

# BUAA-OS-lab4

📅 发表于 2023-05-15 | 🔄 更新于 2023-06-19 | 📖 养德（学习）  
| 📄 字数总计: 5.2k | ⌚ 阅读时长: 19分钟 | 👁 阅读量: 3

## Lab4实验报告

### 思考题

#### Thinking 4.1

思考并回答下面的问题：

- 内核在保存现场的时候是如何避免破坏通用寄存器的？
- 系统陷入内核调用后可以直接从当时的 \$a0-\$a3 参数寄存器中得到用户调用 `msyscall` 留下的信息吗？
- 我们是怎么做到让 `sys` 开头的函数“认为”我们提供了和用户调用 `msyscall` 时同样的参数的？
- 内核处理系统调用的过程对 `Trapframe` 做了哪些更改？这种修改对应的用户态的变化是什么？

◦ 解：



- 在 `SAVE_ALL` 中：

先 `move k0, sp`，先把通用寄存器的 `sp` 复制到 `$k0`；

再 `sw k0, TF_REG29(sp)`、`sw $2, TF_REG2(sp)`：保存现场需要使用 `$v0` 作为协寄存器到内存的中转寄存器，写到内存时需要 `sp`，所以在正式保存协寄存器和通用寄存器前先保存这两个寄存器。

- 可以。

从用户函数 `syscall_*`() 到内核函数 `sys_*`() 时，`$a1 - $a3` 未改变，`$a0` 在 `handle_sys()` 的时候被修改为内核函数的地址，但在内核函数 `sys_*`() 仅为占位符，不会被用到。同时，在内核态中可能使用这些寄存器进行一些操作计算，此时寄存器原有值被改变，因此再次以这些参数调用其他函数时需要重新以 `sp` 为基地址，按相应偏移从用户栈中取用这四个寄存器值。

总之，一般情况下，还是从栈中取得这些参数更加保险。

- 用户调用时的参数：

1. 用户进程的寄存器现场（保存在了内核栈的?? `TF_4-TF_7`??）的 `$a1 - $a3`；

2. 用户栈(栈指针为用户现场的 `sp`)的参数 `$a4`、`$a5`；

把上面两部分参数分别拷贝至内核现场寄存器 `$a1 - $a3` 和内核栈。

- 第一，将栈中存储的EPC寄存器值增加4，这是因为系统调用后，将会返回下一条指令，而用户程序会保证系统调用操作不在延迟槽内，所以直接加4得到下一条指令的地址；

第二，将返回值存入 `$v0`。

## Thinking 4.2

思考 `envid2env` 函数: 为什么 `envid2env` 中需要判断 `e->env_id != envid` 的情况? 如果没有这步判断会发生什么情况?

- 解：

- 在我们生成`envid`时，后十位为了方便从`envs`数组中直接取出`Env`，可能会有所重叠，  
`envid`的独一性取决于`mkenvid`里不断增长的 `i`，所以如果不判断`envid`是否相同，会取到错误的或者本该被销毁的进程控制块。

### Thinking 4.3

思考下面的问题，并对这个问题谈谈你的理解：请回顾 `kern/env.c` 文件中 `mkenvid()` 函数的实现，该函数不会返回 `0`，请结合系统调用和 `IPC` 部分的实现与 `envid2env()` 函数的行为进行解释。

- 解：
- 我们可以看到该函数为：

```
1  u_int mkenvid(struct Env *e) {  
2      static u_int i = 0;  
3      return ((++i) << (1 + LOG2NENV)) | (e - env  
4  }
```

`++i` 保证一定不会为0；`envid2env()` 的`envid`为0时返回 `curenv`；

- 由于 `curenv` 为内核态的变量，用户态不能获取 `curenv` 的 `envid`，所以用 `0` 代表 `curenv->envid`；
- 目的是方便用户进程调用 `syscall_*`() 时把当前进程的 `envid` 作为参数传给内核函数，即方便用户态在内核变量不可见的情况下调用内核接口。

