

操作系统原理

实 验 报 告

学 院 智能与计算学部
年 级 2019 级
班 级 留学生班
学 号 6319000359
姓 名 张明君

2020 年 12 月 5 日

天津大学

操作系统原理实验报告

题目: xv6 lazy page allocation

学院名称	智能与计算学部
专 业	计算机科学与技术
学生姓名	张明君
学 号	6319000359
年 级	2019 级
班 级	留学生班
时 间	2020. 12. 05

目 录

实验名称	1
实验目的	1
实验内容	1
实验步骤与分析	3
实验结论及心得体会	11

Homework: xv6 lazy page allocation

1. 实验目的

- **Eliminate allocation from sbrk():** first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys_sbrk() in sysproc.c.

- **Lazy allocation:** 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.

2. 实验内容

Part One: Eliminate allocation from sbrk()

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!).

Try to guess what the result of this modification will be: what will break?

Make this modification, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Part Two: Lazy allocation

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 `cprintf` call that produced the "pid 3 sh: trap 14" message. Your code is not required to cover all corner

cases and error situations; it just needs to be good enough to let `sh` run simple commands like `echo` and `ls`.

Hint: look at the `cprintf` arguments to see how to find the virtual address that caused the page fault.

Hint: steal code from `allocuvm()` in `vm.c`, which is what `sbrk()` calls (via `growproc()`).

Hint: use `PGROUNDDOWN(va)` to round the faulting virtual address down to a page boundary.

Hint: `break` or `return` in order to avoid the `cprintf` and the `myproc()->killed = 1`.

Hint: you'll need to call `mappages()`. In order to do this you'll need to delete the `static` in the declaration of `mappages()` in `vm.c`, and you'll need to declare `mappages()` in `trap.c`. Add this declaration to `trap.c` before any call to `mappages()`:

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

Hint: you can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in `trap()`.

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation) in the shell, and perhaps two.

By the way, this is not a fully correct implementation. See the challenges below for a list of problems we're aware of.

Optional challenges: Handle negative `sbrk()` arguments. Handle error cases such as `sbrk()` arguments that are too large. Verify that `fork()` and `exit()` work even if some `sbrk()`'d address have no memory allocated for them. Correctly handle faults on the invalid page below the stack. Make sure that kernel use of not-yet-allocated user addresses works — for example, if a program passes an `sbrk()`-allocated address to `read()`.

3. 实验步骤和分析（要细化如何实现的思路或流程图）

When the process needs more memory, `malloc` is called to apply for more heap memory, and the system calls `sbrk()` to complete the work. However, some processes may request a large amount of memory at a time, but they may not be used at all, such as sparse array. Therefore, the complex kernel involves delaying the actual allocation work to the actual use time. Page fault occurs, and then the actual allocation is carried out.

Part One: Eliminate allocation from `sbrk()`

1/The actual implementation of the system call `sbrk`, `sys_sbrk`, is modified to only increase the size of the process's memory space by `n` without actual allocation. So now we have to go to `sysproc.c` file to add the code below:

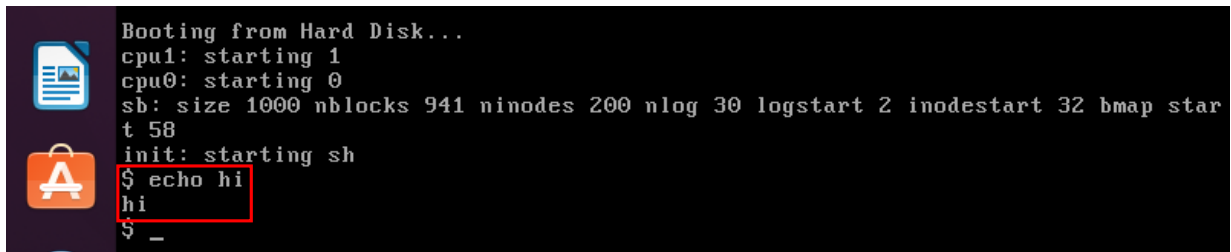
```

44
45 int
46 sys_sbrk(void)
47 {
48     int addr;
49     int n;
50
51     if(argint(0, &n) < 0)
52         return -1;
53     addr = myproc()->sz;
54     myproc()->sz += n;
55     //if(growproc(n) < 0)
56         // return -1;
57     return addr;
58 }

```

The code above, the returned address is the beginning of the newly allocated address space, and here is the end of the original address space. We increased `proc->sz`, but it did not actually increase the process size.

