实 验 报 告

Lab4

姓名：朱瑞媛

学号：14307130373

Part A

Exercise 1. Implement mmio_map_region in kern/pmap.c. To see how this is used, look at the beginning of lapic_init in kern/lapic.c. You'll have to do the next exercise, too, before the tests for mmio_map_region will run.

```
616 //
617 void *
618 mmio_map_region(physaddr_t pa, size_t size)
619 {
620     // Where to start the next region.  Initially, this is the
621     // beginning of the MMIO region.  Because this is static, its
622     // value will be preserved between calls to mmio_map_region
623     // (just like nextfree in boot_alloc).
624     static uintptr_t base = MMIOBASE;
625
626     // Reserve size bytes of virtual memory starting at base and
627     // map physical pages [pa,pa+size) to virtual addresses
628     // [base,base+size).  Since this is device memory and not
629     // regular DRAM, you'll have to tell the CPU that it isn't
630     // safe to cache access to this memory.  Luckily, the page
631     // tables provide bits for this purpose; simply create the
632     // mapping with PTE_PCD|PTE_PWT (cache-disable and
633     // write-through) in addition to PTE_W.  (If you're interested
634     // in more details on this, see section 10.5 of IA32 volume
635     // 3A.)
636     //
637     // Be sure to round size up to a multiple of PGSIZE and to
638     // handle if this reservation would overflow MMIOLIM (it's
639     // okay to simply panic if this happens).
640     //
641     // Hint: The staff solution uses boot_map_region.
642     //
643     // Your code here:
644     //panic("mmio_map_region not implemented");
645     pa = ROUNDDOWN(pa, PGSIZE);
646     size = ROUNDUP(size, PGSIZE);
647     if (base + size >= MMIOLIM)
648         panic("Memory is not enough.\n");
649     boot_map_region(kern_pgdir, base, size, pa, PTE_PCD|PTE_PWT|PTE_W);
650     uintptr_t ret = base;
651     base += size;
652     return (void *) ret;
```

apic为与其相连的cpu传送中断信号，cpu对其控制需要通过寄存器，需要此函数完成映射IO来实现。

静态变量base，记录当前未分配空间的起始地址。

如果地址空间不够，则出现panic.

调用boot_map_region()，从MMIOBASE开始分配size大小的空间。维护base的值。

Exercise 2. Read boot_aps() and mp_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page_init() in kern/pmap.c to avoid adding the page at MPENTRY_PADDR to the free list, so that we can safely

copy and run AP bootstrap code at that physical address. Your code should pass the updated check_page_free_list() test (but might fail the updated check_kern_pgdir() test, which we will fix soon).

```
323     size_t i;
324     /*for (i = 0; i < npages; i++) {
325         pages[i].pp_ref = 0;
326         pages[i].pp_link = page_free_list;
327         page_free_list = &pages[i];*/
328      for (i = 0; i < npages_basemem; i++)
329      {
330         if (i == 0)
331             pages[i].pp_ref = 1;
332         else if (i == MPENTRY_PADDR/PGSIZE)
333             pages[i].pp_ref = 1;
334         else
335         {
336             pages[i].pp_ref = 0;
337             pages[i].pp_link = page_free_list;
338             page_free_list = &pages[i];
339         }
340      }
341      int occup = (int)ROUNDUP(((char *)envs) + NENV * (sizeof(struct Env)) - KERNBASE, PGSIZE)/PGSIZE;
342      for (i = npages_basemem; i < npages; i++)
343      {
344         if (i < occup)
345             pages[i].pp_ref = 1;
346         //else if (i == MPENTRY_PADDR/PGSIZE)
347         //  pages[i].pp_ref = 1;
348         else
349         {
350             pages[i].pp_ref = 0;
351             pages[i].pp_link = page_free_list;
352             page_free_list = &pages[i];
353         }
354     }
```

启动代码占用MPENTRY_PADDR,所以此物理页需要标注为已用，防止被分配出去。

Question

1. Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted inkern/mpentry.S?

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

这里的mpentry.S的代码是链接到KERNBASE之上的但是实际上现在是将这些代码移动到了物理地址0X7000处，而且当前的AP处于实模式下，只支持1MB的物理地址寻址，所以这个时候需要计算相对于0X7000的地址，才能跳转到正确的位置上去。

Exercise 3. Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in inc/memlayout.h. The size of each stack is KSTKSIZE bytes plusKSTKGAP bytes of unmapped guard pages. Your code should pass the new check in check_kern_pgdir().

```
281
282     int i;
283     for (i = 0; i < NCPU; i++)
284         boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE - i * (KSTKSIZE + KSTKGAP), KSTKSIZE, PADDR(percpu_kstacks[i]), PTE
285 }
```