### Thinking 4.4

关于 `fork` 函数的两个返回值，下面说法正确的是：

- A、 `fork` 在父进程中被调用两次，产生两个返回值
- B、 `fork` 在两个进程中分别被调用一次，产生两个不同的返回值
- C、 `fork` 只在父进程中被调用了一次，在两个进程中各产生一

个返回值

D、 `fork` 只在子进程中被调用了一次，在两个进程中各产生一个返回值

- 解：
- 正确答案是C

## Thinking 4.5

我们并不应该对所有的用户空间页都使用 `duppage` 进行映射。那么究竟哪些用户空间页应该映射，哪些不应该呢？请结合 `kern/env.c` 中 `env_init` 函数进行的页面映射、`include/mmu.h` 里的内存布局图以及本章的后续描述进行思考。

- 解：
- 在  $0 \sim USTACKTOP$  范围的内存需要使用 `duppage` 进行映射；
- $USTACKTOP$  到  $UTOP$  之间的 **user exception stack** 是用来进行页写入异常的，不会在处理COW异常时调用 `fork()`，所以 user exception stack 这一页不需要共享；
- $USTACKTOP$  到  $UTOP$  之间的 **invalid memory** 是为处理页写入异常时做缓冲区用的，所以同理也不需要共享；
- $UTOP$  以上页面的内存与页表是所有进程共享的，且用户进程无权限访问，不需要做父子进程间的 `duppage`；
- 其上范围的内存要么属于内核，要么是所有用户进程共享的空间，用户模式下只可以读取。除只读、共享的页面外都需要设置 `PTE_COW` 进行保护。

## Thinking 4.6

在遍历地址空间存取页表项时你需要使用到 `vpd` 和 `vpt` 这两个指针，请参考 `user/include/lib.h` 中的相关定义，思考并回答这几个问题：

- vpt 和 vpd 的作用是什么？怎样使用它们？
- 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？
- 它们是如何体现自映射设计的？
- 进程能够通过这种方式来修改自己的页表项吗？

◦ 解：

- 作用：在用户态下通过访问进程自己的物理内存获取用户页的页目录项页表项的 perm，用于 duppage 根据不同的 perm 类型在父子进程间执行不同的物理页映射；

◦ 使用：

- vpd是页目录首地址，以vpd为基地址，加上页目录项偏移数即可指向va对应页目录项，即  $(*vpd) + (va \gg 22)$  或 `vpd[va >> 22]`；
- vpt是页表首地址，以vpt为基地址，加上页表项偏移数即可指向va对应的页表项，即  $(*vpt) + (va \gg 12)$  或 `vpt[va >> 12]` 即 `vpt[VPN(va)]`；

◦ 自映射设计体现：

```
1  #define vpt ((volatile Pte *)UVPT)
2  #define vpd ((volatile Pde *) (UVPT + (PDX(U
```



vpd的地址在UVPT和UVPT + PDMAP之间，说明将页目录映射到了某一页表位置(即实现了自映射)；

- 不能。该区域对用户只读不写，若想要增添页表项，需要陷入内核进行操作。

## Thinking 4.7

在 `do_tlb_mod` 函数中，你可能注意到了一个向异常处理栈复制 `Trapframe` 运行现场的过程，请思考并回答这几个问题：

- 这里实现了一个支持类似于“异常重入”的机制，而在什么时候会出现这种“异常重入”？

- 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

◦ 解：

- 当出现COW异常时，需要使用用户态的系统调用发生中断，即中断重入；
- 由于处理COW异常时调用的 `handle_mod()` 函数把epc改为用户态的异常处理函数 `env_user_tlb_mod_entry`，退出内核中断后跳转到epc所在的用户态的异常处理函数。

