`sys_sbrk()` 参数为负处理

这里参考原来 `kernel/proc.c` 文件中 `growproc()` 函数的处理方法
在 `kernel/sysproc.c` 文件中的 `sys_sbrk()` 函数对 `n` 的大小进行判断，同时还需要对 `addr + n` 是否溢出进行判断

```
uint64
sys_sbrk(void)
{
  ...
  // lab5-3
  if (n >= 0 && addr + n >= addr) {
    p->sz += n;
  }
  else if (n < 0 && addr + n >= PGROUNDUP(p->trapframe->sp)) {
    p->sz = uvmdealloc(p->pagetable, addr, addr + n);
  }
  else {
    return -1;
  }
  ...
}
```

创建子进程

`fork()` 是通过 `uvmcopy()` 来进行父进程向子进程的复制
这里修改 `kernel/vm.c` 文件中 `uvmcopy()` ，将PTE不存在和无效两种情况引发的 `panic` 改为 `continue`

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  ...
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walk(old, i, 0)) == 0)
      // panic("uvmcopy: pte should exist");
      continue;
    if((*pte & PTE_V) == 0)
      // panic("uvmcopy: page not present");
      continue;
    ...
  }
  return 0;
...
}
```

page fault虚拟地址超出范围

在 `kernel/traps.c` 文件的 `usertrap()` 函数，在该情况时杀死进程

```
void
usertrap(void)
{
  ...
  if(r_scause() == 8){
    ...
  }
  // lab5-2 page fault的处理
  else if (r_scause() == 13 || r_scause() == 15) {
    char* pa;
    // 获得虚拟地址lab5-3
    uint64 va = r_stval();

    // 判断虚拟地址是否超出范围 lab5-3
    if (va >= p->sz) {
      printf("usertrap(): invalid va=%p higher than p->sz=%p\n", va, p->sz);
      p->killed = 1;
      goto end;
    }
    if (va < PGROUNDUP(p->trapframe->sp)) {
      printf("usertrap(): invalid va=%p below the user stack sp=%p\n", va, p->trapframe->sp);
      p->killed = 1;
      goto end;
    }

    // 分配物理页面
    if ((pa = kalloc()) != 0) {
      // 用0填充
      memset(pa, 0, PGSIZE);
      // 引发page fault的虚拟地址向下取整
      // uint64 va = PGROUNDDOWN(r_stval());
      va = PGROUNDDOWN(va);
      ...
    }
    ...
  }
  // lab5-3
  end:
    if (p->killed)
      exit(-1);
  ...
}
```

读写使用未分配的物理内存

`read()` / `write()` 两个函数会调用 `kernel/vm.c` 文件中的 `copyin()` / `copyout()` 来完成用户态到核心态的读写，这两个函数都会调用 `kernel/vm.c` 文件中的 `walkaddr()` 函数完成物理地址到虚拟地址的转换
原本PTE无效、不存在，PTE_U标志位缺失都是异常，但在lazy allocation的情况下，是被允许的
这里需要注意，我们需要先判断虚拟地址是否在用户堆空间范围内，因为lazy allocation是针对于此，然

后才进行相应处理

对 kernel/vm.c 文件中的的 walkaddr() 函数处理

```
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;
  // lab5-3
  struct proc* p = myproc();

  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  // lazy allocation lab5-3
  if (pte == 0 || (*pte & PTE_V) == 0) {
      // 在用户堆空间内
    if (va >= PGROUNDUP(p->trapframe->sp) && va < p->sz) {
      char* pa;
      if ((pa = kalloc()) == 0)
        return 0;
      memset(pa, 0, PGSIZE);
      if (mappages(p->pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)pa, PTE_W | PTE_R | PTE_U) !=
        kfree(pa);
        return 0;
      }
    }
    else
      return 0;
  }
  // if(pte == 0)
  //   return 0;
  // if((*pte & PTE_V) == 0)
  //   return 0;
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte);
  return pa;
}
```

测试程序

在存放 Makefile 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器

键入指令 `lazytests` 进行测试

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
usertrap(): invalid va=0x0000000000004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000001004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000002004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000003004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000004004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000005004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000006004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000007004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000008004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000009004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000a004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000b004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000c004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000d004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000e004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000000f004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000010004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000011004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000012004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000013004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000014004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000015004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000016004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000017004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000018004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000019004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001a004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001b004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001c004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001d004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001e004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000001f004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000020004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000021004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000022004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000023004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000024004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000025004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000026004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000027004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000028004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000029004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000002a004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000002b004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000002c004000 higher than p->sz=0x0000000000003000
```

```
usertrap(): invalid va=0x000000002d004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000002e004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000002f004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000030004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000031004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000032004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000033004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000034004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000035004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000036004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000037004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000038004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x0000000039004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003a004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003b004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003c004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003d004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003e004000 higher than p->sz=0x0000000000003000
usertrap(): invalid va=0x000000003f004000 higher than p->sz=0x0000000000003000
test lazy unmap: OK
running test out of memory
usertrap(): invalid va=0xffffffff80003808 higher than p->sz=0x0000000081003810
test out of memory: OK
ALL TESTS PASSED
```

键入 `Ctrl+a` ， 松开，然后键入 `x` ， 退出xv6系统
退出xv6进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-lazy
make: 'kernel/kernel' is up to date.
== Test running lazytests == (3.5s)
== Test   lazy: map ==
  lazy: map: OK
== Test   lazy: unmap ==
  lazy: unmap: OK
== Test usertests == (62.1s)
== 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
```

3）实验中遇到的问题和解决方法

遇到以下两个错误

```
kernel/vm.c:109:26: error: dereferencing pointer to incomplete type ‘struct proc’
```

```
In file included from kernel/vm.c:8:
kernel/proc.h:87:19: error: field ‘lock’ has incomplete type
   87 |   struct spinlock lock;
      |                   ^~~~
```

根据提示，应该按顺序添加两个头文件

```
#include "spinlock.h"
#include "proc.h"
```

4）实验心得

通过本次实验，我更加深刻地了解到什么是lazy allocation，利用退出资源的分配来节省内存和提高性能，在实现的过程中，一直遵循着“真正需要的时候才分配物理内存”的原则
具体来说，我们在xv6系统中利用page fault来实现，用户态通过 `sbrk()` 进行对heap上内存的增加或减少，如果申请的空间很大，将会花费很多时间。这时候引入了lazy allocation，等到实际用到的时候，会引发page fault，在 `usertrap` 中处理，为其申请物理内存空间。通过这种方法，将一次进行大量申请空

间的开销分散到读写内存中，从而一定程度上提高交互性
我们还需要清楚地了解几个寄存器的定义

- `scause` 存储中断类型
- `stval` 存储发生中断时访问的虚拟地址

同时对于中断类型，本实验用到的有

- `scause` 为13时对应 `Load page fault`
- `scause` 为15时对应 `Store/AMO page fault`