根据cpu的数目NCPU，为每个核都分配一个内核栈，每个内核栈的大小是KSTKSIZE，而内核栈之间的间距是KSTKGAP。

Exercise 4. The code in trap_init_percpu() (kern/trap.c) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.)

```
153     // LAB 4: Your code here:
154     int cid = thiscpu -> cpu_id;
155     assert(cid == cpunum());
156     // Setup a TSS so that we get the right stack
157     // when we trap to the kernel.
158     //ts.ts_esp0 = KSTACKTOP;
159     //ts.ts_ss0 = GD_KD;
160     //ts.ts_iomb = sizeof(struct Taskstate);
161     thiscpu -> cpu_ts.ts_esp0 = KSTACKTOP - cid * (KSTKSIZE + KSTKGAP);
162     thiscpu -> cpu_ts.ts_ss0 = GD_KD;
163
164     // Initialize the TSS slot of the gdt.
165     //gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),sizeof(struct Taskstate) - 1, 0);
166     //gdt[GD_TSS0 >> 3].sd_s = 0;
167     gdt[(GD_TSS0 >> 3) + cid] = SEG16(STS_T32A, (uint32_t) (&(thiscpu -> cpu_ts)), sizeof(struct Taskstate) - 1, 0);
168     gdt[(GD_TSS0 >> 3) + cid].sd_s = 0;
169
170     // Load the TSS selector (like other segment selectors, the
171     // bottom three bits are special; we leave them 0)
172     ltr(GD_TSS0 + cid * 8);
173
174     // Load the IDT
175     lidt(&idt_pd);
```

内核栈的栈地址存储在TSS段中，trap_init_percpu()为每个核的TSS进行初始化任务。在用户态向内核态切换的时候,需要用到TSS段中的部分信息,其中就包括esp0和ss0两个信息,esp0指向相应的CPU的内核栈,ss0指向内核的堆栈段寄存器。

```
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
```

完成以上函数后，make qemu CPUS=4的结果。

Exercise 5. Apply the big kernel lock as described above, by calling lock_kernel() and unlock_kernel() at the proper locations.

进入到内核临界区之前都需要加锁，离开内核临界区之后需要尽快释放锁.

1.在启动的时候，BSP启动其余的CPU之前，BSP需要取得内核锁.

2.Mp_main中，也就是CPU被启动之后执行的第一个函数，应该是调用调度函数，选择一个进程来执行的，但是在执行调度函数之前，必须获取锁.

3.可以访问临界区的CPU只能有一个，所以从用户态陷入到内核态的话，要加锁，因为可能多个CPU同时陷入内核态.

4.Env_run函数执行结束之后，就将跳回到用户态，此时离开内核，也就是需要将内核锁释放.

在以上提到的地方调用lock_kernel和unlock_kernel()函数，代码简单在这里就不做截图，详情见代码。

Question

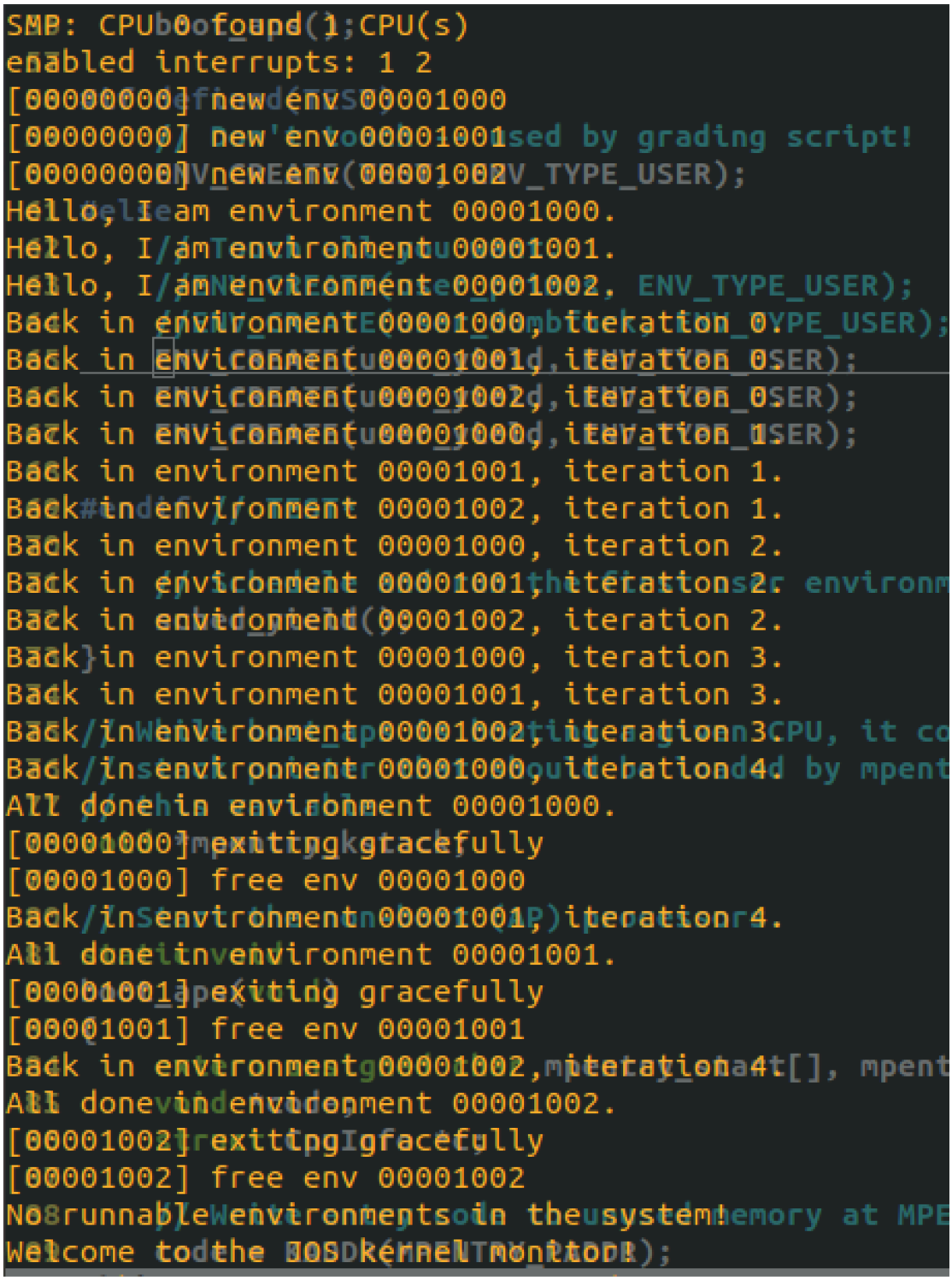
2. It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