由于用户态把异常处理完毕后仍然在用户态恢复现场，所以此时要把内核保存的现场保存在用户空间的用户异常栈。

## Thinking 4.8

在用户态处理页写入异常，相比于在内核态处理有什么优势？

◦ 解：

- 解放内核，不用内核执行大量的页面拷贝工作；
- 内核态处理失误产生的影响较大，可能会使得操作系统崩溃；
- 用户状态下不能得到一些在内核状态才有的权限，避免改变不必要的内存空间；
- 同时微内核的模式下，用户态进行新页面的分配映射也更加灵活方便。

## Thinking 4.9

请思考并回答以下几个问题：

- 为什么需要将 `syscall_set_tlb_mod_entry` 的调用放置在 `syscall_exofork` 之前？
- 如果放置在写时复制保护机制完成之后会有怎样的效果？

◦ 解：



## 系统调用实例

- syscall

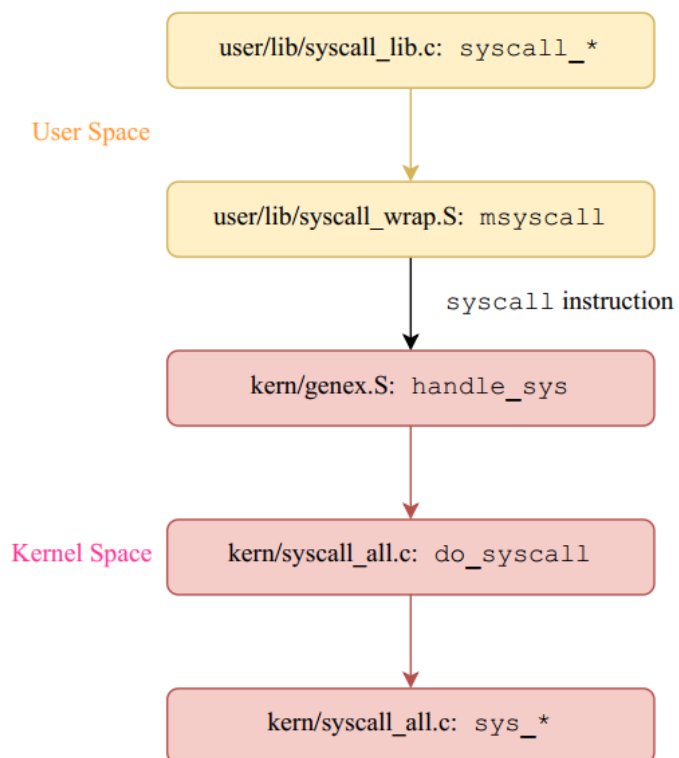
用于执行系统调用的自陷指令，它使得进程陷入到内核的异常处理程序中，由内核根据系统调用时的上下文执行相应的内核函数，完成相应的功能，并最终返回到 syscall 的后一条指令。

- 系统调用的层次结构

高级	用户程序 <b>User Program</b>	
	应用程序编程接口 API	POSIX, C Standard Library等
最底层	系统调用	read, write 等

## 系统调用机制的实现

- syscall 过程流程图



【系统调用使用流程】：

- `syscall_*` (`user/lib/syscall_lib.c`)



该函数构成非常简单，只有一句话：调用 `msyscall` 函数

- `msyscall` (user/lib/syscall\_wrap.S)

该函数构成也非常简单，两步：调用 `syscall` 函数 +  
`jr ra` 返回

(`SYS_*` 系统调用号是在 `include/syscall.h` 里面定义的)

- `syscall` (kern/entry.S)

1. 使用 `SAVE_ALL` 宏：将用户进程的上下文运行环境保存在内核栈中
2. 取出 `CP0_CAUSE` 寄存器中的异常码，系统调用对应的异常码为 8
3. 以异常码为索引在 `exception_handlers` 数组中找到对应异常处理函数 `handle_sys`
4. 转跳至 `handle_sys` 函数处理用户的系统调用请求

- `SAVE_ALL` (include/stackframe.h)

在保存用户态现场时 `sp` 减去了一个 `Trapframe` 结构体的空间大小，此时我们将用户进程现场保存在内核栈中范围为 `[sp, sp + sizeof(TrapFrame))` 的这一空间范围内