2/Before we change the code we can see in the terminal if we input `echo hi` in the qemu we will see only hi like this:

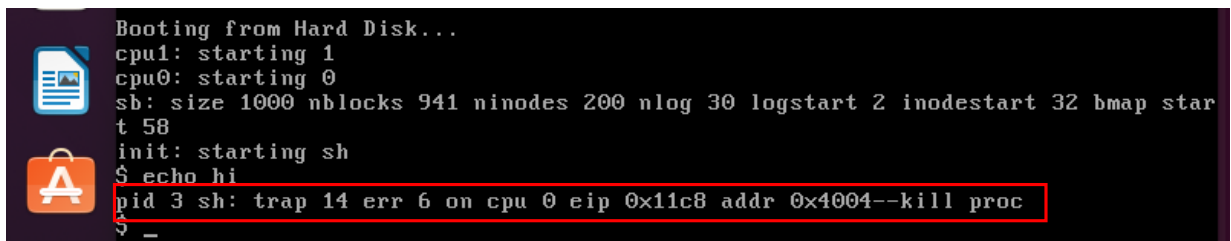


```

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
hi
$ -

```

After we changed the code as above we will see the different between them like this:



```

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x11c8 addr 0x4004--kill proc
$ -

```

Figure 1: Eliminate sbrk

It turn like this because the reason is that when the program tries to manipulate the memory area, it is found that the memory area is not owned by the current process because it is not allocated at all in `sys_sbrk`.

Part Two: Lazy allocation

The newly allocated physical memory page to the fault address and then return to the user space, allowing the process to continue execution. So now we have to see the tips that the

teacher gave us to pay attention on then and follow it step by step:

Hint: look at the `cprintf` arguments to see how to find the virtual address that caused the page fault.

Hint: steal code from `allocuvm()` in `vm.c`, which is what `sbrk()` calls (via `growproc()`).

Hint: use `PGROUNDDOWN(va)` to round the faulting virtual address down to a page boundary.

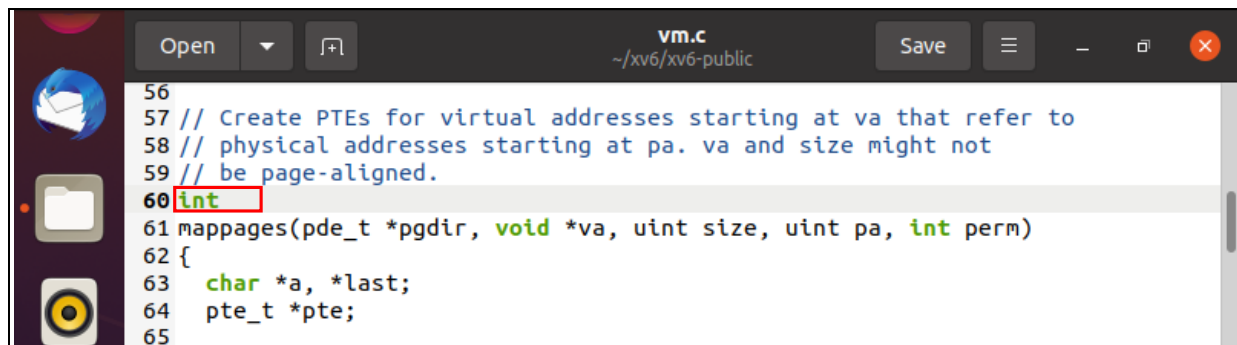
Hint: break or return in order to avoid the `cprintf` and the `myproc()->killed = 1`.