因为不同的内核栈上可能保存有不同的信息，在一个CPU从内核退出来之后，有可能在内核栈中留下了一些将来还有用的数据，所以一定要有单独的栈。

Exercise 6. Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield(). Make sure to invoke sched_yield() in mp_main. Modify kern/init.c to create three (or more!) environments that all run the program user/yield.

```
31      // LAB 4: Your code here.
32      idle = curenv;
33      size_t cur, i;
34      if (idle)
35          cur = (ENVX(idle -> env_id) + 1) % NENV;
36      else
37          cur = 0;
38      for (i = 0; i < NENV; i++, cur = (cur + 1) % NENV)
39      {
40          if (envs[cur].env_status == ENV_RUNNABLE)
41          {
42              env_run(envs + cur);
43              return;
44          }
45      }
46      if (idle && idle -> env_status == ENV_RUNNING)
47      {
48          env_run(idle);
49          return;
50      }
51      // sched_halt never returns
52      sched_halt();
```

sched_yield()函数实现了一个轮转调度。轮转调度思想是：当env[i]进程来调用sched_yield()函数让出CPU，此时，系统会从i开始，不停的往下寻找状态为runnable的进程，然后执行那个进程。如果遍历了所有的进程队列，发现没有进程满足运行条件，若原进程满足运行条件，即状态是runnable,则运行原进程，若原进程不满足运行条件，即原进程被阻塞或者被杀死，则调用 sched_halt()， 让CPU停止工作，直到下次时钟中断，再重新执行上面的过程。



在kern/init.c中创建3个运行yield.c的进程后，make qemu CPU=2后的运行结果。

Question

3. In your implementation of env_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e,

the argument to env_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

4. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

3. 因为当前是运行在系统内核中的，而每个进程的页表中都是存在内核映射的，之前也说过，每个进程页表中虚地址高于 UTOP 之上的地方，只有 UVPT 不一样，其余的都是一样的，只不过在用户态下是看不到的。所以虽然这个时候的页表换成了下一个要运行的进程的页表，但是 curenv 的地址没变，映射也没变，还是依然有效。

4. 这个是进程上下文切换，保存寄存器发生在执行系统调用（例如 sys_yield）或时钟中断时，相关代码在 trapentry.S 中。

Exercise 7. Implement the system calls described above in kern/syscall.c. You will need to use various functions in kern/pmap.c and kern/env.c, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

```
77 static envid_t
78 sys_exofork(void)
79 {
80     // Create the new environment with env_alloc(), from kern/env.c.
81     // It should be left as env_alloc created it, except that
82     // status is set to ENV_NOT_RUNNABLE, and the register set is copied
83     // from the current environment -- but tweaked so sys_exofork
84     // will appear to return 0.
85
86     // LAB 4: Your code here.
87     struct Env *e;
88     int r = env_alloc(&e, curenv -> env_id);
89     if (r < 0)
90         return r;
91     e -> env_status =ENV_NOT_RUNNABLE;
92     e -> env_tf = curenv -> env_tf;
93     e -> env_tf.tf_regs.reg_eax = 0;
94     return e -> env_id;
95
96     //panic("sys_exofork not implemented");
97 }
```

调用env_alloc()函数。此函数做一系列的准备工作，生成存放页表的页，初始化页表内容等。然后将父进程的上下文内容全部拷贝过来，除了返回值eax。但是此时应该将这个进程设置为不可运行，因为还没有将页表映射复制过来。

```
106 static int
107 sys_env_set_status(envid_t envid, int status)
108 {
109     // Hint: Use the 'envid2env' function from kern/env.c to translate an
110     // envid to a struct Env.
111     // You should set envid2env's third argument to 1, which will
112     // check whether the current environment has permission to set
113     // envid's status.
114
115     // LAB 4: Your code here.
116
117     if (status != ENV_RUNNABLE && status != ENV_RUNNING)
118         return -E_INVAL;
119     struct Env *e;
120     if (envid2env(envid, &e, 1) < 0)
121         return -E_BAD_ENV;
122     e -> env_status = status;
123     return 0;
124     //panic("sys_env_set_status not implemented");
125 }
```