- `handle_sys` 用宏 `BUILD_HANDLER` 实现 (kern/genex.S)

1. 由 `SAVE_ALL` 得到的 `sp` 寄存器中保存的是 `Trapframe` 结构体的起始地址，将该起始地址存入 `a0` 寄存器作为 `do_syscall` 的传入参数
2. 调用 `do_syscall` 实现处理系统调用
3. 调用 `ret_from_exception` 从内核态返回用户程序

- `do_syscall` (kern/syscall\_all.c)

1. 改 `epc` 使得由内核态返回用户态之后能够执行 `msyscall` 函数中的 `jr ra` 指令

2. 获得参数，通过 `func(arg1, arg2, arg3, arg4, arg5)` 直接调用内核中相应的系统调用函数，也就是 `sys_*` 函数

- `sys_*` (kern/syscall\_all.c)

- `syscall_*` 和 `sys_*`

`syscall_*` 的函数与内核中的系统调用函数（`sys_*` 的函数）是一一对应的；

- `syscall_*` 的函数是我们在用户空间中最接近的内核的函数（不允许在延迟槽中使用）；

- `sys_*` 的函数是内核中系统调用的具体实现部分。

- `msyscall`

- 每个 `syscall_*` 都调用了函数 `msyscall`，`msyscall` 的第一个参数都是一个与调用名相似的宏（如 `SYS_print_cons`），我们叫这个参数为系统调用号（定义在 `include/syscall.h` 中）。

除了系统调用号之外，`msyscall` 还有5个参数，这些参数是系统调用时需要传递给内核的参数。（之所以还要另外的5个参数，是因为系统调用所需要的最多参数数量，就是是“`syscall_mem_map` 函数需要 5 个参数”）

- `msyscall` 函数是叶函数，没有局部变量，不需要分配栈帧，只需执行自陷指令 `syscall` 来陷入内核态并在处理结束后正常返回即可。

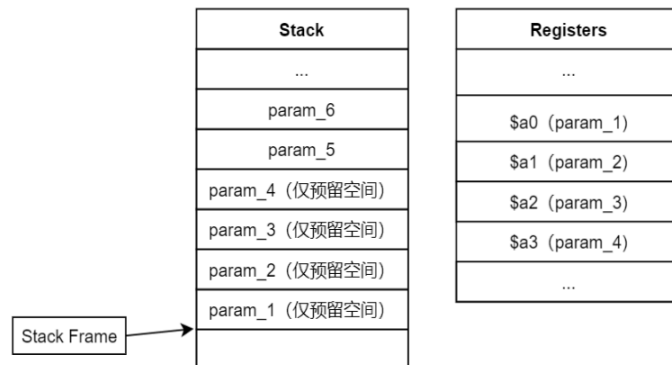
- *stack frame* 栈帧

- 栈帧：进入函数体时会通过对栈指针做减法（压栈）的方式为该函数自身的局部变量、返回地址、调用函数的参数分配存储空间，在函数调用结束之后会\*\*对栈指针做加法（弹栈）\*\*来释放这部分空间，该空间就是栈帧。

- 调用方在自身栈帧的底部预留被调用函数的参数存储空间，由被调用方从调用方的栈帧中读取参数

- 寄存器 `$a0-$a3` 用于存放函数调用的前四个参数（但在栈中仍然需要为其预留空间），剩余的参数仅存放在栈中。

- 例子：msyscall 函数一共有 6 个参数，前 4 个参数会被 syscall\_\* 的函数分别存入 \$a0-\$a3 寄存器（寄存器传参的部分）同时栈帧底部保留 16 字节的空间（不求存入参数的值），后 2 个参数只会被存入在预留空间之上的 8 字节空间内（没有寄存器传参），于是总共 24 字节的空间用于参数传递。



- void do\_syscall(struct Trapframe \*tf)