Hint: you'll need to call `mappages()`. In order to do this you'll need to delete the `static` in the declaration of `mappages()` in `vm.c`, and you'll need to declare `mappages()` in `trap.c`. Add this declaration to `trap.c` before any call to `mappages()`:

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

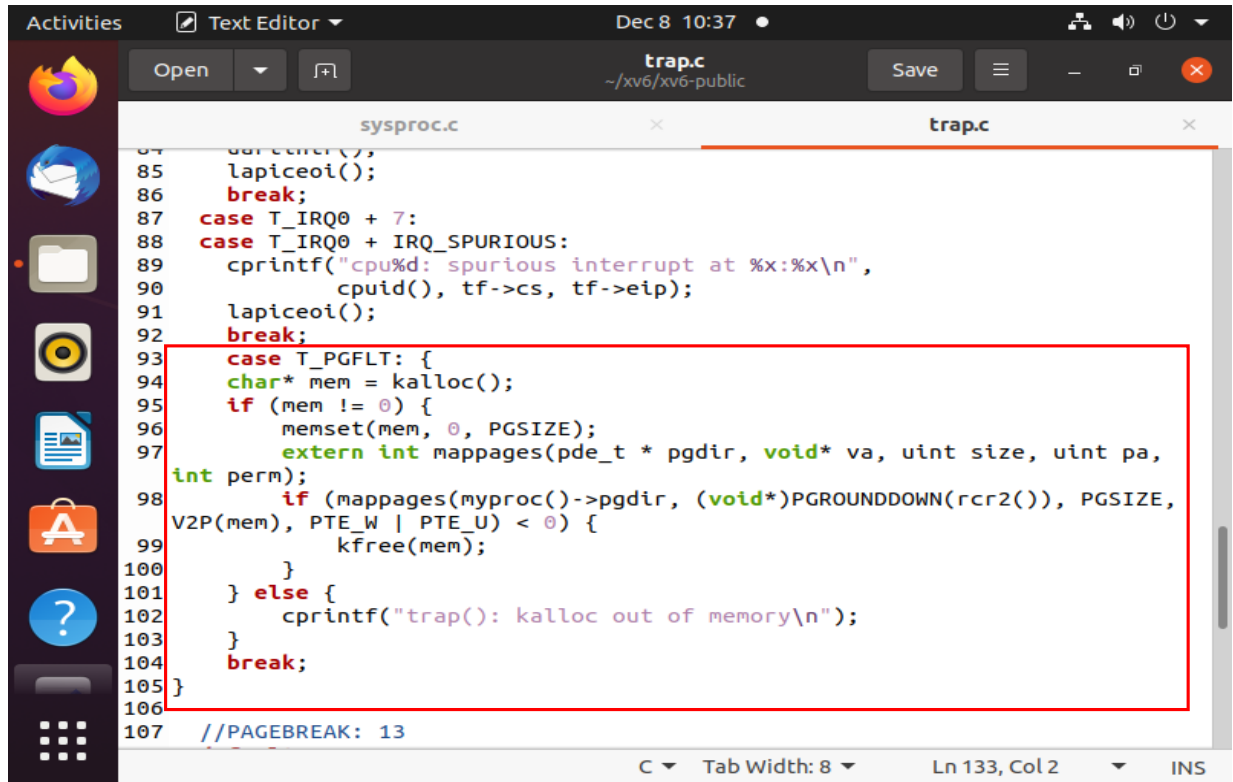
Hint: you can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in `trap()`.