将状态设置为可执行或者不可执行。

```
163 static int
164 sys_page_alloc(envid_t envid, void *va, int perm)
165 {
166     // Hint: This function is a wrapper around page_alloc() and
167     //   page_insert() from kern/pmap.c.
168     //   Most of the new code you write should be to check the
169     //   parameters for correctness.
170     //   If page_insert() fails, remember to free the page you
171     //   allocated!
172
173     // LAB 4: Your code here.
174     if ((uintptr_t) va >= UTOP || (uintptr_t)va % PGSIZE)
175         return -E_INVAL;
176     if ((perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P))
177         return -E_INVAL;
178     struct Env *e;
179     if (envid2env(envid, &e, 1) < 0)
180         return -E_BAD_ENV;
181     struct PageInfo *pp = page_alloc(ALLOC_ZERO);
182     if (pp == NULL)
183         return -E_NO_MEM;
184     if (page_insert(e -> env_pgdir, pp, va, perm) < 0)
185     {
186         page_free(pp);
187         return -E_NO_MEM;
188     }
189     return 0;
190     //panic("sys_page_alloc not implemented");
191 }
```

此函数申请一页物理内存，然后把它映射到虚拟地址va上去。

```
209 static int
210 sys_page_map(envid_t srcenvid, void *srcva,
211          envid_t dstenvid, void *dstva, int perm)
212 {
213     // Hint: This function is a wrapper around page_lookup() and
214     //   page_insert() from kern/pmap.c.
215     //   Again, most of the new code you write should be to check the
216     //   parameters for correctness.
217     //   Use the third argument to page_lookup() to
218     //   check the current permissions on the page.
219
220     // LAB 4: Your code here.
221     struct Env *srce, *dste;
222     if (envid2env(srcenvid, &srce, 1) < 0 || envid2env(dstenvid, &dste, 1) < 0)
223         return -E_BAD_ENV;
224     if ((uintptr_t)srcva >= UTOP || (uintptr_t)srcva % PGSIZE || (uintptr_t)dstva
225         return -E_INVAL;
226     if (perm && (PTE_U | PTE_P) != (PTE_U | PTE_P))
227         return -E_INVAL;
228     pte_t *pte;
229     struct PageInfo *pp = page_lookup(srce -> env_pgdir, srcva, &pte);
230     if (pp == NULL || ((~(* pte) & PTE_W) && (perm & PTE_W)))
231         return -E_INVAL;
232     if (page_insert(dste -> env_pgdir, pp, dstva, perm) < 0)
233         return -E_NO_MEM;
234     return 0;
235     //panic("sys_page_map not implemented");
236 }
```

此函数将进程id为srcenvid的进程的srcva处的物理页的内容，映射到进程id为dstenvid的进程的dstva处。

```
245 static int
246 sys_page_unmap(envid_t envid, void *va)
247 {
248     // Hint: This function is a wrapper around page_remove().
249
250     // LAB 4: Your code here.
251     if ((uintptr_t)va >= UTOP || (uintptr_t)va % PGSIZE)
252         return -E_INVAL;
253     struct Env *e;
254     if (envid2env(envid, &e, 1) < 0)
255         return -E_BAD_ENV;
256     page_remove(e -> env_pgdir, va);
257     return 0;
258
259     //panic("sys_page_unmap not implemented");
260 }
```

解除以上的映射。

dumbfork.c做的事情就是首先获取新生成的子进程的进程id。然后将父进程的进程空间中的每一页的内容都复制过去，对于处在地址addr处的一页，首先为子进程申请一个物理页，然后将这个物理页映射到子进程虚拟地址addr处。然后同时将这个物理页页映射到进程Id为0的进程的虚拟地址PTSIZE处，然后利用

memmove，将父进程addr处的内容移动到进程id为0de进程的虚地址PTSIZE处，此时对应的物理页也就有了相同的内容。

```
dumbfork: OK (1.6s)
Part A score: 5/5
```

make grade ，通过Part A测试。

Part B

Exercise 8. Implement the sys_env_set_pgfault_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

```
136 sys_env_set_pgfault_upcall(envid_t envid, void *func)
137 {
138     // LAB 4: Your code here.
139     //panic("sys_env_set_pgfault_upcall not implemented");
140     struct Env *e;
141     if (envid2env(envid, &e, 1) < 0)
142         return -E_BAD_ENV;
143     e -> env_pgfault_upcall = func;
144     return 0;
145 }
```