- 功能：
- 使用例子：
- 实现：

- 往年通信：

实现思路如下：

- 选择开一个结构体数组记录每次信息发送的相关值和一个记录是否完成的标记。
- 接收进程：首先查表，有无自己可以接受的信息，有的话就接收，设置发送进程状态为RUNNABLE 并正常退出，否则阻塞。
- 发送进程：检查接收进程的状态，若阻塞，直接进程信息发送同时设置接收进程状态为RUNNABLE。若接收进程没有阻塞，将待发送的信息添加到信息表中，阻塞。

## fork

- 要查看系统调用就去 user/lib/syscall\_lib.c 和 syscall\_all.c 文件中看吧

```

1  #define vpt ((volatile Pte *)UVPT)
2  #define vpd ((volatile Pde *) (UVPT + (PDX(UVP
3  #define envs ((volatile struct Env *)UENVS)

```

```

4  #define pages ((volatile struct Page *)UPAGES
5
6  #define BY2PG 4096                // bytes to a
7  #define PDMAP (4 * 1024 * 1024) // bytes mapp
8  #define PGSHIFT 12
9  #define PDSHIFT 22 // log2(PDMAP)
10 #define PDX(va) (((u_long)(va)) >> 22) & 0x0
11 #define PTX(va) (((u_long)(va)) >> 12) & 0x0
12 #define PTE_ADDR(pte) ((u_long)(pte) & ~0xFFF
13
14 // Page number field of an address
15 #define PPN(va) (((u_long)(va)) >> 12)
16 #define VPN(va) (((u_long)(va)) >> 12)
17
18 #define ROUND(a, n) (((u_long)(a)) + (n)-1)
19 #define ROUNDDOWN(a, n) ((u_long)(a) & ~(n

```

- `int fork(void)` (user/lib/fork.c)
- 【注意】：`env = envs + ENVX(syscall_getenvid());`  
`syscall_getenvid()`: 获得当前进程的envid  
`envs + ENVX(...)`: 由envid获得env
- 【注意】
  - vpd是页目录首地址，以vpd为基地址，加上页目录项偏移数即可指向va对应页目录项，即  $(*vpd) + (va \gg 22)$  或 `vpd[va >> 22]` ；  
 二级页表的物理地址：`vpd[va >> 22] & (~0xfff)`  
 提前判断有效位：`(vpd[va >> 22] & PTE_V)` 或 `(vpd[VPN(va) >> 10] & PTE_V)`
  - vpt是页表首地址，以vpt为基地址，加上页表项偏移数即可指向va对应的页表项，即  $(*vpt) + (va \gg 12)$  或 `vpt[va >> 12]` 即 `vpt[VPN(va)]` ；  
 物理页面地址：`vpt[va >> 12] & (~0xfff)`  
 提前判断有效位：`(vpt[va >> 12] & PTE_V)` 或 `(vpt[VPN(va)] & PTE_V)`
  - `vpn = VPN(va) = va >> 12` ( 虚拟页号)

- static void ... cow\_entry(...) (user/lib/fork.c)

## 往年题lab4-2-exam

```

1  int make_shared(void *va) {
2      u_int perm = PTE_D | PTE_V;
3      if (!(vpd[va >> 22] & PTE_V) || !(vpt[va >> 12]
4          //当前进程的页表中不存在该虚拟页
5          if (syscall_mem_alloc(0, ROUNDDOWN(va, BY2PG),
6              //将envid设为0, 表示默认curenv
7              return -1;
8          }
9      }
10     perm = vpt[VPN(va)] & 0xfff; //获得va的perm
11     if (va >= (void *)UTOP ||
12         ((vpd[va >> 22] & PTE_V) && (vpt[va >> 12]
13         return -1;
14     }
15     perm = perm | PTE_LIBRARY;
16     u_int addr = VPN(va) * BY2PG;
17     if (syscall_mem_map(0, (void *)addr, 0, (void *)
18         return -1;
19     }
20     return ROUNDDOWN(vpt[VPN(va)] & (~0xfff), BY2PG);
21 }