1/Firstly, Since we need to call the `int mappages()` function in `vm.c`, we need to remove the original `static` keyword



```
56
57 // Create PTEs for virtual addresses starting at va that refer to
58 // physical addresses starting at pa. va and size might not
59 // be page-aligned.
60 int
61 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
62 {
63     char *a, *last;
64     pte_t *pte;
65
```

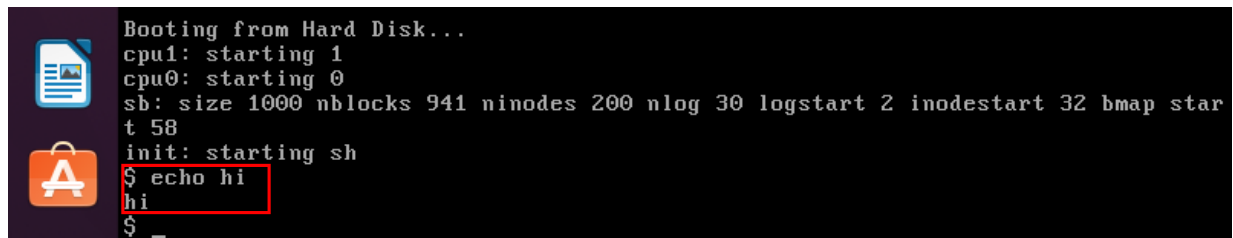
2/ And then add the new case in the following code to part of `void trap(struct trapframe *tf)` in `trap.c`, and place it after case `T_IRQ0 + IRQ_SPURIOUS`:



```
84  case T_IRQ0 + 7:
85      lapiceoi();
86      break;
87  case T_IRQ0 + IRQ_SPURIOUS:
88      cprintf("cpu%d: spurious interrupt at %x:%x\n",
89              cpuid(), tf->cs, tf->eip);
90      lapiceoi();
91      break;
92  case T_PGFLT: {
93      char* mem = kalloc();
94      if (mem != 0) {
95          memset(mem, 0, PGSIZE);
96          extern int mappages(pde_t * pgdir, void* va, uint size, uint pa,
97                          int perm);
98          if (mappages(myproc()->pgdir, (void*)PGROUNDDOWN(rcr2()), PGSIZE,
99                  V2P(mem), PTE_W | PTE_U) < 0) {
100              kfree(mem);
101          } else {
102              cprintf("trap(): kalloc out of memory\n");
103          }
104          break;
105      }
106  }
107  //PAGEBREAK: 13
```

Figure 2: Add new Case

3/After restarting xv6, enter the echo hi command, the output result is as follows



```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
hi
$ _
```

After we correct the code so now the `echo hi` is working well And then we use `ls` command to see if it working or not. After use `ls` command we will see the things as below:

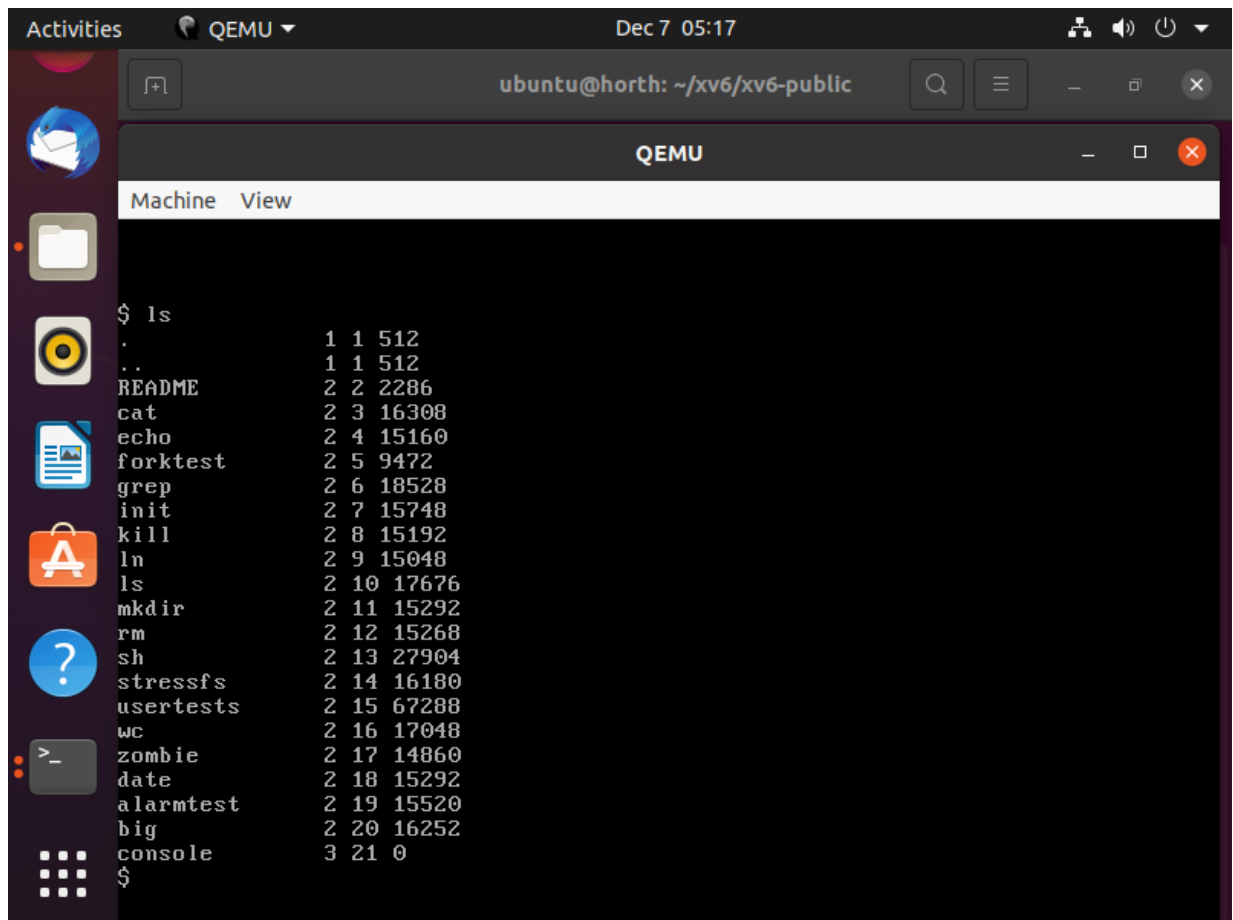
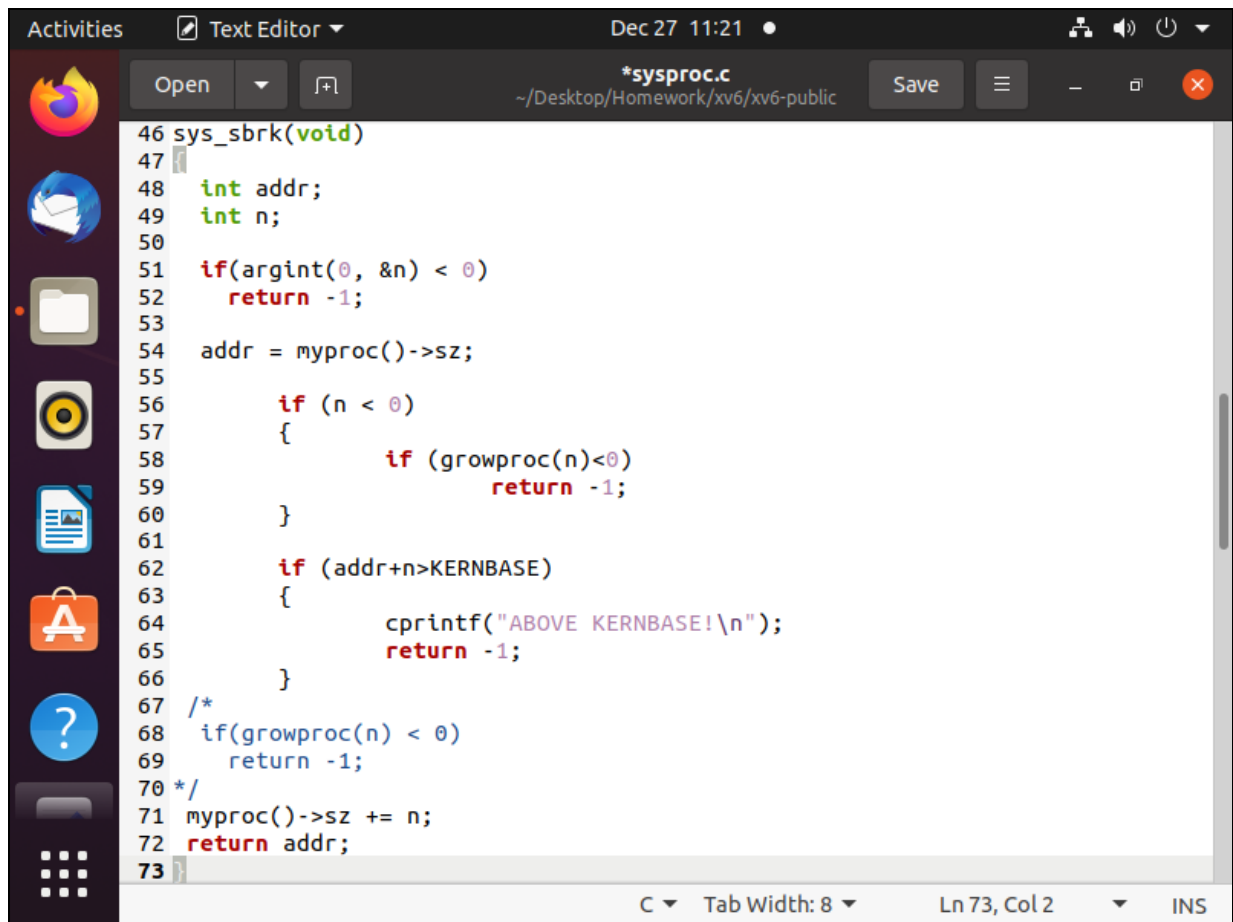