sys_env_set_pgfault_upcall 函数()将当前进程的 page fault 处理例程设置为 func 指向的函数

Exercise 9. Implement the code in page_fault_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

```
74
75      if (curenv -> env_pgfault_upcall)
76      {
77          uintptr_t esp;
78          if (tf -> tf_esp >=  UXSTACKTOP - PGSIZE && tf -> tf_esp < UXSTACKTOP)
79              esp = tf -> tf_esp - sizeof(struct UTrapframe) - 4;
80          else
81              esp = UXSTACKTOP - sizeof(struct UTrapframe);
82          user_mem_assert(curenv, (void *)esp, sizeof(struct UTrapframe), PTE_W | PTE_P | PTE_U);
83
84          struct UTrapframe *utf = (struct UTrapframe *)esp;
85          utf -> utf_fault_va = fault_va;
86          utf -> utf_err = tf -> tf_err;
87          utf -> utf_regs = tf -> tf_regs;
88          utf -> utf_eip = tf -> tf_eip;
89          utf -> utf_eflags = tf -> tf_eflags;
90          utf -> utf_esp = tf -> tf_esp;
91          tf -> tf_esp = esp;
92          tf -> tf_eip = (uintptr_t) curenv -> env_pgfault_upcall;
93          env_run(curenv);
94      }
95      // Destroy the environment that caused the fault.
96      cprintf("[%08x] user fault va %08x ip %08x\n",
97          curenv->env_id, fault_va, tf->tf_eip);
98      print_trapframe(tf);
99      env_destroy(curenv);
00  }
```

如果当前已经在用户错误栈上了，那么需要留出4个字节，否则不需要。即在当前的错误栈顶的位置向下留出保存UTrapframe的空间，然后将tf中的参数复制过来。修改当前进程的程序计数器和栈指针，然后重启这个进程，此时就会在用户错误栈上运行中断处理程序了。

如果异常栈发生了overflow怎么办？

用户异常栈只有一页，一旦溢出，访问的就是内核都没有访问权限的空间，会发生内核空间中的page fault，直接panic.

Exercise 10. Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

```
68
69      movl 0x28(%esp), %edx
70      subl $0x4, 0x30(%esp)
71      movl 0x30(%esp), %eax
72      movl %edx, (%eax)
73      addl $0x8, %esp
74
75      // Restore the trap-time registers.  After you do this, you
76      // can no longer modify any general-purpose registers.
77      // LAB 4: Your code here.
78      popal
79
80      // Restore eflags from the stack.  After you do this, you can
81      // no longer use arithmetic operations or anything else that
82      // modifies eflags.
83      // LAB 4: Your code here.
84      addl $0x4, %esp
85      popfl
86
87      // Switch back to the adjusted trap-time stack.
88      // LAB 4: Your code here.
89      popl %esp
90
91      // Return to re-execute the instruction that faulted.
92      // LAB 4: Your code here.
93      ret
```

1：往 trap-time esp 所指的栈顶后面的空位写入 trap-time eip 并将 trap-time esp 往下移指向该位置。

2：跳过 fault_va 和 err，然后恢复通用寄存器。

3：跳过 eip，然后恢复 efalgs，如果先恢复 eip 的话，指令执行的位置会改变，所以这里必须跳过。

4：恢复 esp，如果第一处不将 trap-time esp 指向下一个位置，这里 esp 就会指向之前的栈顶。

5：由于第一处的设置，现在 esp 指向的值为 trap-time eip，所以直接 ret 即可达到恢复上一次执行的效果。

Exercise 11. Finish set_pgfault_handler() in lib/pgfault.c.

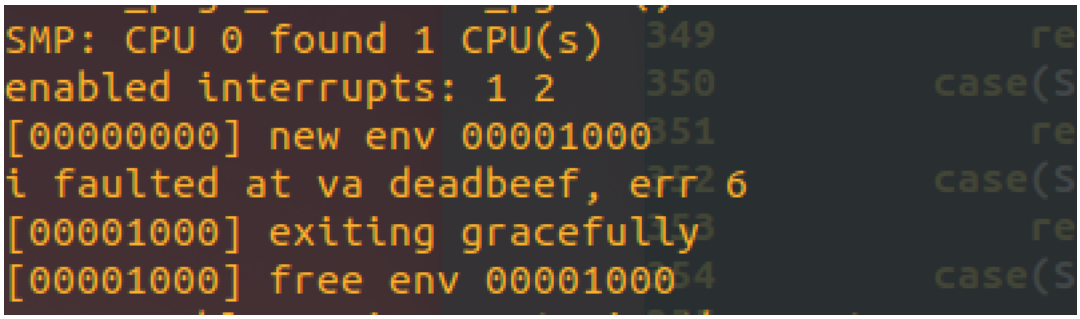
```
24 void
25 set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
26 {
27     int r;
28
29     if (_pgfault_handler == 0) {
30         // First time through!
31         // LAB 4: Your code here.
32         void *addr = (void *)(UXSTACKTOP - PGSIZE);
33         r = sys_page_alloc(sys_getenvid(), addr, PTE_W | PTE_U | PTE_P);
34         if (r < 0)
35             panic("No memory.\n");
36         //panic("set_pgfault_handler not implemented");
37     }
38
39     // Save handler pointer for assembly to call.
40     _pgfault_handler = handler;
41     if(sys_env_set_pgfault_upcall(sys_getenvid(), _pgfault_upcall) < 0)
42         panic("sys_env_set_pgfault_upcall failed.\n");
43 }
```

set_pgfault_handler()函数主要是为进程设定处理历程，同时分配错误栈.