```

- 附：由于 ROUNDDOWN(va, BY2PG) 本质上将后12位置0，所以对于 vpd[] 、 vpt[] 并无影响，但是对于 syscall\_mem\_map 还是有影响的。

## 难点分析

- Trapframe 结构体中 regs[32] 的各个含义：

Reg	Name	
0	zero	
1	at	
2-3	v0-v1	
4-7	a0-a3	

Reg	Name	
8-15	t0-t7	
24-25	t8-t9	
16-23	s0-s7	
26-27	k0-k1	
28	gp	
29	sp	
30	s8/fp	
31	ra	

## 课上测试

### lab4-1-Exam

主要考察添加一个系统调用的步骤，如下以用户进程调用函数 `user_lib_func(u_int whom, u_int val, const void *srcva, u_int perm)` 过程中，会使用到系统调用 `syscall_func` 为例归纳步骤：


1. 在 `user/include/lib.h` 中添加：

```
void user_lib_func(u_int whom, u_int val, const
void *srcva, u_int perm);

void syscall_func(u_int envid, u_int value, const
void *srcva, u_int perm);
```

2. 在 `user/lib/syscall_lib.c` 中添加：

```
1 void syscall_func(u_int envid, u_int value, co
2     msyscall(SYS_func, envid, value, srcva, pe
3 }
```

3. 在 *user/lib* 中的使用 *user\_lib\_func* 函数的目标文件中编写实现该函数（注意在该函数过程中会调用 *syscall\_func* 函数）
  4. 在 *include/syscall.h* 中的 enum 的 *MAX\_SYSNO* 前面加上 *SYS\_func*,
  5. 在 *kern/syscall\_all.c* 的 void *\*syscall\_table[MAX\_SYSNO]* 的最后加上 *[SYS\_func] = sys\_func*, （注意最后有逗号）
  6. 在 *kern/syscall\_all.c* 的 void *\*syscall\_table[MAX\_SYSNO]* 的前面具体编写实现函数
- ```
1  int sys_func(u_int envid, u_int value, u_int s
2      //.....
3  }
```
- 

#### lab4-1-Extra

lab4-1-extra需要实现一种广播通讯机制 *ipc\_broadcast* 函数，具体题目见文章Lab4-1-Extra-Broadcast题干。

主要在于引入全局变量 *envs* 数组，然后遍历判断后代进程。

- 我的答案：【太感动了~ 小女子菜菜，本学期第一次拿到 extra 的100分 感激涕零ing】

```
1  //kern/syscall_all.c
2  extern struct Env envs[NENV]; //注意 extern!
3  int sys_ipc_try_broadcast(u_int value, u_int
4      struct Env *e;
5      struct Page *p;
6
7      /* Step 1: Check if 'srcva' is either
8      /* 抄的sys_ipc_try_send */
9      if (srcva != 0 && is_illegal_va(srcva)
10          return -E_IPC_NOT_RECV;
11      }
12
13      /* 函数核心: 遍历envs找后代进程 */
14      int signal[NENV];
15      for (u_int i = 0; i < NENV; i++) {
16          if (curenv->env_id == envs[i]
```

```

17             signal[i] = 1;
18         } else {
19             signal[i] = 0;
20         }
21     }
22     int flag = 0;
23     while(flag == 0) {
24         flag = 1;
25         for (u_int i = 0; i < NENV; i
26             if (signal[i] == 1) {
27                 for (u_int j
28                     if (s
29
30
31                         }
32                     }
33                 }
34             }
35     }
36
37     /* Step 3: Check if the target is wai
38     /* 基于sys_ipc_try_send修改 */
39     for (u_int i = 0; i < NENV; i++) {
40         if(signal[i] == 1) {
41             e = &(envs[i]);
42             /* 以下都是抄的sys_ipc_try_send */
43             if (e->env_ipc_recving == 0) {
44                 return -E_IPC
45             }
46             e->env_ipc_value = va
47             e->env_ipc_from = cur
48             e->env_ipc_perm = PTE
49             e->env_ipc_recving =
50             e->env_status = ENV_R
51             TAILQ_INSERT_TAIL(&en
52             if (srcva != 0) {
53                 p = page_look
54                 if(p == NULL)
55                 if (page_inse
56                 return -E_INVAL;
57             }
58         }
59     }
60 }
61 return 0;
62 }