Figure 3: Final answer

Optional challenges

First, We have to add some codes into the **sysproc.c** file because we have to handle the negative `sbrk()` and to handle the error cases such as `sbrk()` arguments that are too large. And make sure that kernel use of not yet allocated user addresses works.

So now we have the final code in **sysproc.c** of the `sbrk()` function below:

A screenshot of a Linux desktop environment. The top bar shows 'Activities', 'Text Editor', and the date 'Dec 27 11:21'. The text editor window is titled '*sysproc.c' and shows the following C code:

```
46 sys_sbrk(void)
47 {
48     int addr;
49     int n;
50
51     if(argint(0, &n) < 0)
52         return -1;
53
54     addr = myproc()->sz;
55
56     if (n < 0)
57     {
58         if (growproc(n)<0)
59             return -1;
60     }
61
62     if (addr+n>KERNBASE)
63     {
64         cprintf("ABOVE KERNBASE!\n");
65         return -1;
66     }
67     /*
68     if(growproc(n) < 0)
69         return -1;
70     */
71     myproc()->sz += n;
72     return addr;
73 }
```

The status bar at the bottom of the editor shows 'C', 'Tab Width: 8', 'Ln 73, Col 2', and 'INS'. The left sidebar contains icons for various applications like Firefox, Mail, Files, and a terminal.

Figure 4: Optional challenge answer

4. 实验结论与心得体会

This experiment is not that easy also because syscall consumes performance, the C standard library provides malloc / free to manage the heap memory used by users. However, when the user applies for heap memory, it may not be used immediately. In order to improve the performance, the homework adopts lazy allocation mechanism. When users apply for heap memory, the kernel does not allocate physical memory, but only when it is needed. This will minimize unnecessary syscalls and improve performance.

The specific logic is that when users apply for memory, they do not allocate physical memory. When physical memory is actually used and page fault occurs, an exception of type faults (case T_PGFLT) . Capture page fault in interrupt / exception handler (trap.c: trap()) to realize lazy allocation.