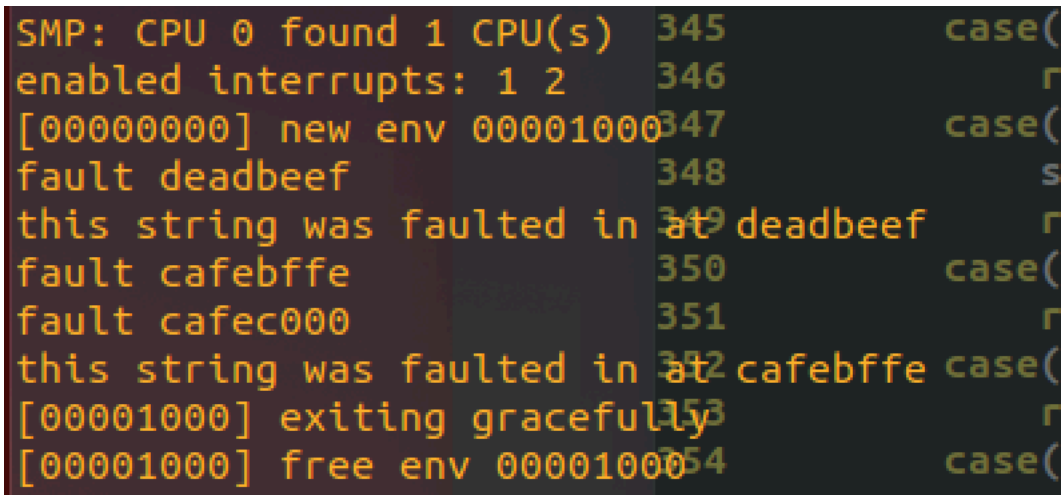
make run-faultread

```
[00000000] new env 00001000
[00001000] user fault va 00000000 ip 00800039
TRAP frame at 0xf02af000 from CPU 0
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x0000000e Page Fault
  cr2  0x00000000
  err  0x00000004 [user, read, not-present]
  eip  0x00800039
  cs   0x----001b
  flag 0x00000096
  esp  0xeebfdfb8
  ss   0x----0023
[00001000] free env 00001000
```

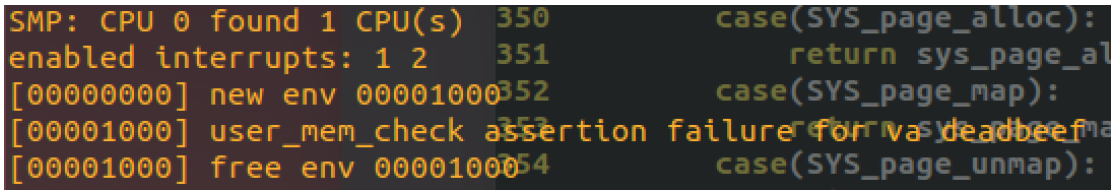
make run-faultdie



make run-faultalloc



make run-faultallocbad



根据要求的测试，结果如图所示。

Exercise 12. Implement fork, duppage and pgfault in lib/fork.c. Test your code with the forktree program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
16 {
17     void *addr = (void *) utf->utf_fault_va;
18     uint32_t err = utf->utf_err;
19     int r;
20
21     // Check that the faulting access was (1) a write, and (2) to a
22     // copy-on-write page.  If not, panic.
23     // Hint:
24     //   Use the read-only page table mappings at uvpt
25     //   (see <inc/memlayout.h>).
26
27     // LAB 4: Your code here.
28
29     if (!(err & FEC_WR) || !(uvpt[PGNUM(addr)] & PTE_COW) || !(uvpt[PGNUM(addr)] & PTE_
30         panic("Not copy on write.\n");
31     // Allocate a new page, map it at a temporary location (PFTEMP),
32     // copy the data from the old page to the new page, then move the new
33     // page to the old page's address.
34     // Hint:
35     //   You should make three system calls.
36
37     // LAB 4: Your code here.
38
39     addr = ROUNDDOWN(addr, PGSIZE);
40     r = sys_page_alloc(0, (void *)PFTEMP, PTE_P | PTE_W | PTE_U);
41     if (r < 0)
42         panic("sys_page_alloc failed.\n");
43     memcpy(PFTEMP, addr, PGSIZE);
44     r = sys_page_map(0, (void *)PFTEMP, 0, addr, PTE_P | PTE_W | PTE_U);
45     if (r < 0)
46         panic("sys_page_map failed.\n");
47     r = sys_page_unmap(0, (void *)PFTEMP);
48     if (r < 0)
49         panic("sys_page_unmap failed.\n");
50
51     //panic("pgfault not implemented");
52 }
```

首先检查 err 和 pte 是否符合条件。然后借用了一个一定不会被用到的位置PFTEMP，专门用来发生page fault的时候拷贝内容。分配页面到地址PFTEMP复制内容到刚分配的页面中。将虚拟地址 addr（需要向下对齐）映射到分配的页面。最后取消地址 PFTEMP 的映射。