```



- 附：我后续在完成lab4-2的任务时，发现有一个 `env = envs + ENVX(enuid)`；可以由 `enuid`得到 `env`

### lab4-2-Exam

- 考察：系统调用+fork+ipc

最终只得分了70分，具体原因还在求助老师和助教中。

【后来重测啦~ 下面的三种写法都是100分~ 耶耶耶！！！！】

- 版本1:

```

1 //测试数据点3和6不过,得分55分
2 u_int sys_barrier_wait(u_int* p_barrier_num,
3     static u_int env_not[100];
4     static u_int N = 0;
5     static u_int num = 0;
6     static u_int useful = 0;
7     if ((*p_barrier_num) > N) {
8         N = (*p_barrier_num);
9         num = N;
10        useful = (*p_barrier_useful);
11    }
12    if (useful == 1) {
13        for (u_int i = 0; i < N - num
14            if (env_not[i] == cur
15                retur
16            }
17        }
18        env_not[N - num] = curenv->en
19        num--;
20        if (num == 0) { //first versi
21            useful = 0;
22            return ENV_RU
23        }
24        return ENV_NOT_RUNNABLE;
25    }
26    return ENV_RUNNABLE;
27 }
```

- 版本2:

```

1 //测试数据点6不过, 得分70分
2 u_int sys_barrier_wait(u_int* p_barrier_num,
```

```

3      static u_int env_not[100];
4      static u_int N = 0;
5      static u_int num = 0;
6      static u_int useful = 0;
7      if ((*p_barrier_num) > N) {
8          N = (*p_barrier_num);
9          num = N;
10         useful = (*p_barrier_useful);
11     }
12     if (useful == 1) {
13         if (num == 0) { //second vers
14             useful = 0;
15             return ENV_RU
16         }
17         for (u_int i = 0; i < N - num
18             if (env_not[i] == cur
19                 retur
20             }
21         }
22         env_not[N - num] = curenv->en
23         num--;
24         return ENV_NOT_RUNNABLE;
25     }
26     return ENV_RUNNABLE;
27 }

```

○ 版本3:

```

1  //经过和其他同学的讨论，以及代码对拍，下面这个代码
2  u_int sys_barrier_wait(u_int* p_barrier_num,
3      static u_int env_not[100];
4      static u_int N = 0;
5      static u_int num = 0;
6      static u_int useful = 0;
7      if ((*p_barrier_num) > N) {
8          N = (*p_barrier_num);
9          num = N;
10         useful = (*p_barrier_useful);
11     }
12     if (num == 0) { //third version
13         useful = 0;
14         return ENV_RUNNABLE;
15     }
16     if (useful == 1) {
17         for (u_int i = 0; i < N - num
18             if (env_not[i] == cur

```

```
19                                     retur
20                                     }
21                                 }
22                                 env_not[N - num] = curenv->en
23                                 num--;
24                                 return ENV_NOT_RUNNABLE;
25                             }
26                             return ENV_RUNNABLE;
27     }
```

## lab4-2-Extra

【哭唧唧~ exam没做出来，extra连题目都没看，只是据说很难~】

## 体会与感想

lab4主要需要掌握：系统调用，IPC通信机制，fork进程创建，页面写入异常处理。

在本次实验中，脑子里一定要清楚现在是在改内核还是改用户，因为之前写的都是内核，而内核函数是不能在用户空间调用的，这一点要注意区分。

让人很难受的是，自己的lab4-2-exam挂了，感觉非常无助，一方面个人觉得题目的表述不是很清晰，另一方面也自己反思还是五一假期过于沉溺与玩乐了。

总之，lab4要想获得比较好的成绩，还是需要投入大量的时间和精力。