```
65 static int
66 duppage(envid_t envid, unsigned pn)
67 {
68     int r;
69
70     // LAB 4: Your code here.
71
72     void *addr = (void *)(pn * PGSIZE);
73     if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))
74     {
75         r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_U | PTE_P);
76         if (r < 0)
77             panic("duppage: %e\n", r);
78         r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_U | PTE_P);
79         if (r < 0)
80             panic("duppage; %e\n", r);
81     }
82     else
83         sys_page_map(0, addr, envid, addr, PTE_U | PTE_P);
84     //panic("duppage not implemented");
85     return 0;
86 }
```

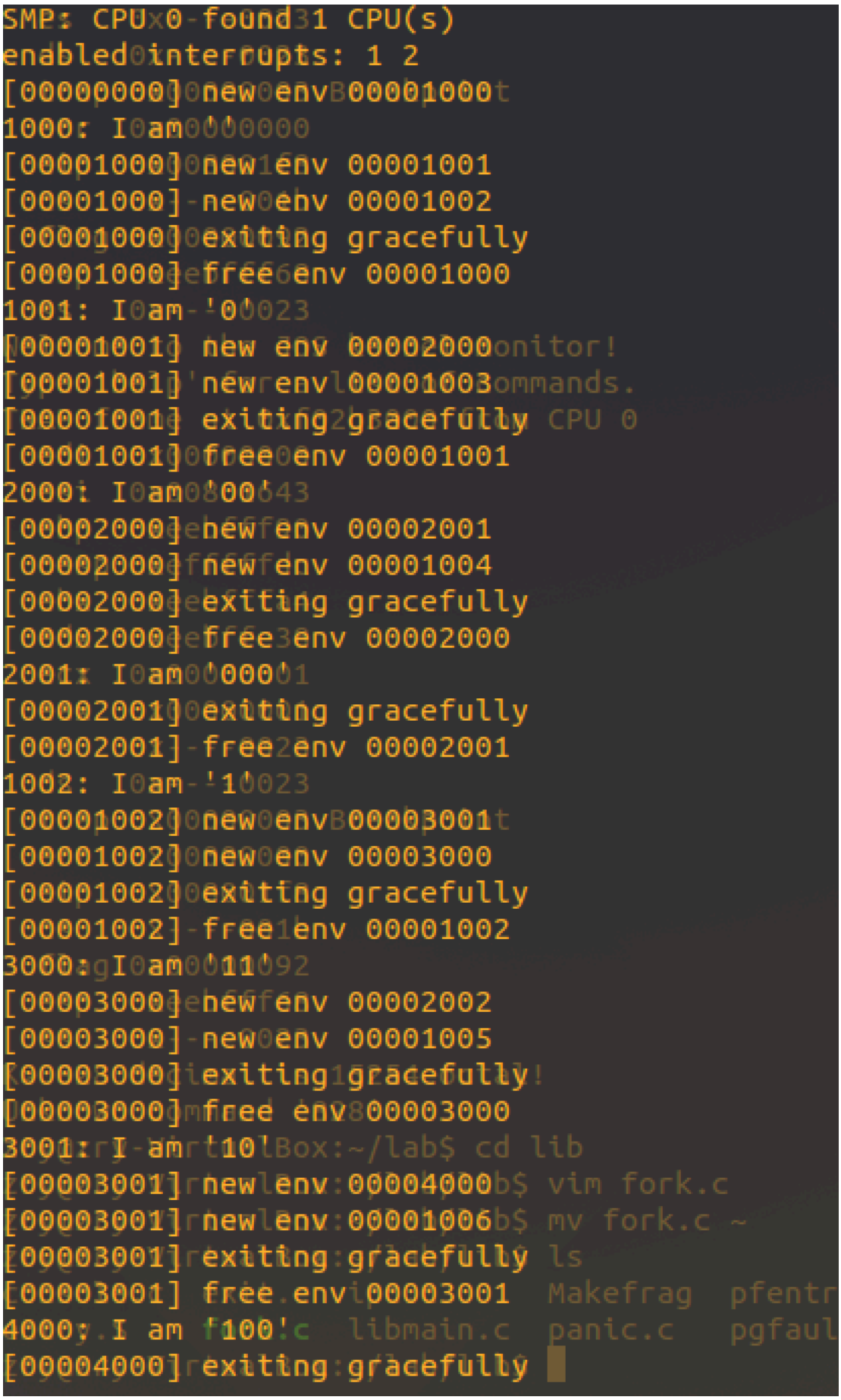
duppage()函数作用就是将当前进程的第 pn 页对应的物理页的映射到 envid 的第

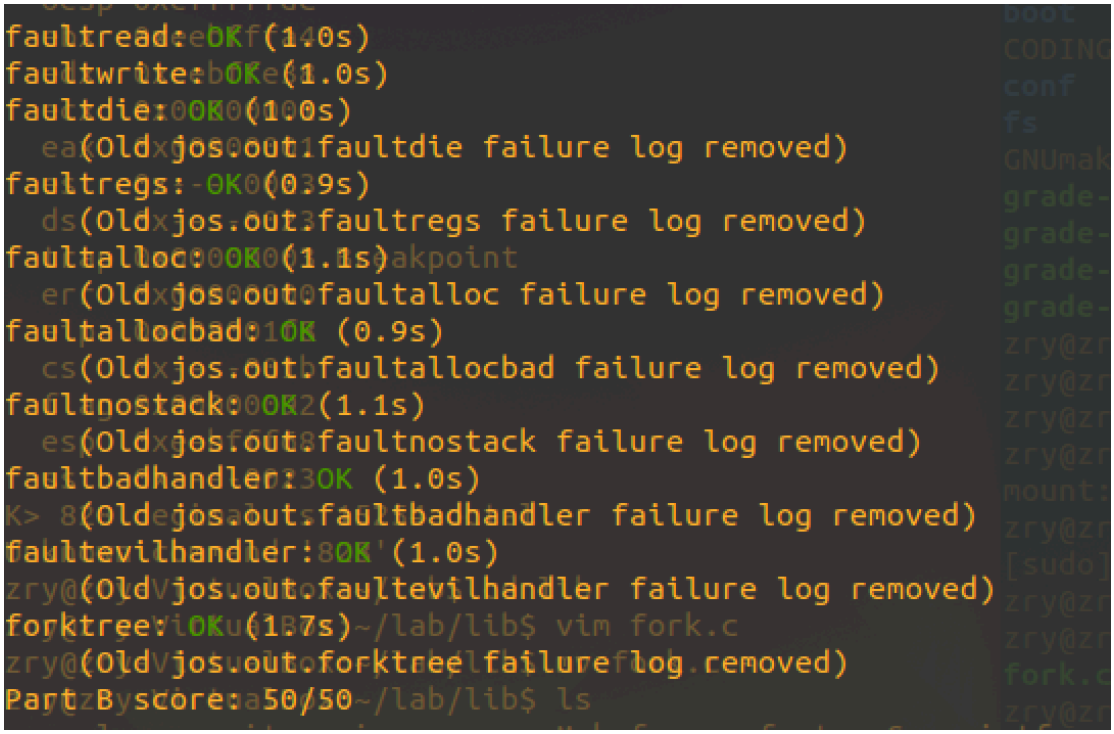
pn 页上去，同时将这一页都标记为 COW.

```
104 envid_t
105 fork(void)
106 {
107     // LAB 4: Your code here.
108
109     int r;
110     set_pgfault_handler(pgfault);
111     //cprintf("==========\n");
112     envid_t envid = sys_exofork();
113     uintptr_t addr;
114     if (envid == 0)
115     {
116         thisenv = envs + ENVX(sys_getenvid());
117         return 0;
118     }
119     if (envid < 0)
120         panic("sys_exofork failed.\n");
121     for (addr = 0; addr < USTACKTOP; addr += PGSIZE)
122     {
123         if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
124             duppage(envid, PGNUM(addr));
125     }
126     if ((r = sys_page_alloc(envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P)) < 0)
127         return r;
128     extern void _pgfault_upcall(void);
129     if ((r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0)
130         return r;
131     sys_env_set_status(envid, ENV_RUNNABLE);
132     return envid;
133     //panic("fork not implemented");
134 }
```

首先需要为父进程设定错误处理例程，这里调用set_pgfault_handler函数是因为当前并不知道父进程是否已经建立了错误栈，没有的话就会建立一个。而sys_env_set_pgfault_upcall则不会建立错误栈。调用sys_exofork准备出一个和父进程状态相同的子进程，状态暂时设置为ENV_NOT_RUNNABLE。然后进行拷贝映射的部分，在当前进程的页表中所有标记为PTE_P的页的映射都需要拷贝到子进程空间中去。用户错误栈必须要新申请一页来拷贝内容。因为copy-on-write就是依靠用户错误栈实现的，所以在fork完成的时候每个进程都有一个栈。

make run-forktree:


SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
1000: I am 0
[00001000] new env 00001001
[00001000] new env 00001002
[00001000] exiting gracefully
[00001000] free env 00001000
1001: I am 0
[00001001] new env 00002000
[00001001] new env 00001003
[00001001] exiting gracefully
[00001001] free env 00001001
2000: I am 0
[00002000] new env 00002001
[00002000] new env 00001004
[00002000] exiting gracefully
[00002000] free env 00002000
2001: I am 0
[00002001] exiting gracefully
[00002001] free env 00002001
1002: I am 1
[00001002] new env 00003001
[00001002] new env 00003000
[00001002] exiting gracefully
[00001002] free env 00001002
3000: I am 1
[00003000] new env 00002002
[00003000] new env 00001005
[00003000] exiting gracefully
[00003000] free env 00003000
3001: I am 1
[00003001] new env 00004000
[00003001] new env 00001006
[00003001] exiting gracefully
[00003001] free env 00003001
4000: I am 100
[00004000] exiting gracefully
lBox:~/lab$ cd lib
b$ vim fork.c
b$ mv fork.c ~
b$ ls
Makefrag pfentr
c libmain.c panic.c pgfaul



make grade ， 通过Part B测试。